

# MPI Lab

# Getting Started

- Login to ranger.tacc.utexas.edu

- Untar the lab source code

```
lslogin3$ cd
```

```
lslogin3$ tar -xvf ~train00/mpibasic_lab.tar
```

- Part 1: Getting Started with simple parallel coding

- hello mpi-world

- Part 2: Calculating PI

```
cd mpibasic_lab/pi
```

- Part 3: 1-D domain decomposition

```
cd mpibasic_lab/decomp1d
```

# Part 1: Hello MPI world!

- Starting with serial hello.f (or hello.c)
- Make a copy, name it mpi-hello.f or your own.
- Include mpi.h header file.
- Insert MPI\_INIT & MPI\_FINALIZE
- Getting basic parallelization info using MPI\_COMM\_SIZE & MPI\_COMM\_RANK
- Modify print\* statement
- Now you have made a parallel code with MPI.

# Hello, MPI-World!

- Check out your module set first

- Running serial code

```
lslogin3$ ifort hello.f
```

```
lslogin3$ ./a.out
```

- After converting serial code to parallel code:

```
lslogin3$ mpif90 mpi-hello.f -o mpi-hello
```

- Edit your job submission script, then submit your job:

```
lslogin3$ qsub job.sge
```

- Edit your job submission script, then submit your job:

```
lslogin3$ more $Job_Name.o$Job_ID
```

# Part 2: Calculation of Pi

- Objective: parallelize serial  $\pi$  calculation, starting with serial code (serial\_pi.c or serial\_pi.f90).

in C

```
for (i = 1; i <= n; i++) {  
    x = h * ((double)(i) - 0.5e0);  
    sum = sum + f(x);  
}  
  
pi = h*sum
```

in Fortran

```
do i = 1, n  
    x = h * (dble(i) - 0.5_KR8)  
    sum = sum + f(x)  
end do  
  
pi = h*sum
```

- Main focus on Parallelization
  - How to split a problem across multiple processors
  - Broadcasting input to other nodes
  - Using MPI\_Reduce to accumulate partial sums

# MPI Parallelization Basic Setup

- Modify the `serial_pi.f` or `serial_pi.c` file.
  - `cp serial_pi.f90 mpi-pi.f90` or `cp serial_pi.c to mpi-pi.c`
  - Include MPI startup and finalization routines at the beginning and end of `pi.c/f90`. Also include declaration statements for the rank and number of processors (`myid` and `numprocs`, respectively)

C: `#include "mpi.h"` or F90: `include "mpif.h"`

`...MPI_Init(...)`

`...MPI_Comm_rank(MPI_COMM_WORLD...)`

`...MPI_Comm_size(MPI_COMM_WORLD...)`

`...`

`...MPI_Finalize(...)`

Top of Code

Serial Code

End of Code

Don't forget:

- Declare `myid`, `numprocs` and `ierrs` as ints in C and integers in fortran.
- Use "call" and an error argument in FORTRAN; and use error return in C code.
- Use `myid` and `numprocs` for the rank and processor count

( and error argument in f90 codes)

# Calculating $\pi$ – Parallel Version

- Each processor will perform partial sum:  
for  $x_i, x_{i+N}, x_{i+2N}, x_{i+3N}, \dots$  where  $N$  is the processor count, and  $i$  is the rank.

```
for (i = myid+1; i <= n; i = i + numprocs)
{
    x = h * ( (double)(i) - 0.5e0 );
    sum = sum + f(x);
}
part_pi = h*sum
```

```
do i = myid+1, n, numprocs
    x = h * (dble(i) - 0.5_KR8)
    sum = sum + f(x)
end do

part_pi = h*sum
```

- Accumulate and add partial sums on processor 0.

```
ierr = MPI_Reduce(&part_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)
```

```
call MPI_Reduce(part_pi, pi, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

# Read & Form Partial Sums

- Have rank 0 processor read n, the total # elements to integrate
  - Make the read statement conditional, only on root, with:  
`if ( myid == 0 ) read...`
  - Broadcast n to the other nodes  
`MPI_Bcast(n,1,<datatype>,0,MPI_COMM_WORLD...)`  
Use MPI\_INTEGER and MPI\_INT for Fortran and C datatypes, respectively. (Use &n address for C).
- Specify integral elements for each processor
  - F90: `do i = 1,n` → `do i = myid+1, n, numprocs`
  - C: `for(i=1; i<=n; i++)` → `for(i=myid+1; i<=n; i=i+numprocs)`

# MPI-Reduce partial sums, print

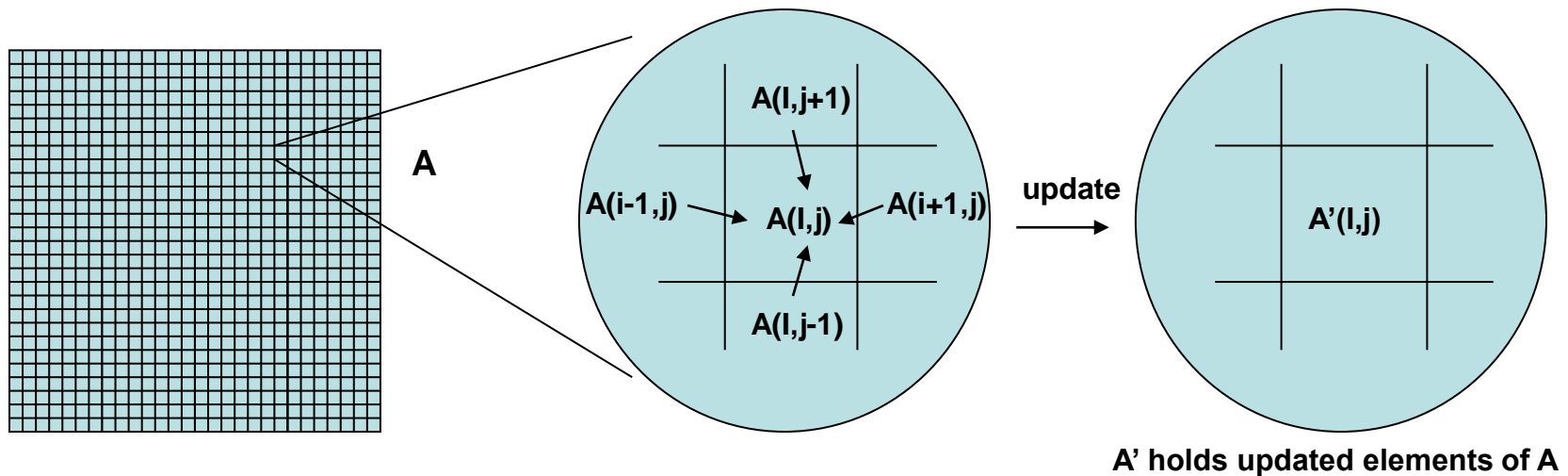
- Assign the sum from each rank to a partial sum
  - declare `part_pi` as a double [ `real(KR8)` in F90 ]
  - after the loop, replace “`pi = h * sum`” with :  
`part_pi = h * sum;` followed by
- Sum the partial sums with an MPI\_Reduce call  
`...MPI_Reduce(part_pi,pi,1,<type>,MPI_SUM,0,  
MPI_COMM_WORLD...)`  
where `<type>` is `MPI_DOUBLE` or `MPI_DOUBLE_PRECISION` for C and F90, respectively; use addresses `&part_pi` and `&pi` in C code
- Write out  $\pi$  & calc. pi, from rank 0 proc (use if)
  - `if (myid == 0) print...`

# Compilation & Running the code

- Compile code:  
`mpif90 -O3 mpi-pi.f90`  
`mpicc -O3 mpi-pi.c`
- Prepare job  
Modify the processor count:  
Keep the # of processors per node set to 16 (keep the “16way”)  
The last argument, divided by 16, is the number of nodes.  
`#$ -pe 16way 16 → change 16 to 32, 48, 64, etc.`
  - create a file called “input” and include the total number of elements (n) on the first line  
`echo 2000 >input`
- Submit job  
`qsub job`
- See `mpi_pi.f90 (.c)` for finished parallel versions.

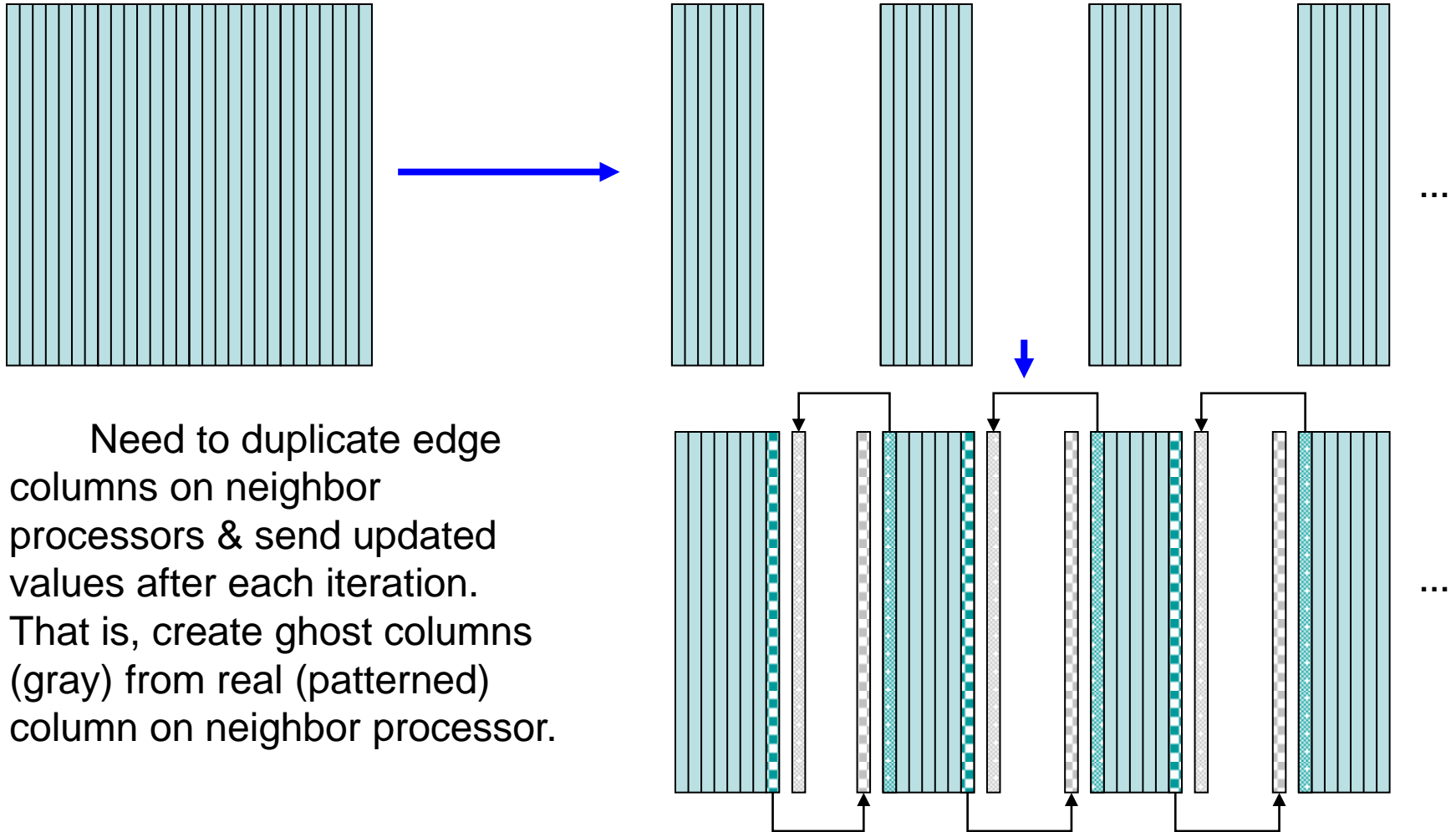
# Part 3: Domain Decomposition

- Solve 2-D partial differential equation (finite difference)
  - Represent x-y domain as 2-D Cartesian grid
  - Solution Matrix= $A(x,y)$
  - Initialize grid elements with guess.
  - Iteratively update Solution Matrix ( $A$ ) until converged.
  - Each iteration uses “neighbor” elements to update  $A$ .



# Domain Decomposition: Sharing Data Across Processors

Decompose 2-D grid into column blocks across  $p$  processors (1-D decomposition)



Need to duplicate edge columns on neighbor processors & send updated values after each iteration. That is, create ghost columns (gray) from real (patterned) column on neighbor processor.

# Domain Decomposition

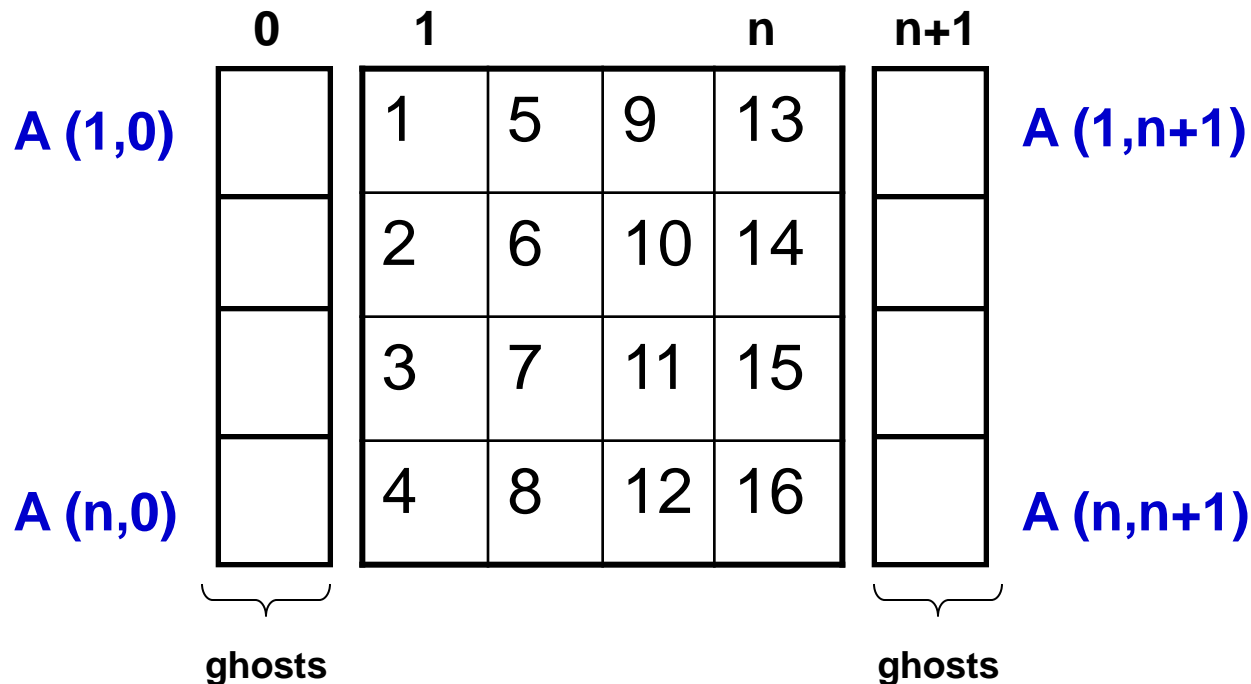
## Matrix Layout with Ghost Cells

Redefine Array for easy ghost access

```
real*8 :: A(n, 0:n+1) Fortran
```

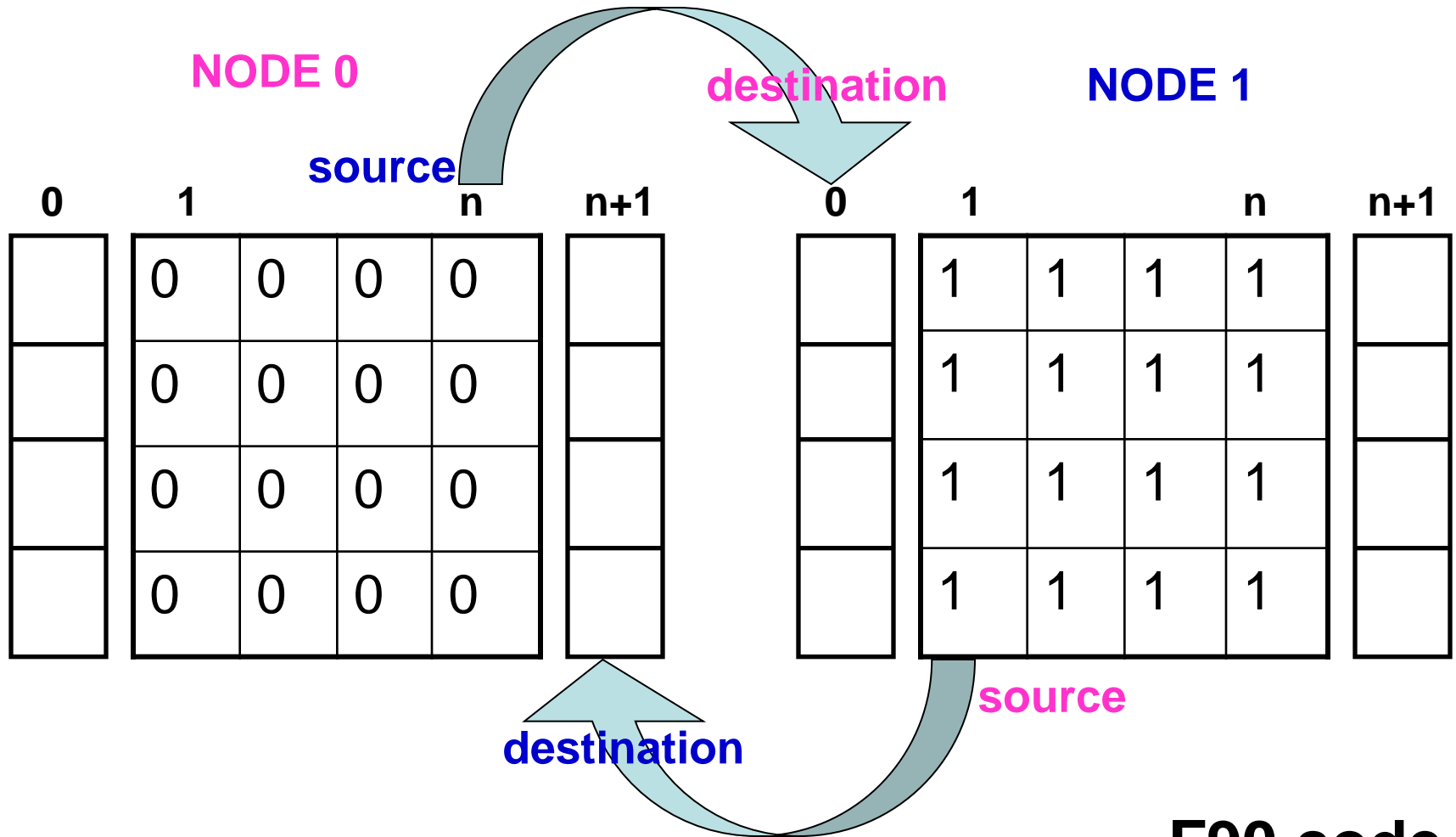
```
#define A(i,j) a( (i-1) + (j)*n )  
double a[n*(n+2)];
```

C



# Domain Decomposition

## Node Exchange

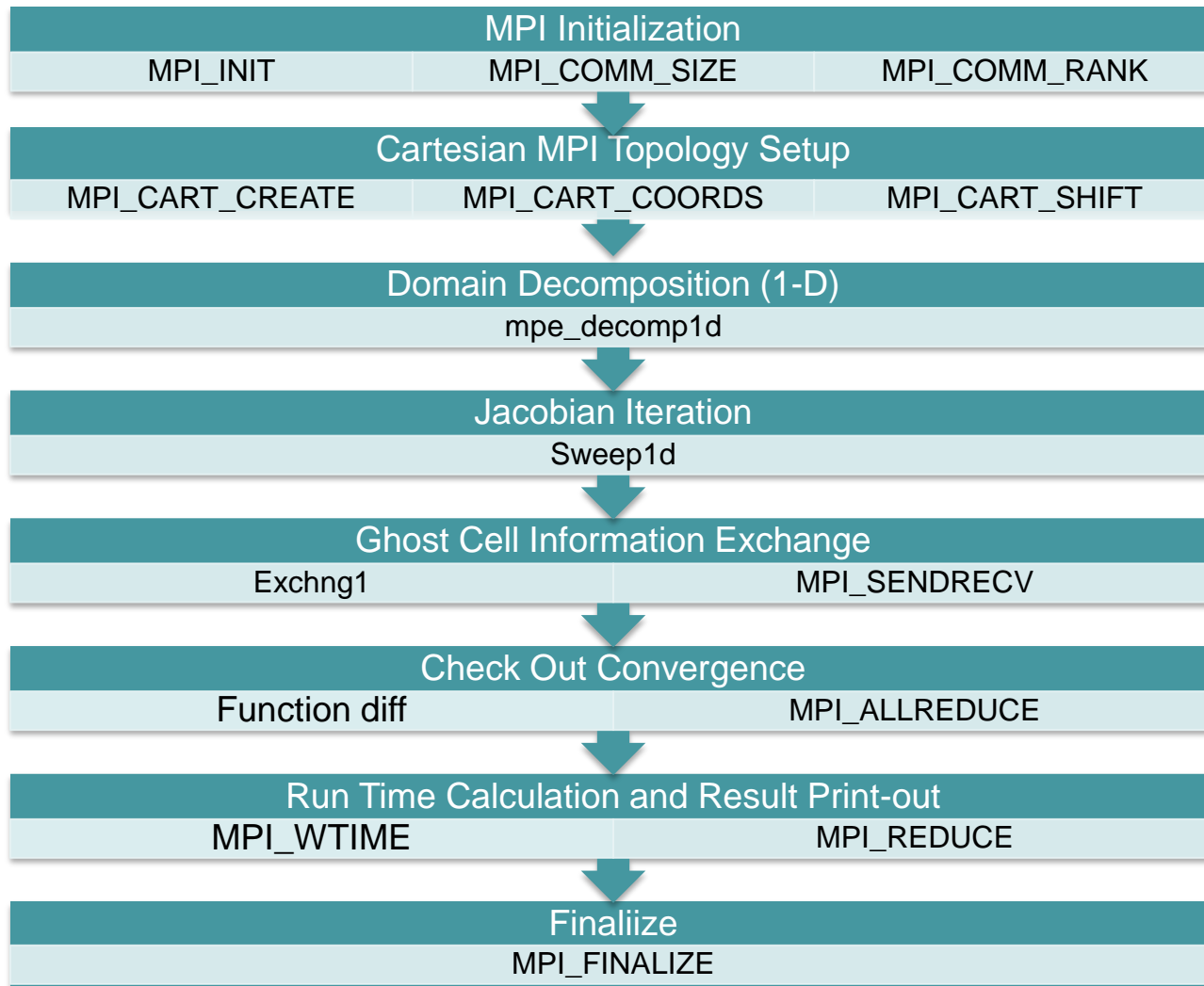


**F90 code**

# Domain Decomposition (cont'd)

- MPI functions used in the example code:
  - MPI\_INIT/FINALIZE
  - MPI\_COMM\_SIZE/RANK
  - MPI\_BCAST
  - MPI\_CART\_CREATE/COORDS/SHIFT
  - MPI\_SENDRECV
  - MPI\_BARRIER
  - MPI\_REDUCE/ALLREDUCE
- Subroutines in the code:
  1. `mpe_decomp1d`: decomposes a domain (matrix) into column or row slices for each processor, redefines local grid index (for the example code, the decomposition is done along **y-axis**)
  2. `onedinit` : initializes the right hand side of poisson equation and the initial solution guess.
  3. `exchng1`: MPI send & receive between ghost columns.
  4. `sweep1d`: performs a Jacobi sweep for a 1-D decomposition

# mpi\_1d\_decomp.f90: work flow



# Domain Decomposition

- Compile code:

```
mpif90 -O3 mpi_1d_decomp.f90
```

- Prepare job

Modify the job submission script:

- Keep the # of processors per node set to 16 (keep the “16way”)
- The last argument, divided by 16, is the number of nodes.

```
#$ -pe 16way 16 → change 16 to 32, 48, 64, etc.
```

If the number of cores is not a multiple of 16, then set the MY\_NSLOTS environment variable with the number of cores. E.g. “setenv MY\_NSLOTS 31”. But the last argument of the `-pe` option must be a multiple of 16, so as to determine the number of nodes.

- Submit job

```
qsub job
```