GWDG – Kurs
Parallel Programming with MPI

MPI Applications

Oswald Haan
ohaan@gwdg.de

Applications

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

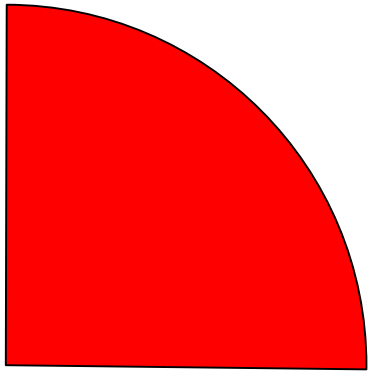
Approximate Calculation of π by Numerical Integration

Learning Objectives:

- MPI Parallelization of Sequential Code
- Farmer-Worker Model for Parallelization
- Second Farmer Thread as Worker

Calculating π

Aerea of a quarter circle = π / 4

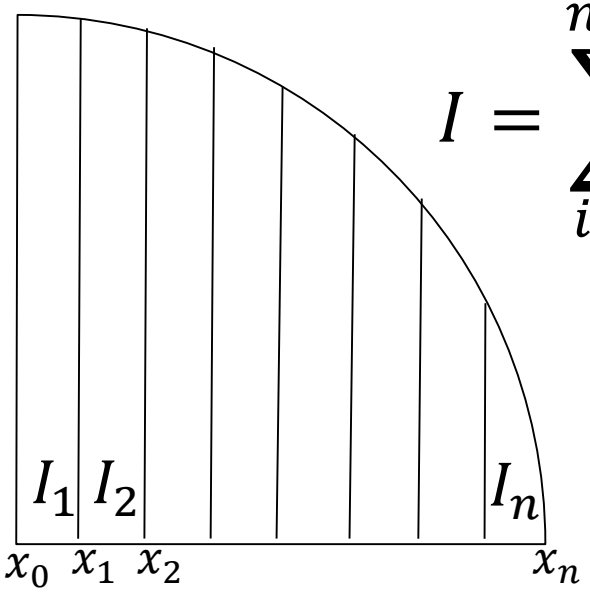


$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx$$

Numerical Integration

$$I = \int_l^h f(x)dx$$

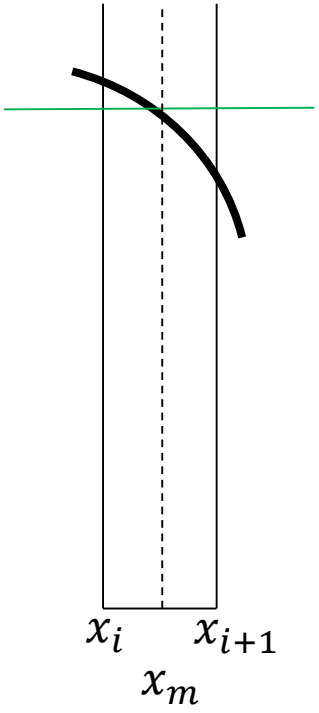
divide integration domain $[l, h]$ into n strips of width $\Delta = (h-l)/n$



$$I = \sum_{i=0}^{n-1} I_i \qquad I_i = \int_{x_i}^{x_{i+1}} f(x)\,dx$$

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

Approximation: I = A + R



I_i exact area of a strip

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

$$x_m = \frac{1}{2}(x_i + x_{i+1})$$

$$I = \sum_{i=0}^{n-1} I_i$$

$$A = \sum_{i=0}^{n-1} I_i^{(m)} = \Delta \sum_{i=0}^{n-1} f(x_m)$$

Fortran Implementation of Numerical Integration

```
double precision function f(x)
implicit none
double precision x
f = sqrt(1.d0-x*x)
return
end

double precision function numint(lo, delt, n, pia)
implicit none
real*8 f, lo, delt, pia, res, del, xi, su
integer i, n

su = 0.0
do i = 1 , n
   xi = lo + (dble(i)-0.5d0)*delt
   su = su + f(xi)
end do
numint = pia+4.*delt*su
return
end
```

The function **`numint(lo,delt,n,0.)`** returns the approximation for the integral $\int_{lo}^{lo+delt*n} f(x)\,dx$, using **n** strips

Python Implementation of Numerical Integration

```
import math
def f(x):
    return math.sqrt(1.-x*x)

def numint(lo,delt,n,pia):
    su = 0.0
    for i in range(0,n):
        xi = lo + (i+0.5)*delt
        su = su +  f(xi)
    return (pia + 4.*delt*su)
```

The function **`numint(lo,delt,n,0.)`** returns the approximation for the integral $\int_{lo}^{lo+delt*n} f(x)\,dx$, using **n** strips

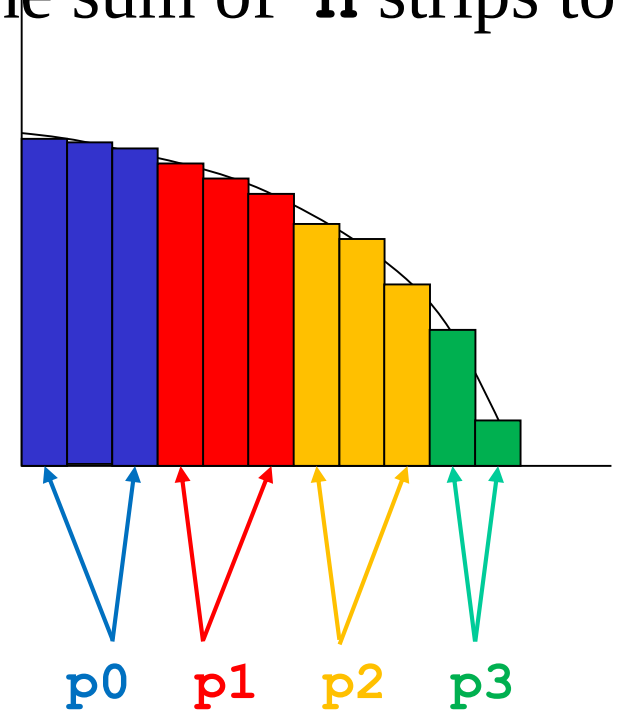
Approximate Calculation of π

```
      program piapp
   ...
!   input of n
   ...
!   numerical approximation
      lo= 0.0; hi = 1.0; delt = (hi-lo)/dble(n); pia = 0.0
      pia = numint(lo,delt,n,pia)
!   output of result
    ...
```

piapp.f , numint.f in directory **mpiexercises/f/pi**
piapp.c , numint.c in directory **mpiexercises/c/pi**
piapp.py, numint.py in directory **mpiexercisespy/pi**

Exercise 1

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



Steps for Solving Exercise 1

`program piapp_mpi`

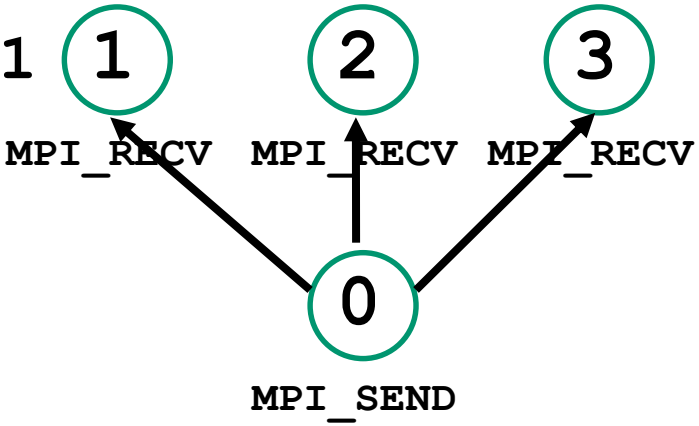
1) set up MPI environment **`np, me`**
2) for **`me=0`** : read the value for **`n`** from standard input,
3) distribute **`n`** to all tasks
4) Distribute **`pia=numint (0.,1./n,n,0.0)`** to **`np`** processes:
   determine **`nl, lo, hi`** for process **`me`**
   `pial = 0.0; pial=numint (lo,1./n,nl,pial)`
5) add up all local sums **`pial`** 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

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: distribute n strips to np processes :

define the smallest integer greater than n/np:
`nla` = integer part of **`(n+np-1)/np`**

the local number **`nl`** of strips for process **`me`**:

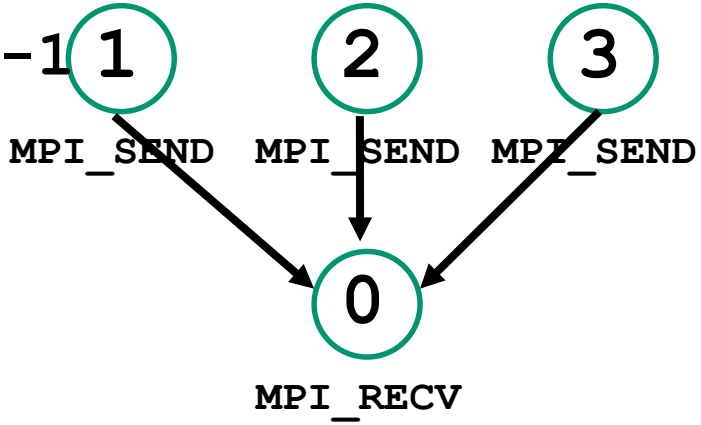
```
nl= min(nla,n-me*nla)
```

the width of strips for all processes remains **`delt = 1.d0/n`**
the integration boundaries **`lo, hi`** for process **`me`**:

```
lo = me*delt*nl   ,     hi = lo + delt*nl
```

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

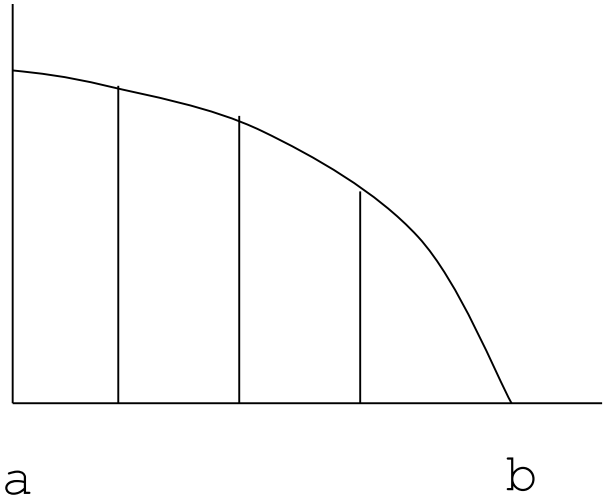
1 SEND of pial in tasks 1,…,np-1



np-1 RECV's of pial in task 0
adding up pial to pia

Solution to this exercise in
~ohaan/mpisolutions/[f,c,py]/piapp_mpi_r[f,c,py]

Numerical Approximation with Higher Precision

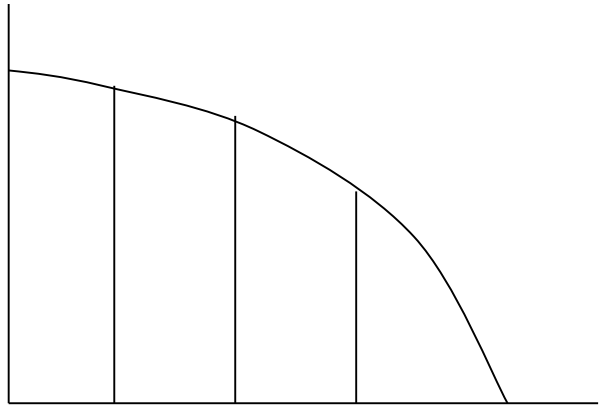


$$\int_a^b f(x)dx = \sum_{i=0}^{nin-1} \int_{a_i}^{b_i} f(x)dx$$

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

divide integration region [**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 $\mathbf{n_i}$ strips.

Use more strips if function is steeper

-> same precision in each interval

$\mathbf{n_i = n\ (f(a_i)-f(b_i))}$

$\mathbf{n_1 + n_2 + \ldots = n}$

Fortran Implementation

```
program piprec
      ...
width = 1.d0/dble(ndom)
pia = 0.0d0
do i = 1 , ndom
   hi = i * width
   lo = hi - width
   nl = n * (f(lo) - f(hi))+1
   delt = 1.d0/nl
   pia = numapp(lo,delt,nl,pia)
end do
```

Python Implementation

```
width = 1./ndom
pia = 0.
for i in range(ndom) :
    lo = width*i
    hi = lo + width
    nl = int(n*(f(lo)-f(hi)))+1
    delt = (hi-lo)/nl
    pia = numint(lo,delt,nl,pia)
```

Example Code

piprec.f and **numint.f**	in directory **mpiexercises/f//pi**
piprec.c and **numint.c**	in directory **mpiexercises/c/pi**
piprec.py and **numint.py**	in directory **mpiexercises/py/pi**

C and Fortran

Compile and link with makefile

```
> make piprec
```

Run with

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

Python

Run with

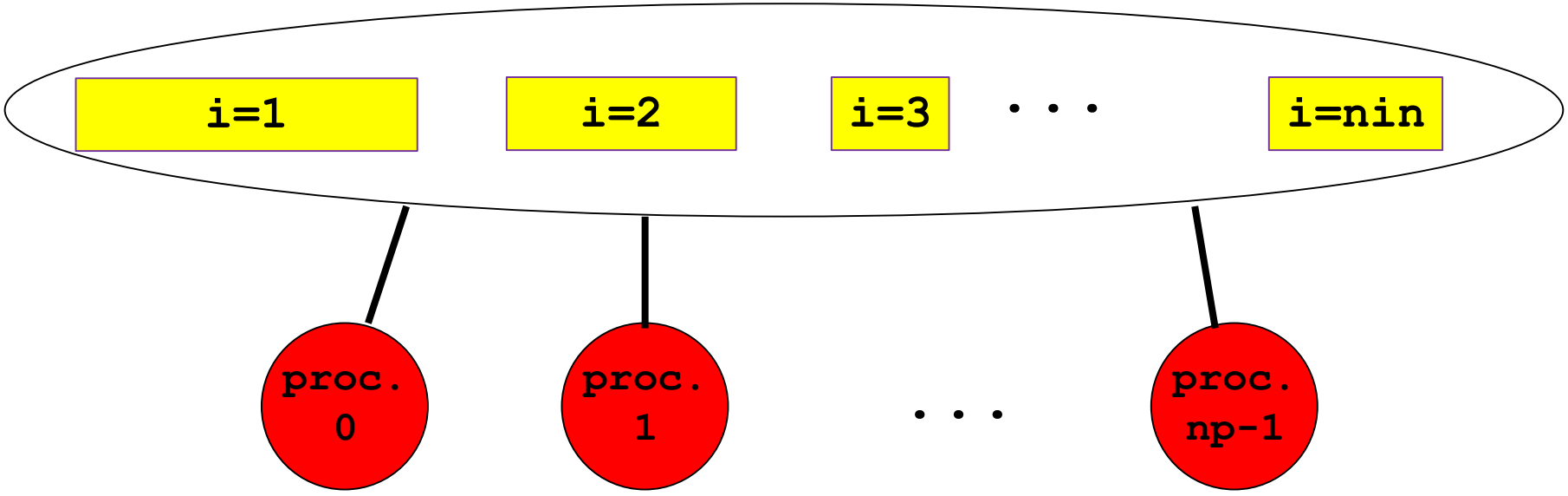
```
> python piprec.py
```

```
number of strips     :              100000000
number of intervals :                      1
domain  1  with 100000001  strips in 0.2827E+00  s

pi estimated           =     3.14159265359050
error                  =           0.710E-12
time :  0.2828E+00 s , rate :  353.58 [Mstrips/s]
```

```
number of strips     :              100000000
number of intervals :                      6
domain  1  with   1398671  strips in 0.3933E-02  s
domain  2  with   4320426  strips in 0.1216E-01  s
domain  3  with   7678364  strips in 0.2161E-01  s
domain  4  with  12066942  strips in 0.3395E-01  s
domain  5  with  19258520  strips in 0.5418E-01  s
domain  6  with  55277080  strips in 0.1555E+00  s

pi estimated           =     3.14159265358988
error                  =           0.839E-13
time :  0.2815E+00 s , rate :  355.26 [Mstrips/s]
```

Exercise 2

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



Problem: unequal workload for different domains
workload not known at compile time

Farmer Distributes Work to Idle Workers

```
farmer:  me == ipfa

pia = 0
for iw = 1 to nin
    recv  ipw from anytask
    send iw to ipw
for ipw = 0 to np-1
    if ipw = ipfa : skip
    send (nin+1) to ipw
    recv  resl from ipw
    add resl to pia
```

```
worker: me != ipfa

resl = 0
for i = 1 to nin+1
    send me to farmer
    recv iw from farmer
    if iw >  nin : exit
    resl = work(iw)
    add resl to res
send resl to farmer
```

Python Implementation for farmer

```
from numint import numint, f
def farmer(idle,ndom):
    comm = MPI.COMM_WORLD
    nproc= comm.Get_size()
    ipfa = comm.Get_rank()
    tag_fa = 0; tag_wo = 1; tag_cl = 2; pia = 0.
    for iw in range( 1,ndom+1):
        ipw = comm.recv(source=MPI.ANY_SOURCE, tag=tag_wo)
        comm.send(iw, dest=ipw, tag=tag_fa)
    for ipw in range( nproc):
        if (ipw == ipfa) & (idle == 0) :
            continue
        comm.send(ndom+1, dest=ipw, tag=tag_fa)
        pial = comm.recv(source=ipw, tag=tag_cl)
        pia = pia + pial
    return (pia)
```

Python Implementation for worker

```
from numint import numint, f
def worker(ipfa,n,ndom):
    comm = MPI.COMM_WORLD
    ipwo = comm.Get_rank()
    tag_fa = 0; tag_wo = 1; tag_cl = 2; pia = 0.
    width = 1./ndom
    comm.send(ipwo, dest= ipfa,tag=tag_wo)
    for i in range( 1,ndom+2):
        iw = comm.recv(source=ipfa, tag=tag_fa)
        if iw > ndom :
            break
        lo = 1.0 - iw*width; hi = lo + width
        nl = int(n * (f(lo) - f(hi)))+1
        delt =(hi-lo)/nl
        pia = numint(lo, delt, nl, pia)
        comm.send(ipwo, dest=ipfa, tag=tag_wo)
    comm.send(pia, dest=ipfa, tag=tag_cl)
    return
```

Python Implementation for main program

```
from mpi4py import MPI
from agents import worker, farmer

# ========== main program for farmer-worker calculation
comm = MPI.COMM_WORLD
nproc= comm.Get_size(); me = comm.Get_rank()
ipfa = 0; idle = 0; pia = 0.; n=0; ndom=0
# input of n, ndom
   ...
# ============  Worker-Teil (me!=ipfa) ================
if me != ipfa:
    worker(ipfa,n,ndom)
# ============  Farmer-Teil (me=ipfa) ================
if me == ipfa:
    pia = farmer(idle,ndom)
# output of results
    ...
```

Example Code

piprec_mpi.f , **agents.f**, **numint.f** in directory **mpiexercises/f/pi**
piprec_mpi.c, **agents.c**, **numint.c** in directory **mpiexercises/c/pi**
piprec_mpi.py, **agents.py**, **numint.py** in directory **mpiexercises/py/pi**

C and Fortran

compile and link with makefile

```
> make piprec_mpi < inp_piprec
```

run with

```
> ./piprec_mpi.exe < inp_piprec
```

Python

run with

```
> mpirun -n 2 python piprec_mpi.py < inp_piprec
```

Performance with Python

```
 number of strips       : 10000000
 number of subintervals    : 26
 process 1:  26 domains in 3.963227 sec
 Naeherung fuer pi =              3.1415926536
 error                  7.727e-13
 time[s], rate[Mstrips/s] :    3.96409,    2.52e+00
```

```
 number of strips       : 10000000
 number of subintervals    : 26
 process 1:   8 domains in 1.022055 sec
 process 2:   8 domains in 1.020599 sec
 process 3:   1 domains in 1.077928 sec
 process 4:   9 domains in 1.015351 sec
Naeherung fuer pi =              3.1415926536
error                  7.727e-13
 time[s], rate[Mstrips/s] :    1.07840,    9.27e+00
```

Performance with Fortran (Intel MPI)

```
number of strips        :    100000000
number of subintervals:  :         50
  process        1 :   50   domains in        0.139 sec

pi estimated       =       3.14159265358980688 3
error              =              0.138E-13
time :   0.1403E+00 s , rate :   712.73 [Mstrips/s]
```

```
number of strips        :    100000000
number of subintervals:  :         50
  process        1 :    14   domains in  3.60E-002  sec
  process        2 :    14   domains in  3.61E-002  sec
  process        3 :    13   domains in  3.64E-002  sec
  process        4 :     9   domains in  3.66E-002  sec

pi estimated       =       3.14159265358980732 7
error              =              0.142E-13
time :   0.3683E-01 s , rate : 2714.97 [Mstrips/s]
```

Performance with Fortran (openmpi)

```
number of strips        :    100000000
number of subintervals:  :         50
  process      1 :    50   domains in    1.85    sec

pi estimated       =       3.14159265358982331 4
error              =              0.302E-13
time :   0.1853E+01 s , rate :    53.97 [Mstrips/s]
```

```
number of strips        :    100000000
number of subintervals:  :         50
  process      4 :    15   domains in   0.4630     sec
  process      3 :    14   domains in   0.4631     sec
  process      1 :     8   domains in   0.4637     sec
  process      2 :    13   domains in   0.4641     sec

pi estimated       =       3.14159265358982242 6
error              =              0.293E-13
time :   0.4643E+00 s , rate :   215.39 [Mstrips/s]
```

Compiling Python Code

Interpreted python code is slow
Compiled python code runs orders of magnitude faster

The **`numba`** package in the **`Anaconda Python`** distribution allows „just in time“ and „ahead of time“ compilation of python code to produce fast executables

See: https://numba.readthedocs.io

Compiling the **numint** Code

```
from numba.pycc import CC
from numba import njit, f8
import math

@njit(f8(f8))
def f(x):
    return math.sqrt(1.-x*x)

cc = CC('numint_module')
@cc.export('numint', 'f8(f8,f8,i4,f8)')
def numint(lo,hi,nl,pia):
# python code for numint
   ...
    return (pia + 4.*delt*su)
if __name__ == "__main__":
    cc.compile()
```

Type declaration (return value,(input values)))

just in time compiler

name of the module to be generated

ahead of time compiler

Complete code in **mpiexercises/py/pi_compiled/numint_compiled**

Compiling the **numint** Code

Load the anaconda module containing the numba package:

```
gwdu102 > module load anaconda3
gwdu102 > module list
currently Loaded Modules:
  1) anaconda3/2021.05
```

Compile python file

```
gwdu102 > python numint_compiled.py
```

Rename the generated compiled extension module

```
gwdu102 > ls *so
numint_module.cpython-38-x86_64-linux-gnu.so
gwdu102 > mv numint_module.cpython-38-x86_64-linux-gnu.so
                                                numint_module.so
```

Using the Compiled Module

The module **numint_module.numint** *can be imported into the python script* **agents.py.**
The compiled numint *function will be available as* **numint_module.numint**

```
gwdu102 > more agents.py
from mpi4py import MPI
import math
import numint_module
   ...
def worker(ipfa,n,ndom):
   ...
   pia = numint_module.numint(lo, hi, nl, pia)
   ...
```

Running `piprec_mpi` with Compiled Code

Add the python module containing mpi4py and the openmpi module:

```
gwdu102 > module add python openmpi
gwdu102 > module list
Currently Loaded Modules:
  1) anaconda3/2021.05   2) python/3.9.0   3) openmpi/4.1.1
```

Run piprec_mpi with 10000000 strips and 50 subintervals

```
gwdu102 > mpirun -n 2 python piprec_mpi.py < inp_piprec
 ...
error                  7.616e-13
 time[s], rate[Mstrips/s] :    0.02275,    4.40e+02

gwdu102 > mpirun -n 5 python piprec_mpi.py < inp_piprec
 ...
error                  7.621e-13
 time[s], rate[Mstrips/s] :    0.00750,    1.33e+03
```

Let the Farmer Work!

In the farmer task **`ipfa`***:*

start two threads using openMP, one thread runs the farmer, the other the worker

```
include 'omp_lib.h'
  ...
      idle = 1
! ---------------------------  farmer  -----------------
      if (me.eq.ipfa) then
         call OMP_SET_NUM_THREADS(2)
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(tid)
         tid = OMP_GET_THREAD_NUM()
! ---------------------------  farmer-worker  -----------
         if (tid.eq.1) then
            call worker(ipfa,n,ndom)
         end if
! ---------------------------  farmer-farmer  -----------
         if (tid.eq.0) then
            pia = farmer(idle,ndom)
         end if
!$OMP END PARALLEL
   ...
```

farmer will dispatch work to the farmer task ipfa

Complete code in **`mpiexercises/f/pi/piprec_mpi_omp.f`**

Performance with Farmer Working

```
gwdu102 > mpirun -n 2 ./piprec_mpi_omp.exe < inp_piprec

  process       0 :     24   domains in   6.99E-002   sec
  process       1 :     26   domains in   7.00E-002   sec

error                 =              0.151E-13
time :   0.6979E-01 s , rate : 1432.92 [Mstrips/s]
```

```
gwdu102 > mpirun -n 5 ./piprec_mpi_omp.exe < inp_piprec

  process       0 :     11   domains in   2.78E-002   sec
  process       1 :     12   domains in   2.80E-002   sec
  process       2 :      3   domains in   2.81E-002   sec
  process       3 :     12   domains in   2.83E-002   sec
  process       4 :     12   domains in   2.85E-002   sec

error                 =              0.151E-13
time :   0.2876E-01 s , rate : 3476.59 [Mstrips/s]
```

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/mpisolutions/[f,c,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 $\mathbf{x} = \sum_i r_i\ \mathbf{v}_i \quad \text{mit} \ \ r_0 \neq 0$

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

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

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

Algorithm Raleigh - Ritz

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

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

Schleife

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

$$\lambda \leftarrow x_1$$

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

Approximate Eigenvalue

Learning Objectives:

- Distribution of Global Matrix
- Derived Data Types
- Parallelization of Matrix-Vector Multiplication
- Use of Allreduce
- Use of Gartherv

program ritz

Source code in directory **`mpiexercise/[f,c,py]/Ritz/ritz[f,c,py]`**

generate executable:

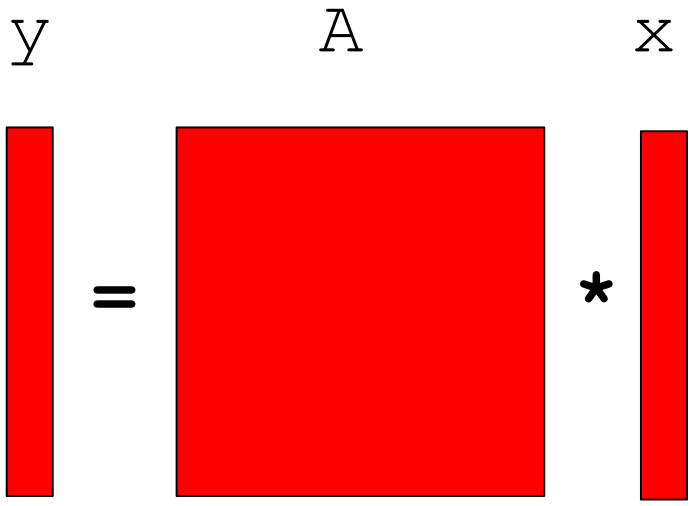
```
> mpifort -o ritz.exe ritz.f mv.f
```

or use makefile:

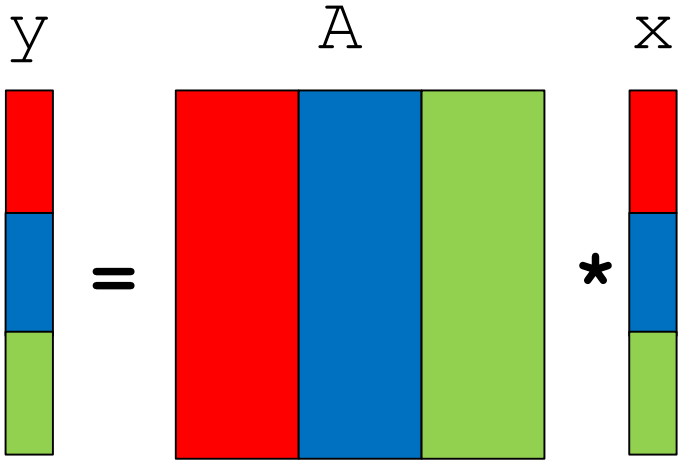
```
> make ritz
```

Matrix-Vector Multiplication

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



Parallel Matrix-Vector Multiplication
Column-block Distribution

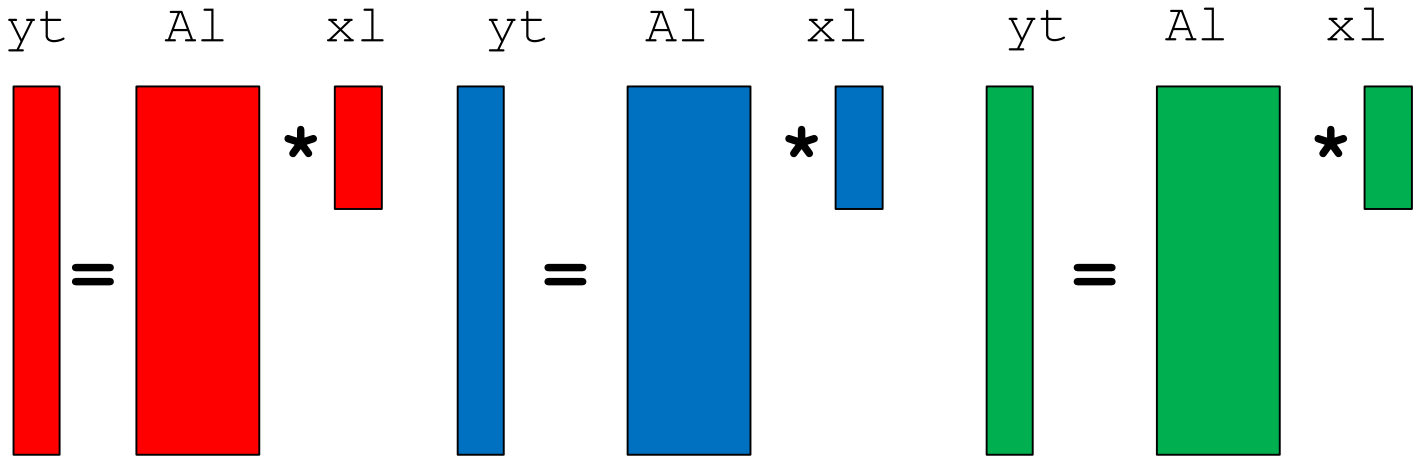


np processes ip = 0,...,np-1

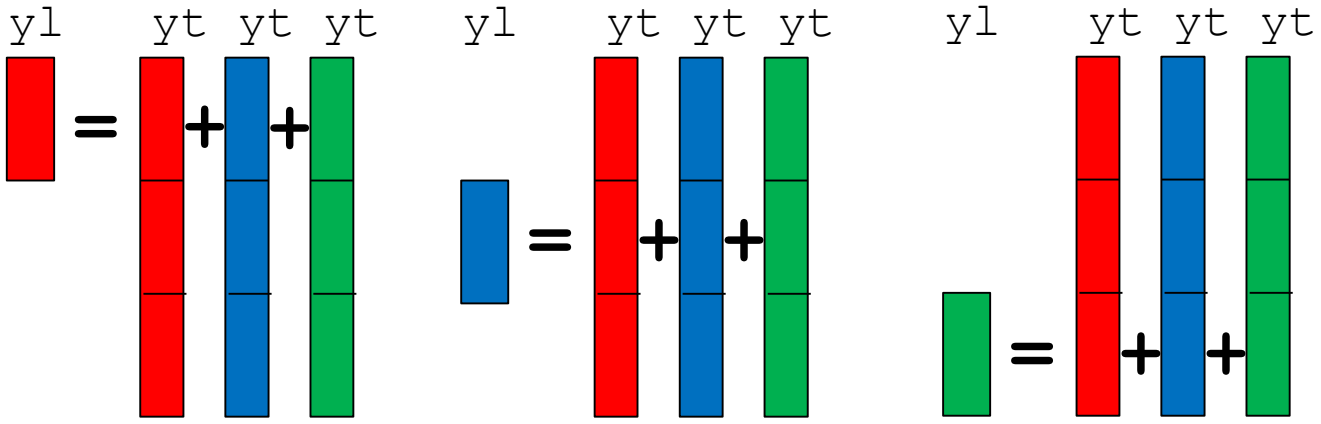
local on each process:
nloc colums of A, nloc elements of x and of y

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
Par	Distribute A to Al‘s
Par	Initialise xl
	Loop
Par	yt = Al * xl
Par	global sum yl
Ser	λ = yl(1)
Par	distribute λ
Par	xl = 1/λ * yl

dist_index
dist_matrix_colblock

DGEMV
reduce_vector

MPI_BCAST

Python Code for `ritz_dist_col`

```
from mpi4py import MPI
import math
import time
import numpy as np
from mv_matmul import dgemv
from dist_index import dist_index
from dist_matrix_colblock import dist_matrix_colblock
from reduce_vector import reduce_vector
                                ...
```

Python Code for `ritz_dist_col`

```
# initializate matrix on root
a = np.zeros((n*n))
for i in np.arange(n):
    for j in np.arange(i,n):
        a[i+n*j] = np.float64(i+j+2)/np.float64(j+1)
    for j in np.arange(i):
        a[i+n*j] = a[j+n*i]
# determine number nl of columns in local column block
firstind = dist_index(n)
nl = firstind[myid+1] - firstind[myid]
# distribute global matrix a to local column blocks al
al = dist_matrix_colblock(n,n,a)
# initialise start vector
x = np.ones(nl)
```

...

Python Code for `ritz_dist_col`

```
# iteration
evo = 1.0
evn = 0.0
for i in np.arange(nloop):
    y = dgemv(n,nl,al,x)
    x = reduce_vector(n,y)
    if myid == 0:
        evn = x[0]
    evn = comm.bcast(evn,root=0)
    res = (evn-evo)/evn
    evo = evn
    if abs(res) < eps:
        break
    x = x/evn
# output of result
print ('eigenvalue after',i,'iterations : ',evn)
```

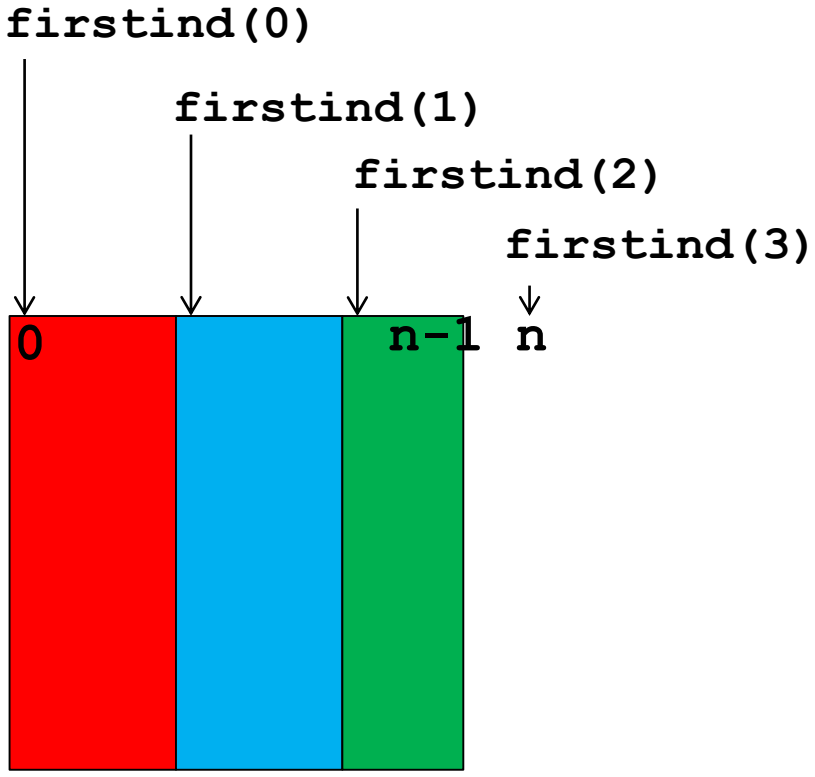
ritz_dist_col

Generate executable

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

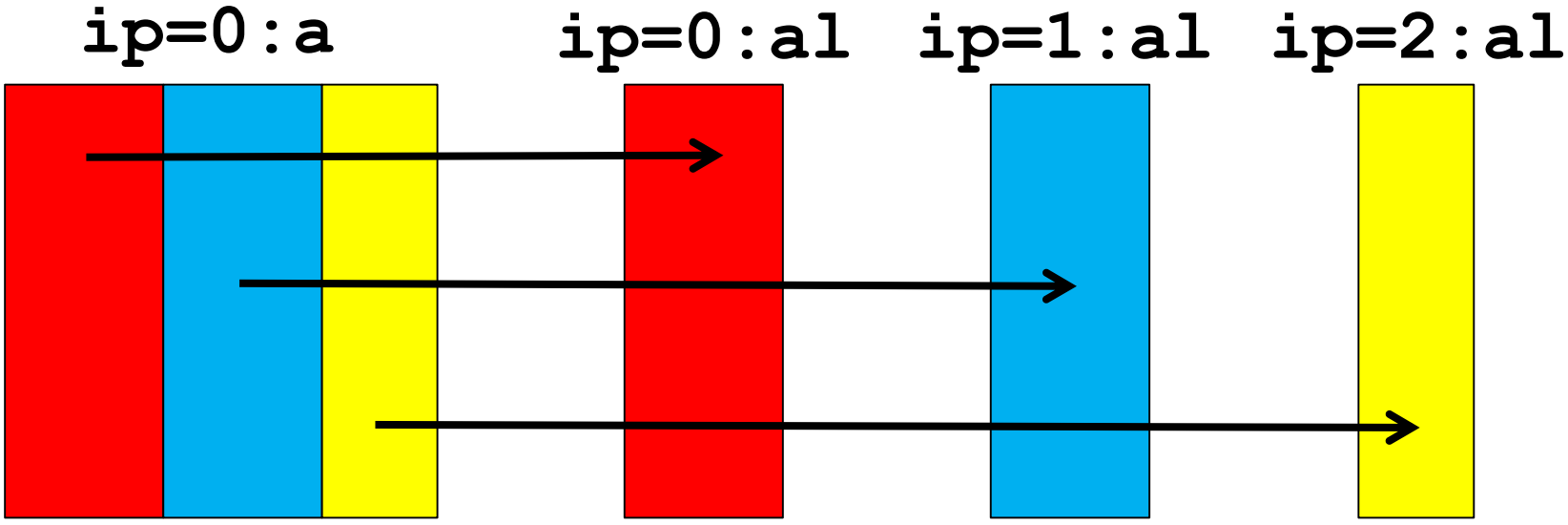
```
> make ritz_dist_col
```

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)



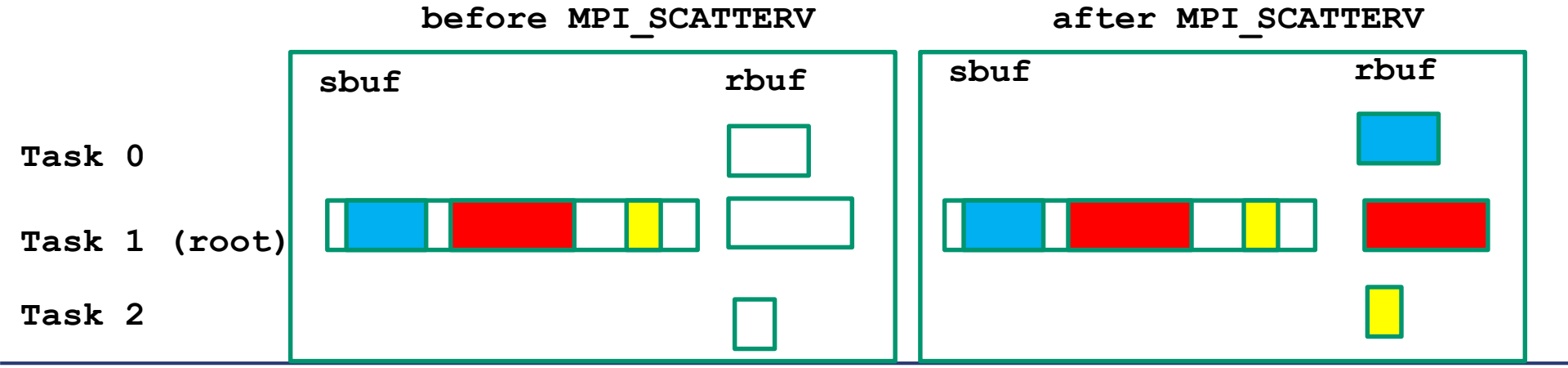
dist_matrix_colblock(m,n,a,al)

```
 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,stype]
```



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 )
```

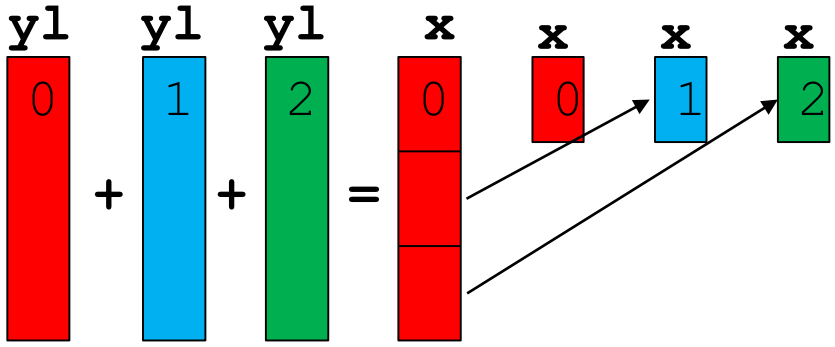
Python: distribute global matrix with dist_matrix_colblock

```
def dist_matrix_colblock( n, nl, a ):
   al = np.empty((n*nl))
   if myid == 0:
      firstind = dist_index(n)
      for ip in range(nproc):
         ia = firstind[ip]
         ie = firstind[ip+1]
         nl = ie -ia
         if ip == 0:
            al = a[ia*n:ie*n]
         else:
            comm.Send(a[ia*n:ie*n], dest= ip)
   else:
      comm.Recv(al, source=0)
   return al
```

Python: distribute global matrix with Scatterv

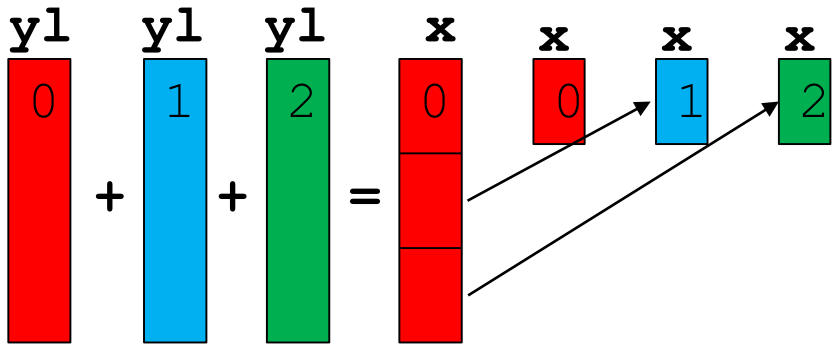
```
#al = dist_matrix_colblock(n,nl,a)

al=np.zeros((n*nl))
counts=np.zeros(nproc)
for i in np.arange(nproc):
   counts[i] = n*(firstind[i+1] - firstind[i])
dspls=np.zeros(nproc)
for i in np.arange(nproc):
   dspls[i] = n*firstind[i]
comm.Scatterv([a,counts,dspls,MPI.DOUBLE],al,root=0)
```

Python: reduce_vector(n,y)




```
comm.Reduce_scatter(y, x, recvcounts=None, op=MPI.SUM)
```

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
```

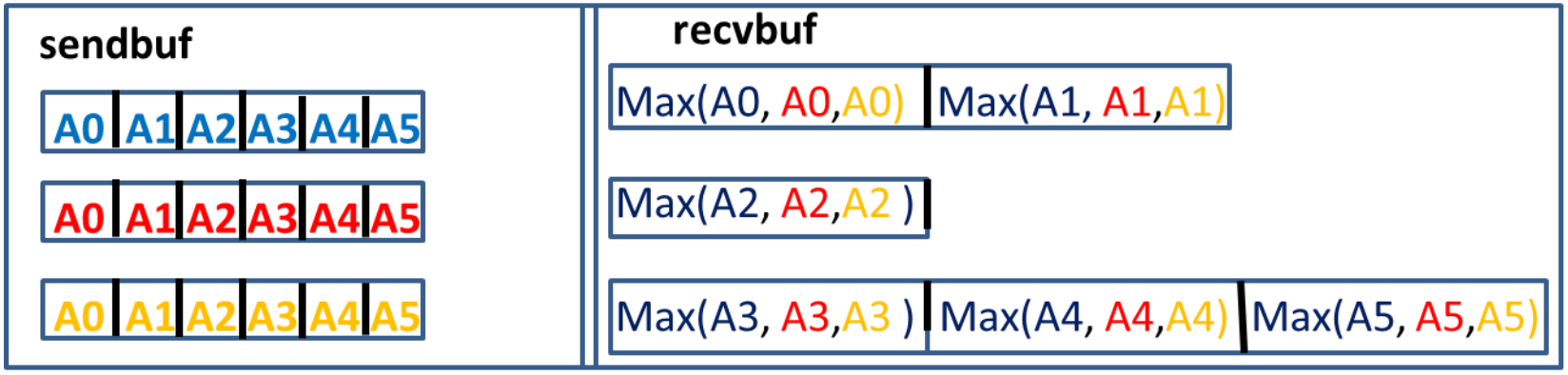
Exercise : MPI_REDUCE_SCATTER

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

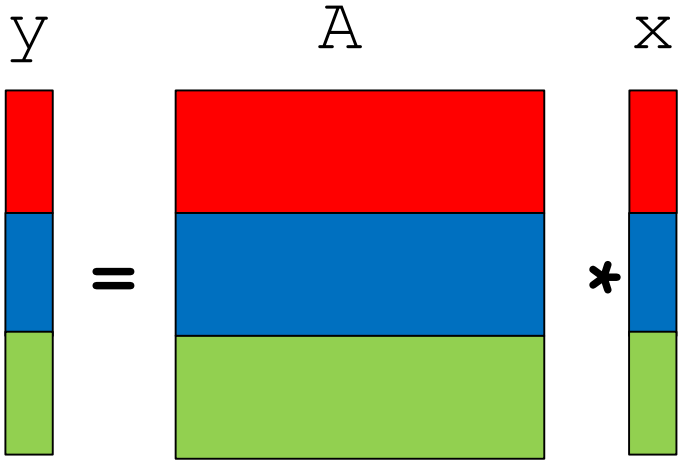
Fortran:

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

                                          ...
!          call reduce_vector(n,y,x)
           call MPI_Reduce_scatter(y, x, counts
     :                            , MPI_DOUBLE_PRECISION, MPI_SUM
     :                            , MPI_COMM_WORLD, ierr)
```

Python:

```
#    x = reduce_vector(n,y)
comm.Reduce_scatter(y, x, recvcounts=None, op=MPI.SUM)
```

Parallel Matrix-Vector Multiplication
Row-block Distribution



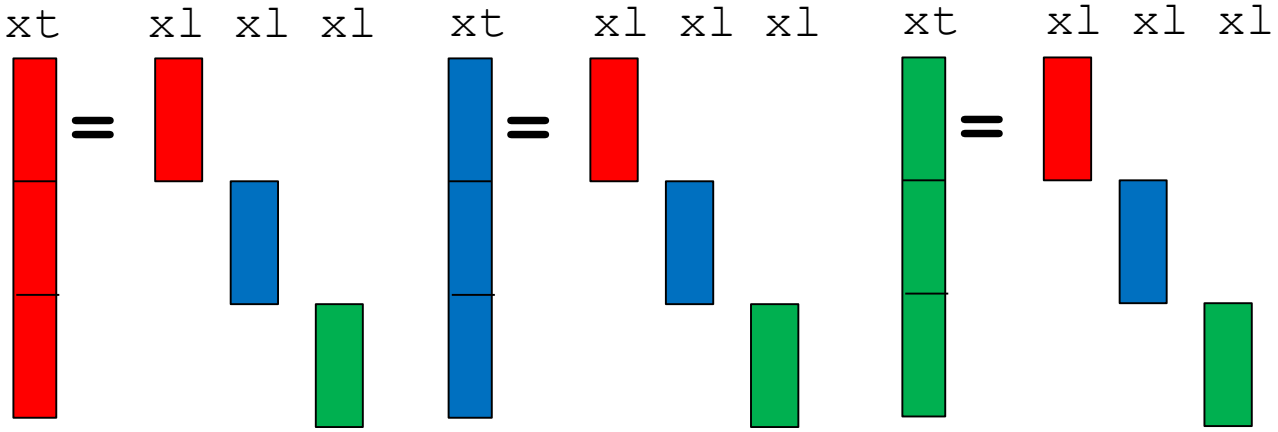
np processes ip = 0,...,np-1

local on each process:

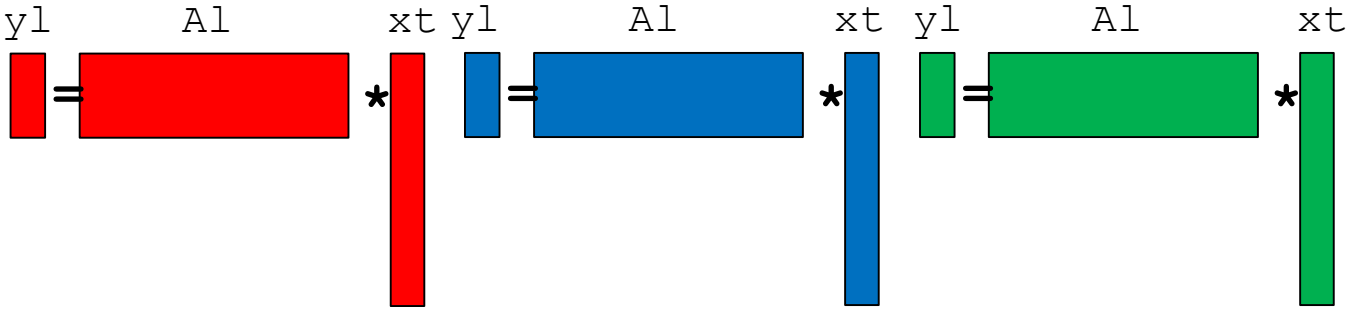
nloc = rows of A, nloc elements of x and of y

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
Par	Initialise xl
	Loop
Par	collect input vector xt
Par	yl = Al * xt
Ser	λ = yl(1)
Par	distribute λ
Par	xl = 1/λ * yl

dist_index
dist_matrix_rowblock

global_vector
DGEMV

MPI_BCAST

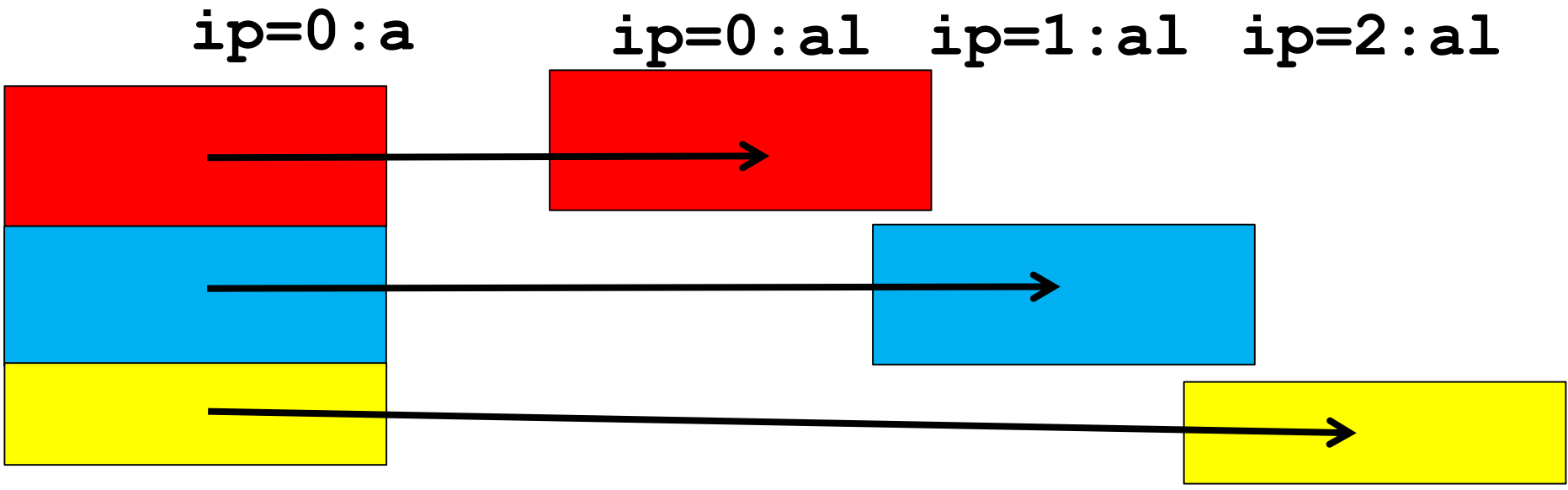
program ritz_dist_row

Generate executable

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

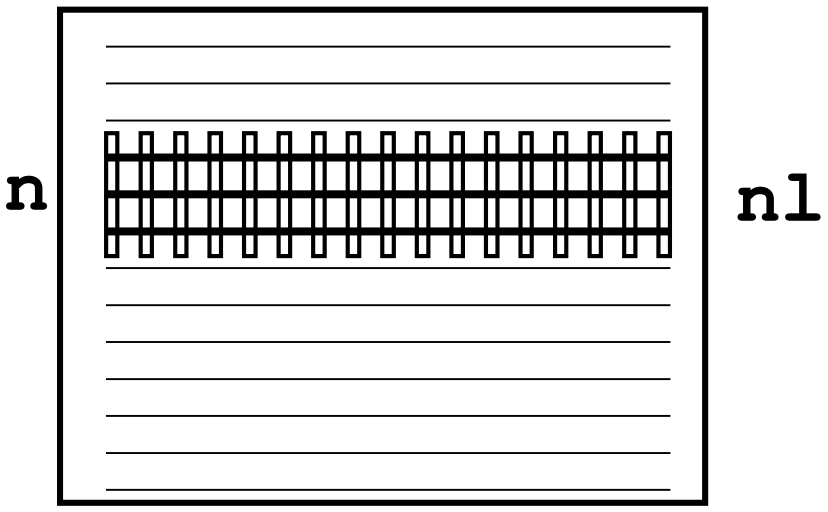
```
> make ritz_dist_row
```

dist_matrix_rowblock(m,n,a,al)

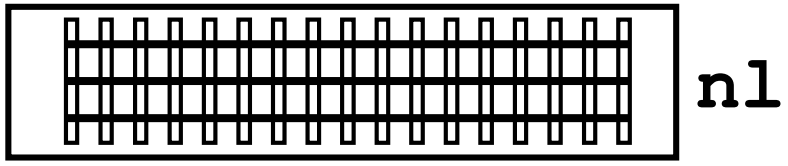


Rowblock Distribution of Global Matrix (1)

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

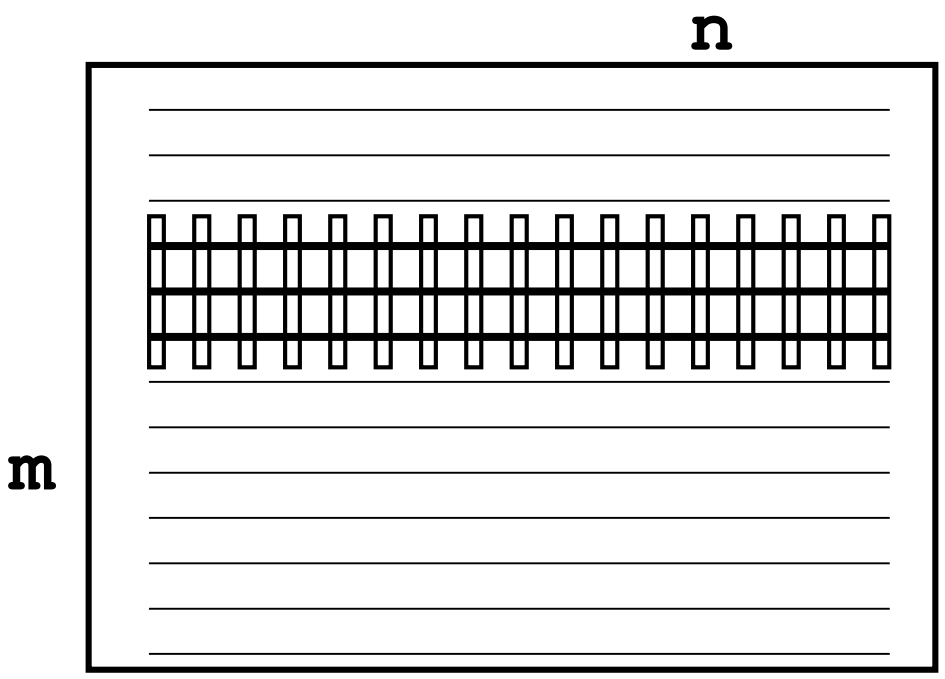
Rowblock Distribution of Global Matrix (2)

```
      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)
```




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

Use of Datatype vector

Modify **dist_matrix_rowblock**

if (myid.eq,0):

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

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

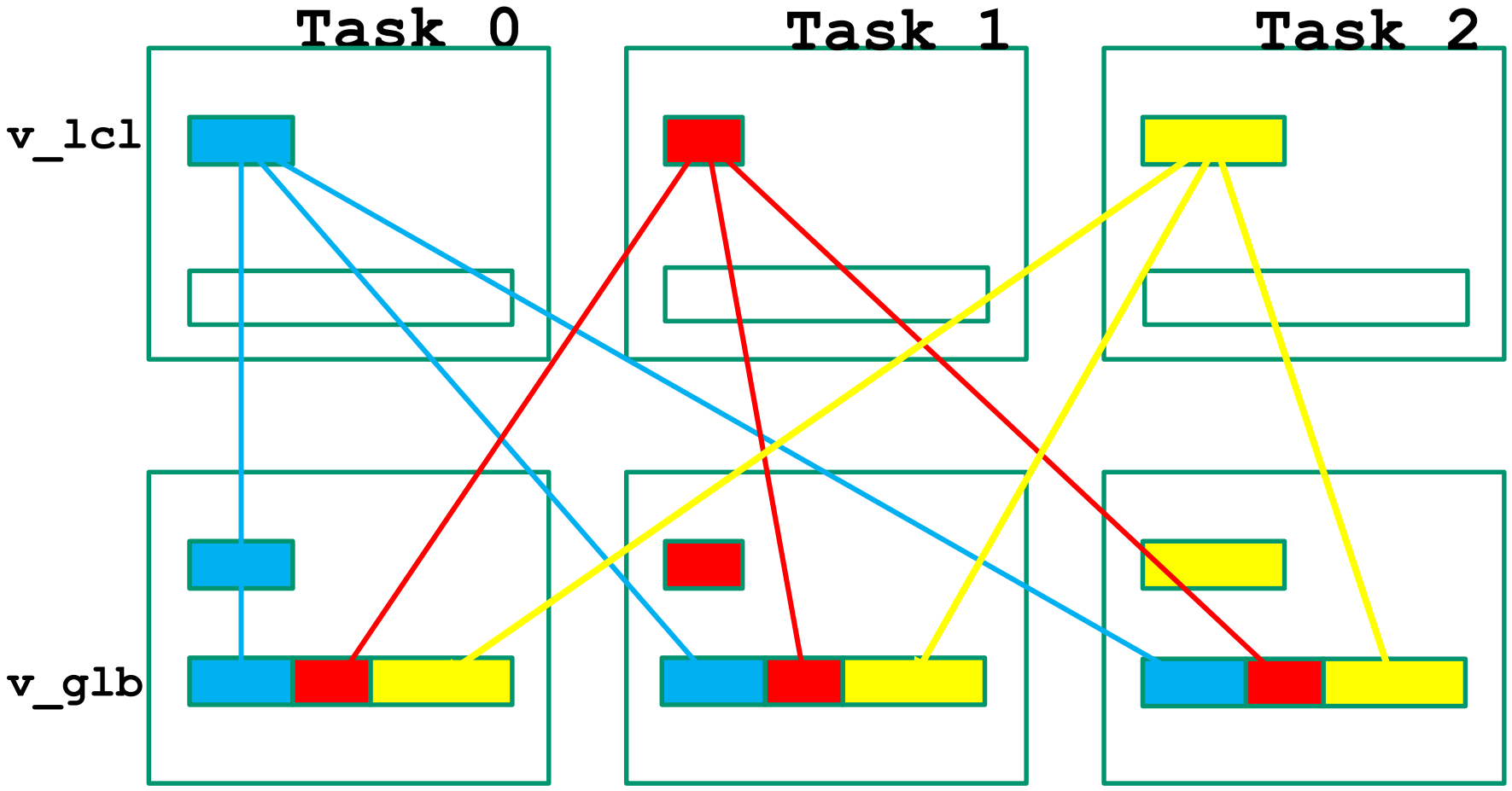
```
ia = firstind(ip)+1
MPI_SEND(a(ia),1,rowblock,ip,tag,
                  MPI_COMM_WORLD,ierr)
```

- Deactivate the new type

```
MPI_TYPE_FREE(rowblock,ierr)
```

Solution in
~/mpisolutions/*/dist_matrix_rowblock_type_ready.*

Gather Data

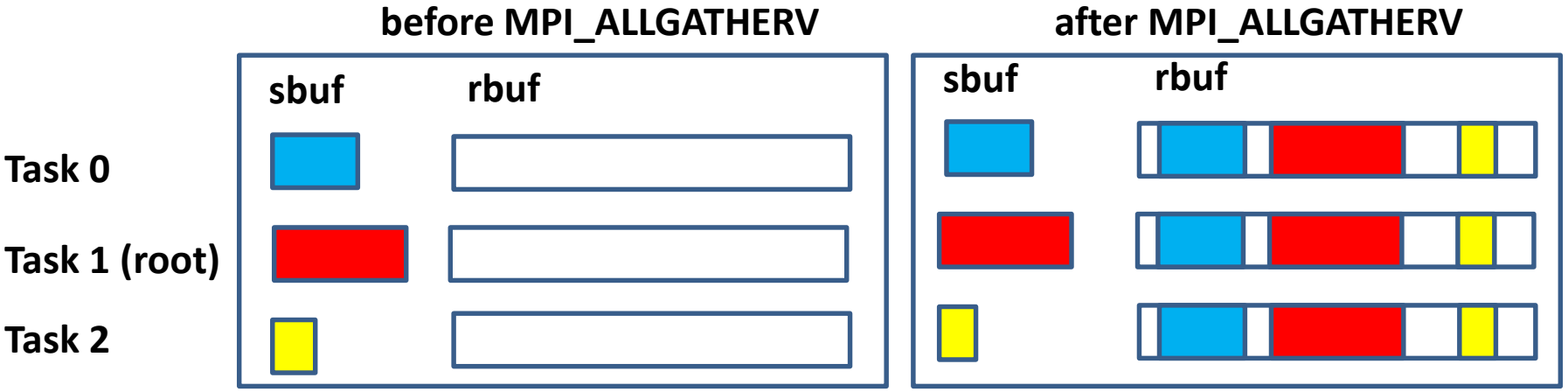


global_vector with MPI_BCAST

```
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
```



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 n = 1000

compare e(np) for different np

determine $np_{1/2}$: $e(np_{1/2}) < ½$

Use jobscript in directory **`mpiexercises/[f,c,py]/Ritz`**

2-dim Heat Equation

Learning Objectives:

- Domain Decomposition with Border Exchange
- Scaling Analysis
- 2-dim Process Grid
- Overlapping of Computation with Communication

Diffusion Equation

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

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

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

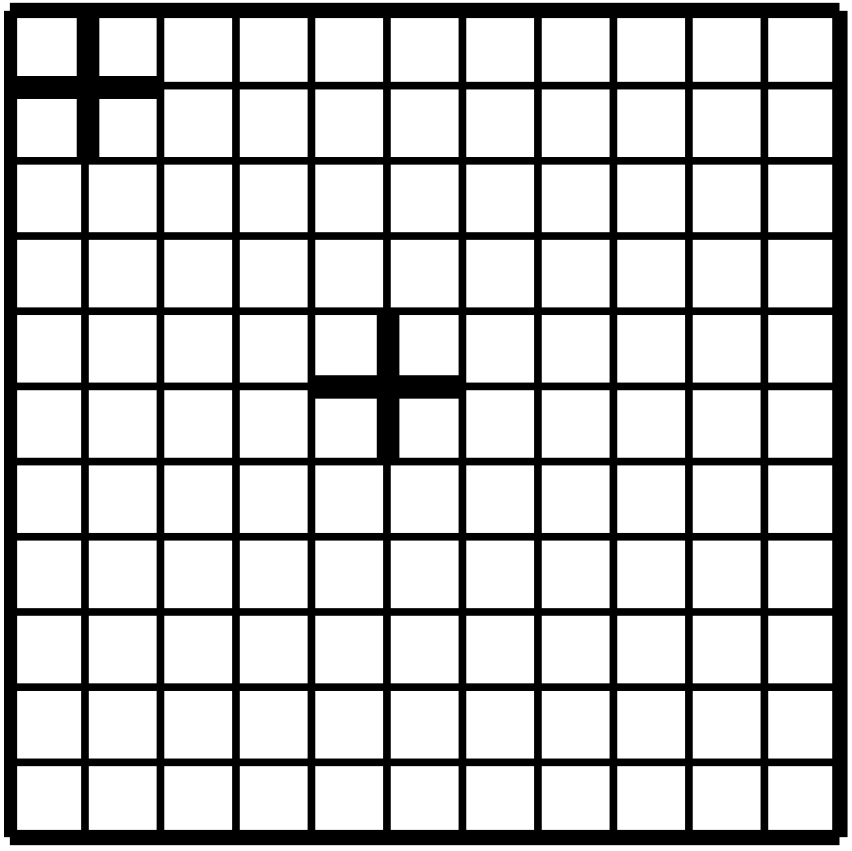
Innere Punkte : $un(i_1,i_2) = s \cdot u(i_1,i_2) +$

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

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 **`mpiexercises/*/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 **mpiexercises/*/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 columns **1** to **n2** in **u**,
using old values from columns **0** to **n2+1** from **a**

Program Execution Wärmeleitung

Source code in directory **mpiexercises/*/Waermeleitung**

```
> make waermeleitung
```

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

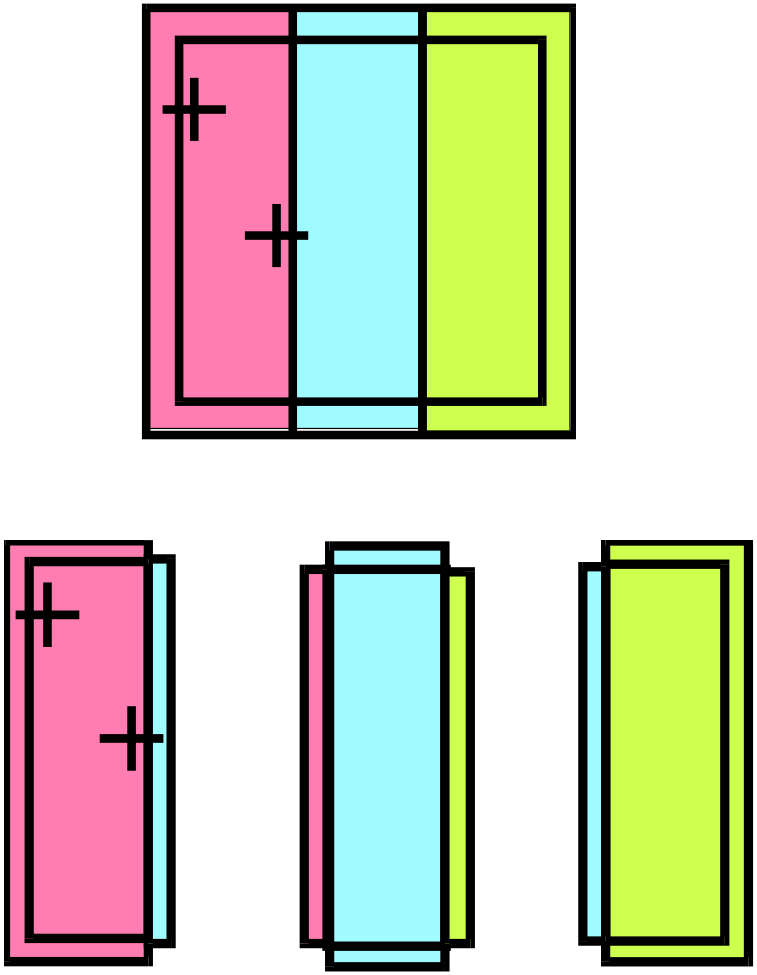
```
500             read(5,*) nt
499             read(5,*) n
0.2             read(5,*) r
```

```
> 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 **`mpiexercises/*/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 n2lth 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
```

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 **`mpiexercises/*/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:

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

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

Scaling with fixed number of grid points per process:

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

Exercise : Parallel Efficiency

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

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

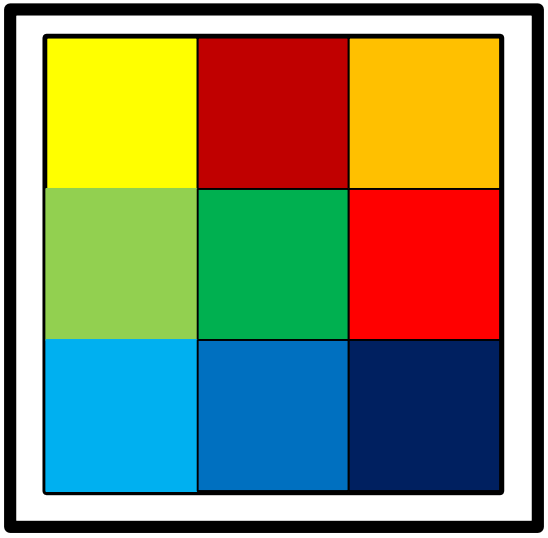
t_{op} and t_{com} are displayed as tupd and trand in the output of the program waermeleitung

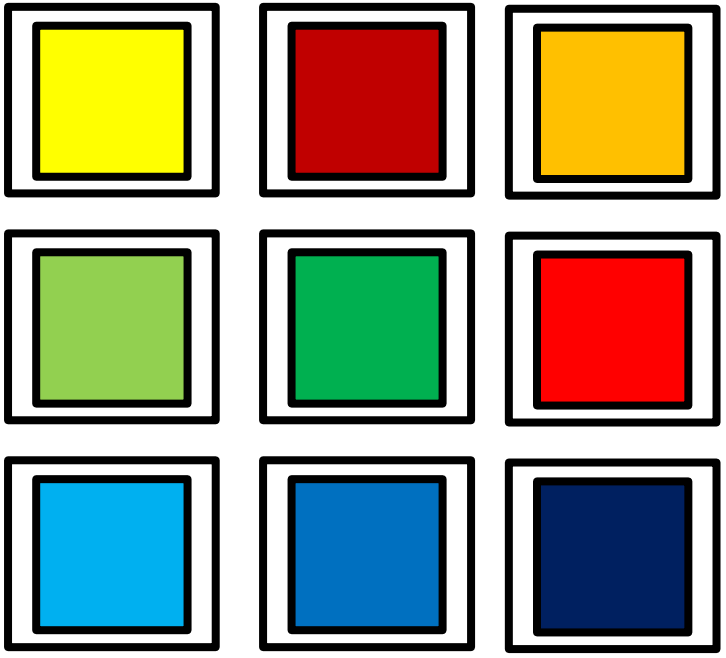
Hint: in program waermeleitung

set the value of n to 50*sqrt(real(nproc))

Use the jobscript job.script to run waermeleitung_mpi for different values of nproc

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:

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

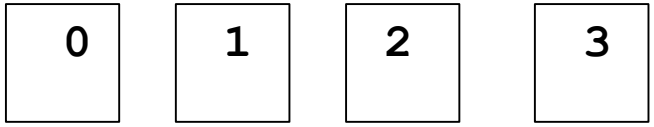
Heat Equation with 2-dim. Block Distribution

Modification

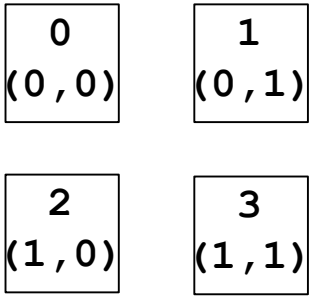
Input: size of 2-dim process-grid: **`nq1,nq2`**
`nq1*nq2 <= nproc ?`
Map **`myid->(myid1,myid2)`**
Calculate blocksizes **`n11,n21`** for all blocks
Initialize local blocks
Boundary exchange for 2-dim distribution

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

Example code for creating 2-dim grid

```
integer com, com_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)
```

Complete code in **mpiexercises/f/WL_2d/test_2dim.f**

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

if periods =.false. :
rank_source and/or rank_dest = MPI_PROC_NULL for boundary nodes
```

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.ne.MPI_PROC_NULL) 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.ne.MPI_PROC_NULL) then
   a(n1+1,1:n2) = ar(1:n2)
 end if
```

Complete code in directory **mpiexercises/f/WL_2d**

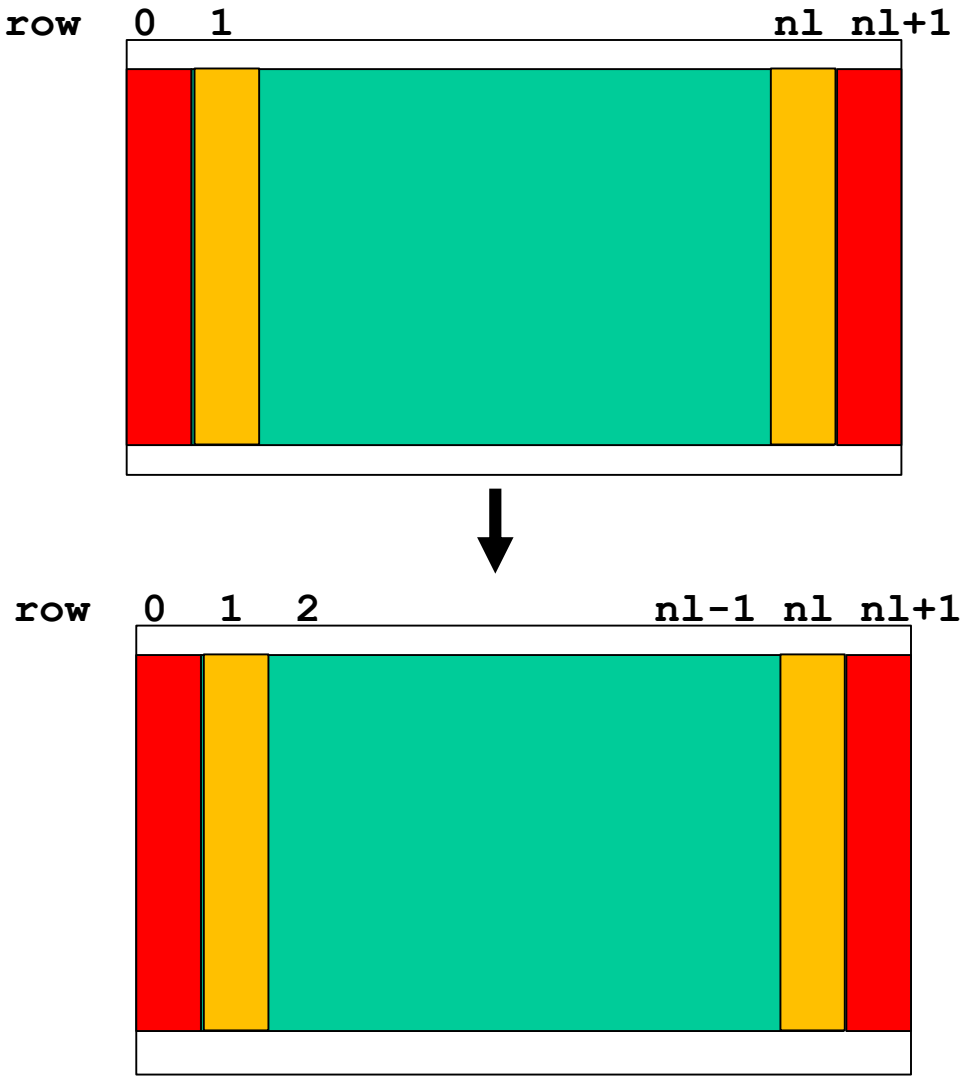
Border Exchange in 2-dim Topology

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 **mpiexercises/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
```

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**