Code parallelization

Institute of Meteorology and Climatology, Leibniz Universität Hannover

Code parallelization

Content

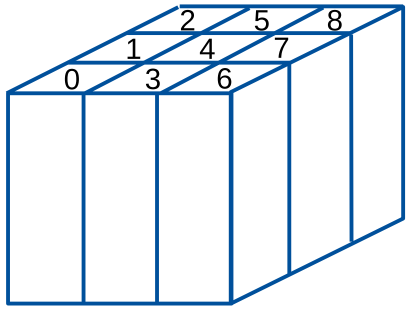
- Basics of parallelization
- PALM parallelization concept
- How to use the parallelized version of PALM
- MPI communication and virtual processor grid within PALM
- Parallel I/O

Code parallelization

Basics of parallelization

- All processor elements (PE, core) carry out the same program code (SIMD)
- Each PE of a computer operates on a different set of data

- **Realization:**
  - Each PE solves the equations for a different subdomain of the total domain.



  - Each PE only knows the variables' values of it's subdomain.
  - Communication / data exchange between PEs required.
  - Done by **M**essage **P**assing **I**nterface (MPI).

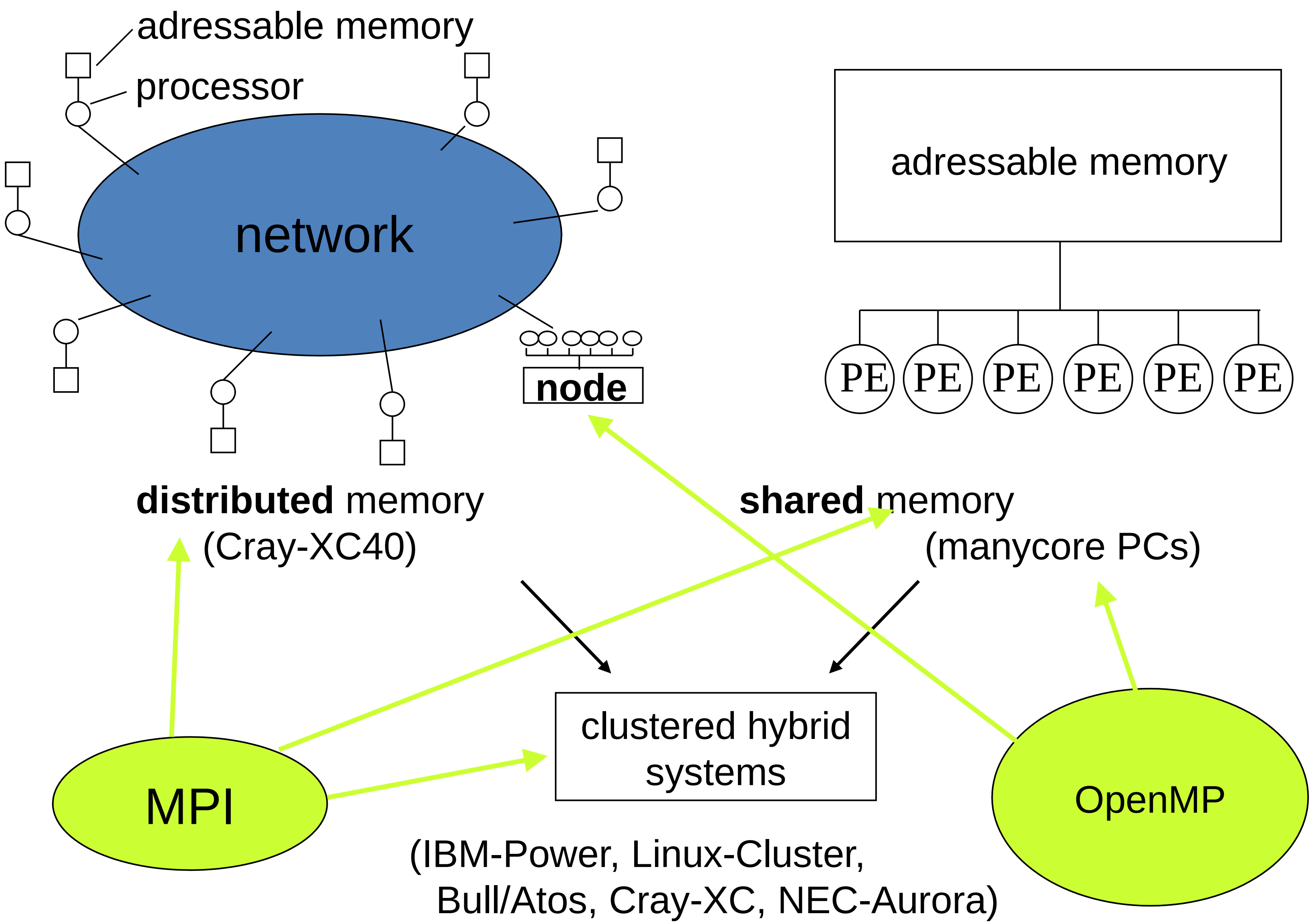
- Program loops are parallelized using directives. Each PE solves for a subset of the total index range.

```
!$OMP DO
    DO  i = 1, 100
           .
           .
    ENDDO
```

- Parallelization can be easily done by the compiler, if all PEs have access to all variables (shared memory).
- Shared memory model (OpenMP), also used for GPU programming (OpenACC).

Code parallelization

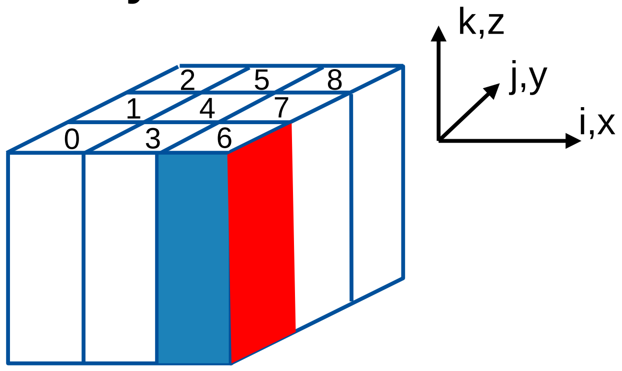
Basic architectures of massively parallel computers



Code parallelization

PALM parallelization concept

- **General demands for a parallelized program:**
  - **Good load balancing** (all cores should have same workload)
  - **Small communication overhead** (otherwise speed-up may get lost)
  - **Scalability** (up to large number of processors, otherwise they can‘t be used)
- **Basic parallelization method used for PALM is a 2D domain decomposition along x and y:**



Data to be communicated should be contiguous in memory (Fortran)!

f(i,j,k)
- columns of i , no contiguous data at all

f(k,j,i)
- columns of k , planes of k,j (all data contiguous)

  - No vertical decomposition because calculations for surface grid points are more expensive ⇨ No load balance!
  - Array indexing f(k,j,i) is used in PALM.
  - Message passing is realized using MPI.
  - OpenMP parallelization as well as hybrid usage of OpenMP and MPI is realized.

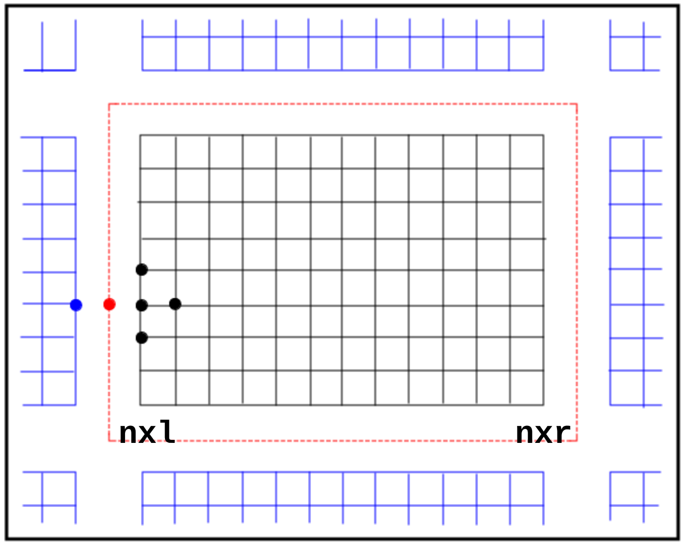
Code parallelization

PALM parallelization model

- Central finite differences cause local data dependencies (only boundary data have to be transferred)

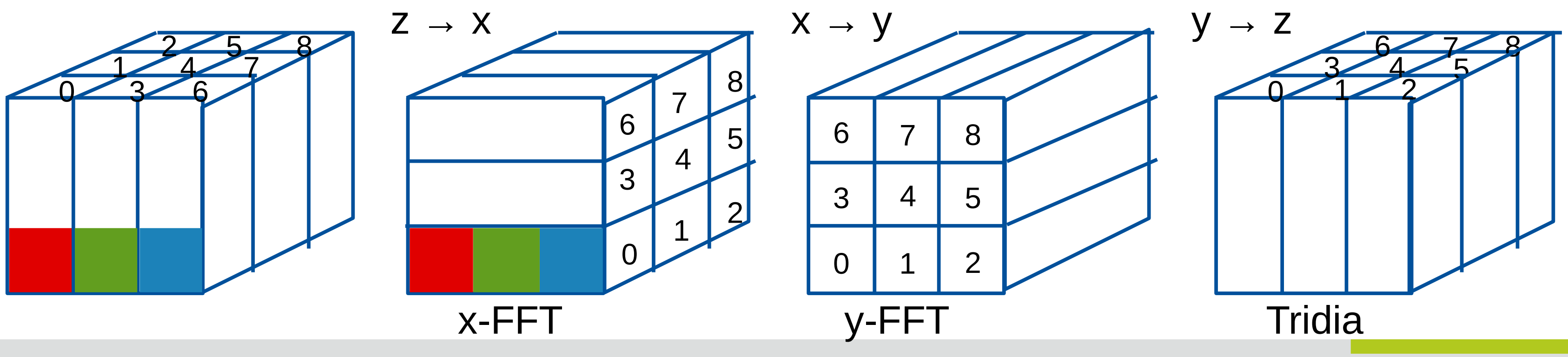
$$\left.\frac{\partial\psi}{\partial x}\right|_i = \frac{\psi_{i+1} - \psi_{i-1}}{2\Delta x}$$

  - **Solution:** introduction of ghost points
    `u(:,:,nxl:nxr), u(:,:,nxl-1:nxr+1)`



- FFT and linear equation solver cause non-local data dependencies (all 3D data have to be transferred)
  - **Solution:** transposition of 3D arrays

Example below shows transpositions for solving the Poisson equation.



Code parallelization

How to use the parallelized version of PALM

- The parallel version of PALM is activated by line

  ```
  %cpp_options -D__parallel
  ```

  in PALM‘s configuration file `.palm.config.<ci>` This is the setting in the default configuration file `.palm.config.default`.

- Additionally, the number of required cores (PEs) and the number of tasks per node (number of PEs to be used on one node) have to be provided. –T option is required in batch mode only.

  ```
  palmrun ... –X64 –T8 ...
  ```

- If “tasks per node“ is not an integral divisor of the total number of requested processor cores (PEs), some PEs of the „last“ node are not used (but the user probably has to pay for them).

- Using the Open-MP parallelization does not yield any advantage over using a pure domain decomposition with MPI (contrary to expectations, it mostly slows down the computational speed), but this may change on cluster systems for very large number of cores (>100000?), or in case of 1D decompositions with large number of cores.

Code parallelization

MPI communication within PALM

- MPI (message passing interface) is a portable interface for communication between PEs (FORTRAN or C library).
- All MPI calls must be within

```
 CALL MPI_INIT( ierror )
 ...
CALL MPI_FINALIZE( ierror )
```

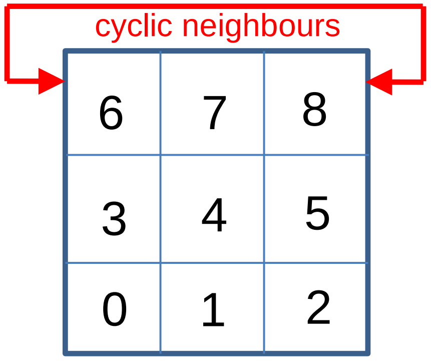
- MPI calls within PALM are available when using `%cpp_options -D__parallel`
- Communication is needed for
  - exchange of ghost points
  - transpositions (FFT-poisson-solver)
  - calculating statistics / global sums (e.g. for calculating horizontal averages)
  - data exchange in case of nesting
  - raytracing within the urban surface model (shading effects)
  - I/O of restart data
- Additional MPI calls are required to define the so-called virtual processor grid and to define special data types needed for more comfortable exchange of data.

Code parallelization

Virtual processor grid in PALM

- The processor grid and special data types are defined in file `init_pegrid.f90`
- PALM uses a two-dimensional virtual processor grid (in case of a 1D-decomposition, it has only one element along y). It is defined by a so-called communicator (here: `comm2d`):

```
ndim = 2
npe_xy(1) = npex     ! # of cores along x
npe_xy(2) = npey     ! # of cores along y
cyclic(1) = .TRUE.
cyclic(2) = .TRUE.
CALL MPI_CART_CREATE( MPI_COMM_WORLD, ndim, npe_xy, &
                      cyclic, reorder, comm2d, ierr )
```



virtual grid with ids

- The processor number (ID) with respect to this processor grid, `myid`, is given by:

```
CALL MPI_COMM_RANK( comm2d, myid, ierr )
```

- The IDs of the neighbouring PEs are determined by:

```
CALL MPI_CARD_SHIFT( comm2d, 0, 1, pleft,  pright, ierr )
CALL MPI_CARD_SHIFT( comm2d, 1, 1, psouth, pnorth, ierr )
```

Code parallelization

Exchange of ghost points

- Ghost points are stored in additional array elements added at the horizontal boundaries of the subdomains, e.g.

```
    u(:,:,nxl-nbgp), u(:,:,nxr+nbgp)    ! left and right boundary
    u(:,nys-nbgp,:), u(:,nyn+nbgp,:)    ! south and north boundary
                                        ! nbgp: number of ghost points
```

- The actual code uses nxlg=nxl-nbgp, etc...
- The exchange of ghost points is located in file `exchange_horiz.f90`
- **Simplified example:** synchroneous exchange of ghost points along `x` (yz- planes: send left, receive right plane):

```
   CALL MPI_SENDRECV( ar(nzb,nysg,nxl),   ngp_yz, MPI_REAL, pleft,  0,
                      ar(nzb,nysg,nxr+1), ngp_yz, MPI_REAL, pright, 0,comm2d,
                      status, ierr )
```

- Special MPI data types (vectors) are defined for exchange of `yz/xz`-planes for performance reasons and because array elements to be exchanged are not consecutively stored in memory for `xz`-planes:

```
ngp_yz(0) = (nzt - nzb + 2) * (nyn - nys + 1 + 2 * nbgp )
CALL MPI_TYPE_VECTOR( nbgp, ngp_yz(0), ngp_yz(0), MPI_REAL, type_yz(0), ierr )
CALL MPI_TYPE_COMMIT( type_yz(0), ierr )   ! see file init_pegrid.f90
CALL MPI_SENDRECV( ar(nzb,nysg,nxl), 1, type_yz(grid_level), pleft, 0, … )
```

Code parallelization

Transpositions

- Transpositions are located in file `transpose.f90` (several subroutines for 1D- or 2D-decompositions; they are called mainly from the FFT pressure solver, see `poisfft_mod.f90`.
- The following example is for a transposition from x to y, i.e. for the input array all data elements along x reside on the same PE, while after the transposition, all elements along y are on the same PE:

```
CALL MPI_ALLTOALL( f_inv(nys_x,nzb_x,0),  sendrecvcount_xy, MPI_REAL, &
     work(1,nzb_y,nxl_y,0), sendrecvcount_xy, MPI_REAL, comm1dy, ierr )
```

- The data resorting before and after the calls of `MPI_ALLTOALL` is highly optimized to account for the different processor architectures and also allows for overlapping communication and calculation.
- A further optimized FFT pressure solver is implemented in PALM release 23.04, which requires only two transpositions. It is switched on via parameter
  `psolver` `= ‘poisfft_sm‘`
  and may give a speedup of more than 30% in case of large setups.

Code parallelization
Parallel I/O

- PALM writes and reads some of the input/output files in parallel, i.e. each processor writes/reads his own file. **Each file then has a different name!**
  Example: binary files for restarts are written into a subfolder of PALM‘s temporary working directory

```
$fast_io_catalog/.../BINOUT/_000000
                 .../BINOUT/_000001   etc.
```

- These files can be handled (copied) by `palmrun` using the file attribute `pe` in the configuration file `.palm.iofiles`:

```
BINOUT*  out:lnpe  restart  $fast_io_catalog/$run_identifier/RESTART _d3d
```

- In this case, filenames are interpreted as directory names. The call

```
palmrun -r example_cbl -a "... restart" ...
```

  will copy the local <u>**directory**</u> `BINOUT` to the user <u>**directory**</u>

```
.../RESTART_DATA/example_cbl_d3d
```

General comment:

- Parallel I/O on a large number of files (>1000) may cause severe file system problems (e.g. on Lustre file systems).
- **Workaround:** reduce the maximum number of parallel I/O streams (see `palmrun`-option **`-w`**), or use **`restart_data_format`** = ‘mpi‘ (will soon be PALM‘s default).

Code parallelization

Parallel I/O for 2D/3D data

- 2D- and 3D-data output is also written in parallel by the cores (2D: by default, 3D: generally).
- Because graphics software (`ncview`, `ncl`, `ferret`, etc.) expect the data to be in one file, these output files have to be merged to one single file after PALM has finished. This is done by the utility program `combine_plot_fields`, which is automatically executed by `palmrun` after the PALM simulation has successfully finished.
- The executable `combine_plot_fields` is created during the installation process by calling `palmbuild.`
- PALM writes 2D-data of the total domain directly into one NetCDF file (without invoking `combine_plot_fields`) if runtime parameter **`data_output_2d_on_each_pe`** `= .FALSE.` has been set.
- If you have a NetCDF4/HDF5 library which has parallel I/O support, PALM can write the 3D-data from the total domain directly into one NetCDF file without invoking `combine_plot_fields`.

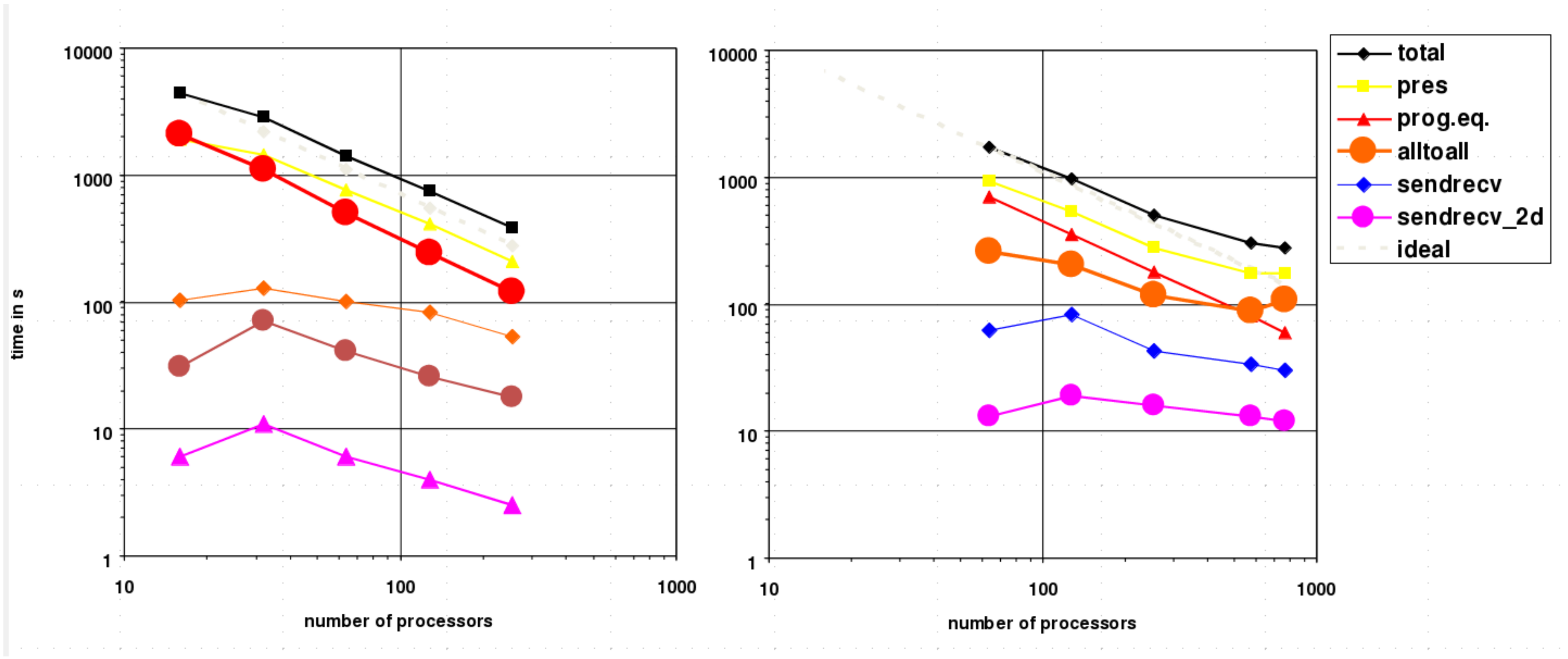
  This requires setting of runtime parameter **`netcdf_data_format`** `= 5` and PALM to be compiled with cpp-options

  ```
  %cpp_options ... –D__netcdf –D__netcdf4 –D__netcdf4_parallel
  ```
- **Attention:** Installation of NetCDF4/HDF5 can be very tricky.

Code parallelization

Performance examples (I)

- Simulation using 1536 * 768 * 242 grid points (~ 60 GByte)



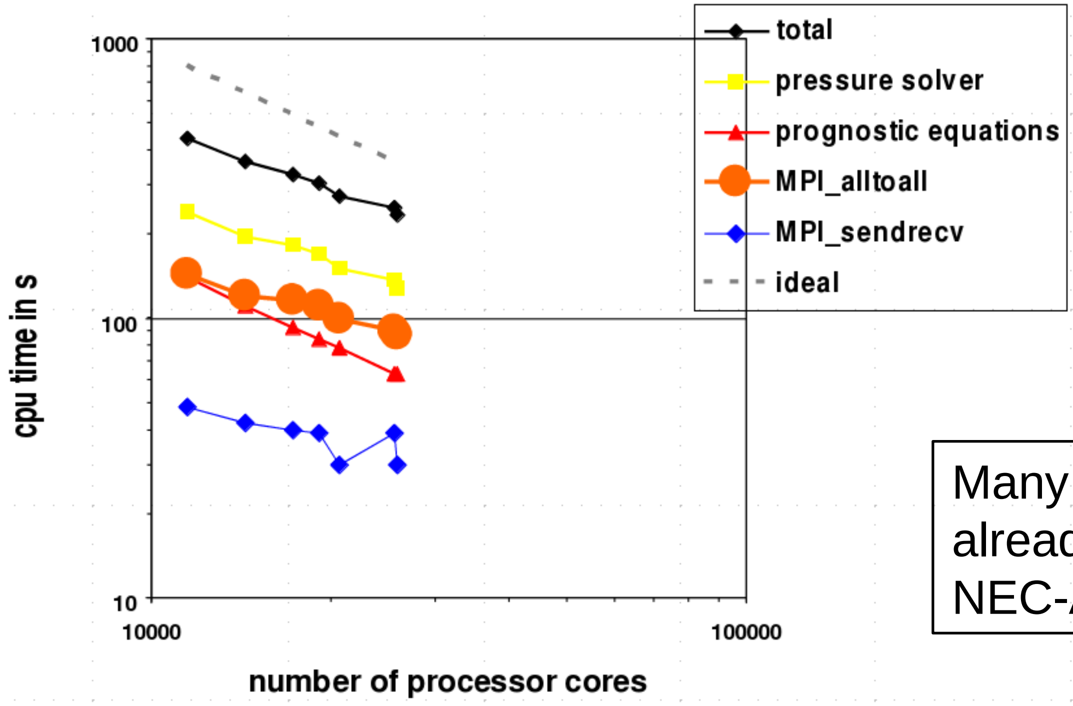
IBM-Regatta, HLRN, Hannover (1D domain decomposition)

Sun Fire X4600, Tokyo Institute of Technology (2D domain decomposition)

Code parallelization

PALM - Scalability

- Simulation with 4320^3 grid points (~ 13 TByte memory) with overlap
- `(MPICH_GNI_MAX_EAGER_MSG_SIZE=16384)`



Many parts of PALM code are already optimized for the new NEC-Aurora vector engines.

Cray XC40, HLRN-III 2nd stage, Haswell, Berlin