GWDG – Kurs
Parallel Programming with MPI

MPI Applications

Oswald Haan
ohaan@gwdg.de

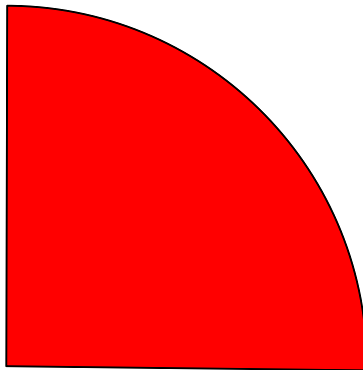
GWDG

Applications

- Approximate Calculation of π by Numerical Integration
- Largest Eigenvalue of a Matrix
- 2-dim Heat Equation

Calculating π

Aerea of a quarter circle = π / 4

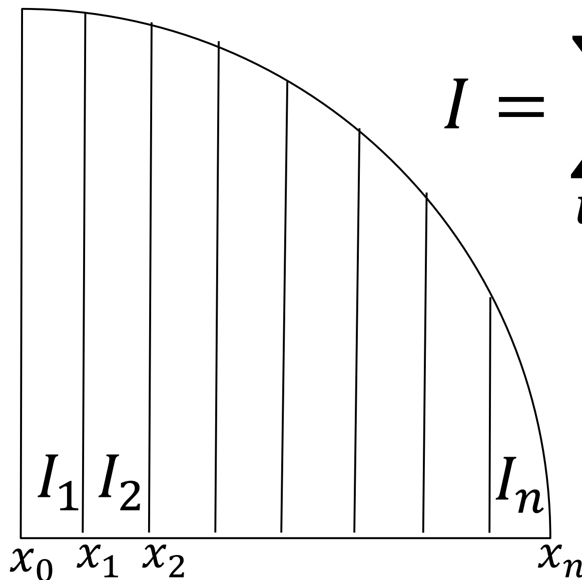


$$\pi = 4\int_0^1 \sqrt{1 - x^2}\,dx$$

Numerical Integration

$$I = \int_{l}^{h} f(x)dx$$

divide integration domain $[l, h]$ into n strips of width $\Delta = (h-l)/n$

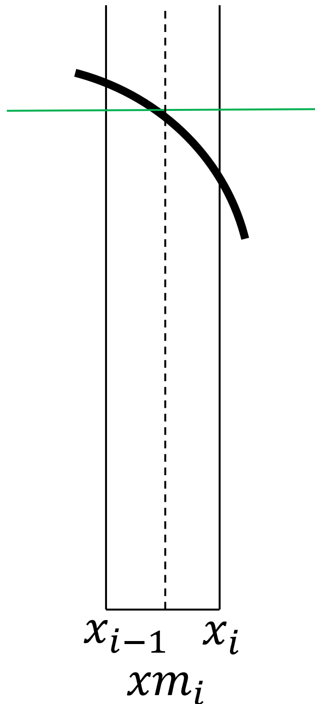


$$I = \sum_{i=1}^{n} I_i$$

$$I_i = \int_{x_{i-1}}^{x_i} f(x)\, dx$$

$$x_0 = l\,,\ x_n = h\,,\ x_i = i\,\Delta$$

Approximation: I = A + R



I_i exact area of a strip

$I_i^{(m)} = \Delta f(xm_i)$ approximate area of a strip

$$xm_i \mathbb{Z} \frac{1}{2}(x_{i-1} + x_i)$$

$$I = \sum_{i=1}^{n} I_i$$

$$A = \sum_{i=1}^{n} I_i^{(m)} = \Delta \sum_{i=1}^{n} f(xm_i)$$

Implementation

`program piapp`

1) reads the value for **`n`** from standard input,
2) Sum over n strips
3) writes results to standard output

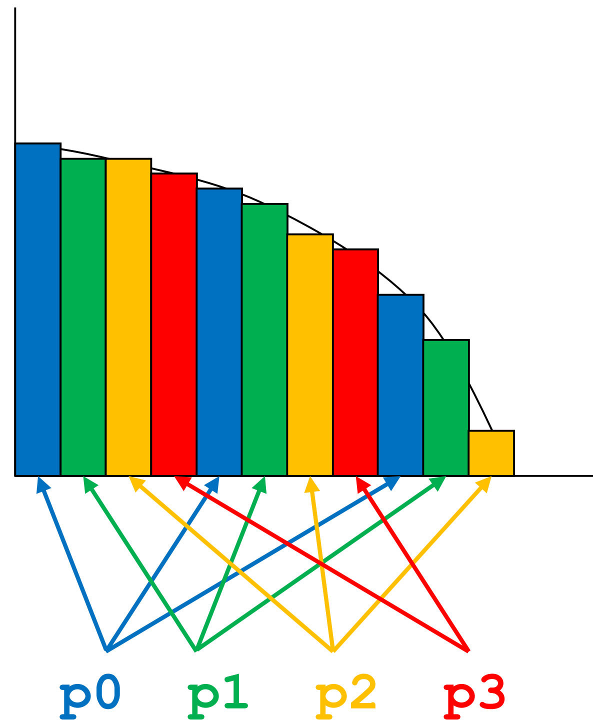
`piapp.f`	in directory **`Uebungen_f/pi`**
`piapp.c`	in directory **`Uebungen_c/pi`**
`piapp.py`	in directory **`Uebungen_py/pi`**

Fortran Implementation

```
del = 1.d0/dble(n)
app = 0.0d0
do i = 1 , n
   xi = (i-0.5d0)* del
   app = app + sqrt(1.d0-xi*xi)
end do
pia = 4.d0*del*app
```

Exercise 1

Distribute the sum of the **n** strips to **np** processes



Steps for Solving Exercise 1

`program piapp_mpi`

1) set up mpi environment **`np, me`**
2) for **`me=0`** : read the value for **`n`** from standard input,
3) distribute **`n`** to all tasks
4) starting from strip **`1+me`** sum every **`np`**-th strip on task me
5) add up all local sums to the total result **`pia`** on task **`0`**
6) for **`me=0:`** write results to standard output
7) exit from mpi

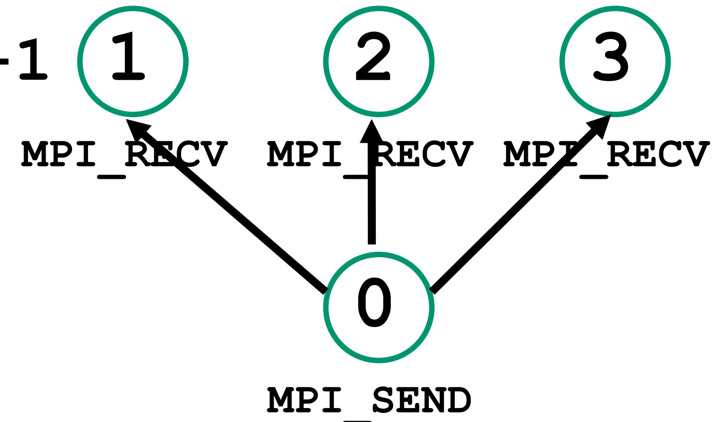
Use file **`piapp_mpi.[f,c,py]`** as a starting point

GWDG

Step 3: distribute **n** to all tasks

1 RECV of n in task 1,...,np-1

np-1 SEND's of n in task 0



Syntax

```
call MPI_SEND( n, 1, MPI_INTEGER, dst, tag, comm, ierr )
call MPI_RECV( n, 1, MPI_INTEGER, src, tag, comm, stat, ierr )

MPI_Send( &n, 1, MPI_INT, dst, tag, comm);
MPI_Recv( &n, 1, MPI_INT, src, tag, comm, &stat);

comm.Send(n,dest=dst)
comm.Recv(n,source=src)
```

Step 4: starting from strip **1+me** sum every **np**-th strip on task me

Fortran:

```
do i = me+1 , n, np
```

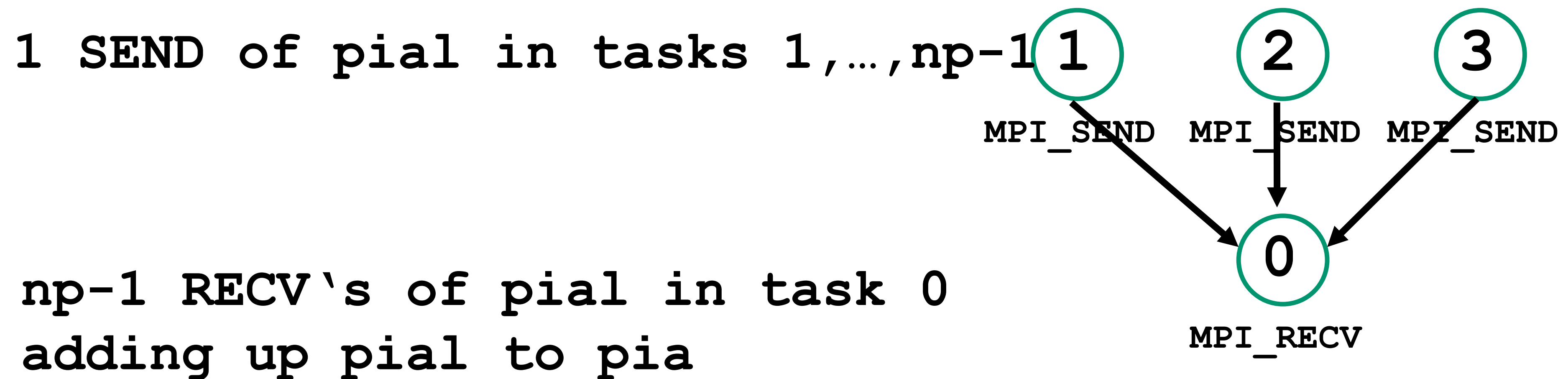
C:

```
for (i=me+1;i<=n;i=i+np){
```

mpi4py:

```
for i in range(1+me,n+1,np+1):
```

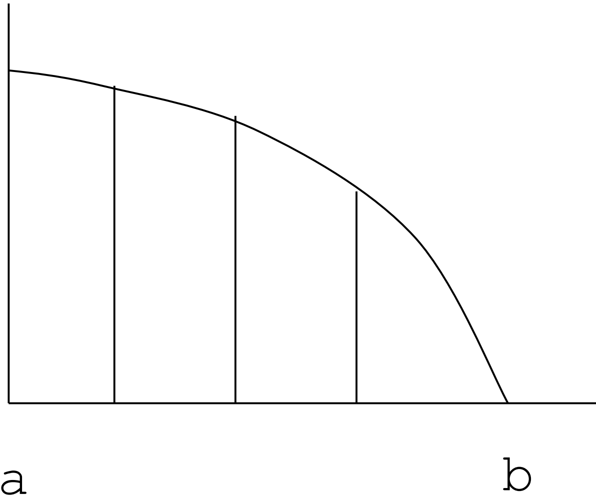
Step 5: add up all local values **pial** to the total result **pia** on process **0**



Solution to this exercise in

~ohaan/mpikurs/[f,c,py]/piapp_mpi_r[f,c,py]

Numerical Approximation with Higher Precision

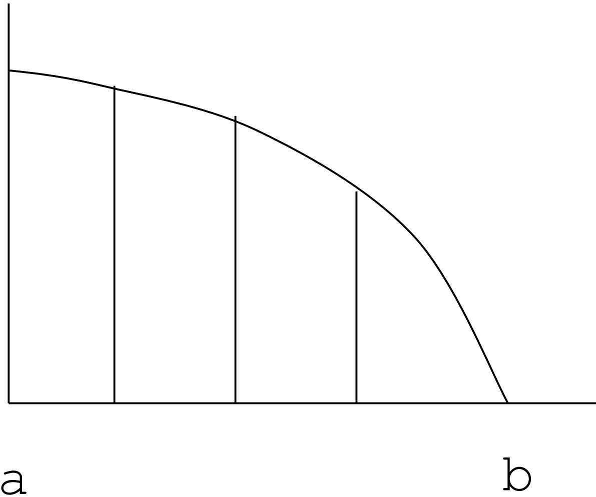


$$\int_a^b f(x)dx = \sum_{i=0}^{nin-1} \int_{a_i}^{b_i} f(x)dx$$

$$a_i = a + i\frac{b-a}{nin} \quad , \quad b_i = a_i + \frac{b-a}{nin}$$

divide integration domain [**`a`**, **`b`**] into **`nin`** intervals of size **`(b-a)/nin`**.

Numerical Approximation with Higher Precision



$$\int_a^b f(x)dx = \sum_{i=0}^{nin-1} \int_{a_i}^{b_i} f(x)dx$$

$$a_i = a + i\frac{b-a}{nin} \quad , \quad b_i = a_i + \frac{b-a}{nin}$$

In each interval **`i`** approximate the integral with **`n_i`** strips.
Use more strips if function is steeper
-> same precision in each interval

```
ni = n (f(ai)-f(bi))
n1 + n2 + ... = n
```

Fortran Implementation

```
program piprec
      ...
 width = 1.d0/dble(nin)
 pia = 0.0d0
 do i = 1 , nin
    hi = i * width
    lo = hi - width
    call numapp(lo,hi,n,app)
    pia = pia + app
 end do
```

Fortran Implementation

```
subroutine numapp(lo, hi, n, app)
returns
app : approximation for the integral from lo to hi
          using number of strips = n(f(lo)-f(hi))
            ...
    flo = sqrt(1.d0-lo*lo)
    fhi = sqrt(1.d0-hi*hi)
    nl = n * (flo - fhi)
    del = (hi-lo)/dble(nl)
    app = 0.0d0
    do i = 1 , nl
       xi = lo + (dble(i)-0.5d0)*del
       app = app + sqrt(1.d0-xi*xi)
    end do
    app = 4.d0*del*app
```

Example Code

`piprec.f` and `numapp.f` in directory `Uebungen_f/pi`
`piprec.c` and `numapp.c` in directory `Uebungen_c/pi`
`piprec.py` and `numapp.py` in directory `Uebungen_py/pi`

C and Fortran

Compile and link with makefile

```
> make piprec
```

Run with

```
> ./piprec.exe
```

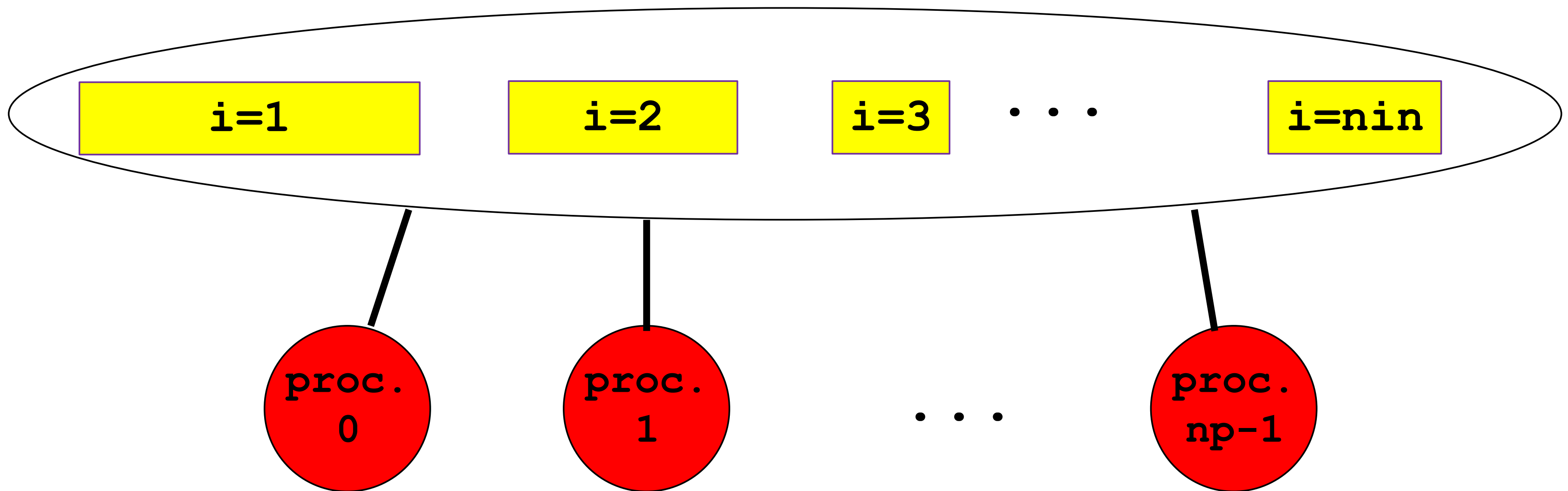
Python

Run with

```
> python piprec.py
```

Exercise 2

distribute **`nin`** evaluations of **`numapp`** to **`np`** processes



Problem: unequal workload for different evaluations
workload not known at run time

Distribute Work to Idle Workers

```
farmer:  me = 0
tres = 0
for iw  = 1 , nintv +(np-1)  :
     recv res from anytask
     tres = tres + res
     ipw = status(mpi_source)
     Send iw to ipw
```

```
worker: me > 0
res = 0
Send res to 0
for i = 1 , nintv :
     Recv iw from 0
     If iw > nintv:  exit
     res = work(iw)
     Send res  to 0
```

Use file **piprec_worker.[f,c,py]**

Distribute Work to Idle Workers (II)

```
Farmer:  me = 0
iw = 1
loop nintv times
     recv  ipidle from anytask
     send iw to ipidle
     iw = iw+1
loop over np-1 workers ip
     send iw to ip
     recv  local result from ip
     add local result to full result
```

```
Worker: me > 0
Loop nintv+1 times
   send me to farmer
   recv iw from farmer
   if index >  intv exit
   res = work(iw)
   add res to local result
send local result to farmer
```

GWDG

Solution for Exercises

If you have tried hard to perform the required exercises and the programs still don‘t work, you are allowed to look into the directories

```
~ohaan/mpikurs_solutions/f
~ohaan/mpikurs_solutions/c
~ohaan/mpikurs_solutions/py
```

where you will find the completed programs for some exercises

Raleigh - Ritz - Method

Eigenvalue problem : $\mathbf{A} \; \mathbf{v}_i = \lambda_i \; \mathbf{v}_i \quad , \; \lambda_0 > \lambda_1 \geq \lambda_2 \geq \ldots$

Choose start vector $\mathbf{x} = \sum_i r_i \; \mathbf{v}_i \;$ mit $\; r_0 \neq 0$

$$\mathbf{A}^n \, \mathbf{x} = \sum_i r_i \, \lambda_i^n \, \mathbf{v}_i = r_0 \, \lambda_0^n \left(\mathbf{v}_0 + \sum_{i \geq 1} r_i / r_0 \cdot (\lambda_i / \lambda_0)^n \cdot \mathbf{v}_i \right)$$

$$\xrightarrow{n \to \infty} \quad \alpha_n \, \mathbf{v}_0$$

$$\lambda_0 = \lim_{n \to \infty} (\mathbf{A}^{n+1} \mathbf{x})_1 \, / \, \mathbf{A}^n \mathbf{x})_1$$

Algorithm Raleigh - Ritz

Initialisiere Matrix $\mathbf{A} \in R^{n \times n}$

Wähle $\mathbf{x} \in R^n$ mit $x_1 = 1$

Schleife

$$\mathbf{x} \leftarrow \mathbf{A}\mathbf{x}$$

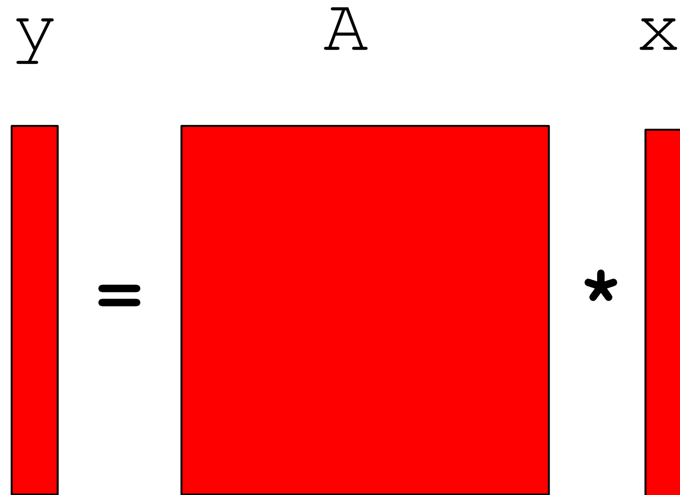
$$\lambda \leftarrow x_1$$

$$\mathbf{x} \leftarrow \lambda^{-1}\mathbf{x}$$

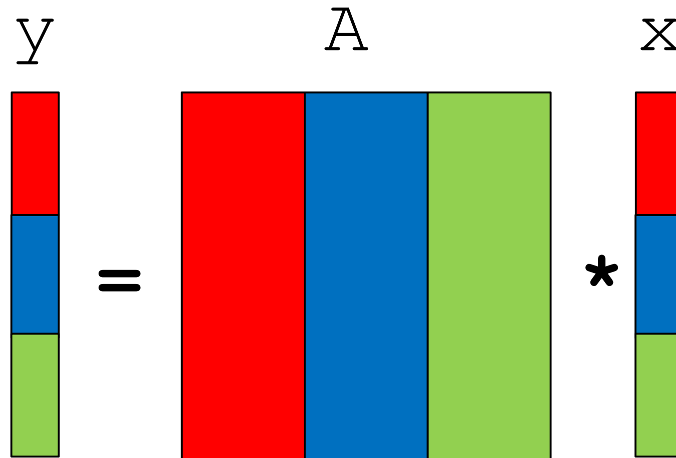
Source code in directory **`Uebungen_*/Ritz`**

Matrix-Vector Multiplication

$$y(i) = \sum_{j=1}^{n} A(i,j) * x(j) \quad , i = 1, \ldots, n$$



Parallel Matrix-Vector Multiplication Column-block Distribution

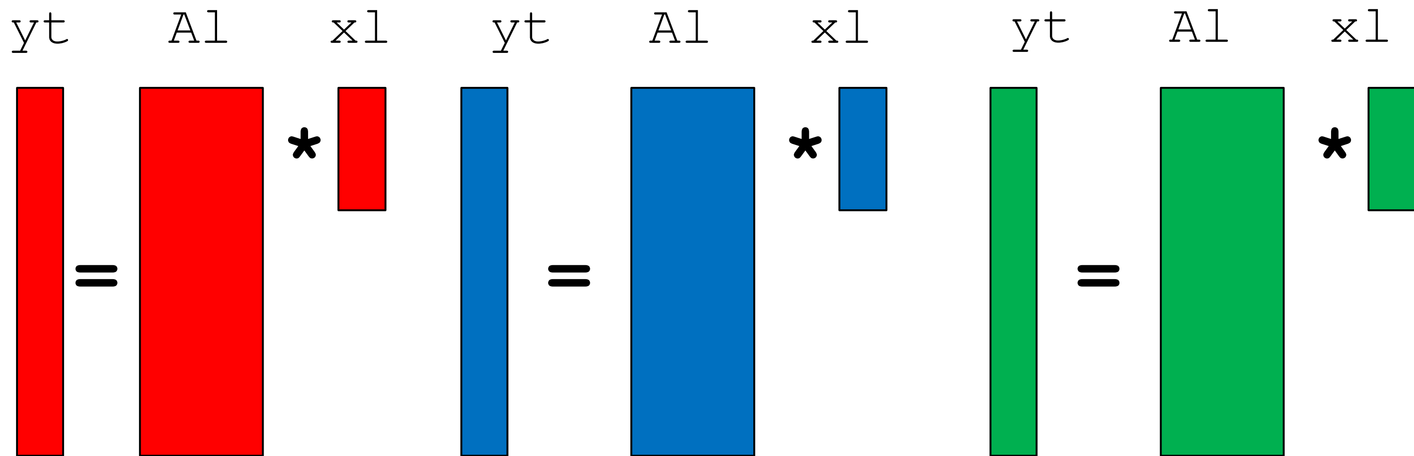


np processes ip = 0,...,np-1

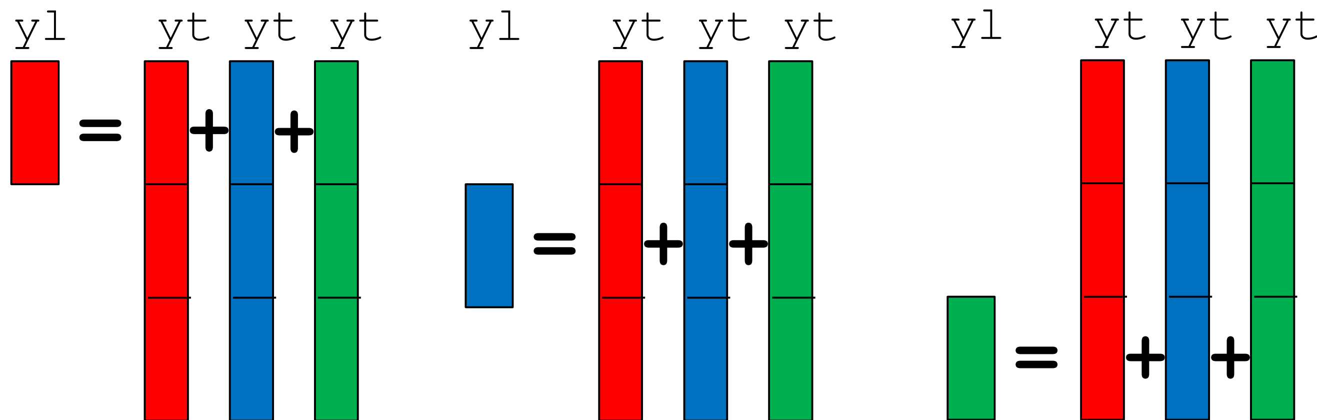
local on each process:
nloc colums of A, nloc elements of x and of y

nloc = n/np if n is multiple of np

Parallel Matrix-Vector Multiplication Column-block Distribution



local
partial results



local full results:
sum of partial results

program ritz_dist_col

Parallel Raley-Ritz Algorithm
with column-block distribution

`ritz_dist_col`

Ser	Input: Matrix-dimension n	
Ser	Initialise A	`dist_index`
Par	Distribute A to Al's	`dist_matrix_colblock`
Par	Initialise xl	
	Loop	
Par	yt = Al * xl	`DGEMV`
Par	global sum yl	`reduce_vector`
Ser	λ = yl(1)	
Par	distribute λ	`MPI_BCAST`
Par	xl = 1/λ * yl	

GWDG

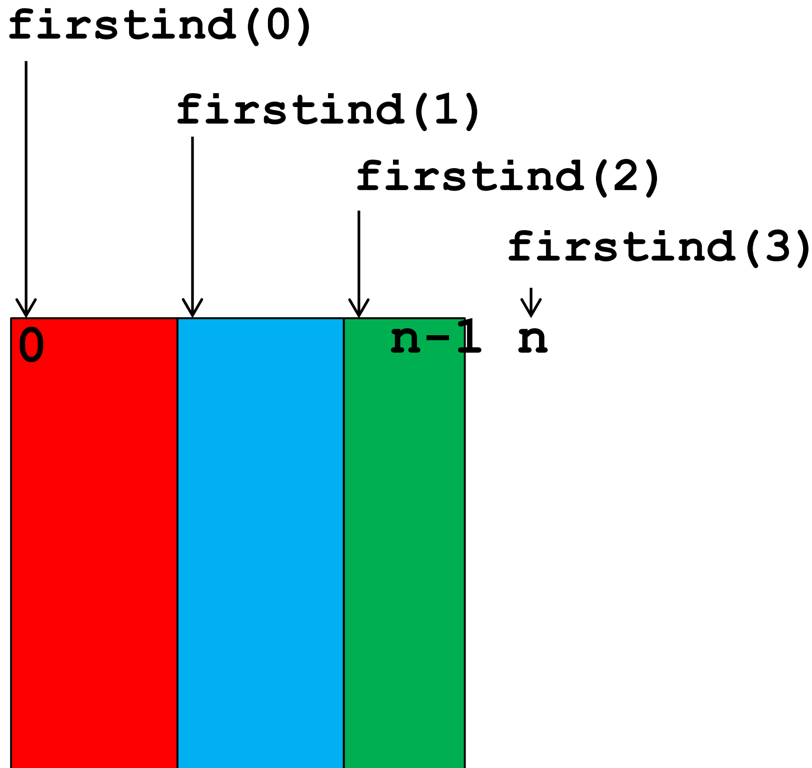
program ritz_dist_col

Generate executable

```
mpifort  -o ritz_dist_col.exe
            ritz_dist_col.f
               dist_index.f
     dist_matrix_colblock.f
                       mv.f
            reduce_vector.f
```

```
> make ritz_dist_col
```

dist_index(n,firstind)

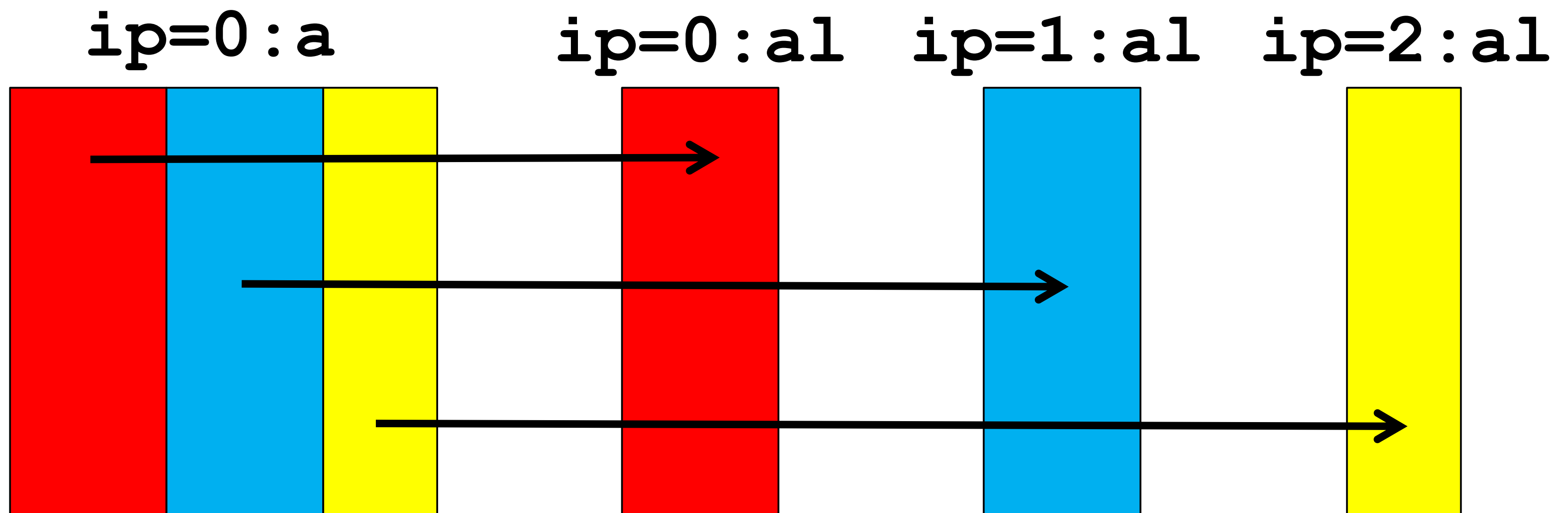



```
nl = int((n+nproc-1)/nproc)

firstind(0) = 0

ip = 1,…,nproc:
firstind(ip) =
    min(firstind(ip-1)+nl,n)
```

dist_matrix_colblock(m,n,a,al)



dist_matrix_colblock(m,n,a,al)

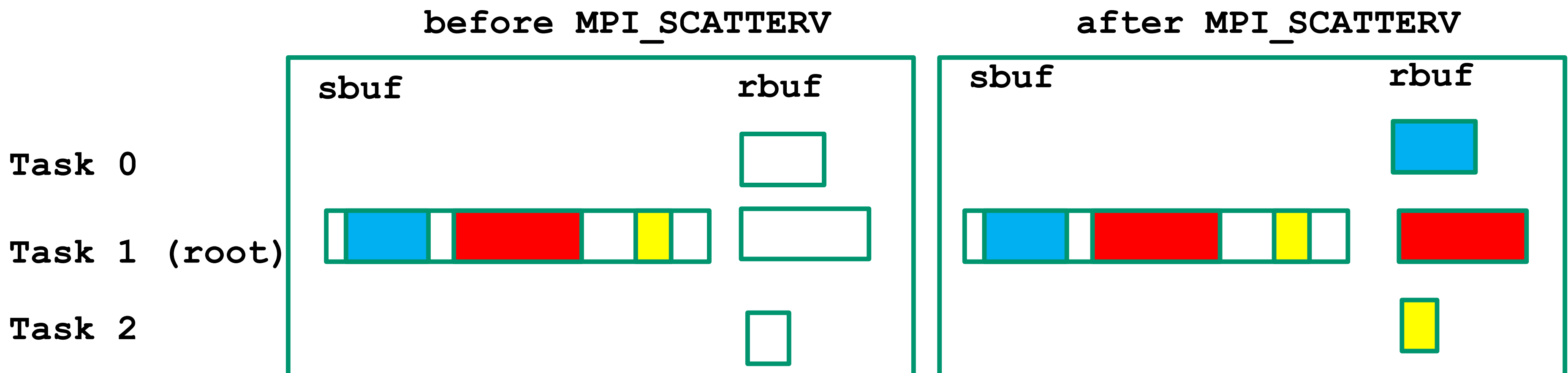
```
 real*8               a(*), al(*)
 call dist_index( n, firstind )
 ncom = m * ( firstind(myid+1) - firstind(myid) )
 call MPI_IRECV( al(1), ncom, MPI_DOUBLE_PRECISION, 0, 0,
:                  MPI_COMM_WORLD, req, ierr )
 if (myid.eq.0) then
    do ip = 0 , nproc-1
       ncom = m * ( firstind(ip+1) - firstind(ip) )
       ia = 1 + m*firstind(ip)
       call MPI_SEND( a(ia), ncom, MPI_DOUBLE_PRECISION
    :                   , ip, 0, MPI_COMM_WORLD, ierr )
     end do
 end if
 call MPI_WAIT( req, MPI_STATUS_IGNORE, ierr )
```

!!non-blocking MPI_IRECV prevents deadlock !!

Exercise: Use MPI_SCATTERV

```
C:  MPI_Scatterv( void *sbuf, int *scounts, int *displs, MPI_Type stype
                 , void *rbuf, int rcount, MPI_Type rtype
                 , int root, MPI_Comm comm )
Fortran:  MPI_SCATTERV( sbuf, scounts, displs, stype
                     , rbuf, rcount, rtype, root, comm, ierr )
         <type>sbuf(*), rbuf(*)
         INTEGER scounts(*), dipls(*), stype, rcount, rtype, comm, ierr

mpi4py: comm.Scatterv(sar, rar, root= 0)
         sar = [senddata,scounts,dspls,dtype]
```



Exercise: Use **MPI_SCATTERV**

Define:

```
i = 0 , nproc-1 :
  counts(i) = n*(firstind(i+1)-firstind(i))
  dspls(i) = n * firstind(i)
```

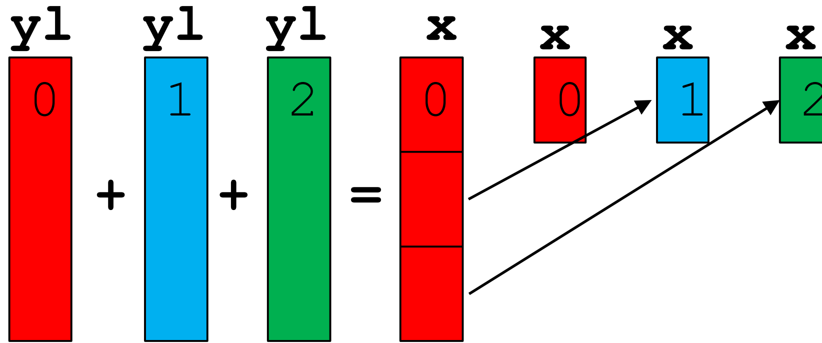
Replace

```
dist_matrix_colblock(n,n,a,al)
```

by

```
MPI_SCATTERV( a,counts,dspls,MPI_DOUBLE_PRECISION
         , al, counts(myid), MPI_DOUBLE_PRECISION
         , 0, MPI_COMM_WORLD, ierr )
```

reduce_vector(n,y,x)




```
 call MPI_REDUCE(y, x, n, MPI_DOUBLE_PRECISION,
:                    MPI_SUM, 0, MPI_COMM_WORLD, ierr )
 if (myid.eq.0) then
    do ip = 1 , nproc-1
       displ = firstind(ip)
       count = firstind(ip+1) - firstind(ip)
       call MPI_SEND(x(displ),count,MPI_DOUBLE_PRECISION
:                  , ip, 0, MPI_COMM_WORLD, ierr )
    end do
 else
    count = firstind(myid+1) - firstind(myid)
    call MPI_RECV(x,count,MPI_DOUBLE_PRECISION
:               , 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr )
 end if
```

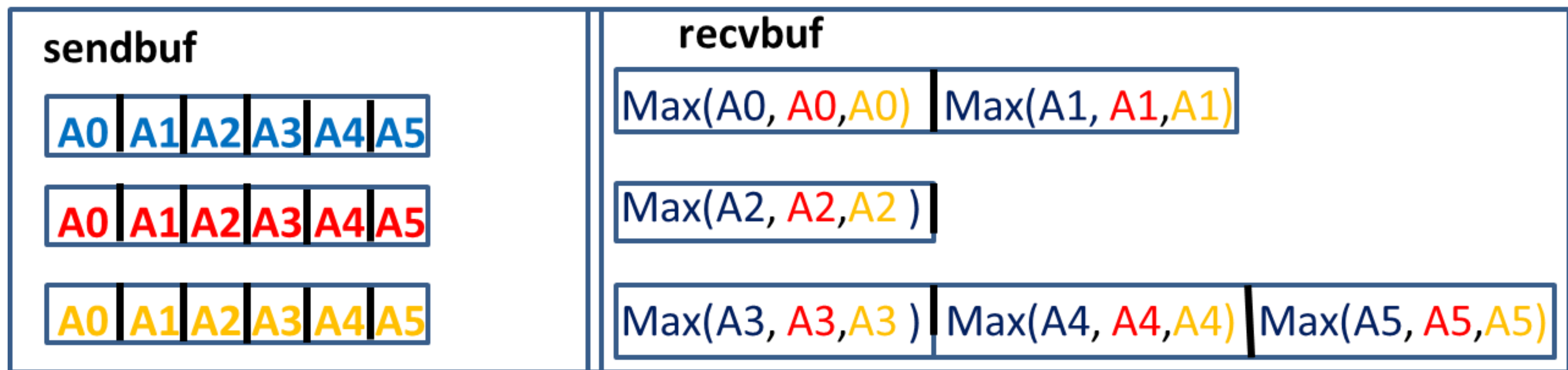
Exercise : MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTER(sendbuf, recvbuf, counts, datatype,
op, comm)

The number of elements in sendbuf to be reduced over nproc tasks is

counts(0)+ . . . +counts(nproc-1)

counts(ip) results are stored in process ip



Exercise : MPI_REDUCE_SCATTER

Define :

```
 for i = 0 , nproc-1
   counts(i) = firstind(i+1)-firstind(i)
```

Replace

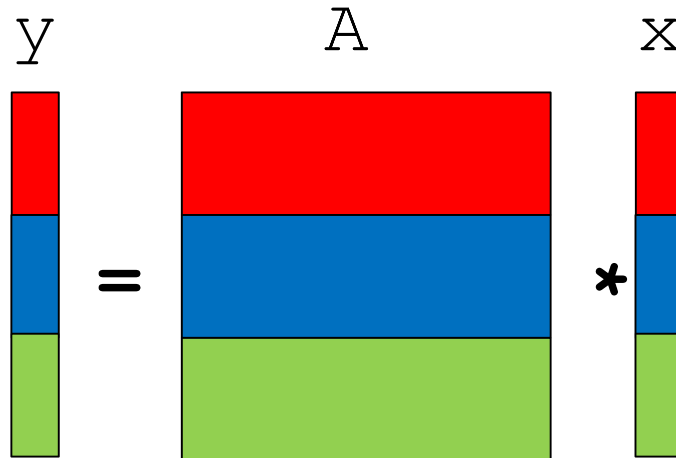
```
reduce_vector(n,y,x)
```

by

```
MPI_Reduce_scatter(y, x, counts
      , MPI_DOUBLE_PRECISION, MPI_SUM
      , MPI_COMM_WORLD, ierr)
```

GWDG

Parallel Matrix-Vector Multiplication
Row-block Distribution



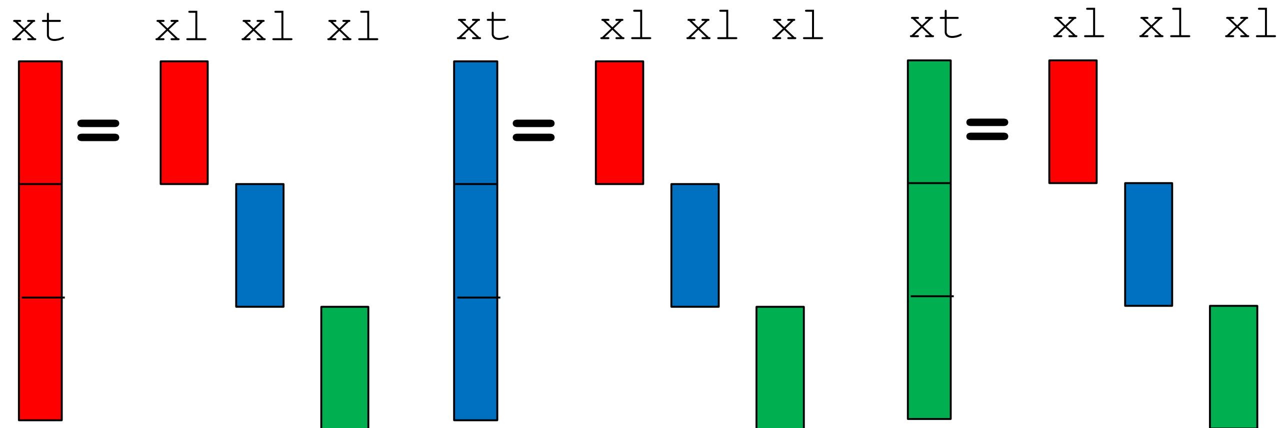
np processes ip = 0,...,np-1

local on each process:

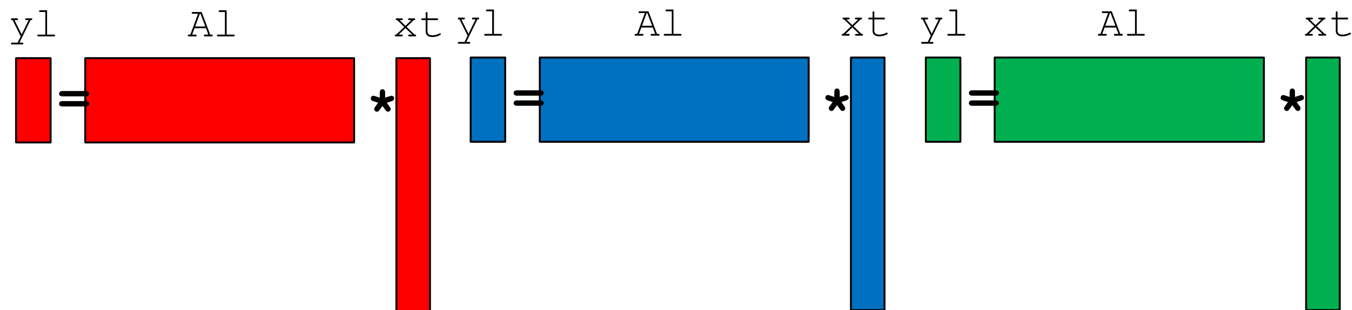
nloc = rows of A, nloc elements of x and of y

nloc = n/np if n is multiple of np

Parallel Matrix-Vector Multiplication Column-block Distribution



Collect the total input vector



calculate local elements of the full result

Programm ritz_dist_row

Paralleler Raley-Ritz Algorithmus mit Zeilenblock-Verteilung

`ritz_dist_row`

Ser	Input: Matrix-dimension n	
Ser	Initialise A	
Par	Distribute A to Al‘s	`dist_index` `dist_matrix_rowblock`
Par	Initialise xl	
	Loop	
Par	collect input vector xt	`global_vector`
Par	yl = Al * xt	`DGEMV`
Ser	λ = yl(1)	
Par	distribute λ	`MPI_BCAST`
Par	xl = 1/λ * yl	

GWDG

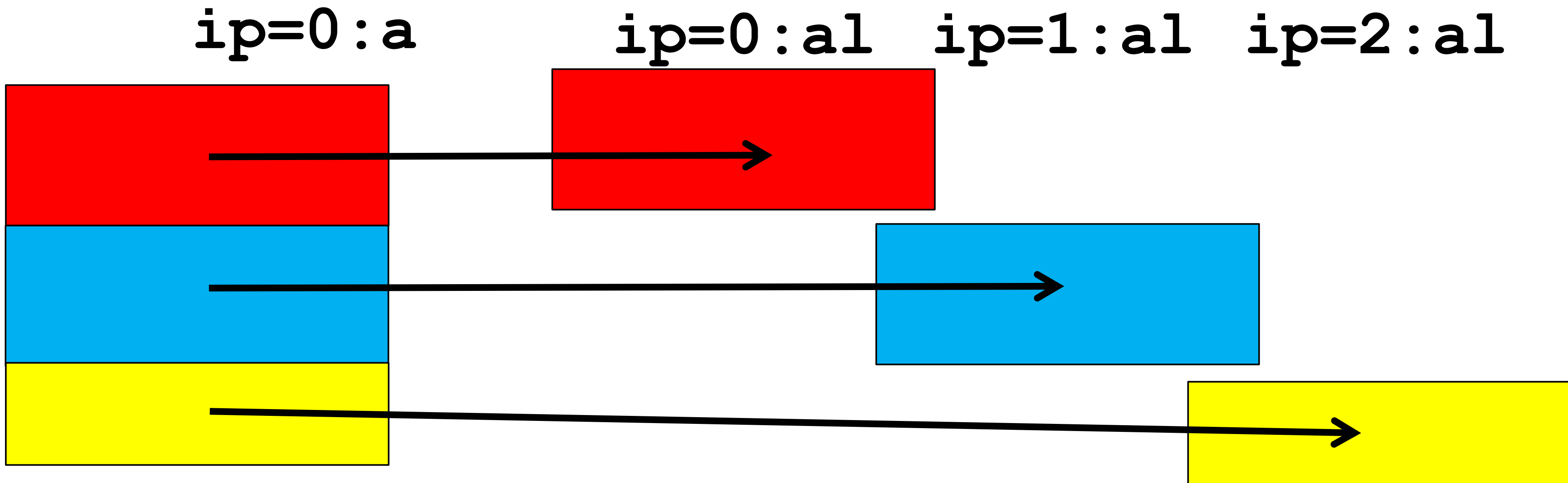
program ritz_dist_row

Generate executable

```
mpifort  -o ritz_dist_row.exe
            ritz_dist_row.f
               dist_index.f
     dist_matrix_rowblock.f
            global_vector.f
                       mv.f
```

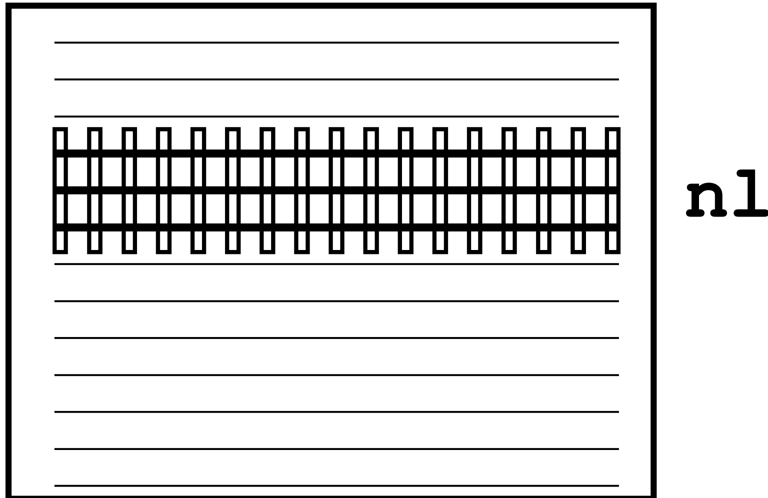
```
> make ritz_dist_row
```

dist_matrix_rowblock(m,n,a,al)

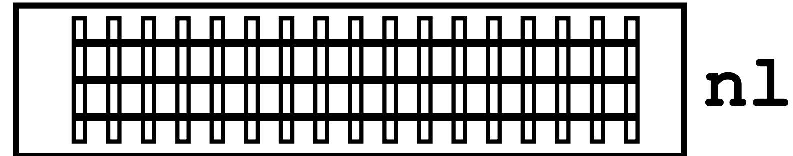


Rowblock Distribution of Global Matrix (1)

global **a**



local **al**




```
ia = firstind(ip)
ic = 0
do j = 1 , n
   do i = 1 , nl
      ic = ic+1
      al(ic) = a(ia+i,j)
   end do
end do
```

dist_matrix_rowblock(m,n,a,al)

Loop over all processes ip

Step1:

on proc. 0 collect rowblock from a for proc ip into contiguous memory in al:

Step 2:

send rowblock al for proc. ip from proc.0 to proc.ip

!! Proceed in reverse order: starting with proc. np-1
!! ending with process 0,
!! then al in proc. 0 contains the rowblock for proc. 0

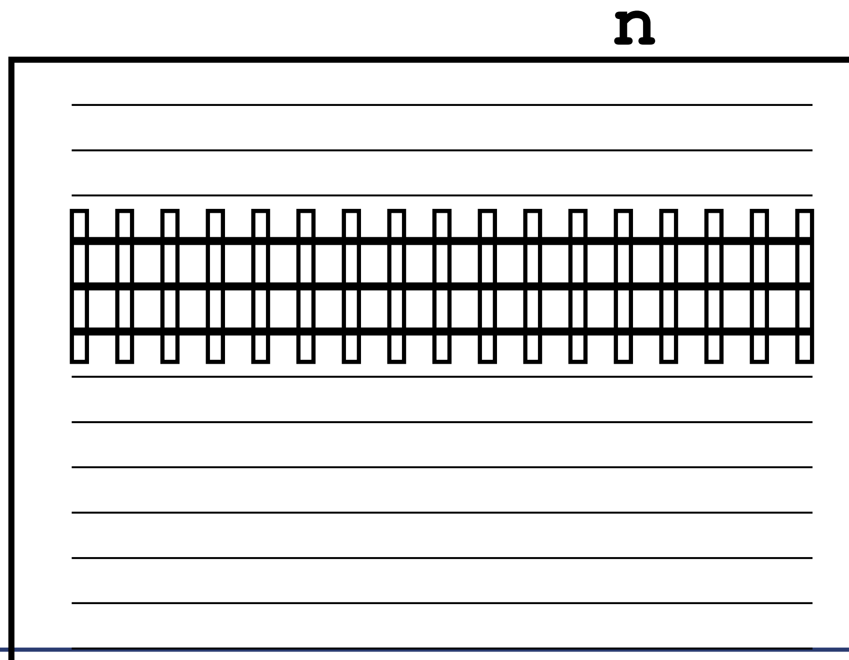
Rowblock Distribution of Global Matrix (2)

```
      if (myid.eq.0) then
         do ip = nproc-1 , 0 , -1
            nl = firstind(ip+1) - firstind(ip)
            ncom = n * nl
!    Copy rows from a to al
            ...
            if (ip.ge.1)
     :      call MPI_SEND( al, ncom,MPI_DOUBLE_PRECISION,
     :                ip, 0, MPI_COMM_WORLD, ierr )
         end do
       else
           ncom = n * (firstind(myid+1) - firstind(myid))
           call MPI_RECV( al, ncom, MPI_DOUBLE_PRECISION,
     :          0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
       end if
```

Derived Datatype with `MPI_Type_vector`

Declaration of **`newtype`**:

```
int newtype (Fortran)
MPI_Datatype newtype; (C)
MPI_Type_vector(count, blocklen, stride,
                oldtype, newtype)
```



ml

```
count = n
blocklen = ml
stride = m
```

Use of Datatype vector

Modify <u>**dist_matrix_rowblock**</u>

if (myid.eq,0):

- Define a new type **rowblock** mit **MPI_TYPE_VECTOR**
- Activate the new type **MPI_TYPE_COMMIT(rowblock,ierr)**

  For **ip = 0, nproc-1** send the new type to process **ip**

```
ia = firstind(ip)+1
MPI_SEND(a(ia),1,rowblock,ip,tag,
                MPI_COMM_WORLD,ierr)
```

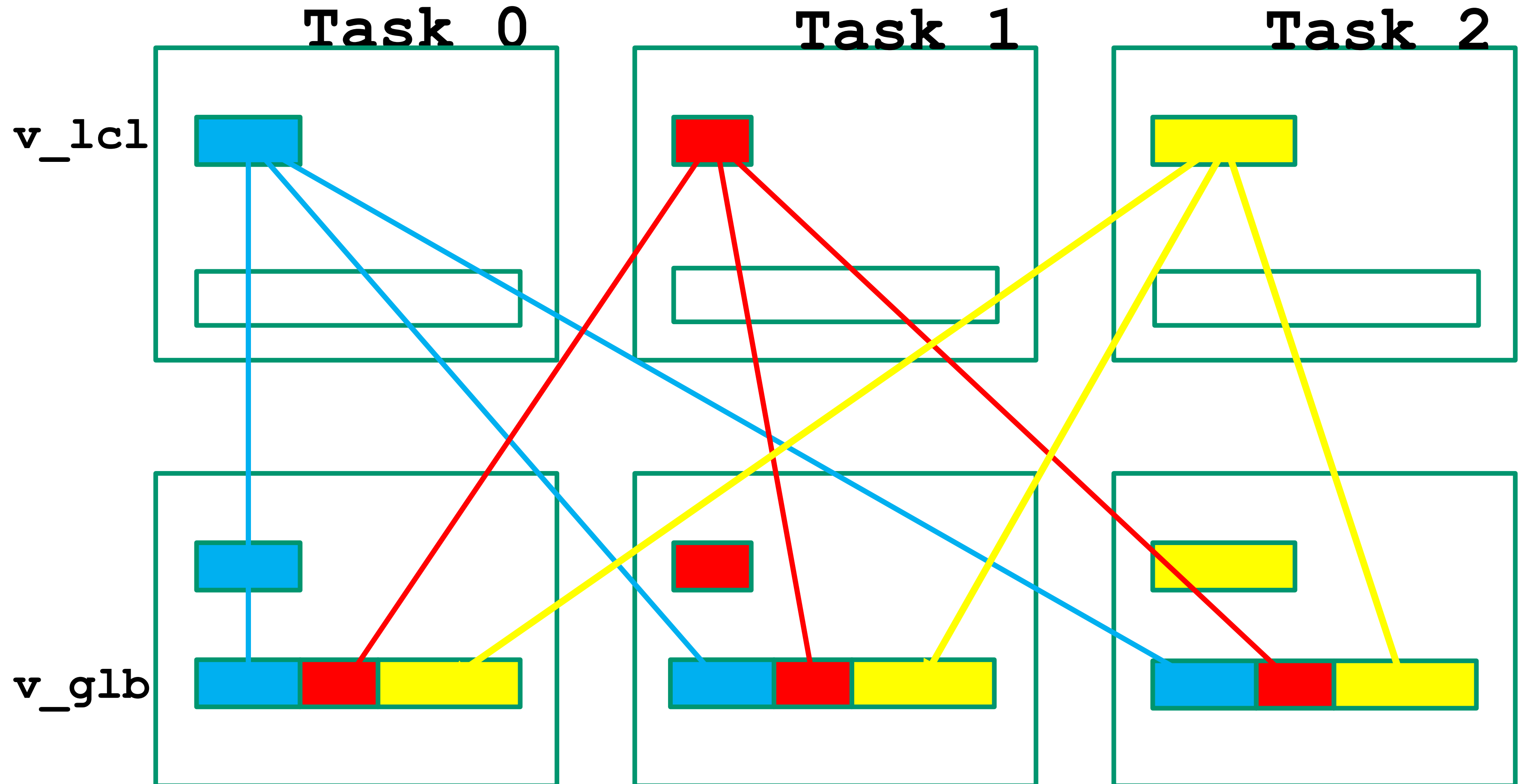
- Deactivate the new type

  MPI_TYPE_FREE(rowblock,ierr)

Solution in

~/mpikurs_solutions/*/dist_matrix_rowblock_type_ready.*

Gather Data



global_vector with MPI_BCAST

```
  call dist_index( n, firstind )
  nl = firstind(myid+1) - firstind(myid)
  ia = firstind(myid)
  do i = 1 , nl
     x(ia+i) = y(i)
  end do
  do ip = 0 , nproc-1
      nl = firstind(ip+1) - firstind(ip)
      ia = firstind(ip)+1
      call MPI_BCAST( x(ia), nl ,
            MPI_DOUBLE_PRECISION, ip,
:           MPI_COMM_WORLD, ierr )
  end do
```

Exercise: use MPI_ALLGATHERV

```
MPI_ALLGATHERV( sbuf, scount, stype
```

- ```
                , rbuf, rcounts, displs, rtype, comm, ierr )
  ```
- ```
          <type>sbuf(*), rbuf(*)
  ```
- ```
          INTEGER scount, stype, rcounts(*), dipls(*), rtype,
                  comm, ierr
  ```
-

GWDG
  ```

Exercise : Parallel Efficiency

parallel efficiency:

e(np) = speed on np processors / (np *speed on 1 processor)

the speed in units Mflop/s is displayed
in the output of the ritz-program.

For different values of n = 200, 400, 800, . . .

compare e(np) for different np

determine $np_{1/2}$: $e(np_{1/2}) < 1/2$

Diffusion Equation

$$\rho c \tfrac{\partial}{\partial t} u(t,\mathbf{x}) = k \Delta u(t,\mathbf{x}) + f(t,\mathbf{x}), \mathbf{x} \in \Omega = [0,1] \times [0,1]$$

Anfangs - u. Randwerte : $u(0,\mathbf{x})$, $u(t,\mathbf{x}), \mathbf{x} \in \partial\Omega$

Diskretisierung : $x_{1,2} \in [0,1] \Rightarrow (i_{1,2}) \in (0 : n+1)$

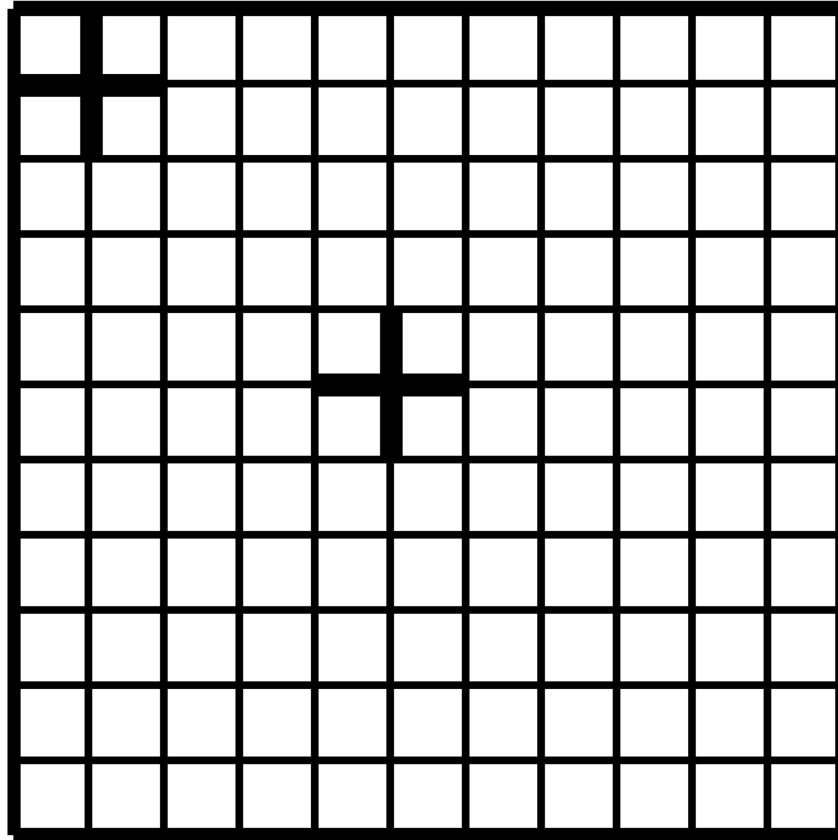
Innere Punkte : $un(i_1,i_2) = s \cdot u(i_1,i_2) +$

$$r \cdot \left(u(i_1-1,i_2) + u(i_1+1,i_2) + u(i_1,i_2-1) + u(i_1,i_2+1) \right)$$

Anfangswerte: $u0(i_1,i_2)$

Randwerte : $u(i_1,0)$, $u(i_1,n+1)$, $u(0,i_2)$, $u(n+1,i_2)$

Finite-Difference Grid



Algorithmus Wärmeleitung

Source code in directory **`Uebungen_*/Waermeleitung`**

input :

```
r,nt,n
```

initialize :

```
u(1:n,1:n)                              initial values

u(0,0:n+1)  u(n+1,0:n+1)                boundary values
u(0:n+1,0)  u(0:n+1,n+1)
```

loop:

```
u(1:n,1:n)  →  un(1:n,1:n)              update temperature
un(1:n,1:n) →  u(1:n,1:n)
```

Update Temperature

Source code in directory **`Uebungen_*/Waermeleitung`**

```
subroutine zeitschritt(r,n1,n2,a,u)
 real*8 a(0:n1+1,0:n2+1), u(0:n1+1,0:n2+1)
 s = 1. - 4.*r
 do j2 = j2a , j2e   ,  do j1 = 1 , n1
  u(j1,j2) = s*a(j1,j2)
    + r*( a(j1-1,j2) + a(j1,j2-1)
    +     a(j1,j2+1) + a(j1+1,j2) )
```

calculates new temperature in rows **`1`** to **`n2`** in **`u`**,
using old values from rows **`0`** to **`n2+1`** from **`a`**

Program Execution Wärmeleitung

Source code in directory **`Uebungen_*/Waermeleitung`**

```
> make waermeleitung
```

Input data in file **`wl.inp`**:

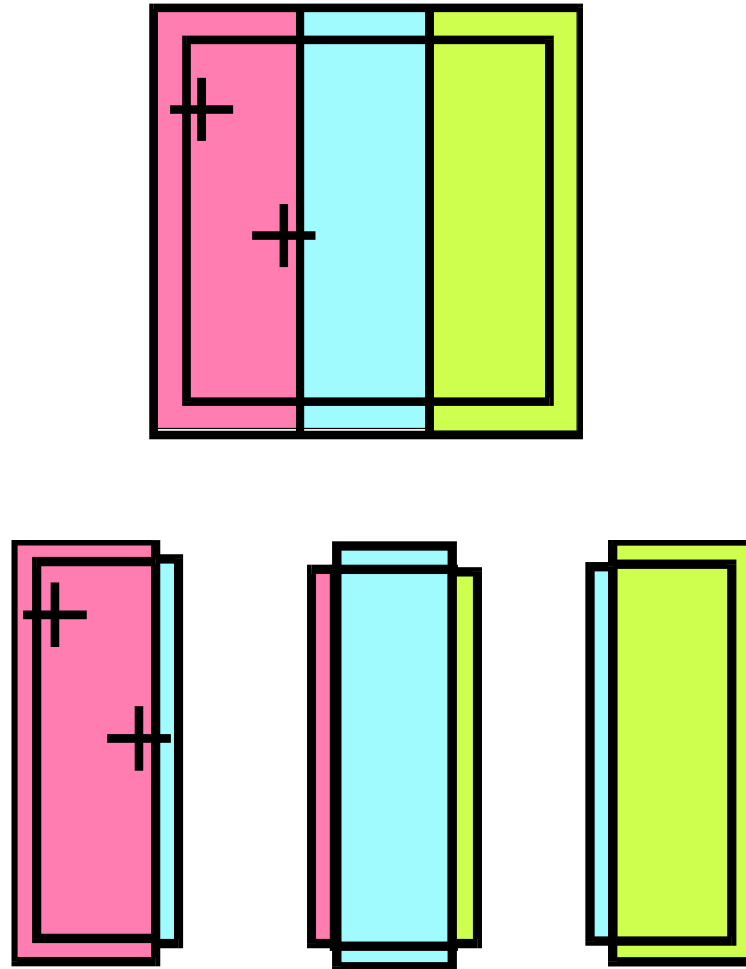
```
500           read(5,*) nt
499           read(5,*) n
0.2           read(5,*) r
```

```
> mpirun -n 1 waermeleitung.exe <wl.inp
```

Python:

```
> mpirun -n 1 python waermeleitung.py <wl.inp
```

Domain Decompositioin with Boundary Exchange



Algorithmus Wärmeleitung-parallel

Source code in directory **`Uebungen_*/Waermeleitung`**

input:

```
r,nt,n
u(1:n,1:n)   initial values
u(0,0:n+1)   boundary values
```

loop:

```
randaustausch( n, nl, u )
zeitschritt( r, n, nl, u, un )
randaustausch( n, nl, un )
zeitschritt( r, n, nl, un, u )
```

Exchange Boundary Values

With MPI_SENDRECV:

Every process sends its 1st row to the left,
receives values for its 0th row from the left.

Every process sends its $n2l^{th}$ row to the right,
receives values for its $(n2l+1)^{th}$ row from the right.

Global grid borderlines are exchanged with process MPI_PROC_NULL!

```
subroutine randaustausch
```

GWDG

Fortran Implementation

```
subroutine randaustausch ( n1, n2, a )
      ...
ipl = myid - 1
if (myid.eq.0) ipl = MPI_PROC_NULL
ipr = myid + 1
if (myid.eq.nproc-1) ipr = MPI_PROC_NULL

call MPI_SENDRECV(a(1,1), n1, MPI_DOUBLE_PRECISION,ipl,0,
                  a(1,0), n1, MPI_DOUBLE_PRECISION,ipl,0,
                  com, istat, ierr)
call MPI_SENDRECV(a(1,n2),n1,MPI_DOUBLE_PRECISION,ipr,0,
                  a(1,n2+1),n1,MPI_DOUBLE_PRECISION,ipr,0,
     :            com, istat, ierr)
```

Parallel Program Execution Wärmeleitung

Source code in directory **`Uebungen_*/Waermeleitung`**

Fortran, C:

```
> make waermeleitung_mpi
```

Input data in file **`wl.inp`**:

```
500            read(5,*) nt
500            read(5,*) n
0.2            read(5,*) r
```

```
> mpirun -n 4 waermeleitung_mpi.exe <wl.inp
```

Python:

```
> mpirun -n 4 python waermeleitung_mpi.py <wl.inp
```

Scaling Analysis

Number of operations: $(6n^2)/np$

Number of words to transfer: $2n$

Execution time and speed:

$$t = t_{op} + t_{com} = \frac{6n^2}{np} r^{-1} + 2(t_{lat} + nc^{-1})$$

$$r_{np} = r \cdot np \frac{1}{1 + \frac{1}{3}\left(\frac{np}{n^2} \cdot r t_{lat} + \frac{np}{n} \cdot \frac{r}{c}\right)}$$

Scaling with fixed number of grid points per process:

$$n^2 = \alpha \cdot np, \qquad r_{np} \to 3\sqrt{\alpha} \cdot r \cdot \sqrt{np}$$

GWDG

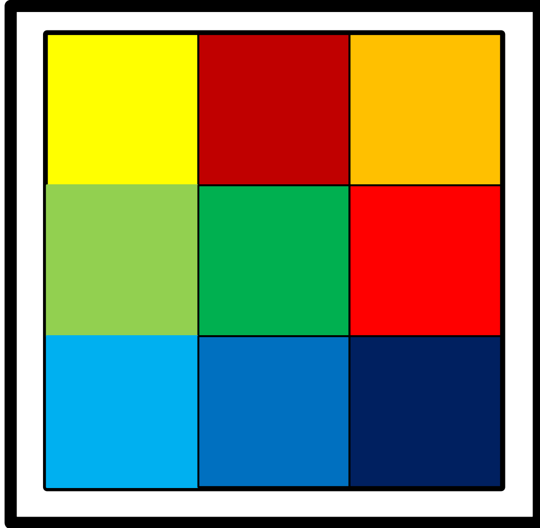
Exercise : Parallel Efficiency

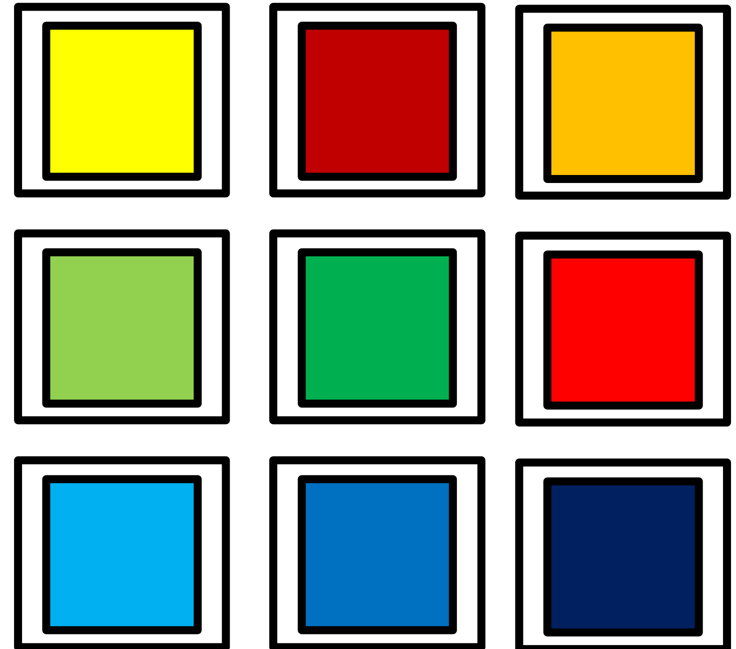
determine t_{op} and t_{com} for np = 2, 4, 8, 16

$$n^2 = 50^2 \cdot np$$

t_{op} and t_{com} are displayed as tupd and trand in the output of the program waermeleitung

2-dimensional Domain Decomposition





2-dimensional Domain Decomposition

Number of processes: $np = nq^2$

Number of words to transfer: $4\frac{n}{nq} = 4\sqrt{n^2/np}$

Execution speed:

$$r_{np} = r \cdot np \frac{1}{1 + \frac{2}{3}\left(\frac{np}{n^2} \cdot rt_{lat} + \frac{nq}{n} \cdot \frac{r}{c}\right)}$$

Scaling with fixed number of grid points per process:

$n^2 = \alpha \cdot np,$ $\quad r_{np} \rightarrow 1.5\sqrt{\alpha} \cdot r \cdot np$

Heat Equation with 2-dim. Block Distribution

Modification

Input: size of 2-dim process-grid: **`nq1,nq2`**

`nq1*nq2 <= nproc ?`

Map **`myid->(myid1,myid2)`**

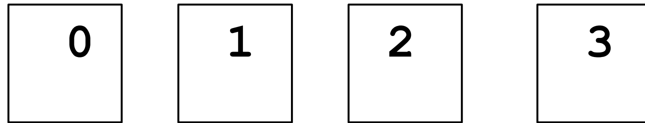
Caculate blocksizes **`n11,n21`** for all blocks

Initialize local blocks

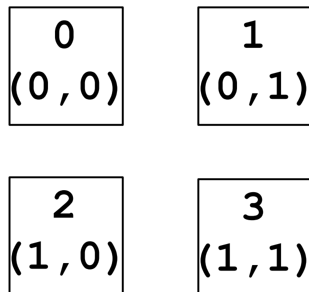
Boundary exchange for 2-dim distribution

Virtual Topology

Flat set of 4 processes. flat_pid = 0, … , 3



2-dim grid of 4 processes: 2-dim_pid = (0,0), … , (1,1)



`np = np1*np2`

map linear numbering

`ip = 0,..., np-1`

to 2-dim numbering

`(ip1,ip2) = (0,0),...,(np1-1,np2-1)`

MPI_CART_CREATE

generates cartesian grid of any dimension

```
MPI_CART_CREATE(comm_old, ndim, dims,
                periods, reorder, comm_cart)
```

comm_old	input communicator (**MPI_COMM_WORLD**)
ndim	number of dimension
dims	integer array of sizes of each dimension
periods	logical array specifying the property of each dimension: periodic (**true**) or not periodic(**false**)
reorder	logical: ranks may be reorderd (**true**) or not (**false**)
comm_cart	new communicator with cartesian topology

MPI_CART_GET

returns the cartesian grid-coordinates of the calling process

MPI_CART_GET(comm_cart, ndim, dims, periods, coords)

comm_cart	input communicator (**MPI_COMM_WORLD**)
ndim	number of dimension
dims	integer array of sizes of each dimension
periods	logical array specifying the property of each dimenension
coords	coordinates of the calling process

GWDG

Example code for creating 2-dim grid

```
integer comm, comm_2d, nproc, myid, ierr
integer dims(2), np1, np2, myid_2d(2)
logical periods(2), reorder
com = MPI_COMM_WORLD
call MPI_INIT( ierr )
call MPI_COMM_SIZE( com, nproc, ierr )
call MPI_COMM_RANK( com, myid, ierr )
dims(1) = np1 ; dims(2)= np2; periods = .false.
call MPI_CART_CREATE(com,2,dims,periods,.true.,com_2d,ierr)
call MPI_CART_GET(com_2d,2,dims,periods,myid_2d,ierr)
write(6,*)myid,myid_2d(1),myid_2d(2)
```

GWDG

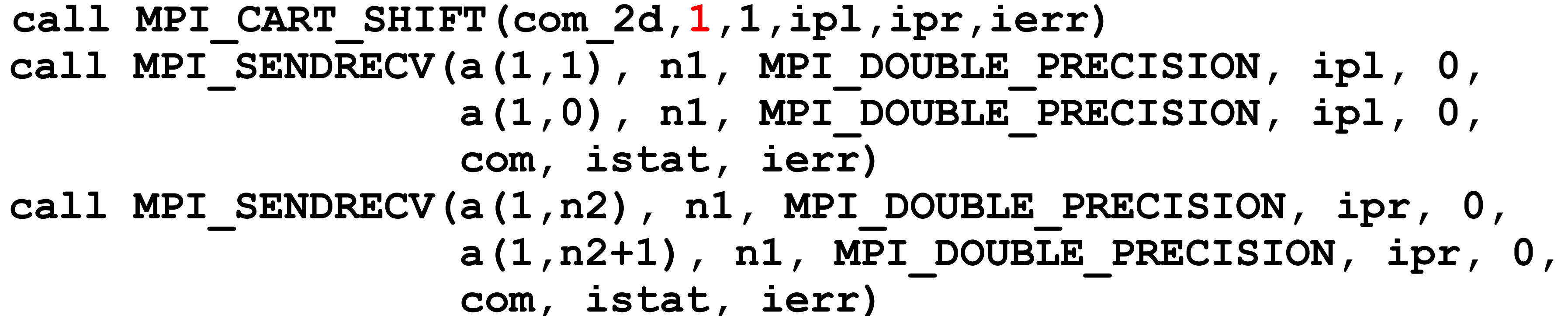
Border Exchange in 2-dim Topology

```
subroutine randaustausch( com_2d, n1, n2, a )
     implicit none
     include 'mpif.h'
     integer    com_2d, n1, n2
     real*8     a(0:n1+1,0:n2+1), as(1000), ar(1000)
     integer    i, ipu, ipd, ipl, ipr, com, nproc, myid, ierr

     com = MPI_COMM_WORLD
     call MPI_COMM_SIZE( com, nproc, ierr )
     call MPI_COMM_RANK( com, myid, ierr )
```

Border Exchange in 2-dim Topology

exchange of vertical boundaries



```
 call MPI_CART_SHIFT(com_2d,1,1,ipl,ipr,ierr)
 call MPI_SENDRECV(a(1,1), n1, MPI_DOUBLE_PRECISION, ipl, 0,
:                  a(1,0), n1, MPI_DOUBLE_PRECISION, ipl, 0,
:                  com, istat, ierr)
 call MPI_SENDRECV(a(1,n2), n1, MPI_DOUBLE_PRECISION, ipr, 0,
:                  a(1,n2+1), n1, MPI_DOUBLE_PRECISION, ipr, 0,
:                  com, istat, ierr)
```

```
MPI_CART_SHIFT(com_2d, dir, disp, rank_source, rank_dest)
  dir           direction (0,1,...,dims-1)
  disp          displacement
  rank_source   rank of source process
  rank_dest     rank of destination process
```

Border Exchange in 2-dim Topology

exchange of horizontal boundaries

```
 call MPI_CART_SHIFT(com_2d,0,1,ipu,ipd,ierr)
 as(1:n2) = a(1,1:n2)
 call MPI_SENDRECV(as, n2, MPI_DOUBLE_PRECISION, ipu, 0,
:                  ar, n2, MPI_DOUBLE_PRECISION, ipu, 0,
:                  com, istat, ierr)
 if (ipu.ge.0) then
   a(0,1:n2) = ar(1:n2)
 end if

 as(1:n2) = a(n1,1:n2)
 call MPI_SENDRECV(as, n2, MPI_DOUBLE_PRECISION, ipd, 0,
:                  ar, n2, MPI_DOUBLE_PRECISION, ipd, 0,
:                  com, istat, ierr)
 if (ipd.ge.0) then
   a(n1+1,1:n2) = ar(1:n2)
 end if
```

Complete code in directory **Uebungen_f/WL_2d**

GWDG

Border Exchange in 2-dim Topology

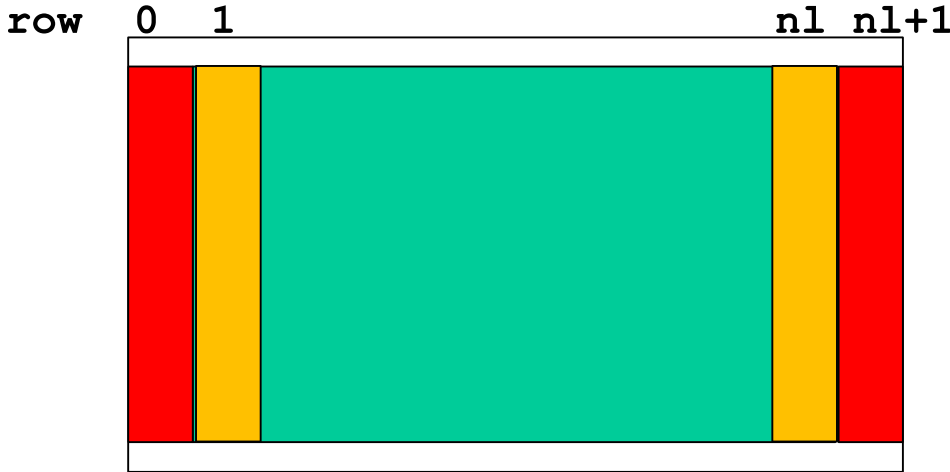
using derived datatype for row exchange

```
 call MPI_TYPE_VECTOR(n2, 1,  n1+2, MPI_DOUBLE_PRECISION,
:                                  row, ierr)
 call MPI_TYPE_COMMIT(row,ierr)

 call MPI_SENDRECV(a(1,1), 1, row, ipu, 0,
:                  a(0,1), 1, row, ipu, 0,
:                  com, istat, ierr)
 call MPI_SENDRECV(a(n1,1), 1, row, ipd, 0,
:                  a(n1+1,1), 1, row, ipd, 0,
:                  com, istat, ierr)
 call MPI_Type_free(row,ierr)
```

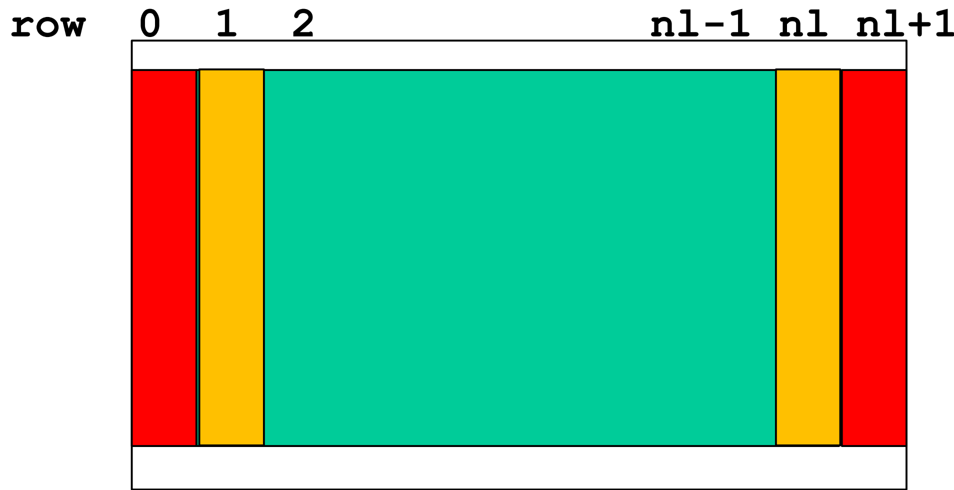
Complete code in file **`Uebungen_f/WL_2d/randaustausch_type.f`**

Communication and Computation



communication of old values:

rows 1,…,nl locally available
rows 0, nl+1 received from neighbour processes



computation of new values:

rows 2,…,nl-1 determined from locally available old data
rows 1, nl use data from neighbour processes

Overlapping Computation and Communication using Non-Blocking Communication

On each process:

Start two non-blocking sends :

to send 1st row to left and nlth row to right neigbour

Start two non-blocking receive calls

to receive 0th row from left and (nl+1)th from right neigbour

(this generates 4 request handels)

update temperature for rows 2,..,nl-1

wait for completion of communication calls

update temperature for rows 1, nl

Overlapping Computation and Communication using Non-Blocking Communication

New routine for boundary exchange:

```
subroutine iexchang ( n1, n2, a, req )
     implicit none
     include 'mpif.h'
     integer              n1, n2, req(*)
     real*8               a(0:n1+1,0:n2+1),
     integer              ipr, ipl, com, nproc, myid, ier
     com = MPI_COMM_WORLD
     call MPI_COMM_SIZE( com, nproc, ierr )
     call MPI_COMM_RANK( com, myid, ierr )
     ipl = myid - 1; if (myid.eq.0) ipl = MPI_PROC_NULL
     ipr = myid + 1; if (myid.eq.nproc-1) ipr = MPI_PROC_NULL
     call MPI_ISEND(a(1,1), n1, MPI_DOUBLE_PRECISION, ipl, 0,com, req(1), ierr )
     call MPI_ISEND(a(1,n2), n1, MPI_DOUBLE_PRECISION, ipr, 0,com, req(2), ierr )
     call MPI_IRECV(a(1,0), n1, MPI_DOUBLE_PRECISION, ipl, 0,com, req(3), ierr )
     call MPI_IRECV(a(1,n2+1), n1, MPI_DOUBLE_PRECISION, ipr, 0,com, req(4), ierr )
     return
     end
```

GWDG

Overlapping Computation and Communication using Non-Blocking Communication

New routine for timestep:

```
subroutine timestep ( r, n1, n2, a, u, j2a, j2e )
     implicit none
     integer                 n1, n2, j2a, j2e
     real*8                  a(0:n1+1,0:n2+1), u(0:n1+1,0:n2+1)
     real*8                  r
     integer                 j1, j2
     real*8                  s
     s = 1. - 4.*r
     do j2 = j2a , j2e
        do j1 = 1 , n1
           u(j1,j2) = s*  a(j1,j2) + r * (
    :                                 a(j1-1,j2) +
    :            a(j1,j2-1)  +                         a(j1,j2+1) +
    :                                 a(j1+1,j2) )
        end do
     end do
     return
     end
```

Overlapping Computation and Communication using Non-Blocking Communication

Modified iteration loop for temperature update:

```
integer req(4)
do i = 1,nt
    call iexchange(n,nl,a,req)
    call timestep(r,n,nl,a,u,2,nl-1)
    call MPI_WAITALL(4,req,MPI_STATUSES_IGNORE,ierr)
    call timestep(r,n,nl,a,u,1,1)
    call timestep(r,n,nl,a,u,nl,nl)
                ...
         exchange a and u
                ...
end do
```

The call to MPI_WAITALL returns, when all communication steps, to which the 4 request-handels refer, are completed.

code in **heat_mpi.f, timestep.f, iexchange.f**