Parallelization

PALM group

Institute of Meteorology and Climatology, Leibniz Universität Hannover

last update: 21st September 2015

Basics of Parallelization

Parallelization:

Basics of Parallelization

Parallelization:

- All processor elements (PE, core) are carrying out the same program code (SIMD).

Basics of Parallelization

Parallelization:

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

Basics of Parallelization

Parallelization:

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

Realization:

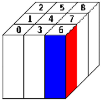
Basics of Parallelization

Parallelization:

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

Realization:

each PE solves the equations for a different subdomain of the total domain



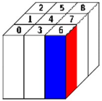
Basics of Parallelization

Parallelization:

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

Realization:

each PE solves the equations for a different subdomain of the total domain



program loops are parallelized, i.e. each processor solves for a subset of the total index range

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

```
!$acc kernels
DO i = 1, 100
   .
   .
   .
ENDDO
```

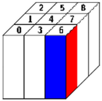
Basics of Parallelization

Parallelization:

- All processor elements (PE, core) are carrying out the same program code (SIMD).
- Each PE of a parallel 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 variable values from its subdomain, communication / data exchange between PEs is necessary

program loops are parallelized, i.e. each processor solves for a subset of the total index range

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

```
!$acc kernels
DO i = 1, 100
   .
   .
   .
ENDDO
```

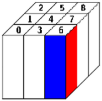
Basics of Parallelization

Parallelization:

- All processor elements (PE, core) are carrying out the same program code (SIMD).
- Each PE of a parallel 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 variable values from its subdomain, communication / data exchange between PEs is necessary

program loops are parallelized, i.e. each processor solves for a subset of the total index range

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

```
!$acc kernels
DO i = 1, 100
   .
   .
   .
ENDDO
```

parallelization can easily be done by the compiler, if all PEs have access to all variables (shared memory)

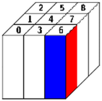
Basics of Parallelization

Parallelization:

- All processor elements (PE, core) are carrying out the same program code (SIMD).
- Each PE of a parallel 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 variable values from its subdomain, communication / data exchange between PEs is necessary

message passing model (MPI)

program loops are parallelized, i.e. each processor solves for a subset of the total index range

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

```
!$acc kernels
DO i = 1, 100
  .
  .
  .
ENDDO
```

parallelization can easily be done by the compiler, if all PEs have access to all variables (shared memory)

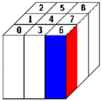
Basics of Parallelization

Parallelization:

- All processor elements (PE, core) are carrying out the same program code (SIMD).
- Each PE of a parallel 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 variable values from its subdomain, communication / data exchange between PEs is necessary
message passing model (MPI)

program loops are parallelized, i.e. each processor solves for a subset of the total index range

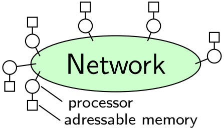
```
!$OMP DO
DO i = 1, 100
   .
   .
   .
ENDDO
```

```
!$acc kernels
DO i = 1, 100
   .
   .
   .
ENDDO
```

parallelization can easily be done by the compiler, if all PEs have access to all variables (shared memory)
shared memory model (OpenMP)
accelerator model (OpenACC)

Basic Architectures of Massively Parallel Computers

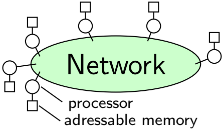
Basic Architectures of Massively Parallel Computers



distributed memory

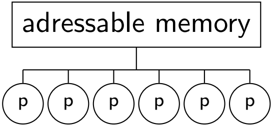
(Cray-XC30)

Basic Architectures of Massively Parallel Computers



distributed memory

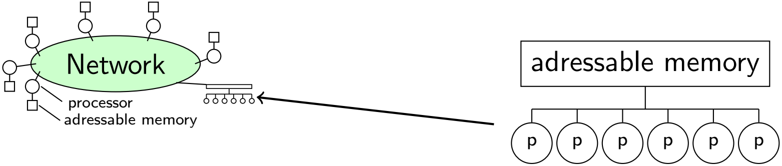
(Cray-XC30)



shared memory

(SGI-Altix, multicore PCs)

Basic Architectures of Massively Parallel Computers



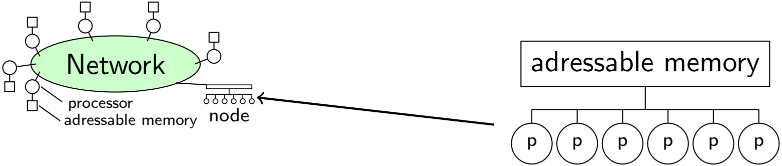
distributed memory

(Cray-XC30)

shared memory

(SGI-Altix, multicore PCs)

Basic Architectures of Massively Parallel Computers



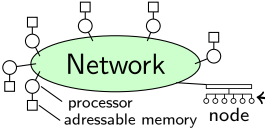
distributed memory

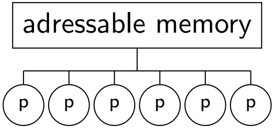
(Cray-XC30)

shared memory

(SGI-Altix, multicore PCs)

Basic Architectures of Massively Parallel Computers





distributed memory

(Cray-XC30)

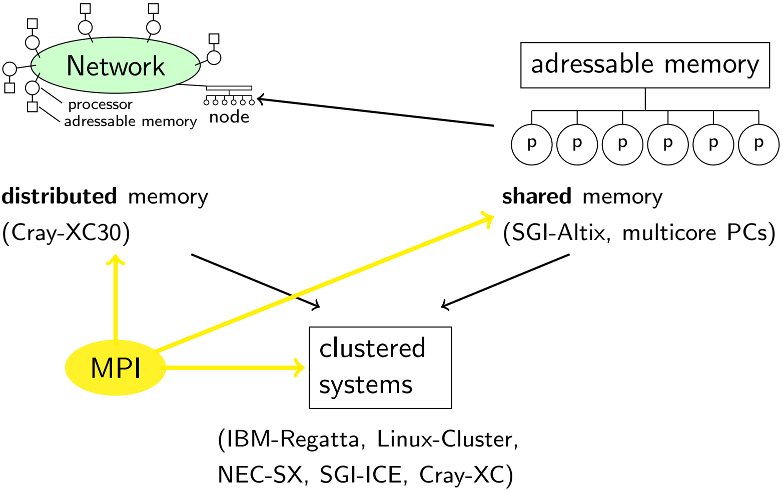
shared memory

(SGI-Altix, multicore PCs)

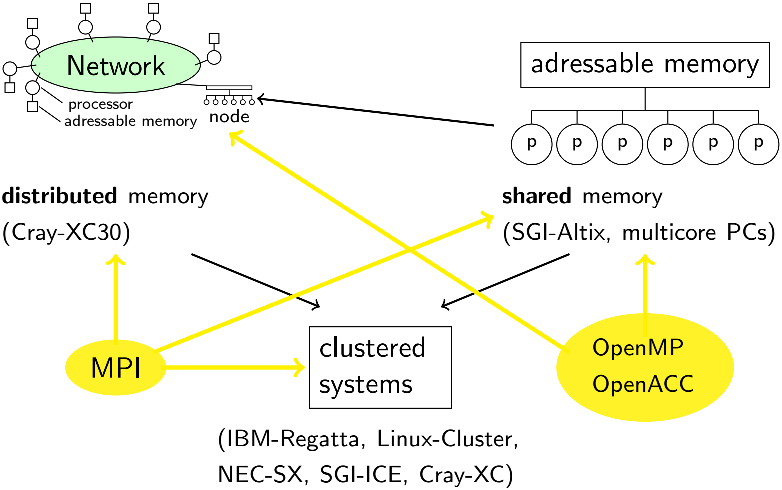
clustered systems

(IBM-Regatta, Linux-Cluster, NEC-SX, SGI-ICE, Cray-XC)

Basic Architectures of Massively Parallel Computers



Basic Architectures of Massively Parallel Computers



PALM Parallelization Model

PALM Parallelization Model

General demands for a parallelized program:

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

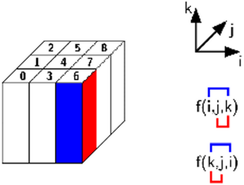
The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:

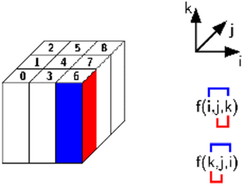


PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:



contiguous data in memory (FORTRAN):

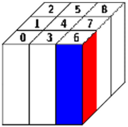
columns of i
no contiguous data at all

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:



contiguous data in memory (FORTRAN):

f(i,j,k)

columns of i
no contiguous data at all

f(k,j,i)

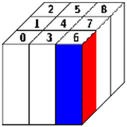
columns of k
planes of k,j (all data contiguous)

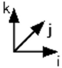
PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:





contiguous data 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)

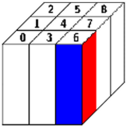
- Alternatively, a 1D-decomposition along x or y may be used.

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:



contiguous data 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)

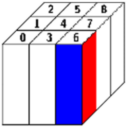
- Alternatively, a 1D-decomposition along x or y may be used.
- Message passing is realized using MPI.

PALM Parallelization Model

General demands for a parallelized program:

- Load balancing
- Small communication overhead
- Scalability (up to large numbers of processors)

The basic parallelization method used for PALM is a 2D-domain decomposition along x and y:



contiguous data 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)

- Alternatively, a 1D-decomposition along x or y may be used.
- Message passing is realized using MPI.
- OpenMP parallelization as well as mixed usage of OpenMP and MPI is realized.

Implications of Decomposition

Implications of Decomposition

- Central finite differences cause local data dependencies

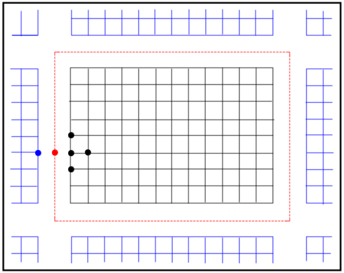
  solution: introduction of ghost points

Implications of Decomposition

- Central finite differences cause local data dependencies

  solution: introduction of ghost points

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

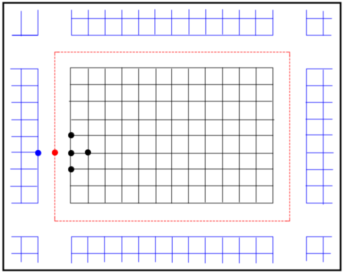


Implications of Decomposition

- Central finite differences cause local data dependencies

  solution: introduction of ghost points

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



- FFT and linear equation solver cause non-local data dependencies

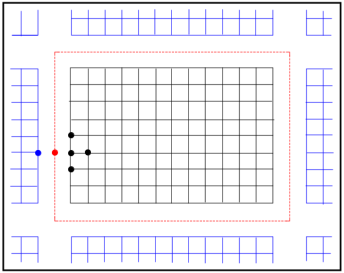
  solution: transposition of 3D-arrays

Implications of Decomposition

- Central finite differences cause local data dependencies

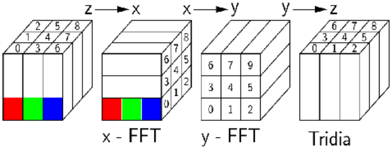
  solution: introduction of ghost points

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



- FFT and linear equation solver cause non-local data dependencies

  solution: transposition of 3D-arrays



Example: transpositions for solving the Poisson equation

How to Use the Parallelized Version of PALM

- The parallel version of PALM is switched on by `mrun`-option "`-K parallel`". Additionally, the number of required processors and the number of tasks per node (number of PEs to be used on one node) have to be provided:

```
mrun ...  -K parallel -X64 -T8 ...
```

How to Use the Parallelized Version of PALM

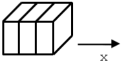
- The parallel version of PALM is switched on by `mrun`-option "`-K parallel`". Additionally, the number of required processors and the number of tasks per node (number of PEs to be used on one node) have to be provided:
  `mrun ...  -K parallel -X64 -T8 ...`
- From an accounting point of view, it is always most efficient to use all PEs of a node (e.g. `-T8`) (in case of a "non-shared" usage of nodes).

How to Use the Parallelized Version of PALM

- The parallel version of PALM is switched on by `mrun`-option "`-K parallel`". Additionally, the number of required processors and the number of tasks per node (number of PEs to be used on one node) have to be provided:
  `mrun ... -K parallel -X64 -T8 ...`
- From an accounting point of view, it is always most efficient to use all PEs of a node (e.g. `-T8`) (in case of a "non-shared" usage of nodes).
- If a normal unix-kernel operating system (not a micro-kernel) is running on each CPU, then there migth be a speed-up of the code, if 1-2 PEs less than the total number of PEs on the node are used.

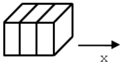
How to Use the Parallelized Version of PALM

- The parallel version of PALM is switched on by `mrun`-option "`-K parallel`". Additionally, the number of required processors and the number of tasks per node (number of PEs to be used on one node) have to be provided:
  `mrun ... -K parallel -X64 -T8 ...`
- From an accounting point of view, it is always most efficient to use all PEs of a node (e.g. `-T8`) (in case of a "non-shared" usage of nodes).
- If a normal unix-kernel operating system (not a micro-kernel) is running on each CPU, then there migth be a speed-up of the code, if 1-2 PEs less than the total number of PEs on the node are used.
- On machines with a comparably slow network, a 1D-decomposition (along x) should be used, because then only two transpositions have to be carried out by the pressure solver. A 1D-decomposition is automatically used for NEC-machines (e.g. `-h necriam`). The virtual processor grid to be used can be set manually by d3par-parameters `npex` and `npey`.



How to Use the Parallelized Version of PALM

- The parallel version of PALM is switched on by `mrun`-option "`-K parallel`". Additionally, the number of required processors and the number of tasks per node (number of PEs to be used on one node) have to be provided:
  `mrun ... -K parallel -X64 -T8 ...`
- From an accounting point of view, it is always most efficient to use all PEs of a node (e.g. `-T8`) (in case of a "non-shared" usage of nodes).
- If a normal unix-kernel operating system (not a micro-kernel) is running on each CPU, then there migth be a speed-up of the code, if 1-2 PEs less than the total number of PEs on the node are used.
- On machines with a comparably slow network, a 1D-decomposition (along x) should be used, because then only two transpositions have to be carried out by the pressure solver. A 1D-decomposition is automatically used for NEC-machines (e.g. `-h necriam`). The virtual processor grid to be used can be set manually by d3par-parameters `npex` and `npey`.
- 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 processors (>10000?) or with Intel-Xeon-Phi accelerator boards.



MPI Communication

- MPI (message passing interface) is a portable interface for communication between PEs (FORTRAN or C library).

MPI Communication

- MPI (message passing interface) is a portable interface for communication between PEs (FORTRAN or C library).
- MPI on the Cray-XC30 of HLRN-III is provided with module `PrgEnv-cray` which is loaded by default.

MPI Communication

- MPI (message passing interface) is a portable interface for communication between PEs (FORTRAN or C library).
- MPI on the Cray-XC30 of HLRN-III is provided with module `PrgEnv-cray` which is loaded by default.
- All MPI calls must be within

```
CALL MPI_INIT( ierror )
:
:
CALL MPI_FINALIZE( ierror )
```

Communication Within PALM

- MPI calls within PALM are available when using the `mrun`-option "`-K parallel`".

Communication Within PALM

- MPI calls within PALM are available when using the `mrun`-option "`-K parallel`".
- Communication is needed for
  - exchange of ghost points

Communication Within PALM

- MPI calls within PALM are available when using the `mrun`-option "`-K parallel`".
- Communication is needed for
  - exchange of ghost points
  - transpositions (FFT-poisson-solver)

Communication Within PALM

- MPI calls within PALM are available when using the `mrun`-option "`-K parallel`".
- Communication is needed for
  - exchange of ghost points
  - transpositions (FFT-poisson-solver)
  - calculating global sums (e.g. for calculating horizontal averages)

Communication Within PALM

- MPI calls within PALM are available when using the `mrun`-option "`-K parallel`".
- Communication is needed for
  - exchange of ghost points
  - transpositions (FFT-poisson-solver)
  - calculating global sums (e.g. for calculating horizontal averages)
- 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.

Virtual Processor Grid Used in PALM

The processor grid and special data types are defined in file `init_pegrid.f90`

Virtual Processor Grid Used 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
pdims(1) = npex   !  # of processors along x
pdims(2) = npey   !  # of processors along y
cyclic(1) = .TRUE.
cyclic(2) = .TRUE.

CALL MPI_CART_CREATE( MPI_COMM_WORLD, ndim, pdims, cyclic, reorder, &
                      comm2d, ierr )
```

Virtual Processor Grid Used 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
pdims(1) = npex    !  # of processors along x
pdims(2) = npey    !  # of processors along y
cyclic(1) = .TRUE.
cyclic(2) = .TRUE.

CALL MPI_CART_CREATE( MPI_COMM_WORLD, ndim, pdims, cyclic, reorder, &
                      comm2d, ierr )
```

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

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

Virtual Processor Grid Used 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
pdims(1) = npex   !  # of processors along x
pdims(2) = npey   !  # of processors along y
cyclic(1) = .TRUE.
cyclic(2) = .TRUE.

CALL MPI_CART_CREATE( MPI_COMM_WORLD, ndim, pdims, cyclic, reorder, &
                      comm2d, ierr )
```

- 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 )
```

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
```

  The actual code uses `nxlg=nxl-nbgp`, etc...

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
```

  The actual code uses `nxlg=nxl-nbgp`, etc...

- The exchange of ghost points is done 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 )
```

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
```

  The actual code uses `nxlg=nxl-nbgp`, etc...

- The exchange of ghost points is done 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 )
```

- In the real code 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, ...
```

Transpositions

- Transpositions can be found in file `transpose.f90` (several subroutines for 1D- or 2D-decompositions; they are called mainly from the FFT pressure solver, see `poisfft.f90`.

Transpositions

- Transpositions can be found in file `transpose.f90` (several subroutines for 1D- or 2D-decompositions; they are called mainly from the FFT pressure solver, see `poisfft.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:

```
!
!-- in SUBROUTINE transpose_xy:
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 )
```

Transpositions

- Transpositions can be found in file `transpose.f90` (several subroutines for 1D- or 2D-decompositions; they are called mainly from the FFT pressure solver, see `poisfft.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:

```
!
!-- in SUBROUTINE transpose_xy:
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 even allows for overlapping communication and calculation.

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 restart are written into a subdirectory of the PALM working directory:

  ```
  BINOUT/_0000
  BINOUT/_0001
  ⋮
  ```

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 restart are written into a subdirectory of the PALM working directory:
  BINOUT/_0000
  BINOUT/_0001
  ⋮
- These files can be handled (copied) by `mrun` using the file attribute `pe` in the configuration file:
  `BINOUT out:loc:pe restart ~/palm/current_version/JOBS/$fname/RESTART _d3d`

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 restart are written into a subdirectory of the PALM working directory:

  ```
  BINOUT/_0000
  BINOUT/_0001
  ⋮
  ```

- These files can be handled (copied) by `mrun` using the file attribute pe in the configuration file:

  ```
  BINOUT out:loc:pe restart ~/palm/current_version/JOBS/$fname/RESTART _d3d
  ```

  In this case, filenames are interpreted as directory names. An `mrun` call using option "`-d example_cbl -r restart`" will copy the local **directory** `BINOUT` to the **directory** `.../RESTART/example_cbl_d3d` .

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 restart are written into a subdirectory of the PALM working directory:

  ```
  BINOUT/_0000
  BINOUT/_0001
  ⋮
  ```

- These files can be handled (copied) by `mrun` using the file attribute pe in the configuration file:

  `BINOUT out:loc:pe restart ~/palm/current_version/JOBS/$fname/RESTART _d3d`

  In this case, filenames are interpreted as directory names. An `mrun` call using option "`-d example_cbl -r restart`" will copy the local **directory** `BINOUT` to the **directory** `.../RESTART/example_cbl_d3d` .

General comment:

- Parallel I/O on a large number of files (>1000) currently may cause severe file system problems (e.g. on Lustre file systems).
  Workaround: reduce the maximum number of parallel I/O streams (see `mrun`-option `-w`)

PALM Parallel I/O for 2D/3D Data

- 2D- and 3D-data output is also written in parallel by the processors (2D: by default, 3D: generally).

PALM Parallel I/O for 2D/3D Data

- 2D- and 3D-data output is also written in parallel by the processors (2D: by default, 3D: generally).
- Because the 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 within the job by calling the utility program `combine_plot_fields.x` after PALM has successfully finished.

PALM Parallel I/O for 2D/3D Data

- 2D- and 3D-data output is also written in parallel by the processors (2D: by default, 3D: generally).
- Because the 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 within the job by calling the utility program `combine_plot_fields.x` after PALM has successfully finished.
- `combine_plot_fields.x` is automatically executed by `mrun`.

PALM Parallel I/O for 2D/3D Data

- 2D- and 3D-data output is also written in parallel by the processors (2D: by default, 3D: generally).
- Because the 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 within the job by calling the utility program `combine_plot_fields.x` after PALM has successfully finished.
- `combine_plot_fields.x` is automatically executed by `mrun`.
- The executable `combine_plot_fields.x` is created during the installation process by the command

```
mbuild -u -h <host identifier>
```

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file
- modules `cray_hdf5_parallel` and `cray_netcdf_hdf5parallel` have to be loaded

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file
- modules `cray_hdf5_parallel` and `cray_netcdf_hdf5parallel` have to be loaded
- cpp-switches `-D__netcdf`, `-D__netcdf4`, `-D__netcdf4_parallel` have to be set

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file
- modules `cray_hdf5_parallel` and `cray_netcdf_hdf5parallel` have to be loaded
- cpp-switches `-D__netcdf`, `-D__netcdf4`, `-D__netcdf4_parallel` have to be set
- Both is done in the default HLRN-III block of the configuration file (`lccrayh`)

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

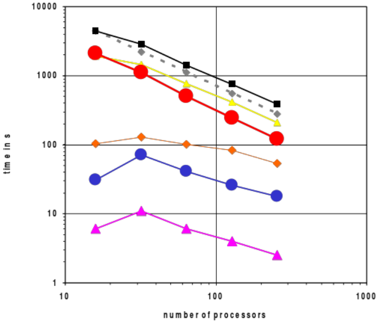
- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file
- modules `cray_hdf5_parallel` and `cray_netcdf_hdf5parallel` have to be loaded
- cpp-switches `-D__netcdf`, `-D__netcdf4`, `-D__netcdf4_parallel` have to be set
- Both is done in the default HLRN-III block of the configuration file (`lccrayh`)
- d3par-parameter `netcdf_data_format=5` has to be set in the parameter file

PALM Parallel I/O for 2D/3D Data with netCDF4/HDF5

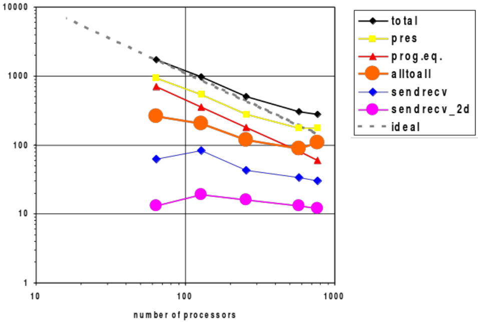
- The Cray XC30 of HLRN-III allows direct parallel I/O to a netCDF file
- modules `cray_hdf5_parallel` and `cray_netcdf_hdf5parallel` have to be loaded
- cpp-switches `-D__netcdf`, `-D__netcdf4`, `-D__netcdf4_parallel` have to be set
- Both is done in the default HLRN-III block of the configuration file (`lccrayh`)
- d3par-parameter `netcdf_data_format=5` has to be set in the parameter file
- `combine_plot_fields.x` is not required in this case

Performance Examples (I)

- Simulation using 1536 * 768 * 242 grid points ($\sim$ 60 GByte)



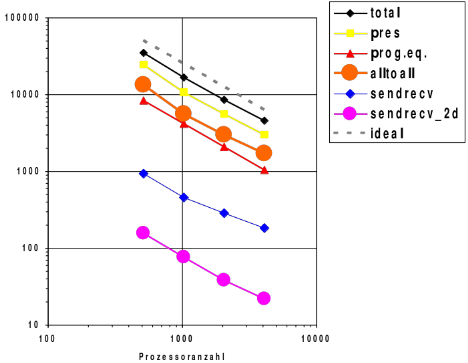
IBM-Regatta, HLRN, Hannover
(1D domain decomposition)



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

Performance Examples (II)

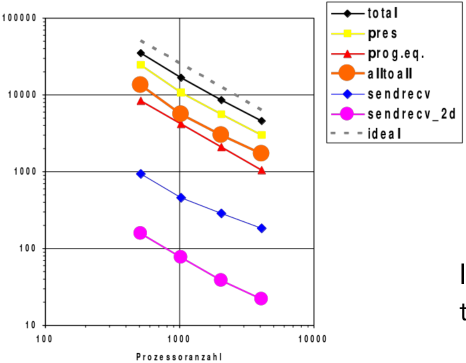
- Simulation with 2048^3 grid points ($\sim$ 2 TByte memory)



SGI-ICE2, HLRN-II, Hannover
(2D-domain decomposition)

Performance Examples (II)

- Simulation with 2048^3 grid points ($\sim$ 2 TByte memory)



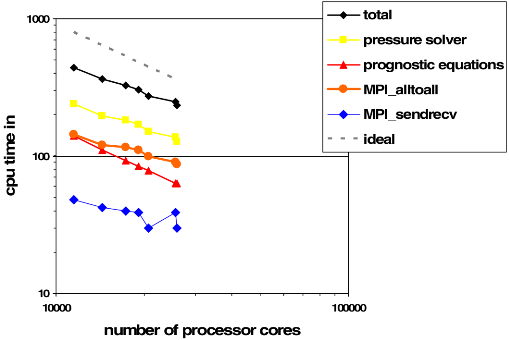
SGI-ICE2, HLRN-II, Hannover
(2D-domain decomposition)

largest simulation feasible on that system:

4096^3 grid points

Performance Examples (III)

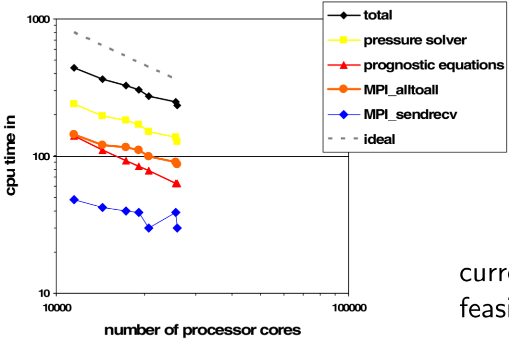
- Simulation with 4320^3 grid points ($\sim$ 13 TByte memory)



Cray-XC40, HLRN-III, Hannover
(2D-domain decomposition)

Performance Examples (III)

- Simulation with 4320^3 grid points ($\sim$ 13 TByte memory)



Cray-XC40, HLRN-III, Hannover
(2D-domain decomposition)

currently largest simulation feasible on that system:

5600^3 grid points