CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Università della Svizzera italiana | Faculty of Informatics | Institute of Computational Science

Core Concepts of Parallel Computing

Applied via Message Passing Interface (MPI)

FoM
ICS

Summer School at USI, July 11th 2013

Roberto Croce, USI
Neil Stringfellow, former CSCS fellow
Claudio Gheller, Andreas Jocksch, CSCS

Plan

- **Programming Parallel – Core concepts**
  - **Task or data parallelism**
  - **Data decomposition**
  - **Halo (or "ghost") cells**
  - **Load balancing, speedup and efficiency**
  - **Strong and weak scaling**
  - **Tightly coupled and embarrassingly parallel**
  - **Amdahl's law**

- **Communicators & Groups & Topology**

- **Collectice Communications:**
  - **MPI_Barrier, MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce**

- **Domain Decomposition via:**
  - **MPI_Cart_create, MPI_Vector**

Programming Parallel – Core Concepts

Data or task parallelism ?

1. **Parallelism can be achieved by:**
   - **getting multiple processes or threads to do the same work on different data**
   - **getting different processes or threads to do different jobs**
   - **a combination of both**

2. **If multiple processes/threads carry out the same job on different data we call this data parallel work**

3. **If different processes/threads are given independent jobs to do then we call this task parallel work**

Task decomposition

For a task parallel model we can employ several strategies:

1. **Some codes are based on Multiple Program Multiple Data (MPMD):**
   - as in the climate community codes such as CCSM have used separate programs for calculations involving:
   i) atmosphere, ii) ocean, iii) sea-ice, iv) land, & v) coupling codes
2. **Some programming models such as StarSS are based on the idea that many tasks within a code are independent**
   - The user adds pragmas into their code to specify the independent units and associated data
   - Tasks can then be spawned on appropriate devices
   i) Cell processors, ii) GPUs, iii) SMP cores
3. **OpenMP allows the user to spawn independent tasks on shared memory machines**
4. **Data locality needs to be considered when spawning tasks on distributed memory machines via MPI**

Domain decomposition

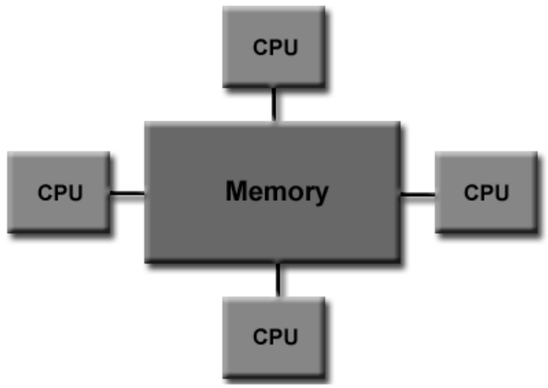
- For data parallel applications we need to decide how the work on the data will be distributed

- For distributed memory machines we also need to decide where the data will be placed
  - ingeneral independent processes can only read/write to their own memory

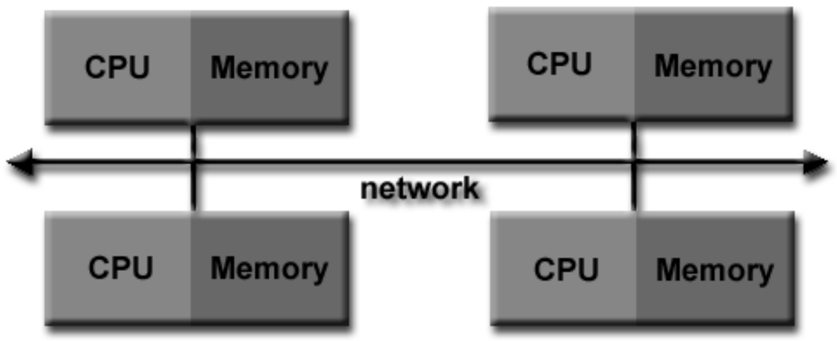
- We refer to the way in which we divide up the data and work as domain decomposition

- Decisions on domain decomposition are a key component of planning how to write a parallel application

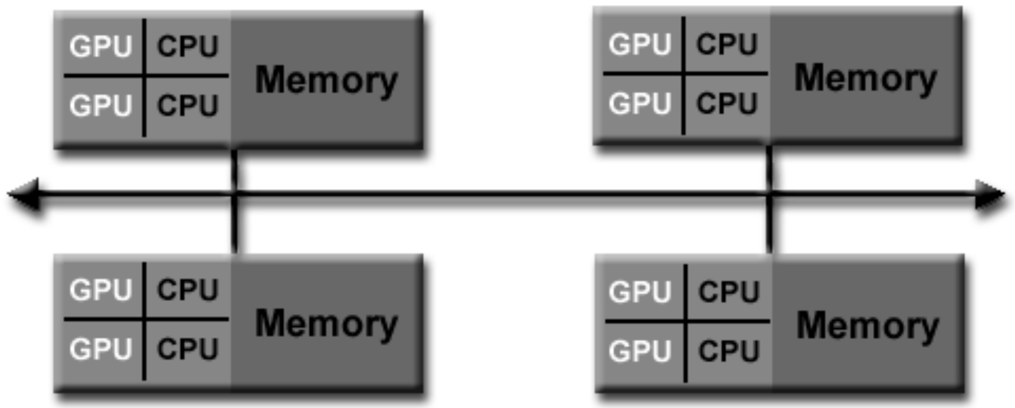
Supercomputer architectures



Shared memory



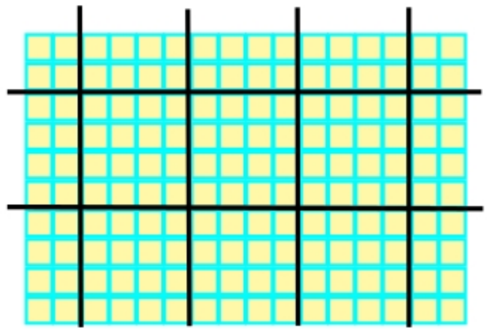
Distributed memory

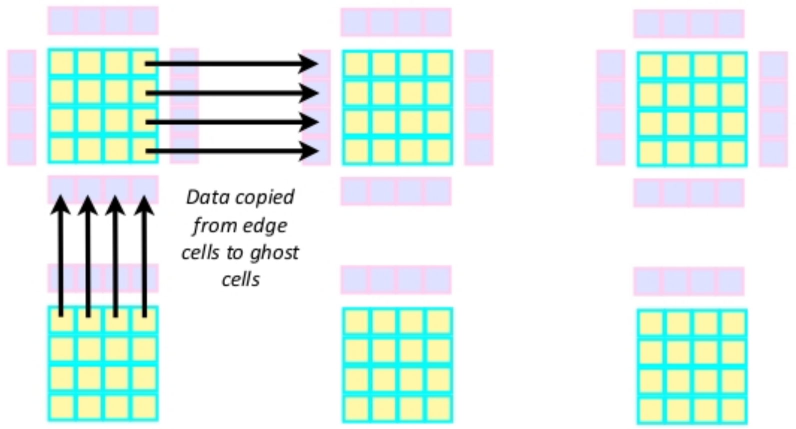


Hybrid GPU-CPU shared/distributed memory

Halo regions also known as “ghost cells”

- In most applications the tasks being done by processors are not independent from the other tasks
  - Tasks need to know about data generated by other tasks
- Since it takes some time to pass messages between processes, copies of essential remote data from other tasks are stored locally
- In the case of grids and meshes the required data is the boundary from adjacent cells in the grid on other processes
- These copies of boundary data are referred to as **halo cells** or **ghost cells**


Grid is decomposed amongst a set of processors



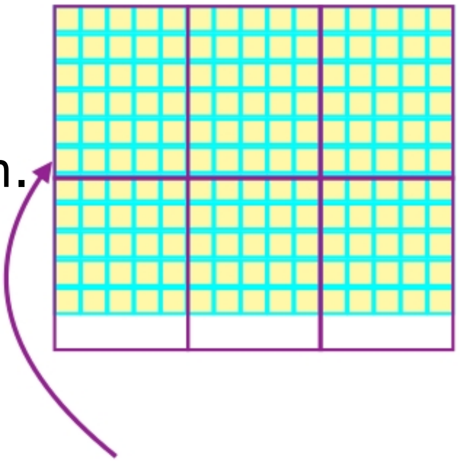
Load balancing

- **When distributing data and work onto processes and threads you want to give each one the same amount to do**
- **The act of equalizing the amount of work to do is called load balancing**
- **Load balancing is one of the most important aspect of ensuring good performance and scalability of parallel applications**
- **Load balancing is often a runtime issue, but it needs to be considered in the design stage of an application**

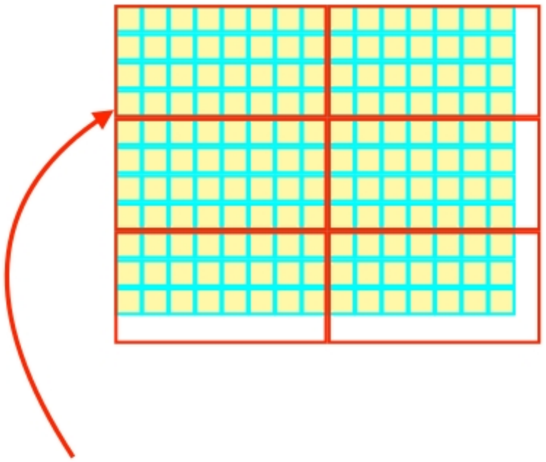
Load balancing example – 2D grid

- **We have 165 grid points (15x11) and 6 process**
- **Try to have a roughly equal number of grid points per task**
- **Minimize the number of grid points in the largest block**
- **Note that some grid points might have more work to do than others – we don't account for this here**
- **Minimize the amount of communication**

less comm.



Largest number of points is 30 (5x6)

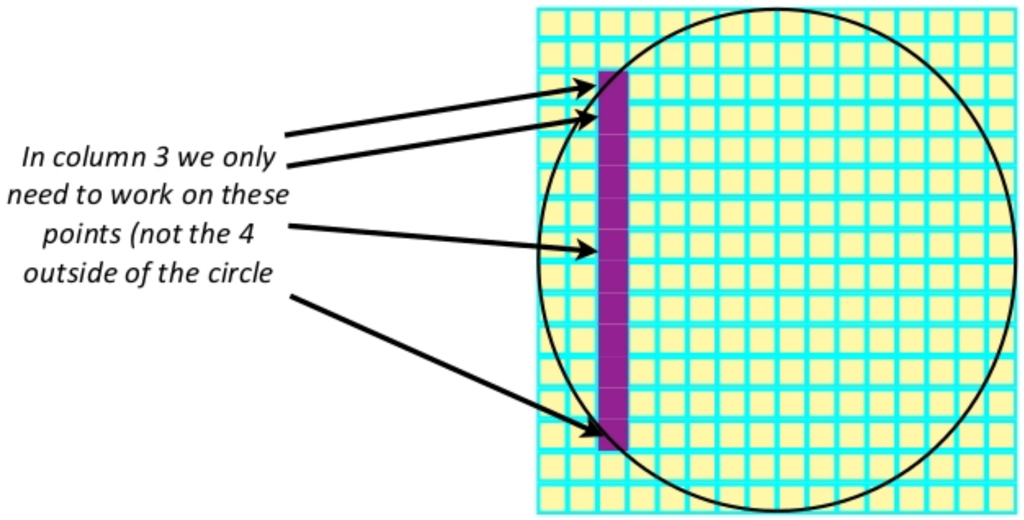


higher comm.

Largest number of points is 32 (8x4)

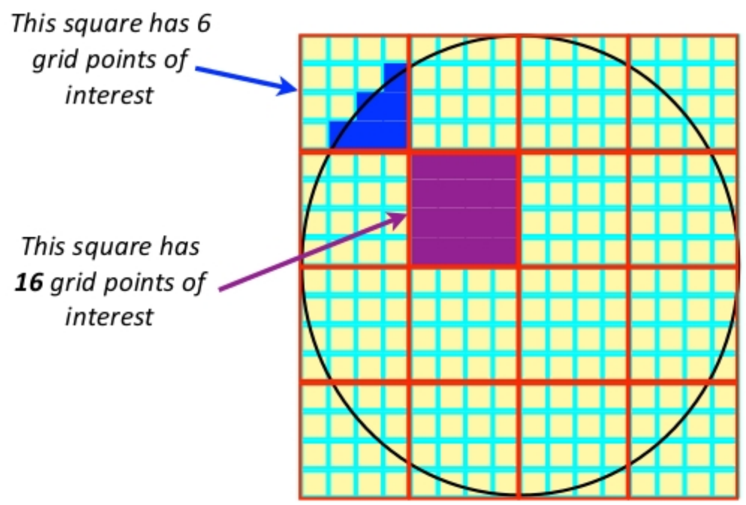
Tricky example – circle in a square

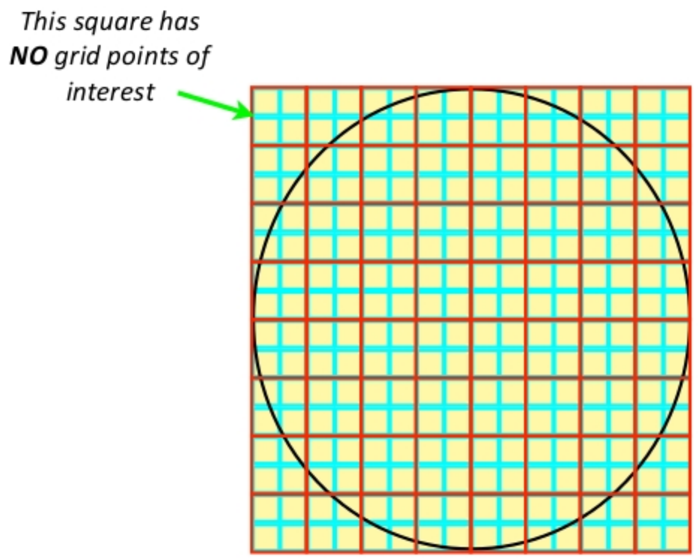
- **In this example the serial code has to work on a sphere contained in a cube from a Cartesian grid**
- **For simplicity we will use a circle in a square**
- **The serial code can find the beginning and end of each column and work on only those points of interest**
  - the points not contained in the circle are of no interest
- **The code works on ~¾ of the points (actually ~π/4)**


In column 3 we only need to work on these points (not the 4 outside of the circle

Circle in a square in parallel

- **A straightforward division into equal-sized domains produces a big load imbalance**
- **In this case we assume 256 (16x16) grid points and 16 processors**
- **Here four processors have no grid points outside the circle**
- **With 64 processors you can have tasks with no work to do**
- **For the case of a cube the situation would be worse**
  - **Only just over half of the grid points are active ($\pi\ r^{(3/8)}$)**


This square has 6 grid points of interest
This square has ***16*** *grid points of interest*



Improving load balance via work stealing

- In some applications it is only possible to see load imbalance at runtime
  - The datasets themselves determine the load
    - i) in weather simulations there might be more work for grid points with precipitation
    - ii) in atmospheric chemistry climate there might be more chemistry during daylight hours
  - Shared resources lead to contention
    - i) more communication for several processes on a node might restrict bandwidth
- In these cases a strategy of work stealing might be employed
- With work stealing a task that has no work left to do looks around for another task that still has lots of work in its queue
- Heuristics need to be employed to determine when it might be too costly to carry out

Speedup and efficiency

- We define the **speedup** to be how much faster a code is on N processors compared to one processor
  - speedup is a measure of reduced time-to-solution
- We define the **efficiency** to be the speedup on N processors divided by the number of processors
  - efficiency is a measure of resource utilisation
  - efficiency is often expressed as a percentage
- If T_1 is the time taken to run on 1 processor and T_N is the time taken to run on N processors then we have

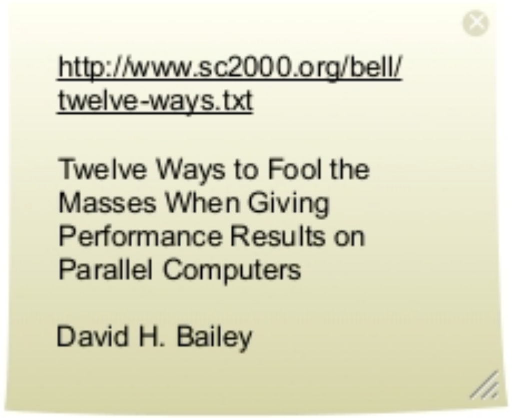
$$\text{Speedup} = T_1 / T_N \qquad \text{Efficiency} = \text{Speedup} / N$$

Strong and weak scaling

- **If you keep the problem size the same as you change the number of processors then we call this strong scaling**
- **If you change the problem size in proportion to the number of processors then we call this weak scaling**
- **Strong scaling is typically harder to achieve than weak scaling**
- **The ratio of memory to flop/s available on modern machines is decreasing and strong scaling is going to become much more important**

Measuring speedup

- When measuring speedup of a program it is important to know what you are measuring!
- On a multi-core systems be aware of shared resources
  - multiple processes sharing the same memory bandwidth
  - multiple processes sharing the same interconnect bandwidth
  - multiple processes sharing the same cache
- Are you measuring the whole application or just a kernel
- Are you measuring weak or strong scaling
- Are you measuring speedup against a 1-processor parallel version or the original serial version


http://www.sc2000.org/bell/twelve-ways.txt

Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers

David H. Bailey

Tightly coupled to embarrassingly parallel

- **Scientific applications can be classified depending upon how much communication is required between tasks during a run**
- **A tightly coupled application requires frequent communication to keep tasks updated as the dependency between tasks is great**
- **A loosely coupled application has tasks which can carry out a reasonable amount of work between communications**
- **An embarrassingly parallel application requires little or no communication between tasks**
- **With tightly coupled applications the cost of communication can be an inhibitor to scaling**

Performing speedup and efficiency tests with the SWE code

Exercise:

- **Compile the parallel SWE-code with scons via:**

```
module switch PrgEnv-cray PrgEnv-gnu
module load scons
module load python/2.7.2
module load git
cd /path/to/SWE
git co master
scons copyenv=true compiler=cray parallelization=mpi
```

- **Allocate processor (for 1h):** `salloc --res=sschool -N1 --time=01:00:00`
- **Run the SWE-code on 1, 2, 4, 8, 16 cores (on one processor) for 320x320 grid-resolution via the following start-script:**

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --time=00:05:00
aprun –n16 ./SWE_cray_release_mpi_augrie -x 320 -y 320 -o output -c 1
```

and investigate the according wall-clock times

- **Compute the according speedup and efficiency values.**
- **Explain the results.**

Amdahl's law

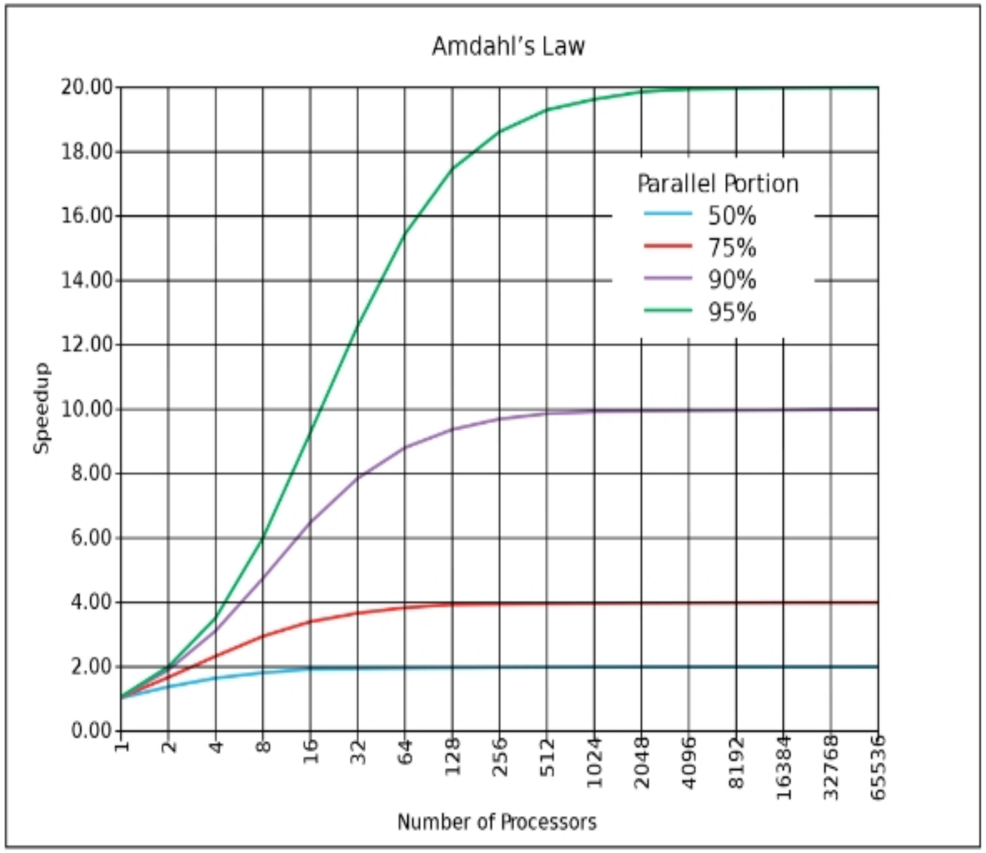
- Amdahl's law is one of the fundamentals of parallel programming
- It states that the speedup that can be achieved is limited by the serial part of an application
- If you have a proportion of your application P that can be perfectly parallelized then the speedup you achieve on N tasks is given by

$$\text{Speedup} = 1 / ((1-P) + P/N)$$

- This means that the maximum achievable speedup (as N tends to infinity) is 1 divided by the serial portion.

Effect of Amdahl's law

- **The diagram (from wikipedia) demonstrates the effect of Amdahl's law**
- **If 50% of a code is perfectly parallelized, the speedup will only ever be 2x**
- **Even where only 5% is serial, you can't do better than a 20x speedup**
- **If 1% of a code is left serial then the speedup can be no more than 100x**
- **Note that this means that for parallelisation via MPI/ OpenMP, you need to parallelize as much of your code as possible**



How many dimensions to parallelise ?

- **Parallelising in all dimensions of a problem allows the maximum possible work distribution**

- **Code could become more complex with more dimensions**

- **Some applications have non-uniformity in all dimensions**
  - most climate and weather codes have large numbers of grid points in the horizontal dimensions but are limited vertically

- **There may be a strict data dependence in some dimensions**

- **Deciding upon the fine-grain nature of parallelism also applies beyond grids**
  - whether to parallelise a model of human population at communities, families, individuals etc.
  - introducing parallelism over atoms, orbitals etc.

Distributing your data

You need to decide how to distribute your data

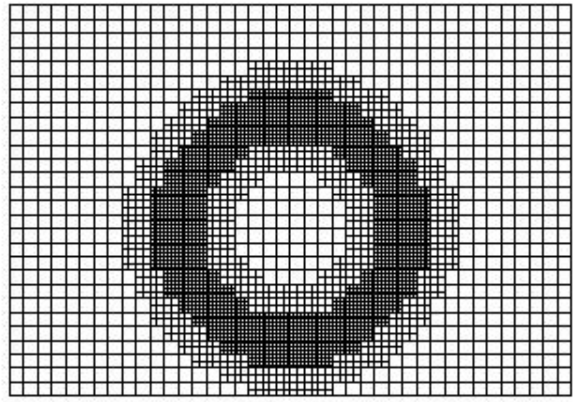
- **Typically you will distribute it in some simple block fashion - each process gets one block of data to work on**
- **You might choose to distribute differently, or a library might force you to distribute data how it wants it**
- **You might also have to decide how you want to distribute the work**

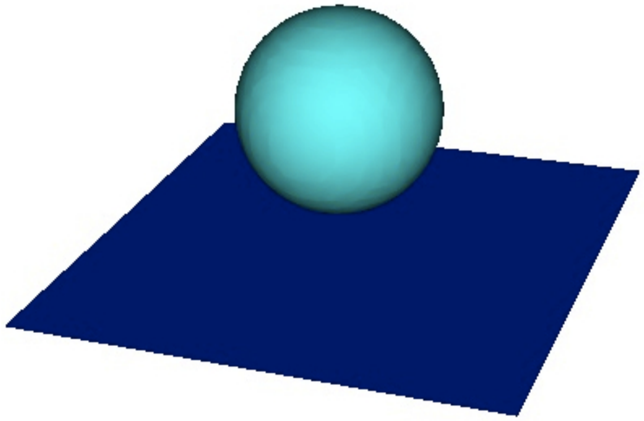
Data transfers, bandwidths and energy

- **When considering the data decomposition you need to keep in mind the amount of data that needs to be transferred**
- **Each data transfer will consume valuable memory bandwidth and interconnect bandwidth**
- **=> Minimising data transfers will improve scalability**
- ***Improvements in memory and interconnect bandwidth are not as fast as improvements in floating point performance***
- **Data transfers consume energy, and greater locality is required as we move towards the Exaflop/s era**

Adaptive grids and distributed memory

- Some people use adaptive grids and mesh refinement to improve algorithmic performance
  - grids and meshes are refined in areas of interest or activity and may be coarsened in other areas
  - Adaptive Mesh Refinement (AMR) is a very powerful tool in this regard. Algorithms and tools are supported in numerical libraries
- On distributed memory machines adaptive mesh refinement algorithms need to redistribute data around the processes
  - refined grids typically need more time steps and take up more memory





Synchronizations and collective communication

In addition to data distribution and point-to-point communication, there are times when all tasks have to communicate together

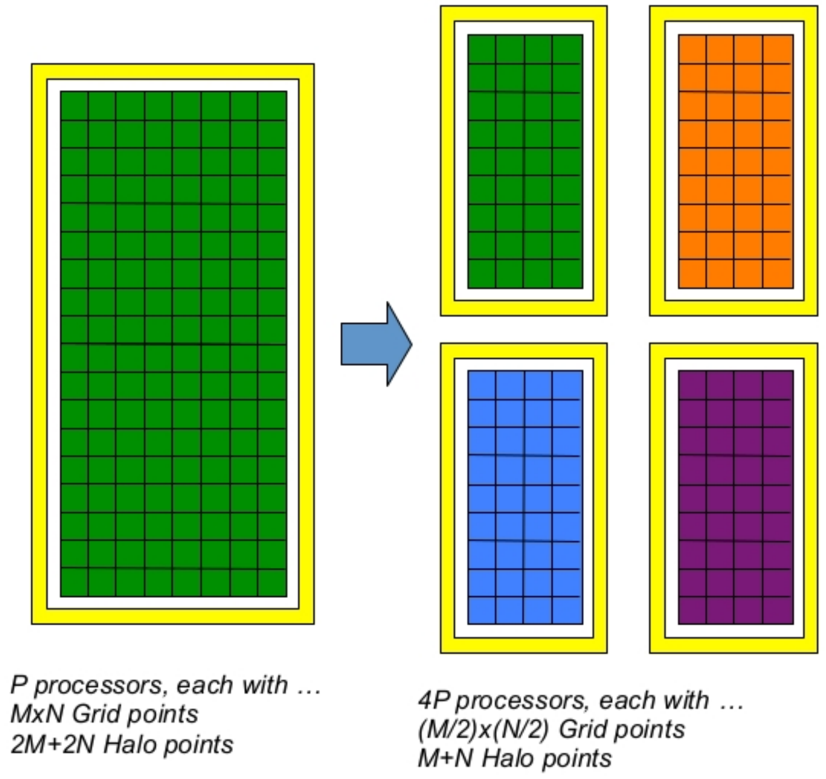
- **These collective communications can be used to**
  - **ensure all tasks have reached a certain point in the code (barrier)**
  - **calculate a common value of interest to all tasks (allreduce)**
  - **distribute some data from one task to all other tasks (broadcast)**
  - **employ efficient data transfer mechanisms when all tasks have to talk to all other tasks (alltoall)**
- **Each of the collective communications causes a synchronization where all tasks have to wait at some point in the code**
  - **different tasks might be at different points when they have to wait**
- **Too many synchronizations can cause a code to slow down**
  - **at large process counts collective communications can often be the dominant part of communication time**

Performance models

It is often helpful to produce a performance model to understand the way that your code behaves

- **The key components of a code might include computation, communication and I/O**
- **Some parts of your code might be memory bandwidth bound, others might be flop bound**
- **A performance model should help you understand the upper limits of realistic speedup**
- **A performance model can help you to identify which parts of a code are most important for performance improvement**

Example of domain decomposition on a 2D grid



Idealised 2D grid layout:

Increasing the number of processors by 4 leads to each processor having

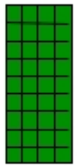
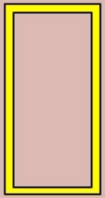
- **one quarter the number of grid points to compute**
- **one half the number of halo points to communicate**

Serial parts of the code do not change.

The same amount of total data needs to be output at each time step.

Using 4 OpenMP threads rather than 4 MPI processes keeps the halo region constant.

Idealised scalability for a 2D grid-based problem

Computation:		Scales O(P) for P processors	*Minor scaling problem – issues of halo memory bandwidth, vector lengths, efficiency of software pipeline etc.*
Communication:		Scales O(√P) for P processors	*<u>Major scaling problem</u> – the halo region decreases slowly as you increase the number of processors*
I/O and serial parts:		No scaling	***<u>Limiting factor in scaling</u>*** *– the same amount of work is carried out, or total data is output at each time step*

Inhibitors to strong scaling

- **There are several factors that can inhibit strong scaling of applications, including**
  - **effect of serial parts of code (Amdahl's law)**
  - **communication latency**
  - **increased computation to communication ratio**
  - **less ability for pipelining and vectorisation**

- **If a code speeds up by a greater amount than the increase in the number of processors added then we say that it is super-scaling**

- **Some codes can exhibit super-scaling for certain datasets**
  - **Frequently this occurs where the local size of a problem becomes small enough to fit into cache at higher process counts**

Limitations for weak scaling

- Typically weak scaling is easier to achieve than strong scaling
- As weak scaling implies a change to size of a problem the complexity might increase
- Increasing problem sizes or using finer resolutions can lead to greater than linear time to solution
  - In computational fluid dynamics the CFL condition requires a reduction in the size of a timestep along with finer resolution
  - In density functional theory the computation might increase as a high power of number of electrons in the system

Parallel Programming via MPI

- **Groups and Communicators**

- **Collective Communications**
  - **MPI_Barrier**
  - **MPI_Bcast**
  - **MPI_Reduce**
  - **MPI_Scatter and MPI_Gather**

- **Parallel Network Topology**
  - **MPI_Cart_Create**
  - **MPI_Type_vector**

Groups and Communicators

- A group is an ordered set of processes, each with a unique integer rank. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. Like groups, communicators are accessible to the programmer only by "handles". The handle for the communicator that comprises all tasks is MPI_COMM_WORLD.

From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which process should be used to construct a communicator.

Groups and Communicators

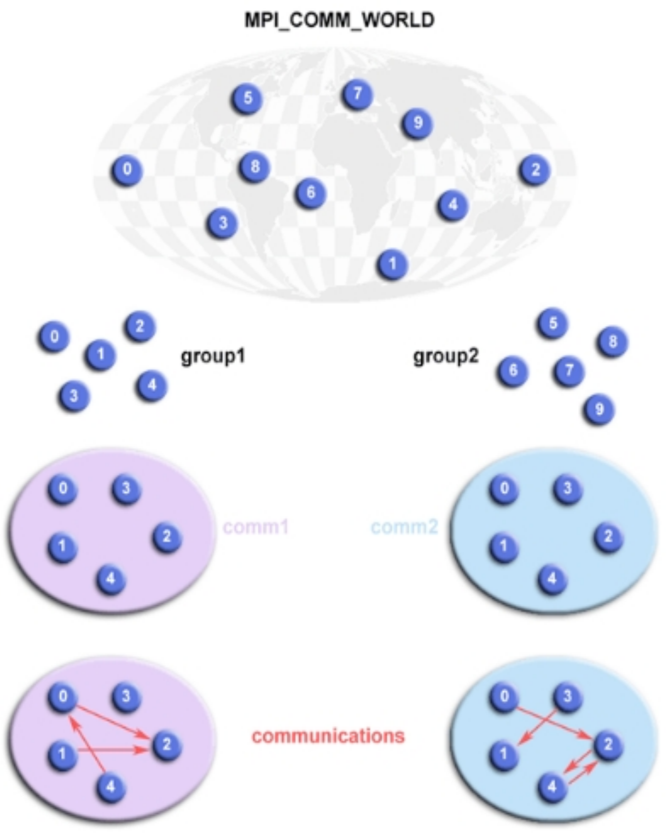
Goals:

- **Allow you to organize tasks, based upon function, into task groups**
- **Enable Collective Communications operations across subset of related tasks**
- **Provide basis for implementing user defined virtual topologies**

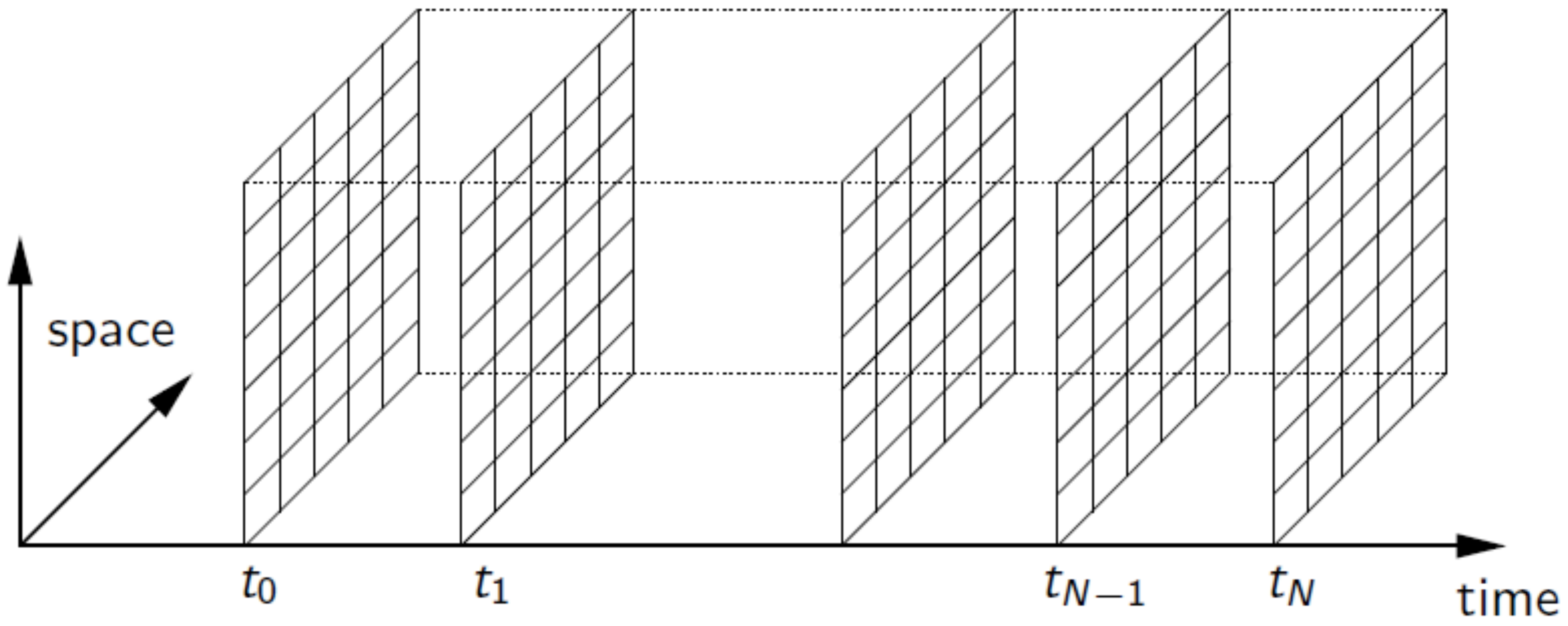
Remarks:

Groups/communicators are dynamic – they can be created and destroyed during program execution.

Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.



Example for processes in more than one group



Parallelisation of PDEs in space & time: each process is contained in a space communicator as well as in a time communicator.

Collective Communications

Communications involving a group of processes called by all processes in a communicator:

- **Barrier**
- **Broadcast**
- **Gather/Scatter**
- **Reduction (sum, max, prod, …)**

Remarks:

-All processes must call the collective routine.

-No non-blocking collective communication.

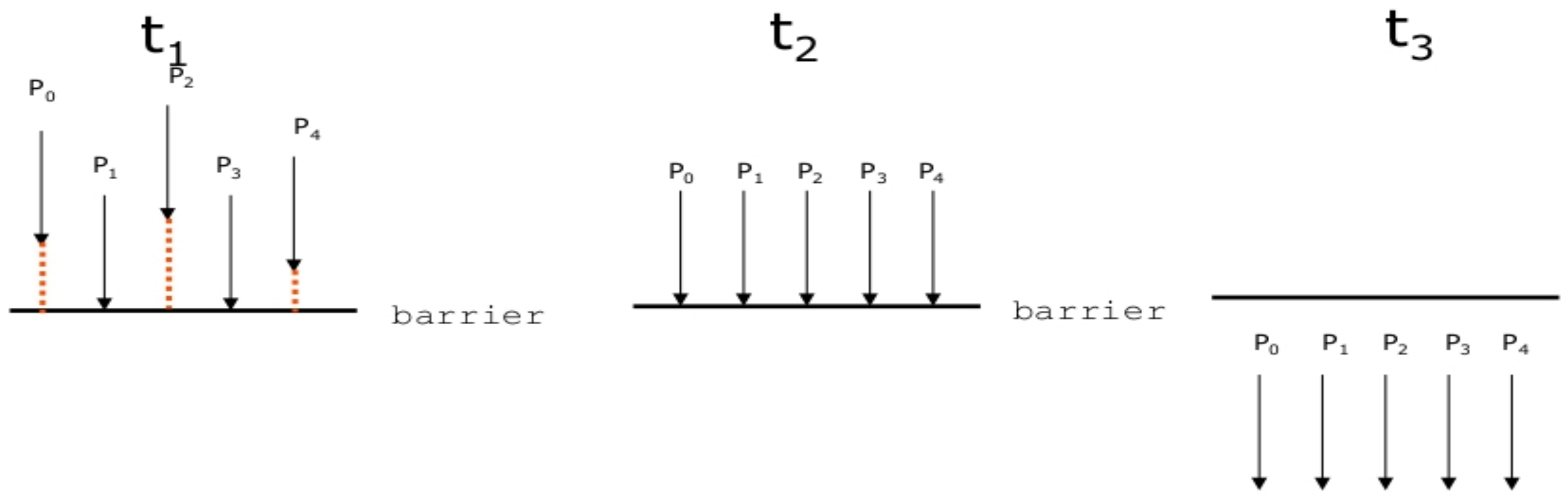
-No tags, the MPI library should use the most efficient communication algorithm for the particular platform.

MPI_Barrier

Stop processes until all processes within a communicator reach the barrier

Fortran:
CALL MPI_BARRIER (comm, ierr) // in fortran: **ierr** in addition

C/C++:
Int MPI_Barrier (MPI_Comm comm)



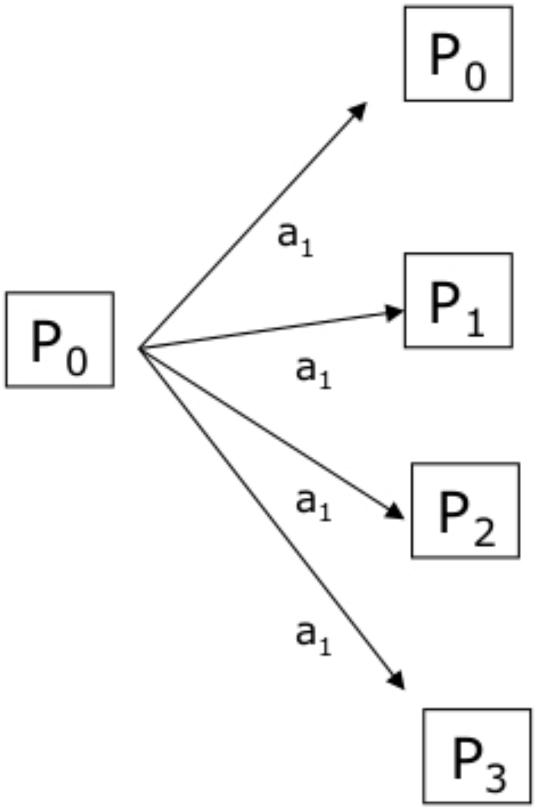
Broadcast (MPI_Bcast)

One-to-all communication: same data sent from root process to all others in the communicator. All processes of a group must call this function.

C/C++:

```
    int MPI_Bcast(void *buf, int count,
    MPI_Datatype datatype, int root, MPI_Comm
    comm)
```

IN/OUT: **buf**=starting address of buffer
IN: **count**=number of entries in buffer (integer)
datatype=data type of buffer (handle)
root=rank of broadcast root (integer)
comm=communicator (handle)



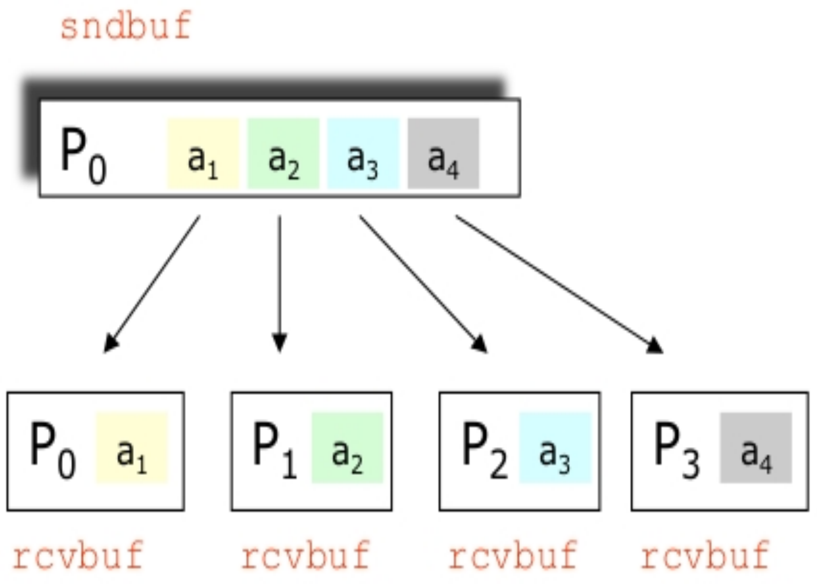
Exercise “broadcast”

- **Compile and run the broadcast-code for different <ProcNr> values between 1-16**

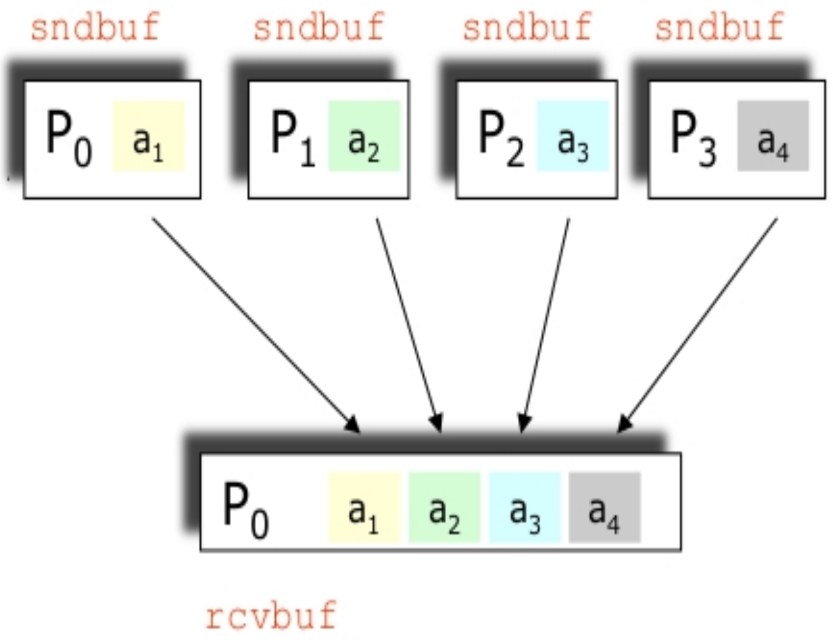
```
CC broadcast_mpi.c –o broadcast
aprun –n<ProcNr.> broadcast
```

Scatter / Gather

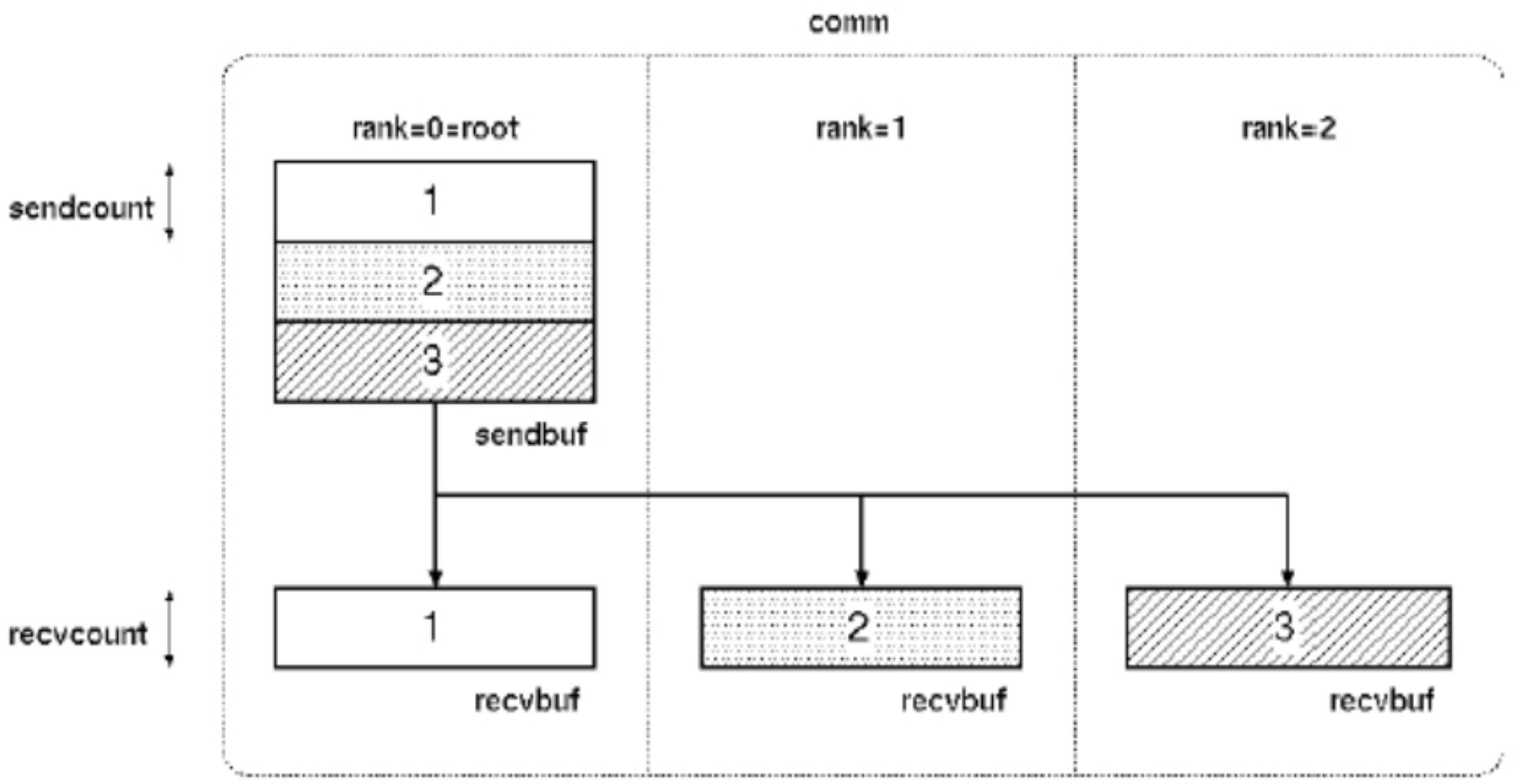
Scatter



Gather



Scatter


comm
rank=0=root
rank=1
rank=2
sendcount
1
2
3
sendbuf
recvcount
1
2
3
recvbuf
recvbuf
recvbuf

MPI_Scatter

One-to-all communication: different data sent from root process to all others in the communicator.

C/C++:

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype
                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

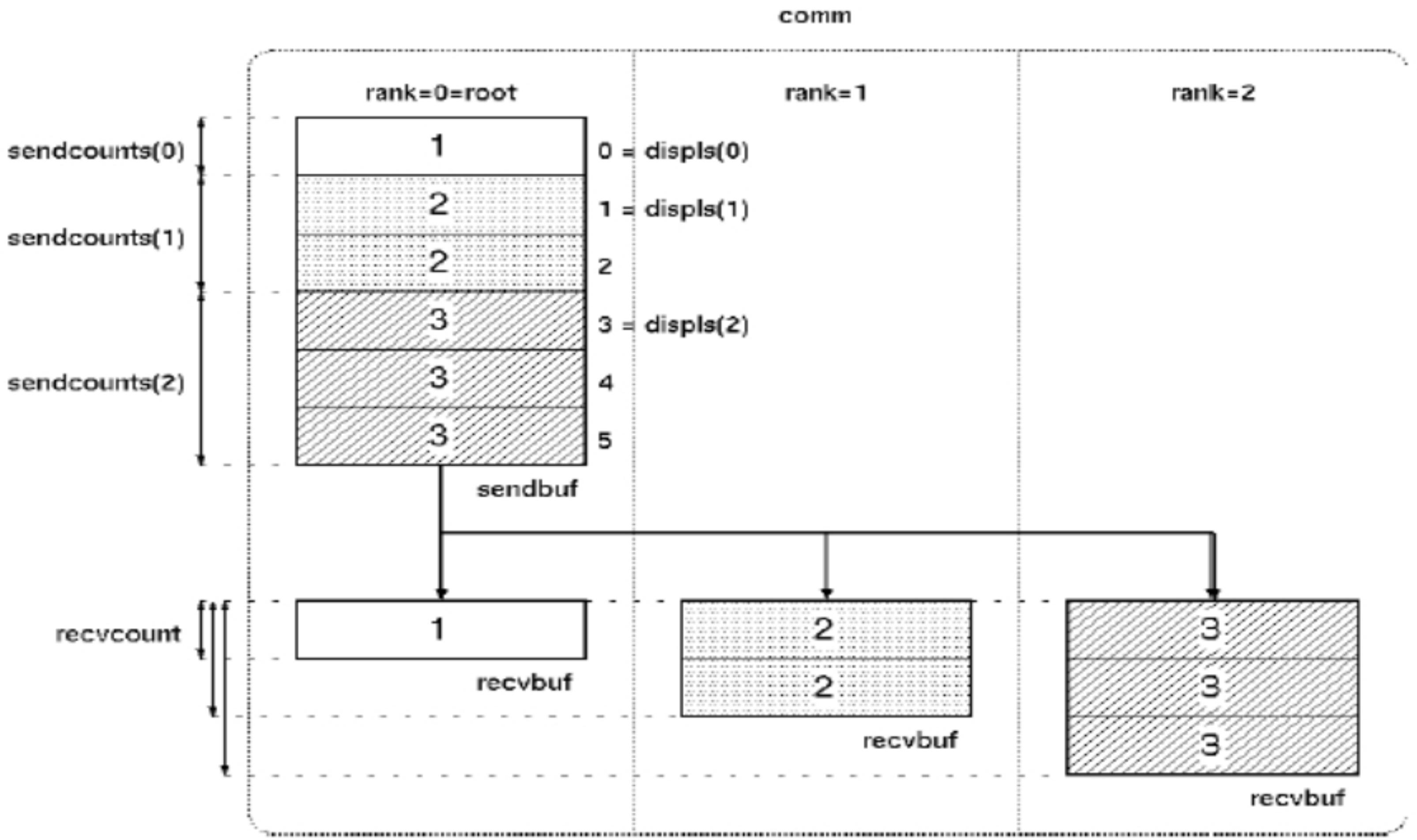
IN:**sendbuf**=address of send buffer (choice, significant only at root)
  sendcount=number of elements sent to each process (integer sig. root)
  sendtype=data type of send buffer elements (significant only at root)
  recvcount=number of elements in receive buffer (integer)
  recvtype=data type of receive buffer elements (handle)
  root=rank of sending process (integer)
  comm=communicator (handle)
OUT: recvbuf=address of receive buffer (choice)

Exercise “scatter”

- **Compile and run the scatter-code for different <ProcNr> values between 1-16**

```
CC scatter_mpi.c –o scatter
aprun –n<ProcNr.> scatter
```

Scatterv: scatter with variable buffer size



MPI_Scatterv

Usage:

C/C++:

```
int MPI_Scatterv( void *sendbuf, int *sendcnts, int *displs,
                  MPI_Datatype sendtype, void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran:

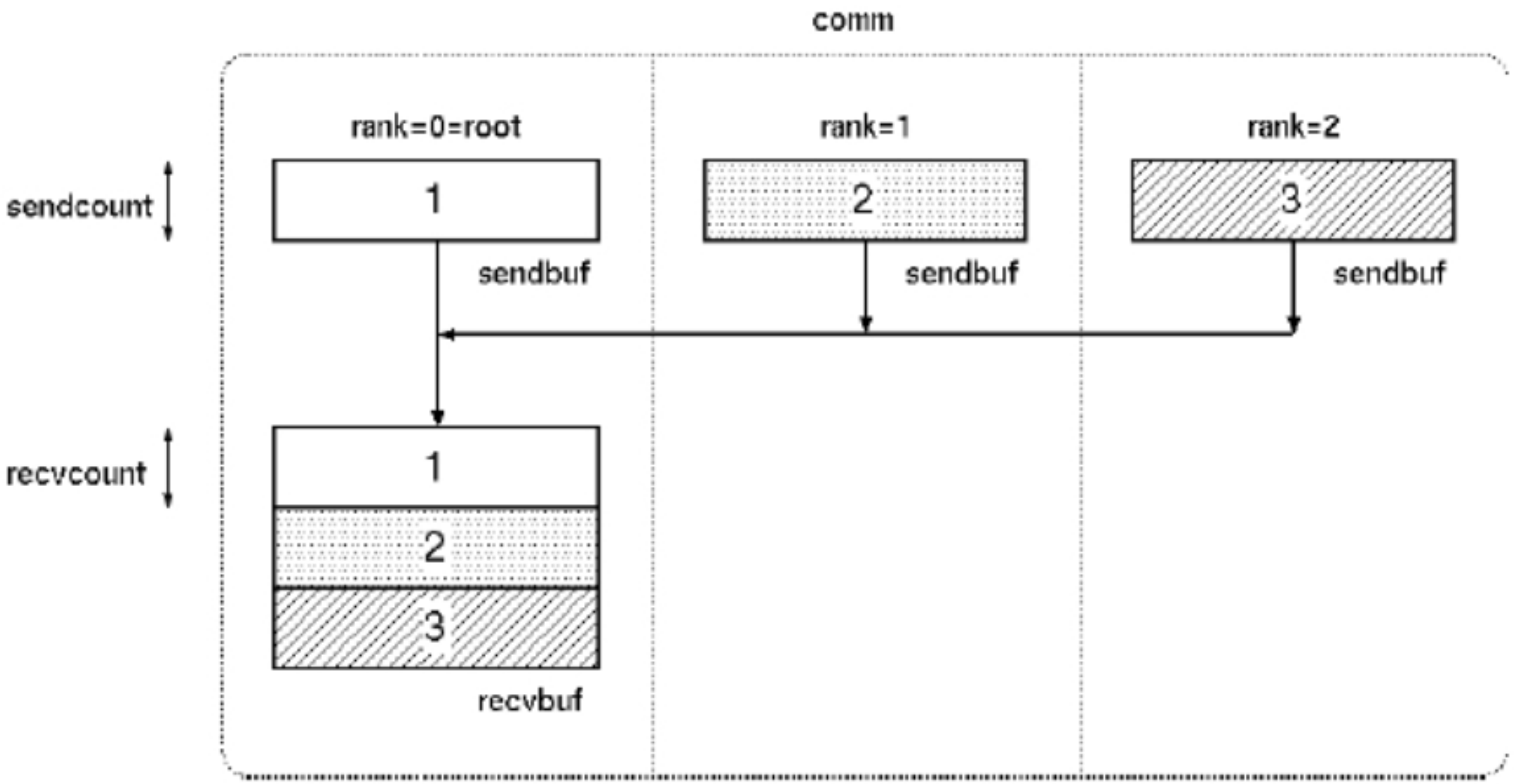
```
CALL MPI_SCATTERV( sendbuf, sendcnts, displs, sendtype, recvbuf,
                   recvcnt, recvtype, root, comm, ierr)
```

Description:

- Distributes individual messages from root to each process in communicator

- Messages can have different sizes and displacements

Gather


comm
rank=0=root
rank=1
rank=2
sendcount
1
2
3
sendbuf
sendbuf
sendbuf
recvcount
1
2
3
recvbuf

MPI_Gather

All-to-one communication: different data collected by the root process from all other processes in the communicator.

C/C++:

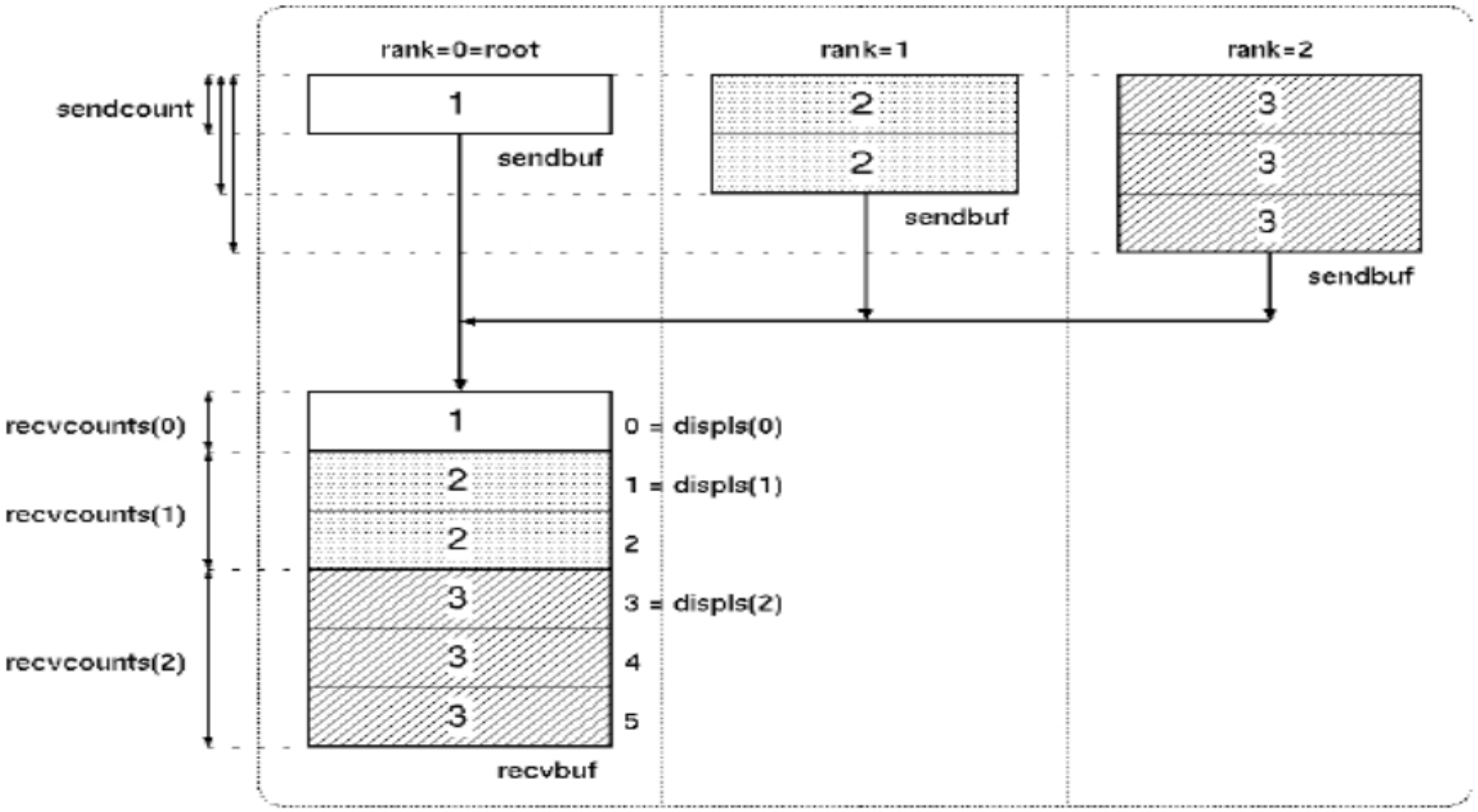
```
    int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype
                   sendtype, void *recvbuf, int recvcnt, MPI_Datatype
                   recvtype, int root, MPI_Comm comm)
```

IN: sendbuf=starting address of send buffer (choice)
sendcount=number of elements in send buffer (integer)
sendtype=data type of send buffer elements (handle)
recvcount=number of elements for any single receive (integer)
recvtype=data type of recv buffer elements
root=rank of receiving process (integer)
comm=communicator,

OUT: recvbuf=address of receive buffer

recvcnt is the number of elements collected from each process, not the size of recvbuf, that should be recvcnt times the number of process in the communicator.

Gatherv: gather with variable buffer size



MPI_Gatherv

Usage:

C/C++:

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

Fortran:

```
    CALL MPI_GATHERV(sendbuf, sendcnt, sendtype, recvbuf,
                     recvcnts, displs, recvtype, root, comm, ierr)
```

Description:

- Collects individual messages from each process in communicator to the root process and store them in rank order

- Messages can have different sizes and displacements

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Università della Svizzera italiana | Faculty of Informatics | Institute of Computational Science

Reduction

The reduction operation allows to:

- **Collect data from each process**
- **Reduce the data to a single value**
- **Store the result on the root processes**
- **Store the result on all processes**
- **Overlap communication and computation**

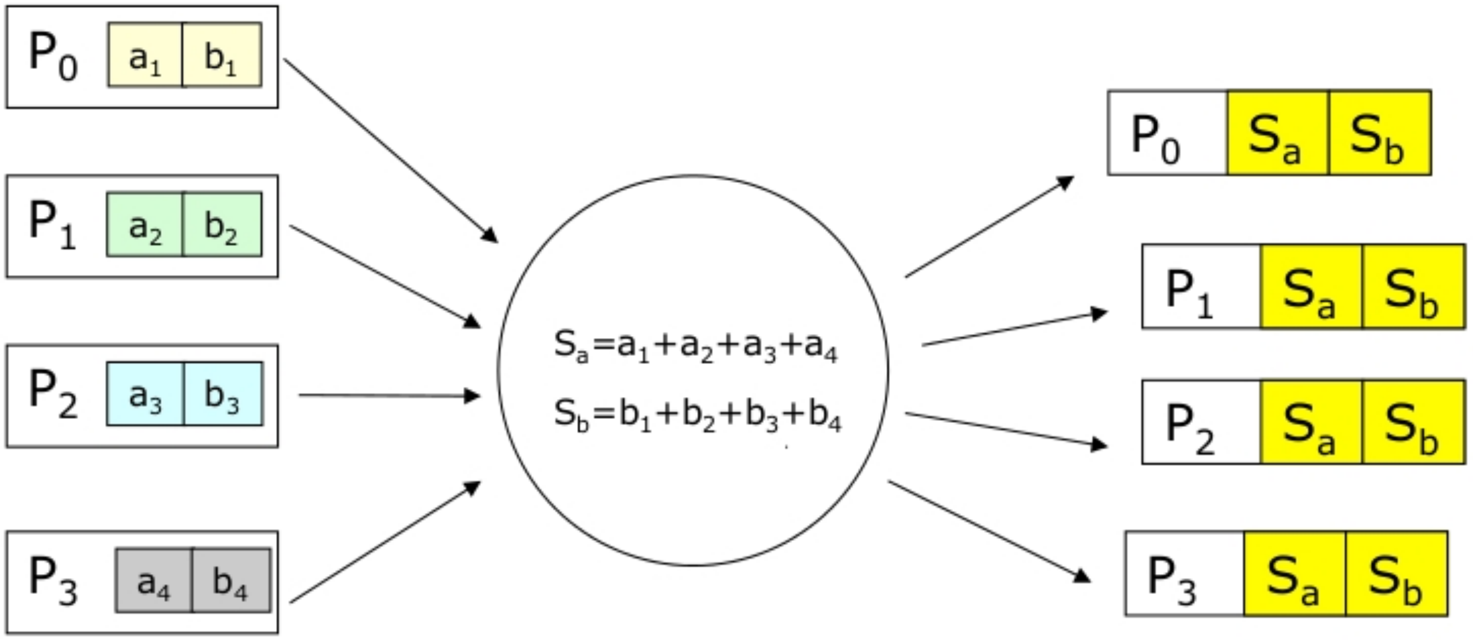
Predefined reduction operations:

MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, etc.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Università della Svizzera italiana | Faculty of Informatics | Institute of Computational Science

Reduce example: Parallel Sum (MPI_SUM)



- **Reduction function works with arrays**

MPI_Reduce and MPI_Allreduce

Usage:

C/C++:

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

Fortran:

```
CALL MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm,
                 ierr )
CALL MPI_ALLREDUCE( sendbuf, recvbuf, count datatype op, comm, ierr)
```

MPI_Allreduce: The argument root **is missing, the result is stored to all processes.**

Exercise “MPI_Reduce & MPI_Allreduce”

- Compile and run the reduce-code for different <ProcNr> values between 1-16

```
CC reduce_mpi.c –o reduce
aprun –n<ProcNr.> reduce
```

- Comment out the Allreduce command line in the source code and restart it

Virtual Topologies

- A virtual topology describes the "connentivity" of MPI processes in a communicator
- The two main types of topologies supported by MPI are Cartesian and Graph
- MPI topologies are virtual – there may be no relation between the physical structure of the parallel machine and the process topology
- Virtual topologies are build upon MPI communicators

Cartesian topology:

- Each process is "connected" to its neighbors in a virtual grid
- Boundaries can be cyclic
- Processes are identified by (discrete) Cartesian coordinates i, j, k

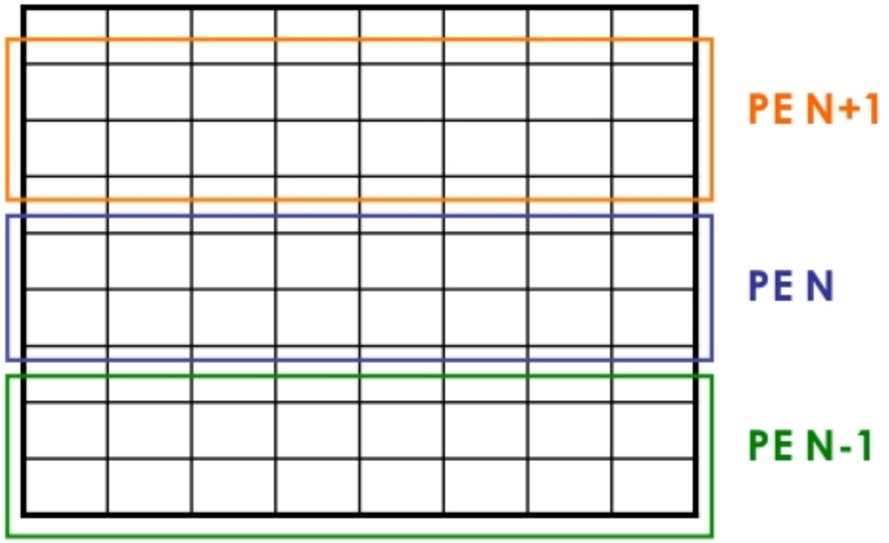
Graph topologies:

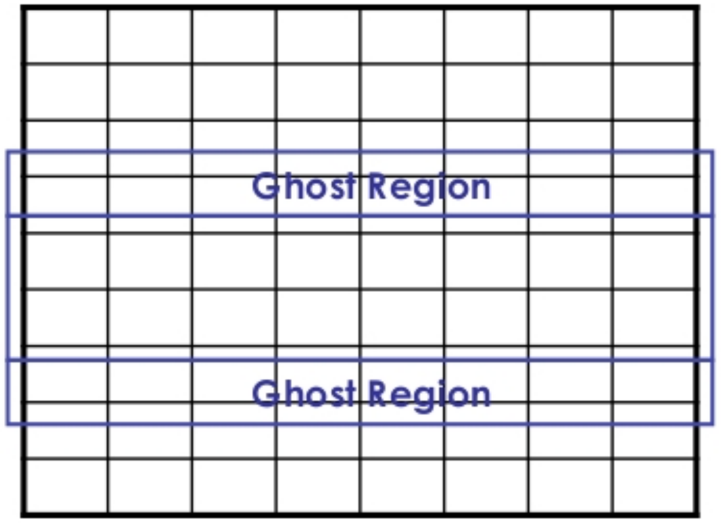
- Graphs are used to describe communication patterns
- The most general description of communication patterns

Domain decomposition: simple distribution

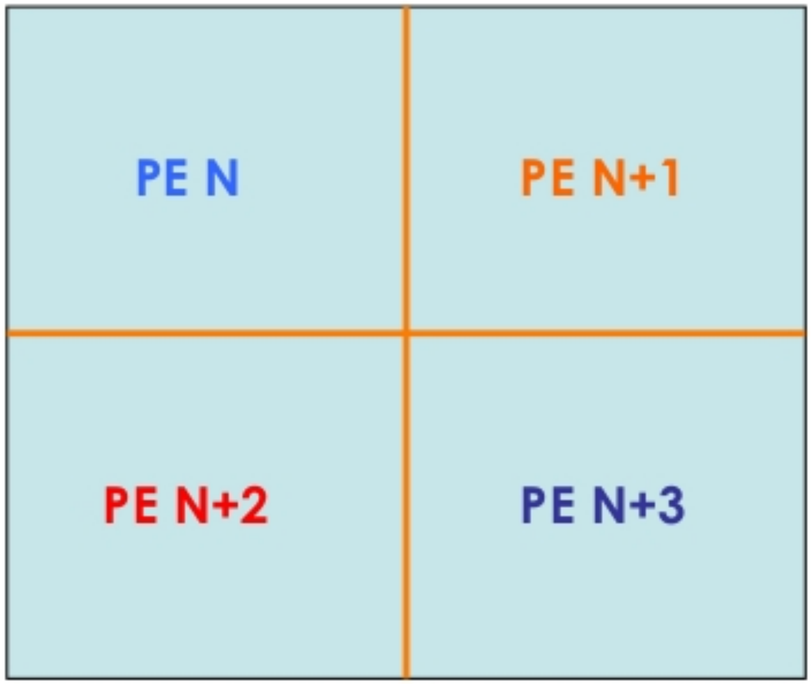
Data is distributed "linearly" between processors.
Maps the MPI_COMM_WORLD towards a linear topology

When halo (ghost) regions are exchanged, processor N communicates with N-1 and N+1





Domain decomposition: Cartesian distribution



This is in general a more effective way of distribute the domain, since:

- **It is much more scalable**
- **Communicated data volume can be smaller (especially when a large number of processors is used)**
- **It can better map the geometry of the problem and the algorithm**

However, it is more difficult to handle (e.g. who are my neighbors?)

MPI_Cart_create

Usage: C/C++

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

Input parameters:

comm_old: input communicator (handle)

ndims: number of dimensions of cartesian grid (integer)

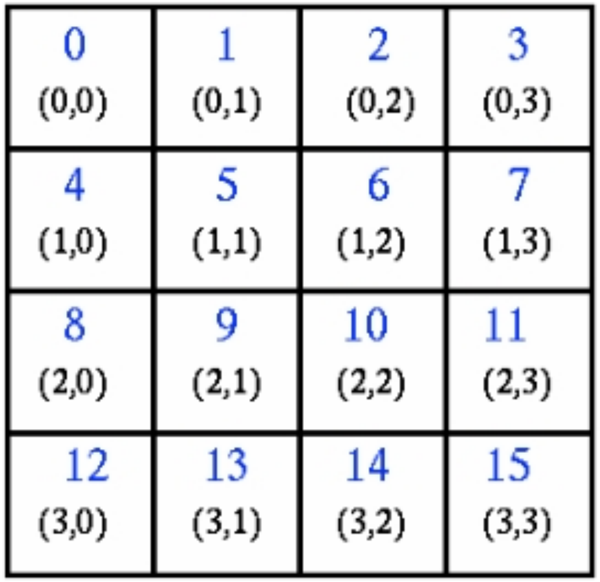
dims: integer array of size ndims specifying the number of processes in each dimension

periods: logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension

reorder: ranking may be reordered (true) or not (false) (logical)

Output parameter:

comm_cart: communicator with new cartesian topology (handle)

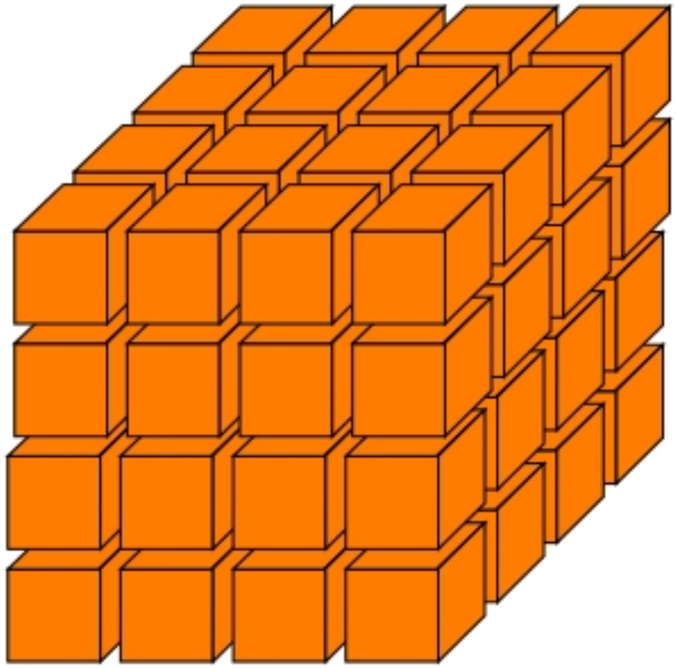


0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

MPI_Cart_create: 3D-example

C/C++ Example:

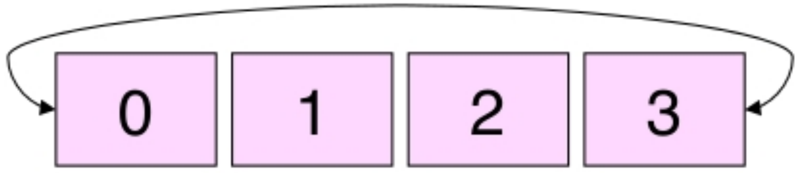
```
MPI_Comm comm_cart;
integer dimsProc[3];
integer periods[3]={1,1,1};


dimsProc[0]=NprocX;
dimsProc[1]=NprocY;
dimsProc[2]=NprocZ;
```



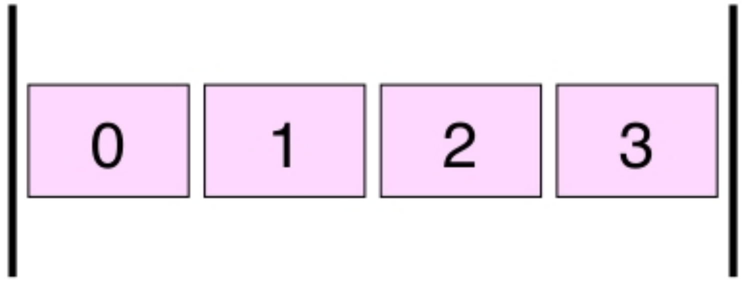
```
int MPI_Cart_create(MPI_COMM_WORLD, 3, dimsProc, periods, 0
                    , &comm_cart)
```

Periodic boundaries

periods[0]=TRUE;
left_0 = 3
right_3 = 0



periods[0]=FALSE;
left_0 = MPI_PROC_NULL
right_3 = MPI_PROC_NULL



Finding neighbors: MPI_Cart_shift

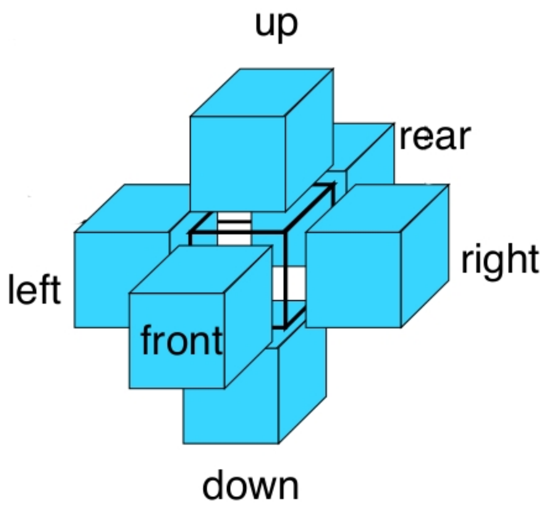
Usage:

int MPI_Cart_shift(MPI_Comm comm, int direction, int displ,
int rank_1, int rank_2)

Returns the shifted source and destination ranks, given a shift direction and amount

IN: **comm:** communicator with cartesion structure (handle)
direction: coordinate dimension of shift (integer)
displ: displacement (integer)

OUT: **rank_1**: rank of 1st neighbour process (integer)
rank_2: rank of 2nd neighbor process (integer)



Example for 3D Cartesian:

int MPI_Cart_shift (comm_cart,0,1,**left,right)**
int MPI_Cart_shift (comm_cart,1,1,**front,rear)**
int MPI_Cart_shift (comm_cart,2,1,**down,up)**

Storing halo cells as MPI_Vector

- **Usage of derived data-types:**

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype old_type, MPI_Datatype *newtype_p)
```

In:

count = number of blocks (nonnegative integer)
blocklength = number of elements in each block (nonnegative int.)
stride = number of elements between start of each block (integer)
oldtype = old datatype (handle)

Out:

newtype_p = new datatype (handle)

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Università della Svizzera italiana | Faculty of Informatics | Institute of Computational Science

Exercise “MPI_Cart_create & MPI_Vector”

- **Compile and run the ghost_cell_ex_cart_mpi_column-code for <ProcNr>=16**

  ```
  CC  –o ghost_cell_ex_cart_mpi_column.c –o column
  aprun –n16 column
  ```

- **Compile and run the ghost_cell_ex_cart_mpi_row-code for <ProcNr>=16**

  ```
  CC  –o ghost_cell_ex_cart_mpi_row.c –o row
  aprun –n16 column
  ```

- **Change the writing-rank for the output and restart.**

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Cart scheme overview

- **Row: C-style.**

Processor distribution: **data on one processor:**

```
|---------------------|        xggggggggggx
|  0 |  1 |  2 |  3  |        gddddddddddg
|---------------------|        gddddddddddg
|  4 |  5 |  6 |  7  |        gddddddddddg
|---------------------|        gddddddddddg
|  8 |  9 | 10 | 11  |        gddddddddddg
|---------------------|        gddddddddddg
| 12 | 13 | 14 | 15  |        gddddddddddg
|---------------------|        gddddddddddg
                               gddddddddddg
                               xggggggggggx
```

- **Column: Fortran-style based on transposed processor distribution.**

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Università della Svizzera italiana | Faculty of Informatics | Institute of Computational Science

Thank you for your attention