Goals for today

- Learn how parallelization works and why it is helpful
- Understand the difference between shared and distributed memory
- Learn about different possibilities for parallelization in Fortran
- Gain insight into parallel programming with OpenMP and MPI

End-of-semester projects

27 June 2024, 9:00* – 12:00 Exner room (2F513)	Presentations of projects**
27 June 2024, 23:59	Deadline for handing in programs

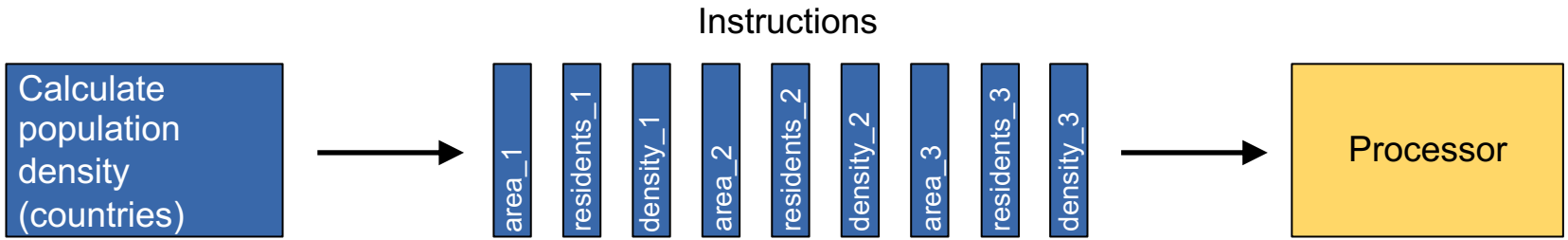
* Is this ok?

** Presentation guidelines:

- 10 minutes per person
- Content: Theory, most important parts of the code, results (plots / animations)

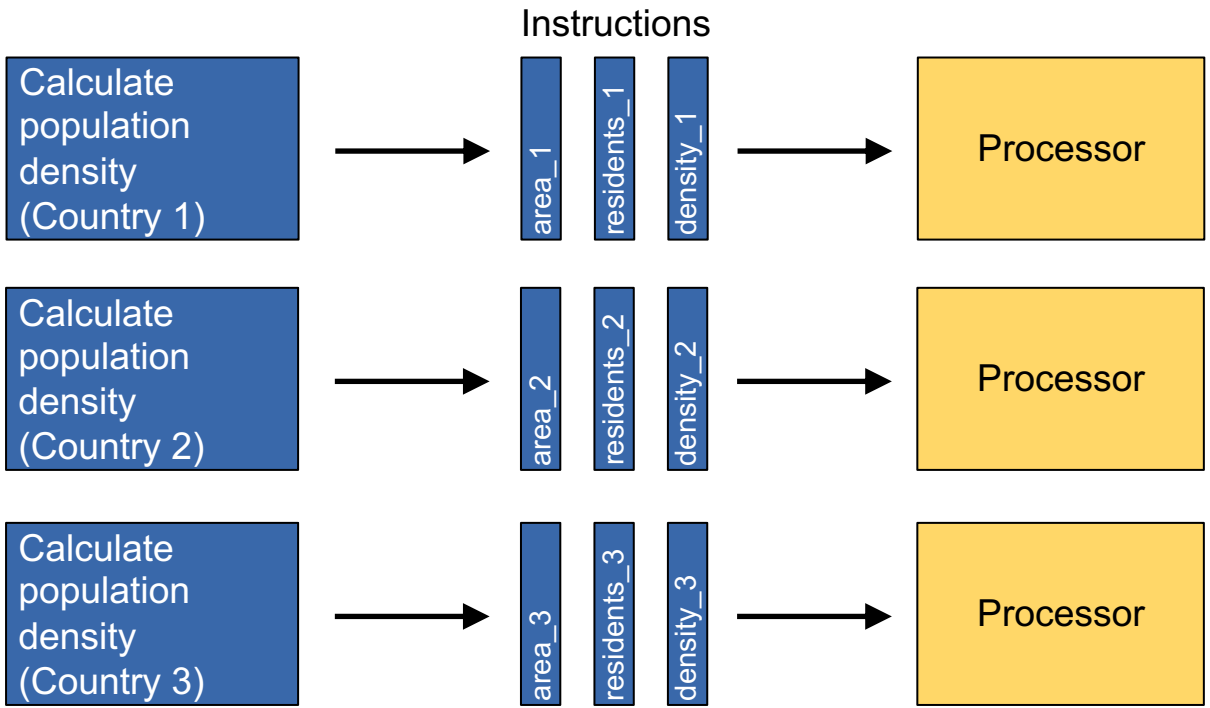
Serial programming

Serial programs process instructions sequentially and run on a single processor.



Parallel programming

Parallel programs split problems and process instructions simultaneously on different processors.



Why do we need parallelization?

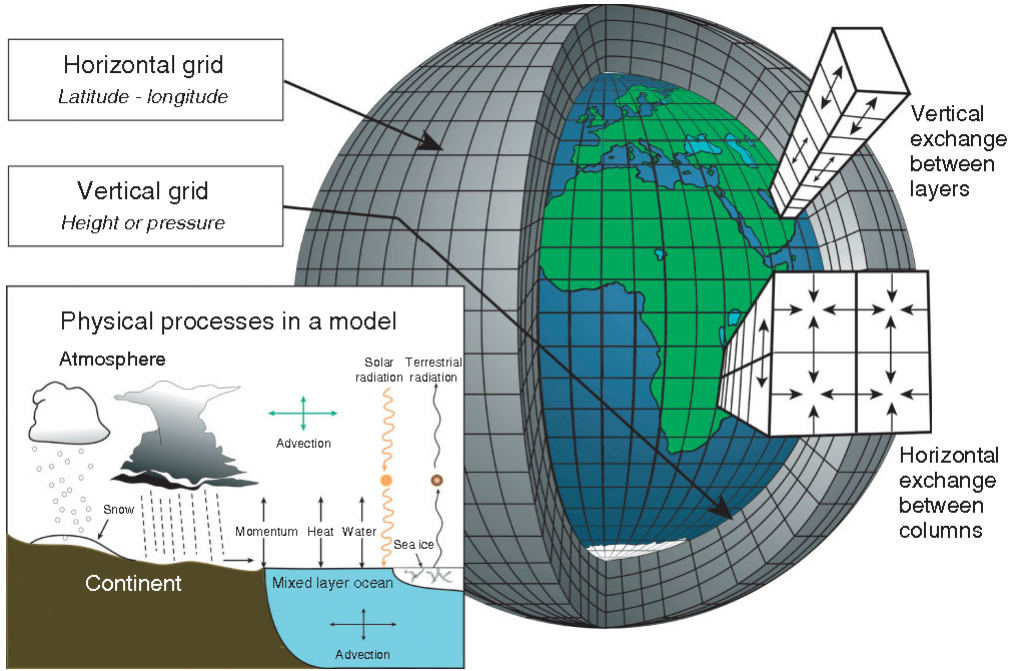
Weather and climate models require a high resolution (= many grid points) to realistically simulate all the complex physical processes.

Example: ECMWF IFS

- 9 km horizontal resolution
- 137 vertical levels

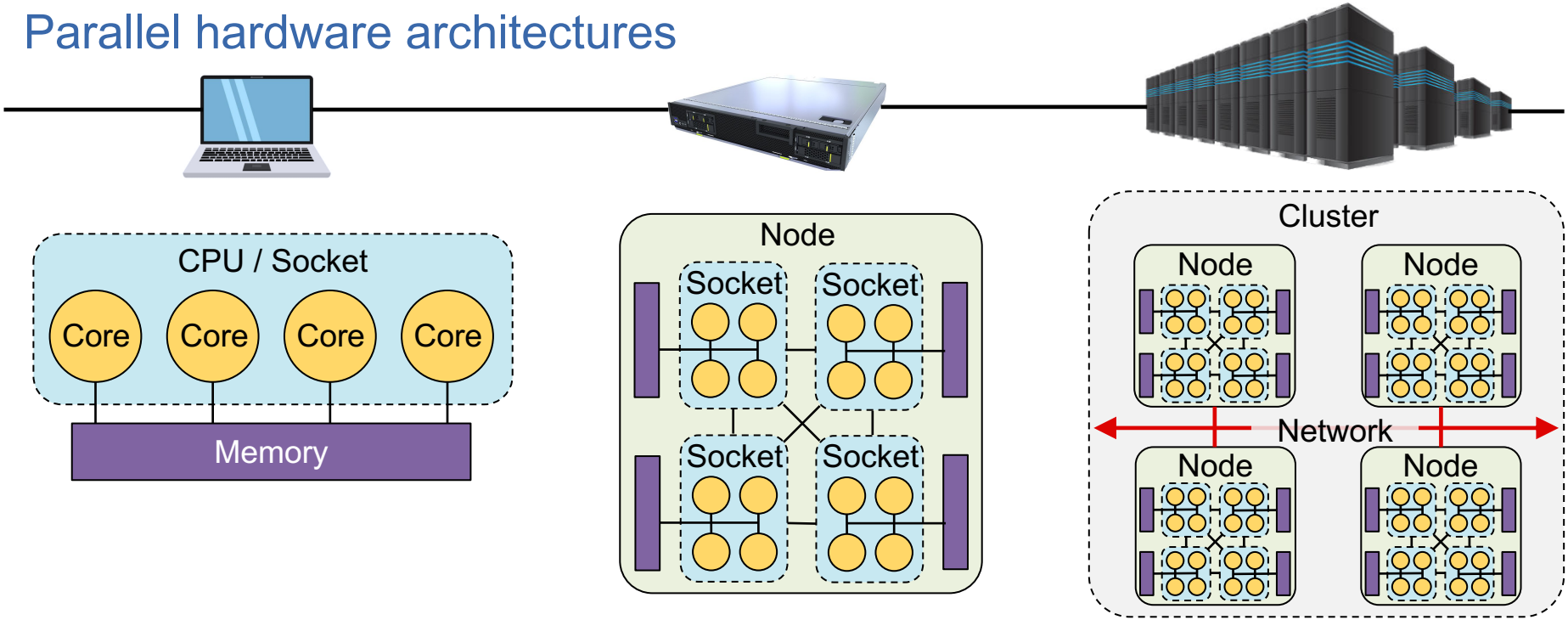
→ 1'795'686'400 grid points

Without parallelization the models would take far too long to compute forecasts.



https://www.semanticscholar.org/paper/History-of-climate-modeling-Edwards/411cee316ad1ffcce9a0fe7cdc3ec3be28712f90

Parallel hardware architectures



Shared memory

- Multiple cores share the same memory
- Parallelization is done by compiler, possibly with help, e.g. OpenMP instructions

Distributed memory

- Each node has its own memory
- Parallelization requires message exchange, e.g. with MPI

Ranking of supercomputers

By speed

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
⋮					
353	**VSC-4** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo Vienna Scientific Cluster Austria	37,920	2.73	3.76	
⋮					
455	**VSC-5** - MEGWARE SLIDESX, AMD EPYC 7713 64C 2GHz, Infiniband HDR, MEGWARE Vienna Scientific Cluster Austria	95,232	2.31	3.05	516

TOP500 The List.

https://top500.org/

Ranking of supercomputers

By energy efficiency

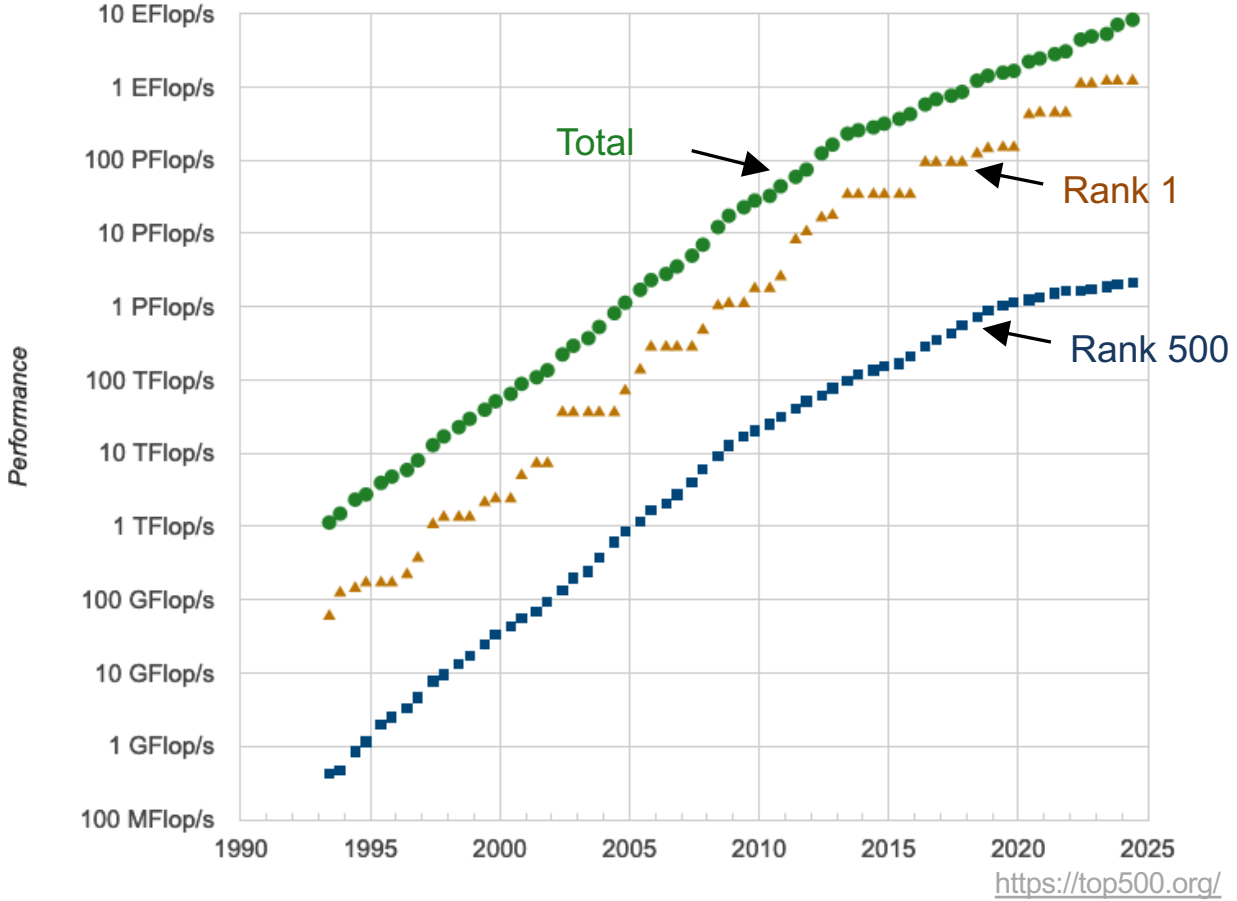
Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	189	**JEDI** - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, ParTec/EVIDEN EuroHPC/FZJ Germany	19,584	4.50	67	72.733
2	128	**Isambard-AI phase 1** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE University of Bristol United Kingdom	34,272	7.42	117	68.835
3	55	**Helios GPU** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cyfronet Poland	89,760	19.14	317	66.948
⋮						
139	455	**VSC-5** - MEGWARE SLIDESX, AMD EPYC 7713 64C 2GHz, Infiniband HDR, MEGWARE Vienna Scientific Cluster Austria	95,232	2.31	516	4.481
⋮						
394	353	**VSC-4** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo Vienna Scientific Cluster Austria	37,920	2.73		0.000

https://top500.org/

Speed increases exponentially

We are already in the exascale era

Exaflop = 10^{18} calculations per second



Parallel programming in Fortran

- **OpenMP**: works only for shared memory (at most 1 node)
- **MPI**: works for distributed and shared memory → any number of nodes
- **Coarrays** (Fortran ≥2008): Alternative to MPI
- **CUDA Fortran** and **OpenACC**: for GPUs

Open Multi-Processing (OpenMP)

- Application programming interface that enables multiprocessing in C, C++, and Fortran.
- Threads have to run on the same node (shared memory).
- OpenMP specification: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
- Supported by most Fortran compilers (gfortran, ifort, nagfor, and more).

```
$ gfortran -fopenmp program.f90
$ export OMP_NUM_THREADS=xx
$ ./a.out
```

xx = number of threads

Hello world with OpenMP

OpenMP functions →

Start parallel region →

Query own thread ID →

Determine number of threads →

End parallel region →

Compile with flag –fopenmp and run normally →

```
PROGRAM helloOpenMP

IMPLICIT NONE
INTEGER :: nthreads, thread_id
INTEGER :: OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS

!$OMP PARALLEL PRIVATE(thread_id)

thread_id = OMP_GET_THREAD_NUM()

nthreads = OMP_GET_NUM_THREADS()

PRINT*, 'I am thread', thread_id, 'of', nthreads

!$OMP END PARALLEL

END PROGRAM helloOpenMP
```

Ensures that each thread has its own copy of thread_id

```
$ gfortran -fopenmp helloOpenMP.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
 I am thread           1 of           4
 I am thread           2 of           4
 I am thread           0 of           4
 I am thread           3 of           4
```

Thread IDs start at 0

Parallelize DO loops with OpenMP

Within a parallel region, the !$OMP DO directive can be used to parallelize DO counting loops.

The loop indices of the loop that follows (here j=1,N) are then divided among the available threads, with each thread executing a portion of the iterations.

```
IMPLICIT NONE
INTEGER, PARAMETER :: N=30000
INTEGER :: i, j, v1, v2
INTEGER :: pos, neg
REAL    :: v
REAL, DIMENSION(N, N) :: matrix

!$OMP PARALLEL PRIVATE(i, v1, v2, v)
!$OMP DO
DO j=1,N
  DO i=1,N
    v1 = N - (j+2)
    v2 = 1 + 6*(j+2) - 2*(i+3)
    v = v1 / v2
    matrix(i,j) = v
  END DO
END DO
!$OMP END DO
!$OMP END PARALLEL

pos = COUNT(matrix >= 0.)
neg = N*N - pos

PRINT*, neg, 'negative and', pos, 'positive values'
```

Message Passing Interface (MPI)

- Standard library that enables communication between processes computing in parallel.
- The processes can run on the same node (shared memory) or on different nodes (distributed memory).
- MPI Standard: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-rc-jun-21.pdf
- Works for Fortran, C, and C++.
- Used in many weather and climate models.

```
$ mpifort program.f90
$ mpirun -n xx ./a.out
```

xx = number of processes

Hello world with MPI

Import MPI library →

Initialize MPI →

Determine number of processes →

Query own process ID →

Exit MPI →

Compile with mpifort and run with mpirun (-n: number of processes) →

```
PROGRAM helloMPI

USE mpi_f08 ! or simply mpi
IMPLICIT NONE
INTEGER :: ierr, nranks, myrank

CALL MPI_Init(ierr)

CALL MPI_Comm_size(MPI_COMM_WORLD, nranks, ierr)

CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)

PRINT*, 'I am process', myrank, 'of', nranks

CALL MPI_Finalize(ierr)

END PROGRAM helloMPI
```

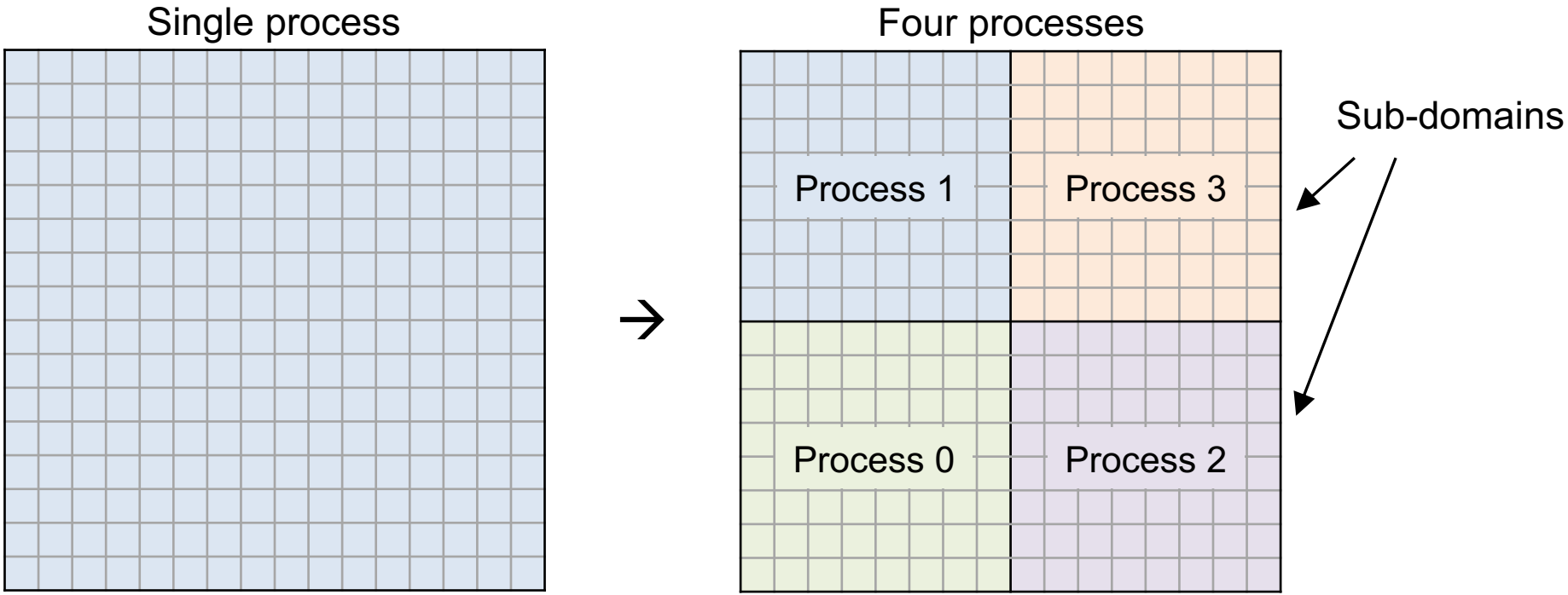
Communicator (MPI variable)

Process IDs start at 0

```
$ mpifort helloMPI.f90
$ mpirun -n 4 ./a.out
 I am process           2 of           4
 I am process           0 of           4
 I am process           1 of           4
 I am process           3 of           4
```

Decomposition of the model domain

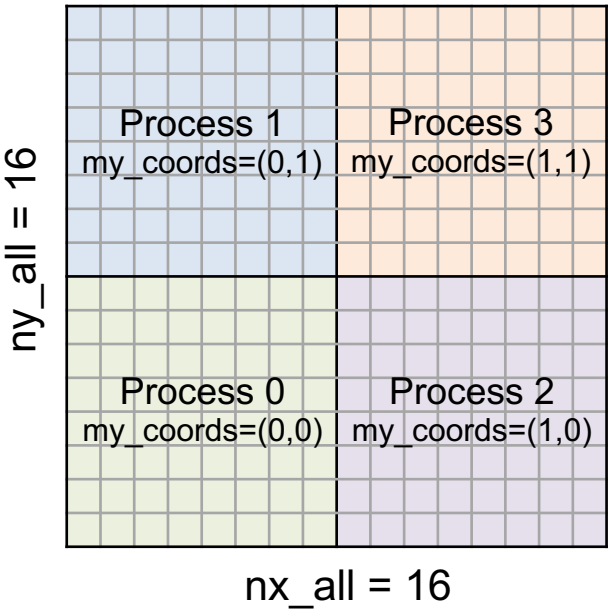
Each process performs the same operations, but on different parts of the model domain.



Create virtual topology

- MPI_Dims_create distributes the processes along the dimensions in dims
  - e.g. 12 MPI processes, 2 dimensions → 4x3
- MPI_Cart_create creates new cartesian MPI communicator comm_cart
- MPI_Comm_rank queries process ID my_rank in the new communicator
  - If reorder = .TRUE. it may differ from the process ID in the original communicator MPI_COMM_WORLD
- MPI_Cart_coords queries coordinates my_coords of the process in the domain
- MPI_Cart_shift finds process IDs of neighboring processes

Example



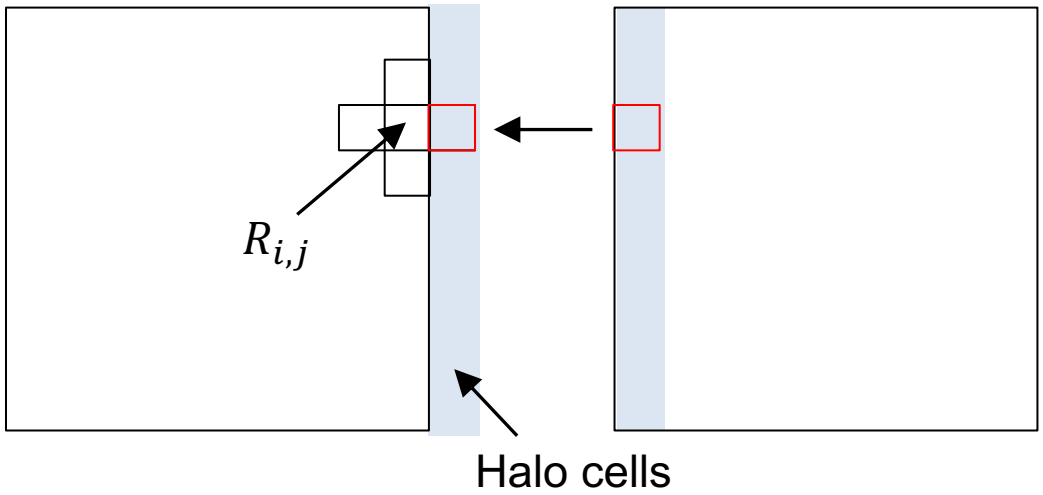

```
$ mpifort domain_decomp.f90
$ mpirun -n 4 ./a.out
I am process 3 and I handle i= 9,16 and j= 9,16
I am process 0 and I handle i= 1, 8 and j= 1, 8
I am process 1 and I handle i= 1, 8 and j= 9,16
I am process 2 and I handle i= 9,16 and j= 1, 8
```

```
CALL MPI_Init()
CALL MPI_Comm_size(MPI_COMM_WORLD, nranks)

! Initialize
dims = 0.; periodic = .FALSE.; reorder = .TRUE.

! Distribute processes
CALL MPI_Dims_create(nranks, ndims, dims)
CALL MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, &
                     periodic, reorder, comm_cart)
CALL MPI_Comm_rank(comm_cart, myrank)
CALL MPI_Cart_coords(comm_cart, myrank, ndims, my_coords)

! Assign grid points
nx = nx_all/dims(1); ny = ny_all/dims(2)
is = my_coords(1)*nx+1   ! first grid point in x direction
ie = (my_coords(1)+1)*nx ! last  grid point in x direction
js = my_coords(2)*ny+1   ! first grid point in y direction
je = (my_coords(2)+1)*ny ! last  grid point in y direction

WRITE(*,'(5(A,I2))') 'I am process', myrank, &
  ' and I handle i=', is, ',', ie, &
  ' and j=', js, ',', je

CALL MPI_Finalize()
```

Halo cells

- The solution at a grid point often depends on neighbor grid points located in other sub-domains → processes must communicate with each other.

- For example, Poisson: $$R_{i,j} = f_{i,j} - \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}}{h^2}$$



Communication between processes

Input arguments
Output arguments

- Processes can send messages to each other.
  - A message consists of elements of a certain data type (intrinsic or derived).
  - Intrinsic data types have the prefix MPI_, e.g. MPI_REAL.
  - Processes are addressed via their process IDs.

- Send: MPI_Send(buf, count, datatype, dest, tag, comm, *ierr*)

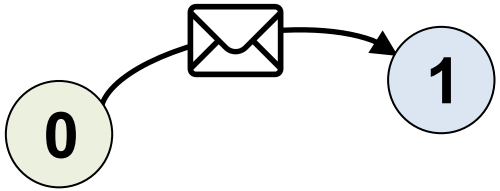
- Receive: MPI_Recv(buf, count, datatype, source, tag, comm, status, *ierr*)

- Both: MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, *ierr*)

source: sender, dest: receiver, buf: message, count: number of elements, datatype: data type, tag: indicator, comm: communicator, status: status.

Example: Ping

Process 0 sends a message to process 1




```
PROGRAM ping
  USE mpi_f08
  IMPLICIT NONE
  TYPE(MPI_Status) :: stat
  INTEGER :: message(1), myrank

  CALL MPI_Init()
  CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank)
  IF (myrank == 0) THEN
    message = 42
    WRITE(*,'(A,I1,A,I2)') 'I am process ', myrank, &
      ' and I am sending the following message: ', message
    CALL MPI_Send(message, 1, MPI_INTEGER, 1, 17, MPI_COMM_WORLD)
  ELSE
    CALL MPI_Recv(message, 1, MPI_INTEGER, 0, 17, MPI_COMM_WORLD, stat)
    WRITE(*,'(A,I1,A,I2)') 'I am process ', myrank, &
      ' and I just received the following message: ', message
  END IF
  CALL MPI_Finalize()
END PROGRAM ping
```

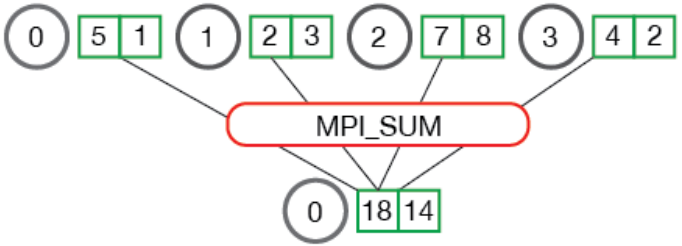
```
$ mpifort ping.f90
$ mpirun -n 2 ./a.out
I am process 0 and I am sending the following message: 42
I am process 1 and I just received the following message: 42
```

Global reduction operations

There are special MPI functions that perform operations over all processes. Intrinsic operations have the prefix MPI_, e.g. MPI_SUM

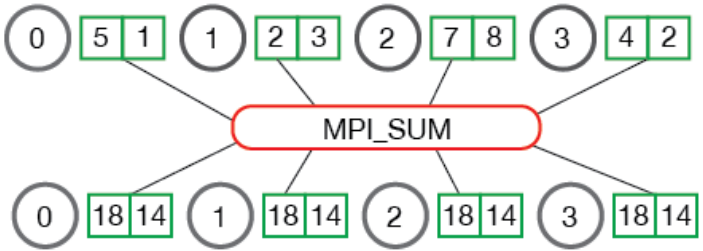
Only root gets the result:

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, *ierr*)



All processes get the result:

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, *ierr*)



sendbuf: values, recvbuf: result, count: number of elements, datatype: data type, op: operation, comm: communicator

Example: Global sum of process IDs

```
PROGRAM reduce
USE mpi_f08
IMPLICIT NONE
INTEGER :: myrank, nranks, sum_ranks

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank)
CALL MPI_Reduce(myrank, sum_ranks, 1, MPI_INTEGER, &
                MPI_SUM, 3, MPI_COMM_WORLD)
WRITE(*,*) "Process", myrank, ": Sum =", sum_ranks
CALL MPI_Finalize()
END PROGRAM reduce
```

```
$ mpifort reduce.f90
$ mpirun -n 6 ./a.out
 Process           0 : Sum =           0
 Process           1 : Sum =           0
 Process           2 : Sum =           0
 Process           3 : Sum =          15
 Process           4 : Sum =           0
 Process           5 : Sum =           0
```

```
PROGRAM allreduce
USE mpi_f08
IMPLICIT NONE
INTEGER :: myrank, nranks, sum_ranks

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank)
CALL MPI_Allreduce(myrank, sum_ranks, 1, MPI_INTEGER, &
                   MPI_SUM, MPI_COMM_WORLD)
WRITE(*,*) "Process", myrank, ": Sum =", sum_ranks
CALL MPI_Finalize()
END PROGRAM allreduce
```

```
$ mpifort allreduce.f90
$ mpirun -n 6 ./a.out
 Process           0 : Sum =          15
 Process           1 : Sum =          15
 Process           2 : Sum =          15
 Process           3 : Sum =          15
 Process           4 : Sum =          15
 Process           5 : Sum =          15
```

Parallelize Poisson solver

Jacobi, without multigrid process

```
CALL MPI_Dims_create(nranks, ndims, dims)
CALL MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, &
                     periodic, reorder, comm_cart)
CALL MPI_Comm_rank(comm_cart, my_rank)
CALL MPI_Cart_coords(comm_cart, my_rank, ndims, my_coords)

! Calculate neighbors' ranks based on my_coords
CALL MPI_Cart_shift(comm_cart, 0, 1, left, right)
CALL MPI_Cart_shift(comm_cart, 1, 1, down, up)
```

Find neighbors

```
! Set up grid
nx = nx_all/dims(1); ny = ny_all/dims(2); h = 1./(ny_all-1.)
ALLOCATE(u(0:nx+1,0:ny+1), f(nx,ny), res(nx,ny))

! Initialize f, u and res
CALL RANDOM_NUMBER(f); u = 0.; res = 0.

! Boundary conditions for f
IF (my_coords(1) == 0) f(1,:) = 0.
IF (my_coords(1) == dims(1)-1) f(nx,:) = 0.
IF (my_coords(2) == 0) f(:,1) = 0.
IF (my_coords(2) == dims(2)-1) f(:,ny) = 0.

sum_f2 = SUM(f**2)
CALL MPI_Allreduce(sum_f2, sum_f2_all, 1, MPI_REAL, MPI_SUM, comm_cart)

f_rms = SQRT(sum_f2_all/(nx_all*ny_all))
res_rms = f_rms
```

Calculate f_rms

```
DO WHILE (res_rms / f_rms > max_err)
  ! Send right
  CALL MPI_Sendrecv(u(nx,1:ny), ny, MPI_REAL, right, 16, &
                    u(0,1:ny), ny, MPI_REAL, left, 16, comm_cart, stat)
  ! Send left
  CALL MPI_Sendrecv(u(1,1:ny), ny, MPI_REAL, left, 17, &
                    u(nx+1,1:ny), ny, MPI_REAL, right, 17, comm_cart, stat)
  ! Send up
  CALL MPI_Sendrecv(u(1:nx,ny), nx, MPI_REAL, down, 18, &
                    u(1:nx,0), nx, MPI_REAL, up, 18, comm_cart, stat)
  ! Send down
  CALL MPI_Sendrecv(u(1:nx,1), nx, MPI_REAL, up, 19, &
                    u(1:nx,ny+1), nx, MPI_REAL, down, 19, comm_cart, stat)
```

Transfer halo cells

```
  ! Calculate residue
  DO j = 1, ny
    DO i = 1, nx
      res(i,j) = f(i,j) - 1./h**2*(u(i,j+1) + u(i,j-1) + u(i+1,j) + &
        u(i-1,j) - 4*u(i,j))
    END DO
  END DO

  ! Correct u
  u(1:nx,1:ny) = u(1:nx,1:ny) - alpha*res*h**2 / 4.

  ! Calculate res_rms over all processes
  sum_res2 = SUM(res**2)
  CALL MPI_Allreduce(sum_res2, sum_res2_all, 1, MPI_REAL, MPI_SUM, comm_cart)
  res_rms = SQRT(sum_res2_all/(nx_all*ny_all))
END DO
```

Calculate res_rms

How much time is used for communication?

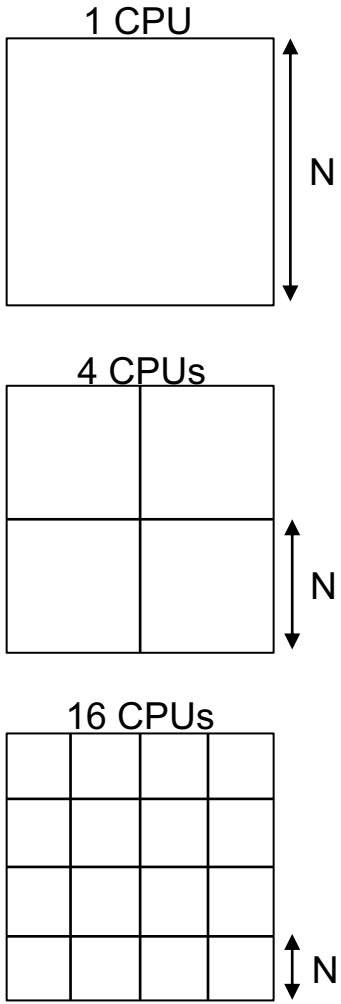
Computations: $t_{comp} = \frac{a}{f} \cdot N^2$

Communication: $t_{comm} = L + \frac{4bN}{B}$

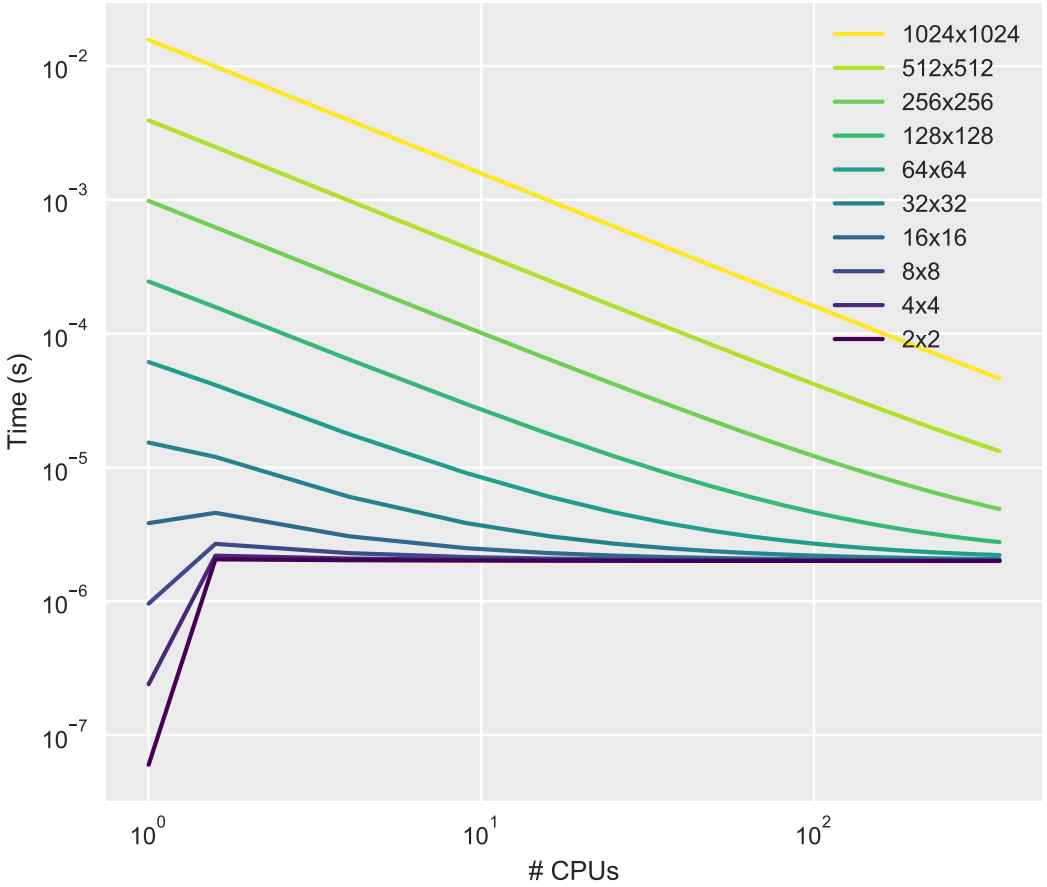
Total: $t = t_{comp} + t_{comm} = \frac{a}{f} \cdot N^2 + L + \frac{4bN}{B}$

a: Number of operations per time step
f: Clock speed
b: Size of message
L: Latency
B: Bandwidth
N: Number of grid points per dimension

Assumptions:
a = 15
f = 10^9 s^{-1}
b = 64 bit
L = $2 \cdot 10^{-6}$ s
B = 10 Gbit/s



Parallelization makes sense only for many grid points

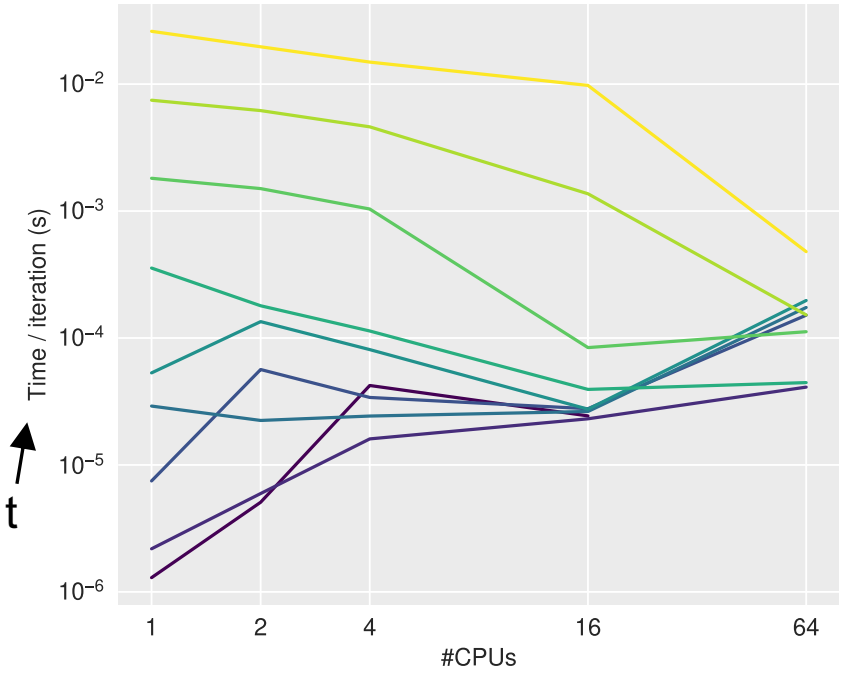


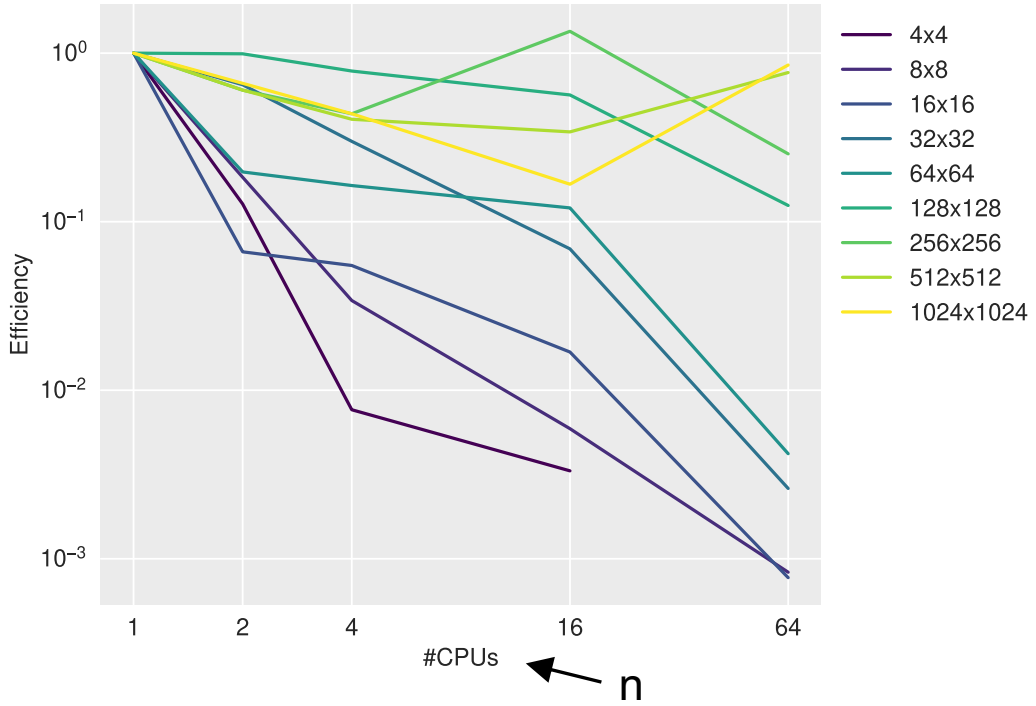
- Bandwidth determines the slope
- Latency determines the limit

Not considered:
Parts of code that cannot be parallelized

Test runs on Jet (cluster at IMGW)

$$\text{Efficiency } E = \frac{t(n)/t(1)}{n}$$





Summary

- Large simulations have to be parallelized if we want them to finish in an acceptable amount of time.

- In weather and climate modeling, parallelization is usually done by dividing the model domain among different processes, with each process performing the same operations on its sub-domain.

- Processes can have either shared or distributed memory. In the case of distributed memory, they (usually) need a way to communicate with each other.

- OpenMP and MPI are widely used interfaces for parallel computing. OpenMP works for shared memory, MPI works for both distributed and shared memory.