# MATRIX MODELS

A Lightning Introduction to Computer Simulations
of (some) Matrix Models

Athens, 2025

KONSTANTINOS N. ANAGNOSTOPOULOS
*National Technical University of Athens*

National Technical University of Athens

5.E Appendix: Wick Rotation . . . . . 177
References . . . . . 180

**6 Parallel Computation** **183**
6.1 Fortran Coarrays . . . . . 186
6.1.1 Images . . . . . 186
6.1.2 Coarrays . . . . . 190
6.1.3 Collective Subroutines . . . . . 194
6.1.4 Procedures - Broadcasting an Array . . . . . 198
6.1.5 Matrix Multiplication . . . . . 204
6.1.6 Transpose and Traces . . . . . 206
6.2 MPI . . . . . 208
6.2.1 Compiling and Running . . . . . 210
6.2.2 Timing your Program . . . . . 218
6.2.3 Collective Operations . . . . . 220
6.2.4 Broadcasting an Array . . . . . 232
6.2.5 Matrix Multiplication . . . . . 237
6.2.6 Transpose and Traces . . . . . 242
6.2.7 Communicators . . . . . 245
6.2.7.1 The Diagonal Communicator . . . . . 245
6.2.7.2 Row and Column Communicators . . . . . 252
6.2.7.3 Matrix Multiplication: Fox's Algorithm . . 255
6.2.8 Scalability . . . . . 259
6.3 ScaLAPACK . . . . . 262
6.3.1 BLACS . . . . . 263
6.3.2 Naming Conventions . . . . . 270
6.3.3 Distributing Arrays . . . . . 273
6.3.4 Matrix Multiplication . . . . . 276
6.3.5 Transpose and Traces . . . . . 278
6.3.6 Eigenvalues and Eigenvectors . . . . . 280
6.4 Working Together: Coarrays, MPI, ScaLAPACK . . . . . 286
6.4.1 Coarrays & ScaLAPACK Integration . . . . . 286
6.4.2 Independent "Simulations" and ScaLAPACK Contexts 290
6.5 A Guide to Further Reading . . . . . 295
6.6 Problems . . . . . 295
6.A Appendix: Array Descriptors and Index Mapping in ScaLAPACK . . . . . 295
References . . . . . 298

**7 Parallel Simulations** **301**
7.1 The Embarrassingly Parallel Approach . . . . . 301
7.2 Parallel Dimensions . . . . . 307
7.3 Parallel Simulations: Multi-Ensemble ScaLAPACK Implementation . . . . . 313

# Chapter 6

# Parallel Computation

This chapter offers an elementary introduction on parallel programming. Since the topic of the book is matrix models, we will focus on how to parallel program elementary matrix operations, like assignment, matrix multiplication, trace calculation, matrix transposition, and the computation of eigenvalues and eigenvectors. We will focus on parallel programming on CPUs, therefore GPU programming is not covered in this chapter.

We will show how to perform linear algebra parallel computations using Fortran coarrays, the Message Parsing Interface (MPI) library, and the ScaLAPACK library. The first method is a Fortran built in standard, which is simple, intuitive, and available in most modern Fortran environments. MPI is the de facto standard for parallel programming in Fortran, C, and C++ High Performance Computing (HPC) applications. Finally, ScaLAPACK is built on LAPACK, and can be run on any computer that supports MPI. It provides a parallel version of a subset of LAPACK routines, which you can use for matrix multiplication, solving linear systems, computing eigenvalues and eigenvectors, etc.

All of the above follow the, so called, Single Program Multiple Data (SPMD) model, where identical copies of the same program are run independently by different *processes* (or *images* in the coarray slang). Each copy of the program has its own copy of the working data in its local memory. Parallelization is achieved by the exchange of data ("communication") between the processes. Each process runs independently, until (explicit or implicit) synchronization commands between all, or a group of processes are executed by the program. See Figure 6.1 for a pictorial representation.

We should stress that the model is not aware of the hardware architecture. Some processes can be run on the same core of a CPU, on multiple cores of a CPU, on multiple CPUs on the same motherboard, on CPUs of different racks of a supercomputer, or independent desktops/laptops connected via fast or slow network! This is controlled by the *running*

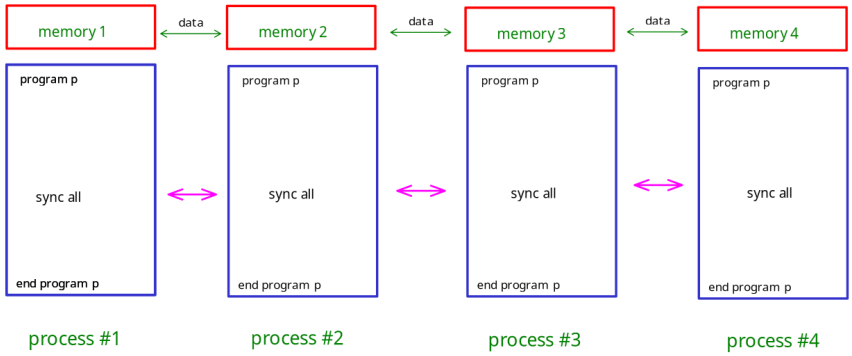



Figure 6.1: The SPMD paradigm: a replica of the same program is created for each running process. Each process has its own memory space containing data which can be manipulated independently. Each process runs its own copy of the program independently, until instructions are given in the program for data exchange and/or synchronization. For example, the `sync all` (Fortran syntax) instruction makes all processes to wait for each other. When this instruction is executed by all processes, then independent execution resumes. Data communication may occur at any moment during the execution of the program, and it can be synchronous, or asynchronous. Each process's memory space, although independent, can be located on the same or different memory chips. Data exchange may occur between any processes, and the topology of exchange is usually not controlled by the programmer (MPI programming, however, provides routines that create different types of *virtual* topologies).

commands, and not by the program. Therefore, you don't need a supercomputer, or a cluster of computers to test your parallel code, you can do that on the single processor of your laptop.

In the examples presented in this chapter, we consider a $N \times N$ matrix $\mathbf{A} = (A_{a,b})$, $a, b = 1, \ldots, N$ distributed on a grid of `nprocs` $= N_\mathrm{p} \times N_\mathrm{p}$ processes, as shown in Figure 6.2. Each process holds in its own memory space a $N_\mathrm{loc} \times N_\mathrm{loc}$ square submatrix $\mathbf{A}_{IJ}$, $I, J = 1, \ldots, N_\mathrm{p}$, where $N_\mathrm{loc} = N/N_\mathrm{p}$ (of course, we assume that $N_\mathrm{p}$ is a divisor of $N$).

In block notation, we will write $\mathbf{A} = (\mathbf{A}_{IJ})$. Under these assumptions, several matrix operations simplify considerably. For example, the matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ can be computed using the relation:

$$\mathbf{C}_{IJ} = \sum_{K=1}^{N_\mathrm{p}} \mathbf{A}_{IK} \cdot \mathbf{B}_{KJ} \quad I, J = 1, \ldots, N_\mathrm{p}\,. \tag{6.1}$$

Therefore, in order to compute it, one needs to communicate with the processes holding the $(IK)$ block of $\mathbf{A}$, and the $(KJ)$ block of $\mathbf{B}$, make

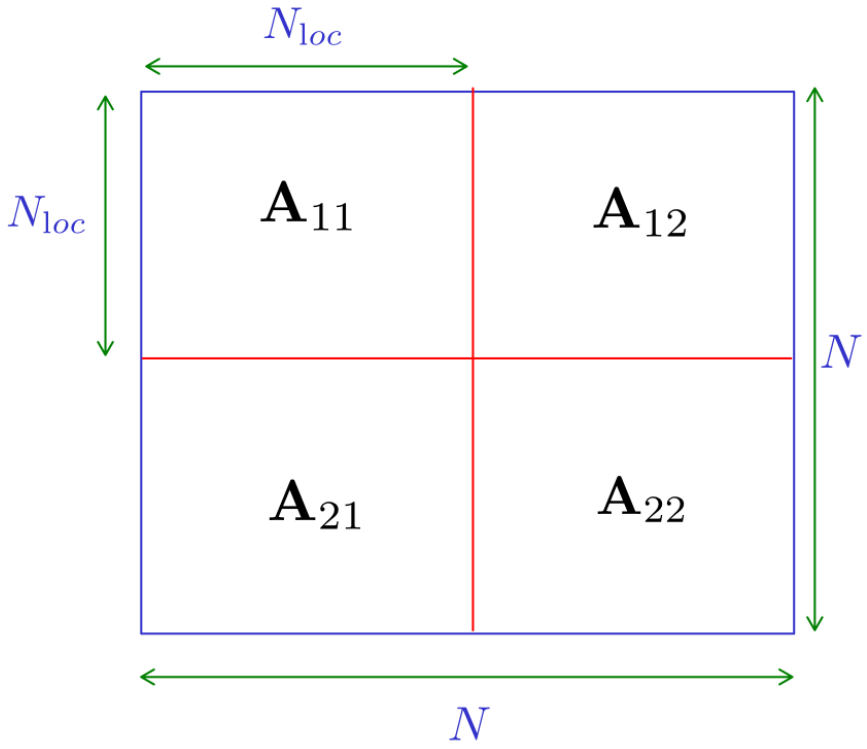



Figure 6.2: An $N \times N$ matrix $\mathbf{A} = (A_{a,b})$, $a, b = 1, \ldots, N$ distributed on a grid of $N_\mathrm{p} \times N_\mathrm{p} = 4$ processes. We choose $N_\mathrm{loc} = N/N_\mathrm{p}$, so that each process holds an $N_\mathrm{loc} \times N_\mathrm{loc}$ submatrix $\mathbf{A}_{IJ} = (A_{(I-1)N_\mathrm{loc}+p,\,(J-1)N_\mathrm{loc}+q})$, $I, J = 1, \ldots, N_\mathrm{p}$, $p, q = 1, \ldots, N_\mathrm{loc}$.

local copies of the $N_\mathrm{loc} \times N_\mathrm{loc}$ matrices $\mathbf{A}_{IK}$ and $\mathbf{B}_{KJ}$, and perform an ordinary matrix multiplication on the process that will be holding the $(IJ)$ block of the $\mathbf{C}$ matrix. Transferring the data from the $(IK)$ and $(KJ)$ processes will require *communication* between the processes. The details of the communication are hiding "under the hood" by the software, and the programmer may have little or no control over it (in favor of programming simplification). The programmer, however, must be aware that such communications are performed *asynchronously* between the processes, and this is usually where the devil hides: Most bugs and performance bottlenecks have their root in the way the data is transferred between the processes, and the programmer makes the wrong assumptions about it.

Other matrix operations that will be of interest are: taking the trace,

$$\operatorname{tr} \mathbf{A} = \sum_{I=1}^{N_\mathrm{p}} \operatorname{tr} \mathbf{A}_{II}\,, \tag{6.2}$$

the transpose $\mathbf{A}^T$,

$$\mathbf{A}^T_{IJ} = (\mathbf{A}_{JI})^T \,, \tag{6.3}$$

and the trace of $\mathbf{A}^2$,

$$\mathrm{tr}\,\mathbf{A}^2 = \mathbf{A}_{ab}\mathbf{A}_{ba} = \mathbf{A}_{ab}\mathbf{A}^T_{ab} \,. \tag{6.4}$$

The software needed in this chapter can be easily installed on a recent Ubuntu version using the commands:

```
sudo apt install openmpi-bin libopenmpi-dev  openmpi-doc
sudo apt install libcoarrays-openmpi-dev
sudo apt install libscalapack-mpi-dev scalapack-doc
sudo apt install scalapack-mpi-test blacs-mpi-test
```

## 6.1 Fortran Coarrays

Coarray Fortran (CAF) was initially introduced in Fortran 2003 and has been officially included in the Fortran standard since Fortran 2008. For the examples presented here, we will utilize the OpenCoarrays implementation [1], which is fully supported within GCC starting with `gcc 5.1`. For more comprehensive and pedagogical introductions to CAF, please refer to [2, 3].

### 6.1.1 Images

In CAF, the parallel executable is replicated onto every running process; each replica is referred to as an `image`. Every image executes asynchronously and maintains its own private set of data objects. This structure adheres to the Single Program Multiple Data (SPMD) model.

The program can be compiled using the `caf` command, which acts as the OpenCoarrays compiler wrapper. The number of images is then specified at runtime using `cafrun`:

```
cafrun -n <num_images> executable
```

Even a simple serial program will execute concurrently across `<num_images>` independent processes.

Consider this basic "Hello World!" program (`hello.f90`), which generates a random number:

```
program hello
 real(8) :: r
```

```
 call random_number(r)
 print *,r

end program hello
```

To compile and run this code with four images:

```
caf  hello.f90 -o exe
cafrun -n 4 ./exe
```

The execution result is the printing of four independent random numbers to `stdout`. Note that the variable `r` exists independently on each image, and without explicit instructions, there is no way for one image to access the data of another.

Alternatively, a CAF program can be compiled using the `mpifort` wrapper, provided the `-fcoarray=lib` flag is used. In this case, the `libcaf_mpi` library must be explicitly loaded with the `-lcaf_mpi` flag.

The program is compiled and executed using the following commands:

```
CAF=/usr/lib/x86_64-linux-gnu/open-coarrays/openmpi/lib
mpifort -fcoarray=lib hello.f90 -L ${CAF} -lcaf_mpi -o exe
mpirun  -n 4 ./exe
```

Note that the precise location of `libcaf_mpi` is system-dependent and must be manually determined for your installation.

Images are aware of their execution environment via intrinsic functions. The total number of images running is returned by `num_images()`, and the `this_image()` function (with no arguments) returns the unique image index, an integer between 1 and the total number of images.

Since all images execute asynchronously, controlling the order of operations, especially output to standard streams, requires explicit commands. Consider this example where only image 1 prints the total number of images:

```
program     hello
 real(8)  :: r

 if(this_image() == 1) print *,'No images: ',num_images()

 call random_number(r)
 print *,r, ' from image ',this_image()

end program hello
```

When run with four images, the output order is non-deterministic. For example, the output from image 4 might appear before the output from image 1:

```
 0.97231413457330162        from image            4
No images:            4
 0.63469125965053375        from image            1
 0.34852943706016526        from image            2
 0.70240645848476002        from image            3
```

Due to this asynchronous nature, it is common practice for each image to write its results to separate files. Alternatively, controlling the output order requires external sorting or explicit synchronization commands.

The `sync all` statement is an *image control* statement that provides a *barrier* where all images pause and wait for each other before continuing execution. Including `sync all` guarantees a specific output order for operations that occur before the barrier. The following program ensures the total image count is printed before any random numbers are generated and printed:

```
program     hello
 real(8)  :: r

 if(this_image() == 1) print *,'No images: ',num_images()

 sync all

 call random_number(r)
 print *,r, ' from image ',this_image()

end program hello
```

Using a loop structure with `sync all` allows for enforced sequential output based on image index, though this approach is not recommended as it can be very slow for large output:

```
program     hello
 real(8)  :: r

 call random_number(r)
 do i = 1, num_images()
  if(this_image() == i) print *,r, ' from image ',this_image()
  sync all
 end do
end program hello
```

All images are implicitly synchronized at program initiation, behaving as if a `sync all` were the first executed statement.

When an image reaches the `end program` statement or executes a `stop` statement, it waits for all other processes to terminate, and any local data remains accessible to other images that are still running.

For example, this program is expected to print output from all images except image 1, which executes `stop` immediately:

```
program      hello
 real(8) :: r
 n = this_image() - 1
 if(n==0 ) stop
 do i = 1, 1000000
  call random_number(r)
 end do
 print *, 1.0/n ,i, r
end program hello
```

If a fatal error occurs and should immediately halt the entire job, the affected image can initiate termination across all processes using the `error stop` statement. For example, `if(n==0 )error stop` guarantees that all images will terminate almost immediately.

**Caution**: The standard Fortran 2018 behavior of `stop` (normal termination, allowing other images to finish) may not be honored by all implementations; some may execute `stop` as an `error stop`. To ensure normal termination where data is preserved for remaining processes, the `end program` statement or a controlled flow using `goto` is the safest method:

```
program      hello
 real(8) :: r
 n = this_image() - 1
 if(n==0 ) goto 100
 do i = 1, 1000000
  call random_number(r)
 end do
 print *, 1.0/n ,i, r
100 continue
end program hello
```

Input/Output operations are performed asynchronously, and file unit numbers are local to each image.

The following code generates a unique filename for each image based on its index, ensuring each image writes to a separate file, even though the unit number (`unit=17`) is the same across all images:

```
program      hello
 character(100) :: filename
```

```
 write(filename,'(A,I0.3)') 'out', this_image()

 open(unit=17,file=trim(filename))

 write(17,*) 'I am image ',this_image(),' out of ',num_images()

end program hello
```

It is permissible for the same physical file to be connected to more than one image simultaneously.

### 6.1.2 Coarrays

Coarrays are objects declared with the codimension attribute. They are accessible locally using standard Fortran syntax, but they also enable one-sided communication by allowing any image to access the data of another using *coindices*. Since a coarray *must have the identical type and shape on every image*, this parallel paradigm is often referred to as symmetric memory.

The simplest coarray is a scalar with codimension one, such as `integer n[*]`. The asterisk allows the array's total size (cosize) to be determined by the number of images at run time.

An image can access the value of `n` on any other image (`iimg`) using the syntax `n[iimg]`. The local value on the current image is referenced simply as `n`.

The following program demonstrates one-sided communication where image 1 accesses and sums the local value `n` from all images (including itself):

```
 program     hello
 integer n[*]

 n = this_image()

 !Ensure every image has finished the assignment above
 sync all

 if( this_image() == 1) then
  ntot     = 0
  do iimg  = 1, num_images()
   print *,          n[iimg]
   ntot    = ntot + n[iimg]
  end do
  print *,'tot= ',ntot
 end if

end program hello
```

When run with 4 images, the output confirms that image 1 successfully accesses the values 1, 2, 3, and 4 across the images, totaling 10:

```
           1
           2
           3
           4
 tot=           10
```

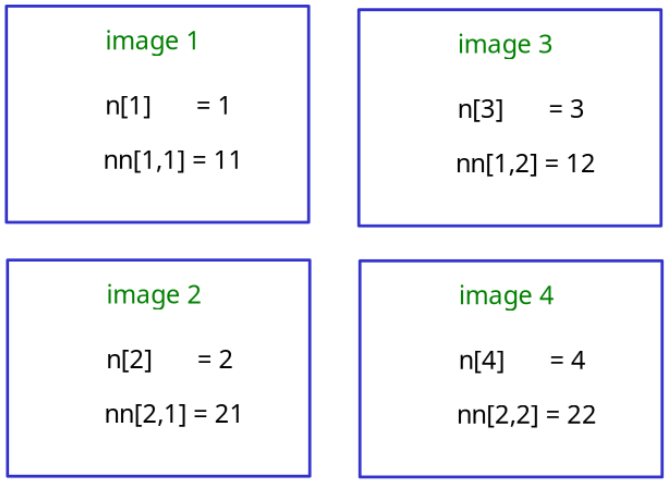



Figure 6.3: The distribution of the coarrays `integer n[*], nn[2,*]` on the images of the programs on pages 190 and 191, when the number of images is 4. The values are computed from `n=this_image()`, and `nn=ip*10+jp`, where `ip = this_image(nn,1)`, and `jp = this_image(nn,2)`. Note the column major distribution in the coindices.

Coarrays can be defined with multiple codimensions, specified by their *corank* (the number of codimensions) and their *coshape* (the size of each codimension). The range of coindices is defined by the *cobounds*.

To define a coarray with a corank greater than one, all but the last cobound must be specified:

```
integer nn[2,*], nnn[0:1,0:1,*]
real(8), codimension[-2:2,4,0:8,*] :: r
```

The next example defines a codimension 2 integer coarray `nn`:

```
program     hello
 integer nn[2,*]

 if(num_images() < 2 .or. mod(num_images(),2) /= 0) error stop

 ip = this_image(nn,1)
 jp = this_image(nn,2)
```

```
 nn   = ip*10 + jp
 sync all

 if(this_image() == 1)then
  do i = lcobound(nn,1), ucobound(nn,1)
   print *, (nn[i,j],j=lcobound(nn,2), ucobound(nn,2))
  end do
 end if

end program hello
```

The coarray `nn` is logically placed on a 2D grid of processes in column-major mode. The declared coshape of the first dimension is 2, requiring the total number of images to be a multiple of 2. The position (`ip`,`jp`) of the current image within the multi-dimensional process grid is found using the `this_image()` function, which accepts the coarray name and the codimension index as arguments. These position indices are referred to as cosubscripts. The size of the second dimension is automatically determined at runtime, equaling `num_images()`/2. The range of coindices (cobounds) is determined using the intrinsic functions `lcobound` and `ucobound`.

Running with 4 images produces the following output on Image 1, visualizing the values across the $2 \times 2$ grid:

```
          11          12
          21          22
```

This output correctly reflects the column-major placement pattern shown in Figure 6.3.

Coarrays are not limited to scalar types; they can be of any type, including arrays. When a coarray is locally an array, its data is accessed through standard Fortran array indexing. This data remains accessible for one-sided communication from any other image using the coindices.

The program below demonstrates how an $N \times N$ matrix is distributed across a grid of images, where $N = 4$ and the block size is $N_{\text{loc}}$:

```
program     coarray
 integer,parameter :: nmat   = 4, nmat_loc = 2
 integer,parameter :: nmat_p = nmat/nmat_loc
 integer           :: amat(nmat_loc,nmat_loc)[nmat_p,*]

 if(num_images() /= nmat_p**2) error stop

 ip = this_image(amat,1)
 jp = this_image(amat,2)

 do  jmat    = 1, nmat_loc
  do imat    = 1, nmat_loc
```

```
    imat_glob = (ip-1) * nmat_loc + imat
    jmat_glob = (jp-1) * nmat_loc + jmat
    amat(imat,jmat) = imat_glob * 10 + jmat_glob
  end do
 end do

end program coarray
```

The coarray `amat(nmat_loc,nmat_loc)[nmat_p,*]` partitions the global $4\times4$ matrix $\mathbf{A}$ into $2\times2$ blocks across a $2\times2$ grid of images. The coindices ($\texttt{ip}\equiv I, \texttt{jp}\equiv J$) map the image to its corresponding global matrix block $\mathbf{A}_{IJ}$ (where $\mathbf{A}_{IJ}$ = `amat(:,:)[ip,jp]`. The local array indices ($\texttt{imat}\equiv p, \texttt{jmat}\equiv q$) specify the position of an element within that block, $(\mathbf{A}_{IJ})_{pq}$. The global indices ($\texttt{imat\_glob}\equiv a, \texttt{jmat\_glob}\equiv b$) map the local indices back to the position $\mathbf{A}_{ab}$ in the full matrix.

The calculation `amat(imat,jmat)= imat_glob * 10 + jmat_glob` sets the element values to display their global index $ab$, as illustrated in Figure 6.4.

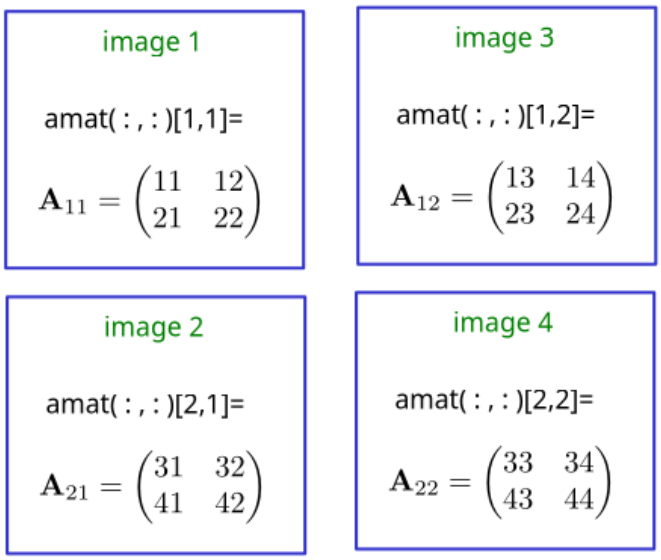



Figure 6.4: The distribution of the coarray `amat(nmat_loc,nmat_loc)[nmat_p,*]` on the images of the program `coarray` on page 192, when the number of images is 4. The values of $(\mathbf{A}_{IJ})_{pq}$, $I,J = 1,\ldots,N_{\rm p}$, $p,q = 1,\ldots,N_{\rm loc}$ are displaying the indices $ab$ of the global array $\mathbf{A}_{ab}$, $a,b = 1,\ldots,N$ (see Figure 6.1). Note the column major distribution in the coindices.

The values stored across the distributed coarray can be accessed by image 1 using coindexing:

```
sync all

if(this_image() == 1) then
 do   ip = 1, nmat_p
  do  jp = 1, nmat_p
   print *,'------------------------'
   iimg    = image_index(amat,[ip,jp])
   do imat = 1, nmat_loc
    print '(5I6)',iimg,ip,jp,amat(imat,:)[ip,jp]
   end do
  end do
 end do
end if
```

The intrinsic function `image_index(amat,[ip,jp])` returns the original integer image index (`iimg`) that owns the block specified by the cosubscripts (`ip`, `jp`).

The output from running the program with 4 images confirms the column-major distribution of the blocks onto the image indices (1, 2, 3, 4) and the mapping of global matrix indices:

```
------------------------
1     1     1    11    12
1     1     1    21    22
------------------------
3     1     2    13    14
3     1     2    23    24
------------------------
2     2     1    31    32
2     2     1    41    42
------------------------
4     2     2    33    34
4     2     2    43    44
```

Coarrays can also be declared with the allocatable attribute and allocated at run time, maintaining the use of the asterisk (*) for the last codimension:

```
integer,allocatable :: nn       [:,:]
integer,allocatable :: bmat(:,:)[:,:]

nmat = 4; nmat_loc = 2; nmat_p=nmat/nmat_loc

allocate(bmat(nmat_loc,nmat_loc)[nmat_p,*],nn[nmat_p,*])
```

### 6.1.3 Collective Subroutines

While coarrays naturally enable data exchange through one-sided communication (using coindices), Fortran also provides collective intrinsic sub-

routines to perform common operations like broadcasting data or accumulating sums across images, often without directly using coarrays.

In high-performance computing, these collective intrinsics are generally preferred because they allow the compiler and runtime system to select highly-optimized communication routines that are typically faster than manually coding a sequence of individual one-sided operations.

Data can be broadcast from one image to all others using either explicit coarray access or the optimized intrinsic subroutine. Using a coarray (`n[*]`), an image can access the value on the source image (e.g., `n[1]`). However, this operation requires explicit synchronization:

```
program      coarray
 integer n[*]

 n=0
 if( this_image() == 1) n = 99

 sync all

 n = n[1] ! now n is equal to 99 on all images

end program coarray
```

Without the `sync all` statement, the final result is non-deterministic.

The `co_broadcast(array,source_image)` subroutine performs the same function more robustly:

```
program      coarray

 n=0
 if( this_image() == 1) n = 99

 call co_broadcast(n,1)

end program coarray
```

Here, `n` does not need to be a coarray. The `sync all` statement is unnecessary because the `co_broadcast` call must be executed by all images, effectively acting as a global barrier where all processes synchronize before completion.

Similarly, collective intrinsic subroutines like `co_sum`, `co_max`, `co_min`, and `co_reduce` are available to perform collective computations on data across images.

For collective intrinsics like `co_sum`, the behavior of the result depends on whether a destination image is specified. The statement `call co_sum(a)`, executed on all images where `a` is a scalar, calculates the total sum of a across all images and overwrites a with the result on all images. Thus,

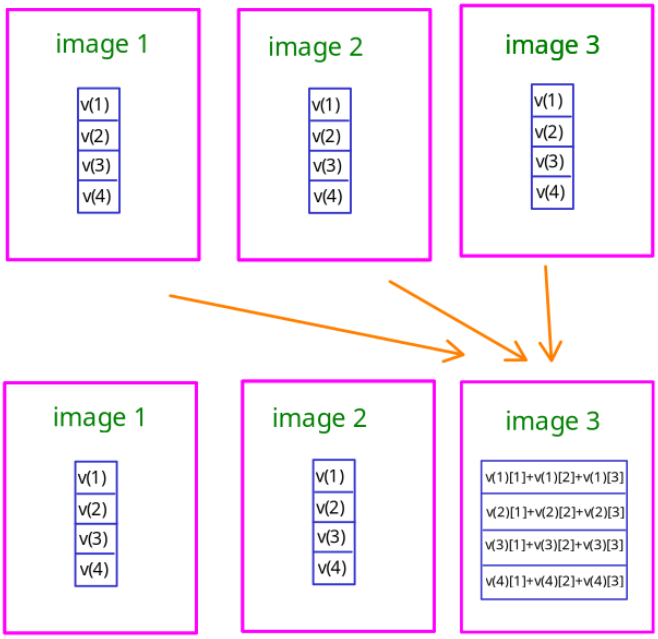



Figure 6.5: The effect of the `co_sum` collective intrinsic when a destination image is specified, as in `call co_sum(v,3)`. The array `v(4)` is defined independently on each image prior to the call. After execution, the elements `v(i)` on the destination image (Image 3) are replaced by the element-wise sum of `v(i)` from all images. On all other images (Images 1 and 2), the array `v` retains its initial values. The array `v` may or may not be a coarray.

after the call, $\mathtt{a} = \sum_{i=1}^{N_{\text{img}}} \mathtt{a}_i$, where $\mathtt{a}_i$ is the initial value of `a` on image $i$. If `a` is an array, the summation is performed element by element (elemental procedure, see Figure 6.6).

If a destination is specified, such as `call co_sum(a,1)`, the result is stored only on the destination image (Image 1 in this case), leaving the value of a unchanged on all other images (see Figure 6.5).

Similarly, the intrinsic subroutines `co_max` and `co_min`, compute the maximum and minimum value of an object across all images, respectively.

The following example demonstrates both selective and global result placement:

```
program     coarray
 real(8) r, v(4)

 call random_number(r)
 call random_number(v)

 call co_max(r,3) ! Image 3 replaces r with the max value of r
```

```
 call co_sum(v)    ! All images replace v with the sum
                   ! of v element by element

end program coarray
```

In this program, all images initialize the scalar `r` and the array `v(4)` with random values. The call `call co_max(r,3)` replaces the value of `r` only on Image 3 with the maximum value of `r` found across all images. The call `call co_sum(v)` replaces the entire array `v(4)` on all images with the element-wise sum of `v(4)` across all images.

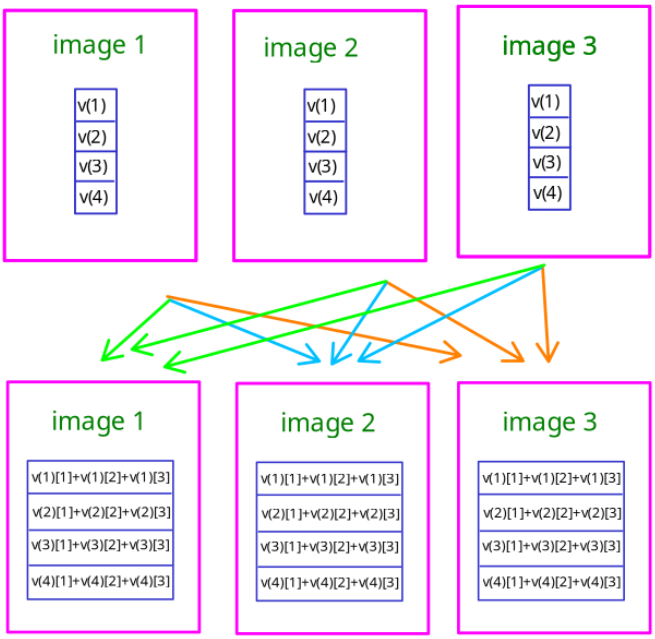



Figure 6.6: The effect of the `co_sum` collective intrinsic without a specified destination, as in `call co_sum(v)`. The array `v(4)` is initialized independently on each image before the call. After execution, the elements `v(i)` on *all images* have been replaced by the element-wise sum of `v(i)` across all images. This is a global reduction operation, and `v` may or may not be a coarray.

The `co_reduce` subroutine generalizes the collective operations (`co_sum`, `co_max`, `co_min`) to any user-defined commutative and associative operation, such as multiplication. The desired operation must be implemented as a *pure function* that accepts two scalar arguments and returns a scalar result.

The code below demonstrates how to calculate the collective product of the elements of the array `v(4)` across all images, storing the final result on Image 1:

```
program     coarray
 real(8) v(4)

 call random_number(v)

 call co_reduce(v,co_mult,1)

contains

 pure function co_mult(a,b) result (c)
  real(8), intent(in ) :: a,b
  real(8)              :: c

  c = a*b
 end function  co_mult

end program coarray
```

The reduction operator can be any commutative and associative function. Here, `co_reduce` is used with Möbius addition, where the result is again stored on Image 1:

```
program     coarray
 real(8) v(4)

 call random_number(v)

 call co_reduce(v,co_mobius,1)

contains

 pure function co_mobius(a,b) result(c)
  real(8), intent(in ) :: a,b
  real(8)              :: c

  c = (a+b)/(1.0_8 + a*b) ! associative
 end function co_mobius

end program coarray
```

### 6.1.4 Procedures - Broadcasting an Array

Coarrays can be passed as dummy arguments to procedures, such as subroutines and functions, provided the procedures are not elemental.

When coarrays are used as arguments:

- **Pass-by-Reference**: Arguments are passed by reference, meaning the coarrays are not copied, and the overhead of copy-in/copy-out is eliminated. This naturally helps avoid unnecessary synchronization requirements.

- **Allocatable Arguments**: If a coarray argument is declared `allocatable`, the actual argument in the calling program must also be `allocatable` and must share the same rank and corank.
- **Synchronization**: Because procedures may not be called by all images simultaneously, the programmer must ensure synchronization ( `sync all` ) is used wherever necessary within the procedure.
- Automatic Coarrays (coarrays whose size depends on a dummy argument) are forbidden because they would implicitly require synchronization on entry.
- Non-allocatable local coarray objects must be declared with the `save` attribute.
- Pure and Elemental procedures cannot define a coindexed object (i.e., assign to it) or contain image control statements (like `sync all` ), although they are permitted to reference the value of a coindexed object.
- An elemental procedure is strictly prohibited from having a coarray dummy argument.

The following example demonstrates both dummy and local coarrays within a subroutine:

```
subroutine mysub(x,y,a,b,c,n,np)
!-----------------------------------------------------------------
!Dummy arguments:
 integer              :: n   , np
 real(8)              :: x[*], y[np,*]
!a and c must be allocatable in the calling program:
 real(8), allocatable :: a(:,:)[:  ,:], c(:,:)[:,:]
 real(8)              :: b(:,:)[np,*]  ! or b(n,n)[np,*]
!real(8)              :: b(:,:)[:  ,:]  ! not allowed
!-----------------------------------------------------------------
!Local variables:
 real(8), allocatable :: d(:,:)[:  ,:]  ! local coarray
!real(8)              :: d(n,n)[np,*]  ! not allowed
 real(8), save        :: z[*],w[np,*]  ! local coarray
!real(8)              :: z[*],w[np,*]  ! oops, need save
 real(8), save        :: u(100,100)[*] ! local coarray
!real(8)              :: u(100,100)[*] ! oops, need save
!-----------------------------------------------------------------

!allocate the coarray of the calling program:
 allocate(c(n,n)[np,*])
!allocate the local coarray:
 allocate(d(n,n)[np,*])
```

```
  c = a ; d = a
  z[1]   = a(1,2)[2,2]
  w[1,1] = a(2,2)[1,2]
  sync all
  print *,this_image(),c(1,1)[1,1],d(1,1)[1,1],z[1],w[1,1]
  ...
end subroutine mysub
```

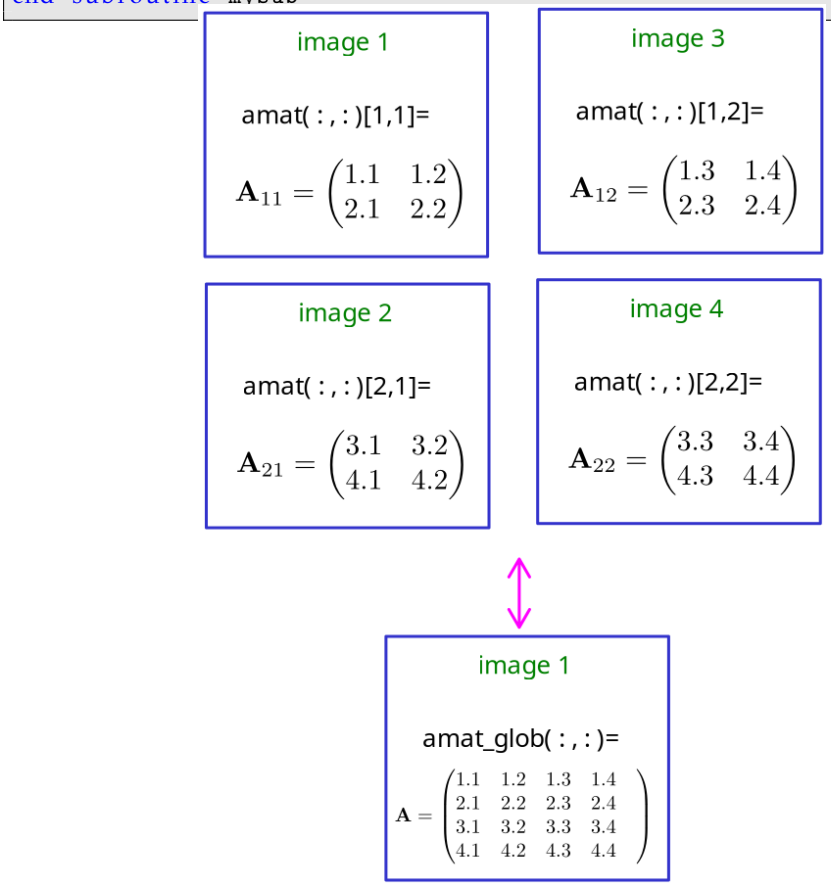



Figure 6.7: Data transfer between local blocks and global array. This figure illustrates the relationship between the distributed blocks ($\mathbf{A}_{IJ}$) stored locally in the coarray `amat` on four images, and the complete global matrix $\mathbf{A}$ (`amat_glob`). The functions `loc2glob` and `glob2loc` manage this data exchange: `loc2glob` collects the local blocks $\mathbf{A}_{IJ}$ = `amat(:,:)`$[I, J]$ from all images to construct the full global matrix $\mathbf{A}$ in `amat_glob` on Image 1. `glob2loc` performs the inverse operation, taking the global matrix $\mathbf{A}$ (available on Image 1) and distributing its constituent blocks back to the local `amat` coarray across all images.

As an example of using coarrays within procedures, we implement two core functions, `loc2glob` and `glob2loc`, to manage the transfer between local distributed data and the global matrix representation.

We consider an $N \times N$ matrix $\mathbf{A}$ distributed across $N_\mathrm{p} \times N_\mathrm{p}$ images, where each image holds a local $N_\mathrm{loc} \times N_\mathrm{loc}$ block $\mathbf{A}_{IJ}$. This local data is stored in the coarray `a(nmat_loc,nmat_loc)[nmat_p,nmat_p]`, where `nmat_loc` $= N_\mathrm{loc}$, and `nmat_p` $= N_\mathrm{p} = N/N_\mathrm{loc}$.

The relationship between the indices of the local blocks and the global matrix elements $\mathbf{A}_{ab}$ is given by:

$$(\mathbf{A}_{IJ})_{pq} = \mathbf{A}_{ab}\,, \quad a = (I-1)N_\mathrm{p} + p, \quad b = (J-1)N_\mathrm{p} + q\,, \tag{6.5}$$

where $a, b = 1, \ldots, N$, $p, q = 1, \ldots, N_\mathrm{loc}$, $I, J = 1, \ldots, N_\mathrm{p}$.

The function `log2glob` takes the distributed coarray `a(N_loc,N_loc)[N_p,N_p]` as input and returns a full global array `a_glob(N,N)`. The reconstructed matrix is meaningful and available only on Image 1:

```
function           loc2glob(a)  result(a_glob)
 real(8), allocatable, intent(in)  :: a     (:,:)[:,:]
 real(8), allocatable              :: a_glob(:,:)
 integer                           :: N,N_loc,N_p
 integer                           :: myimg
 integer                           :: ip, jp, istart, jstart

 myimg = this_image()

 N_loc = size    (a,dim=1)
 N_p   = ucobound(a,dim=1) - lcobound(a,dim=1) + 1
 N     = N_loc * N_p

 allocate(a_glob(N,N))

 sync all ! we need all a(:,:)[:,:] to be synched

 if( myimg == 1 ) then

  do  ip    = 1, N_p
   istart   = (ip-1) * N_loc + 1
   do jp    = 1, N_p
    jstart  = (jp-1) * N_loc +1
    a_glob(istart:istart+N_loc-1,jstart:jstart+N_loc-1) = &
      a(:,:)[ip,jp] !get data from image [ip,jp]
   end do
  end do

 end if

end function       loc2glob
```

The function is designed for modularity: it dynamically calculates the global size ($N$), grid dimension ($N_\mathrm{p}$), and block size ($N_\mathrm{loc}$) from the shape and coshape of the input coarray `a`. The reconstruction of `a_glob(:,:)` is performed by transferring data from the coarray blocks (`a(:,:)[ip,jp]`)

on all images to Image 1. Using the indexing relations of Eq.(6.5), the correct slice `a_glob(istart:istart+N_loc-1,jstart:jstart+N_loc-1)` is targeted to be filled by the data retrieved from the image at coordinates `[ip,jp]`.

The `sync all` statement is required before the data transfer to ensure all images have completed any prior calculations and are ready to be accessed. The structure of this transfer is shown in Figure 6.7

The implementation is shown in the following program, which can be found in the file `coa_03_loc2glob.f90`:

```
program     coarray
 use, intrinsic        :: iso_fortran_env
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p, nimg
 real(dp), allocatable :: amat(:,:)[:,:]
 real(dp), allocatable :: amat_glob(:,:)
 integer               :: myimg,myi,myj
 integer               :: imat,jmat,imat_glob,jmat_glob

 nmat     = 4

 nimg     = num_images() ;
 nmat_p   = int(sqrt(nimg+1.0e-6_dp));
 nmat_loc = nmat/nmat_p

 if(    nimg         /= nmat_p**2) &
  call co_abort("Not correct no. images: nimg /= nmat_p**2")
 if(mod(nmat,nmat_p) /= 0        ) &
  call co_abort("Not correct no. images: mod(nmat,nmat_p) /= 0")

 allocate(amat(nmat_loc,nmat_loc)[nmat_p,*])

 myimg = this_image()
 myi   = this_image(amat,1) ; myj = this_image(amat,2)

 if(myimg == 1) print '(A,3I4      )','00_Nmat:       ',&
  nmat,nmat_loc,nmat_p
 print                '(A,I4,A,8I4)','01_Image:      ',&
  myimg,' :: ',myi,myj,nmat,nmat_loc,nmat_p

!Construct the local matrix
 do concurrent(jmat=1:nmat_loc,imat=1:nmat_loc)
  imat_glob       = (myi-1)*nmat_loc + imat
  jmat_glob       = (myj-1)*nmat_loc + jmat
  amat(imat,jmat) = imat_glob        + jmat_glob/10.0_dp
 end do

 do imat           = 1, nmat_loc
  print '(A,I4,A,100F4.1)','02_amat:       ',&
   myimg,' :: ',amat(imat,:)
 end do
```

```
 amat_glob = loc2glob(amat)

 if(myimg == 1)then
  do imat_glob     = 1, nmat
   print '(A,I4,A,100F4.1)','03_amat_glob: ',&
    myimg,' :: ', amat_glob(imat_glob,:)
  end do
 end if
!...
contains
!...
 function           loc2glob(a)                result(a_glob)
 subroutine         co_abort(errmes)
!...
end program coarray
```

Compile, and run the program with the commands:

```
caf coa_03_loc2glob.f90 -o c
cafrun -n 4 ./c | sort
```

The data transfer can be made in the opposite direction using the function `glob2loc`. This function assumes the full global coarray `a_glob` is available on Image 1, and it distributes the appropriate block of this data to the local array a on every image.

The function `glob2loc` must calculate which block of the global array is required by the current image, and then access that block directly from Image 1:

```
 function           glob2loc(a_glob)           result(a)
  real(dp), allocatable             :: a     (:,:)
  real(dp), allocatable, intent(in) :: a_glob(:,:)[:,:]
  integer                           :: N,N_loc,N_p
  integer                           :: nimg,myimg,myi,myj
  integer                           :: ip, jp, istart, jstart

  nimg  = num_images()
  myimg = this_image()
  myi   = this_image(a_glob,1) ; myj = this_image(a_glob,2)

  N     = size    (a_glob,dim=1)
  N_p   = ucobound(a_glob,dim=1) - lcobound(a_glob,dim=1) + 1
  N_loc = N / N_p

  allocate(a(N_loc,N_loc))

! we need to have a_glob defined in [1,1] before the assignment
  sync all

  istart  = (myi-1) * N_loc + 1
```

```
 jstart  = (myj-1) * N_loc + 1
 a(:,:)  = &                    !get data from image [1,1]
  a_glob(istart:istart+N_loc-1,jstart:jstart+N_loc-1)[1,1]

end function       glob2loc
```

The complete program for performing this broadcast of **A** from Image 1 to all images is found in the file `coa_04_glob2loc.f90`. This entire transfer process is illustrated in Figure 6.7.

### 6.1.5 Matrix Multiplication

Implementing block matrix multiplication using Coarrays is straightforward. The task is to compute the resulting block $\mathbf{C}_{IJ}$ on each image using the summation formula of Eq.(6.1). The operation requires distributed coarrays `amat`, `bmat`, and `cmat`, to hold the matrices **A**, **B**, and **C**. The core loop relies on accessing the required blocks directly from the remote images using coindices:

```
 myi   = this_image(amat,1) ; myj = this_image(amat,2)
 sync all              ! need synched data from amat and bmat

 cmat    = 0.0_dp
 do kp   = 1, nmat_p
  cmat   = cmat + matmul(amat(:,:)[myi,kp],bmat(:,:)[kp,myj])
 end do
```

The parallel matrix multiplication can be cleanly isolated within a function, `pmmult`:

```
function           pmmult(a,b)        result(c)
 real(dp), allocatable, intent(in) :: a(:,:)[:,:]
 real(dp), allocatable, intent(in) :: b(:,:)[:,:]
 real(dp), allocatable             :: c(:,:)
 integer                           :: N,N_loc,N_p
 integer                           :: myi,myj,kp

 N_loc = size    (a,dim=1)
 N_p   = ucobound(a,dim=1) - lcobound(a,dim=1) + 1
 myi   = this_image(a,1) ; myj = this_image(a,2)

 allocate(c(N_loc,N_loc))

 sync all !ensure a and b are computed on all images

 c     = 0.0_dp
 do kp = 1, N_p
  c    = c + matmul(a(:,:)[myi,kp],b(:,:)[kp,myj])
 end do
```

```
  sync all ! Ensure all images finished computing their local c

 end function       pmmult
```

Note that the function returns an array and not a coarray! Therefore, if you want the result to be assigned to a coarray, you have to declare it or allocate it in the calling program before the call to `pmmult`.

The complete program `coa_05_mmult.f90` implements `pmmult` and verifies the result by comparing the parallel block calculation against a serial multiplication of the reconstructed global matrices on Image 1:

```
program     coarray
 use, intrinsic        :: iso_fortran_env
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p, nimg
 real(dp), allocatable :: amat     (:,:)[:,:]
 real(dp), allocatable :: bmat     (:,:)[:,:]
 real(dp), allocatable :: cmat     (:,:)[:,:]
 real(dp), allocatable :: amat_glob(:,:)
 real(dp), allocatable :: bmat_glob(:,:)
 real(dp), allocatable :: cmat_glob(:,:)

 nmat     = 4

 nimg     = num_images() ;
 nmat_p   = int(sqrt(nimg+1.0e-6_dp));
 nmat_loc = nmat/nmat_p

 if(    nimg         /= nmat_p**2) &
  call co_abort("Not correct no. images: nimg /= nmat_p**2"    )
 if(mod(nmat,nmat_p) /= 0        ) &
  call co_abort("Not correct no. images: mod(nmat,nmat_p) /= 0")

 allocate(amat(nmat_loc,nmat_loc)[nmat_p,*])
 allocate(bmat(nmat_loc,nmat_loc)[nmat_p,*])
 allocate(cmat(nmat_loc,nmat_loc)[nmat_p,*])

 call random_number(amat) ; call random_number(bmat)

! cmat must be allocated as coarray to be used
! in loc2glob(cmat), otherwise loc2glob returns
! array and the coarray attribute of cmat is lost
 cmat      = pmmult(amat,bmat)

!Now, test the result:
 amat_glob = loc2glob(amat)
 bmat_glob = loc2glob(bmat)
 cmat_glob = loc2glob(cmat)

 if(this_image() == 1)then
```

```
  print *,'||(a.b)_glob-(a.b)_loc||= ', &
   sum(abs(matmul(amat_glob,bmat_glob)-cmat_glob))/nmat**2
 end if
!-----------------------------------------------------------
contains
!-----------------------------------------------------------
!...
 function           pmmult(a,b)                result(c)
 function           loc2glob(a)                result(a_glob)
 subroutine         co_abort(errmes)
!...
end program coarray
```

### 6.1.6 Transpose and Traces

The computation of the matrix transpose ($\mathbf{A}^T$) is another fundamental matrix operation. Given the block distribution scheme of the matrix (Figure 6.2), the transpose operation is defined by block transposition and exchange: $\mathbf{A}^T_{IJ} = (\mathbf{A}_{JI})^T$ (Eq.(6.3)).

The operation is encapsulated in the function `ptranspose`, which uses coindexing to retrieve the required block ($\mathbf{A}_{JI}$) from the remote image (`myj`,`myi`) and then performs a local transpose:

```
function           ptranspose(a)              result(at)
 real(dp), allocatable, intent(in) :: a(:,:)[:,:]
 real(dp), allocatable             :: at(:,:)
 integer                           :: N,N_loc,N_p
 integer                           :: myi,myj,kp

 N_loc = size(a,dim=1)
 myi   = this_image(a,1) ; myj = this_image(a,2)

 allocate(at(N_loc,N_loc))

 sync all

 at    = transpose(a(:,:)[myj,myi])

 !if at is a coarray, make sure it is computed on all images
 sync all

end function       ptranspose
```

The `sync all` before the assignment ensures the remote data is ready for access, and the final `sync all` ensures the result (`at`) is computed across all images before subsequent use.

The ability to compute $\mathbf{A}^T$ allows for the direct calculation of $\mathrm{tr}\,\mathbf{A}^2$ (Eq.(6.4)). The program `coa_06_trA2.f90` (listed below) calculates $\mathbf{A}^T$,

$\mathrm{tr}\,\mathbf{A}$, and $\mathrm{tr}\,\mathbf{A}^2$, followed by checks against serial computations performed on the full, reconstructed global arrays.

The calculation of $\mathrm{tr}\,\mathbf{A}$ uses $\mathrm{tr}\,\mathbf{A} = \sum_{I=1}^{N_\mathrm{p}} \mathrm{tr}\,\mathbf{A}_{II}$, where only the diagonal blocks ($\mathbf{A}_{II}$) are needed. The co_sum intrinsic aggregates the contributions:

```
program     coarray
 use, intrinsic        :: iso_fortran_env
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p, nimg
 real(dp), allocatable :: amat      (:,:)[:,:]
 real(dp), allocatable :: amatt     (:,:)[:,:]
 real(dp), allocatable ::  amat_glob(:,:)
 real(dp), allocatable :: amatt_glob(:,:)
 real(dp)              :: trA2,trA,trA2_glob,trA_glob
 integer               :: myimg,myi,myj
 integer               :: imat

 nmat     = 4

 nimg     = num_images()
 nmat_p   = int(sqrt(nimg+1.0e-6_dp))
 nmat_loc = nmat/nmat_p

 if(    nimg         /= nmat_p**2) &
  call co_abort("Not correct no. images: nimg /= nmat_p**2"    )
 if(mod(nmat,nmat_p) /= 0        ) &
  call co_abort("Not correct no. images: mod(nmat,nmat_p) /= 0")

 allocate(amat (nmat_loc,nmat_loc)[nmat_p,*])
 allocate(amatt(nmat_loc,nmat_loc)[nmat_p,*])

 myimg  = this_image()
 myi    = this_image(amat,1)
 myj    = this_image(amat,2)

 call  random_number(amat)

 amatt  = ptranspose(amat) ! calculates transpose of amat

 trA    = 0.0_dp
 if(myi == myj)then
  do concurrent(imat=1:nmat_loc)
   trA  = trA + amat(imat,imat)
  end do
 end if

 trA2   = sum(amat * amatt)

 ! All images have trA and trA2:
 call co_sum(trA) ; call co_sum(trA2)
```

```
 print *,'01 tr                = ',myimg,trA,trA2

 ! Check results:
 allocate(amat_glob(nmat,nmat),amatt_glob(nmat,nmat))

 amat_glob  = loc2glob(amat)
 amatt_glob = loc2glob(amatt)

 if(myimg  == 1)then

  print *,'02 ||at.glob-at.loc||= ', &
   sum(abs( transpose(amat_glob) - amatt_glob ))/nmat**2

  trA_glob  = 0.0_dp
  do concurrent (imat=1:nmat)
   trA_glob = trA_glob + amat_glob(imat,imat)
  end do
  print *,'03  |trA_g - trA_l|  = ',abs(trA_glob -trA )/nmat

  trA2_glob = sum( amat_glob * amatt_glob)
  print *,'04  |trA2_g-trA2_l|  = ',abs(trA2_glob-trA2)/nmat

 end if

!-----------------------------------------------------------
contains
!-----------------------------------------------------------
!...
 function           ptranspose(a)              result(at)
 function           loc2glob(a)                result(a_glob)
 subroutine         co_abort(errmes)
!...
end subroutine     co_abort
```

## 6.2 MPI

While CAF provides an elegant, language-integrated approach to parallel programming, the Message Passing Interface (MPI) remains the de facto standard for High-Performance Computing (HPC) applications in Fortran, C, and C++. MPI is not a language but a library specification that defines a rich set of routines for managing processes and enabling communication between them. For an introduction to MPI using Fortran, see [4].

MPI is a standardized, portable, and complete system for writing parallel programs. Unlike Coarrays, which offers a Partitioned Global Address Space (PGAS) model where data can be accessed via coindices (one-sided communication), MPI strictly enforces the Message-Passing paradigm. It is preferred because it provides a message-passing model of parallel computation that is universal, expressive, relatively easy to debug, and can be

used for high performance applications.

In the message-passing model, each running copy of the program, referred to as a *process* (or rank in MPI terminology), has its own distinct and private local memory space. If one process needs data residing in the memory of another process, it must explicitly ask the owning process to *send* the data, and the receiving process must explicitly *receive* it. This communication is achieved through calls to MPI library routines. The underlying hardware performing the data transfer is irrelevant, and the processes can exchange memory by running on the same CPU, different computer racks connected via fast multilevel switched networks, or on different computers connected through the internet.

This structure inherently follows the Single Program Multiple Data (SPMD) model, which forms the foundation of all parallel approaches discussed in this chapter.

The MPI standard provides a Fortran module, typically accessed via `use mpi`, which exposes all necessary routines. Almost all MPI procedures are subroutines and follow a highly standardized interface structure in Fortran:

```
call MPI_Routine_Name(comm, argument_list, ierr)
```

This structure immediately highlights the importance of the Communicator (comm), which is almost always one of the arguments of an MPI routine. The Communicator identifies the group of processes participating and provides a safe communication context. While most initial operations use the predefined `MPI_COMM_WORLD` (representing the universe of all available processes), users can define custom communicators to manage smaller groups of processes, allowing for multiple, independent parallel operations within a single program.

The subsequent arguments, detailed in the following sections, typically include buffers, counts, data types, and identifiers. The final argument, ierr, is an integer status variable used to capture any error codes returned by the MPI library call.

Every MPI program must follow a specific life cycle, characterized by initialization, communication, and finalization.

1. Initialization: The library must be initialized on all processes by calling `MPI_Init`. This routine sets up the communication environment and must be the first MPI routine called.

2. Process Identification: Every process must be able to identify itself and the size of the parallel job.

    - `MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)`: Returns the total number of processes (nprocs) launched for the current job.

- `MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)`: Returns the unique integer identifier (the rank or `myrank`) of the calling process. Ranks are numbered starting from 0 up to $N_{\text{procs}}-1$, where $N_{\text{procs}} =$ `nprocs` is the total number of processes.

3. Communication and Computation: This is the body of the program where all parallel work, computation, and data exchange (message passing) occurs.

4. Finalization: Before exiting, the program must call `MPI_Finalize` on every process to gracefully shut down the MPI environment and clean up internal resources. No MPI routines may be called after finalization.

A minimal MPI program template therefore looks like this:

```
program mpi_hello
  ! The module that provides MPI routines and environment
  use mpi

  integer :: ierr, nprocs, myrank

  ! 1. Initialize MPI
  call mpi_init(ierr)

  ! 2. Get total processes        -> nprocs
  call mpi_comm_size(MPI_COMM_WORLD, nprocs, ierr)
  ! 2. Get my rank                -> myrank  (0 .. Nprocs-1)
  call mpi_comm_rank(MPI_COMM_WORLD, myrank, ierr)

  ! 3. Main parallel work goes here.
  call my_mpi_application

  ! 4. Finalize MPI
  call mpi_finalize(ierr)

end program mpi_hello
```

### 6.2.1 Compiling and Running

Since MPI is a library specification, not an intrinsic part of the Fortran language, programs must be correctly linked to the appropriate MPI libraries during compilation. Because managing the specific compiler flags and linking instructions can be complex, the MPI ecosystem provides compiler wrappers to automate this process.

For the gfortran compiler, the standard wrapper is the command `mpifort`.

The execution of the program, along with instructions specifying the resources to use, depends on the local computing environment. For running programs on a personal computer, the executable is typically launched using the command `mpirun`.

The following Fortran code provides a basic "Hello World!" program (`mpi_01_hello.f90`) that illustrates the essential steps of an MPI application, including initialization, process identification, synchronization, and finalization.

```
use mpi
implicit none

integer :: ierr, nprocs, myrank

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

!All processes synchronize here:
call mpi_barrier  (MPI_COMM_WORLD,ierr)

!There is no standard that sets the order that each
!process will execute the print statement.
!The output can be in any order!
print *,'I am process: ', myrank, ' of ', nprocs

call mpi_finalize(ierr)
```

This program performs the following key functions:

- **Process Identification**: It retrieves the total number of processes (`nprocs`) and the unique rank (`myrank`) for the calling process.
- **Synchronization**:The `MPI_Barrier` call ensures that all participating processes wait at that point until every process has reached the call.
- **Asynchronous Output**: Although a barrier is used, the final `print` statements themselves are asynchronous, meaning there is no guarantee regarding the order in which the output will appear on the standard output stream.

To compile the program, the MPI compiler wrapper `mpifort` must be used:

```
mpifort mpi_01_hello.f90 -o m
```

To run the executable (`m`) using four parallel processes, the launch command `mpirun` is employed, specifying the number of processes with the `-n` flag:

```
mpirun -n 4 ./m | sort
```

The resulting sorted output from the four processes will be:

```
 I am process:   0  of     4
 I am process:   1  of     4
 I am process:   2  of     4
 I am process:   3  of     4
```

As clearly demonstrated by the output, the unique process identifier, `myrank`, is assigned integer values ranging from $0, \ldots, N_{\text{procs}} - 1$.

For actual message passing, processes must issue explicit send and receive instructions to exchange data. Each message must be specified by its *length*, the *type* of data being transferred, the *source* and *destination* processes, and the *communicator* that defines the communication context and process group.

This is the most basic operation in MPI, performed by the following procedures:

```
MPI_Send(address,    count,datatype,destination,tag,&
 comm,        ierr)
MPI_Recv(address,maxcount,datatype,source      ,tag,&
 comm,status,ierr)
```

For `MPI_Send`, the triplet `(address,count,datatype)` describes `count` occurrences of items of the specified `datatype`, starting from memory address `address` (e.g. the name of an array or a scalar variable). `destination` is an integer specifying the rank of the receiving process within the context of the `comm` communicator. `tag` is an integer that uniquely identifies the message within the communicator. Programmers must ensure the tag value is unique for the message being sent.

For `MPI_Recv`, the triplet `(address,count,datatype)` describes up to `maxcount` occurrences of items of type `datatype` to be stored starting from the memory address `address`. It is permitted for the received data to be less than the specified `maxcount`. `source` is an integer specifying the rank of the sending process within the context of the `comm` communicator. `tag` is an integer used to match the message being received. `status` is an integer array that holds essential information about the message actually received, including its actual size, source rank, and tag.

To demonstrate this process, we will write a program that sends the data from an array `real(8) x(2)` on process 0, which is received by process $N_{\text{procs}} - 1$, and stored in the local array `real(8) y(2)`. The code can be found in the file `mpi_02_SendAndRecv.f90`:

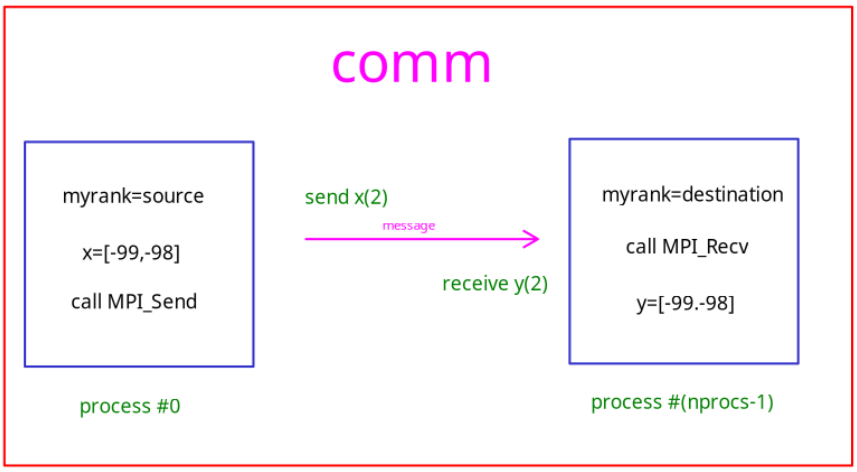



Figure 6.8: Illustration of a Point-to-Point Communication using `MPI_Send` and `MPI_Recv`. Process #0 (`myrank=source`) defines an array `real(8) x(2)` and sends its data as a message, using `MPI_Send`. The message is received by the destination, Process #$(N_{\text{procs}}-1)$ (`myrank=destination`), using `MPI_Recv`, where the data is stored in its local array `real(8) y(2)`. This message exchange occurs within the communication context and process group defined by the communicator `comm`.

```
program     mpi_introduction

 use mpi
 implicit none
 integer, parameter :: dp = 8
 integer            :: ierr, nprocs, myrank
 integer            :: source,destination,tag,count
 real(dp)           :: x(2), y(2)
 integer            :: status(MPI_STATUS_SIZE)

 call mpi_init(ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 source      = 0
 destination = nprocs - 1
 tag         = 100     ! any integer value you want

 if(myrank == source     ) then
  x = [-99.0,-98.0]
  call mpi_send(x , 2, MPI_REAL8,destination,tag,  &
   MPI_COMM_WORLD,ierr)
  print *,'proc ',myrank,                          &
    ': I sent to       ',destination,' x= ',x
 end if
```

```
 if(myrank == destination) then
  call mpi_recv(y , 2, MPI_REAL8,source      ,tag,  &
   MPI_COMM_WORLD,status,ierr)
  print *,'proc ',myrank,                            &
   ': I received from ',source,' y= ',y

!Get the actual size of the message in count
  call mpi_get_count(status,MPI_REAL8,count,ierr)

!Print the actual tag, source, and size of the message:
  print *,'proc ',myrank,                            &
   ': message tag=     ',status(MPI_TAG),            &
   ' source= '          ,status(MPI_SOURCE),          &
   ' count=  ',count
 end if

 call mpi_finalize(ierr)
```

The program begins by defining the source (rank 0) and destination (rank $N_{\text{procs}} - 1$) processes, and sets a message `tag` $= 100$, which uniquely identifies the message within the program.

The process with `myrank` $=$ `source` (rank 0) initializes its local array `x(2)` and executes `MPI_Send` to transmit the data: The `MPI_Send` arguments specify that the message starts at memory location `x`, contains 2 elements of type `real(8)` (specified by the predefined MPI variable `MPI_REAL8`), and is sent to the `destination` process with the chosen `tag`. Both the `source` and `destination` belong to the predefined communicator `MPI_COMM_WORLD`.

The destination process (`myrank=destination=nprocs-1`) executes `MPI_Recv` to capture the message: It specifies that the incoming message should be received from `source` $= 0$ and match the defined `tag` within `MPI_COMM_WORLD`. The call indicates that the process is ready to receive at most 2 elements of type `real(8)`, which will be stored starting at its local memory address `y`.

After a successful `MPI_Recv`, the `status` array contains metadata[1] about the message actually received.

The size of the communication buffer is calculated from `count` and the size of the `datatype` specified in the `MPI_Send` and `MPI_Recv` calls. For basic Fortran types, MPI provides predefined variables that abstract the underlying data representation, such as `MPI_REAL8`.

Table 6.1 lists some of the most important mappings between Fortran types and their corresponding MPI datatypes

[1]The actual tag is stored in `status(MPI_TAG)`. The actual source rank is stored in `status(MPI_SOURCE)`. The actual size (or `count`) of the message must be retrieved separately using the subroutine `MPI_Get_count`. If one is not interested in the actual values, she can use the wildcards `MPI_ANY_SOURCE`, `MPI_ANY_TAG`.

| Fortran Type | Corresponding MPI Datatype |
|---|---|
| LOGICAL | MPI_LOGICAL |
| INTEGER | MPI_INTEGER |
| INTEGER(8) | MPI_INTEGER8 |
| REAL | MPI_REAL |
| REAL(8) | MPI_REAL8 |
| COMPLEX | MPI_COMPLEX |
| COMPLEX(8) | MPI_COMPLEX16 |
| CHARACTER | MPI_CHARACTER |

Table 6.1: Some of the most important MPI data types used in MPI procedures, and their corresponding Fortran type.

To execute the `mpi_02_SendAndRecv.f90` program using 8 processes ($N_{\text{procs}} = 8$), you would use the following commands:

```
mpifort mpi_02_SendAndRecv.f90 -o m
mpirun -n 8 ./m | sort
```

The resulting output, showing the communication between the source (rank 0) and the destination (rank $N_{\text{procs}} - 1 = 7$), is:

```
proc 0: I sent to       7  x= -99.0 -98.0
proc 7: I received from 0  y= -99.0 -98.0
proc 7: message tag= 100 source= 0 count= 2
```

The `MPI_Sendrecv` subroutine is a point-to-point communication primitive designed for the efficient, simultaneous exchange of data between processes. It performs a pair of matching send and receive operations in a single, atomic call.

The process $P$ executing the call simultaneously performs two operations: it sends a message to a designated `sendproc` ($P_s$) and receives a message from a `recvproc` ($P_r$):

$$P_r \xrightarrow{\text{Receive}} P \xrightarrow{\text{Send}} P_s$$

The routine requires two complete sets of arguments to describe both the outbound (send) message and the inbound (receive) message.

The Fortran calling sequence for this routine is:

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,sendproc,sendtag,&
             recvbuf,recvcount,recvtype,recvproc,recvtag,&
             comm   ,status   ,ierr)
```

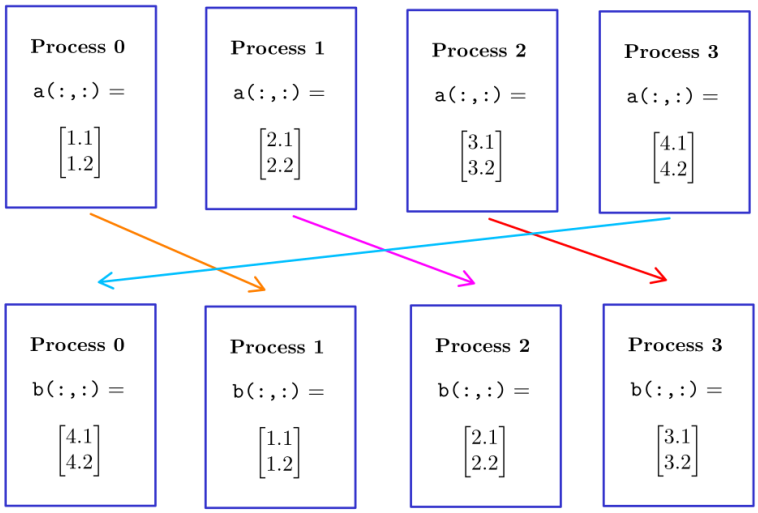



Figure 6.9: Circular Data Exchange using the `MPI_Sendrecv` routine. The local array `a(2)` is initialized independently on each process. The subsequent `MPI_Sendrecv` call, results in a simultaneous exchange: each process sends its data to the process immediately to its right and receives data from the process immediately to its left (in a circular topology). The received message is stored in the local array `b(2)` on the destination process. For example, Process 1 sends the data, `a=[2.1,2.2]`, to Process 2, and receives the data `b=[1.1,1.2]`, from Process 0.

Care must be taken to ensure that the designated partner processes ($P_s$ and $P_r$) execute the matching receive and send calls, respectively, to prevent the program from hanging (deadlock).

While a simple two-process data exchange is possible (as shown in `mpi_02_Sendrecv2Point.f90`), the program `mpi_02_Sendrecv.f90` listed below demonstrates a more interesting and practical pattern: a circular shift involving all participating processes, as illustrated in Figure 6.9.

In this scenario:

- Each process initializes its local array `a(2)`.
- The process sends its `a(2)` data to the neighbor process immediately to its right in a circular topology.
- Simultaneously, it receives data from the neighbor process immediately to its left, storing the incoming message in its local array `b(2)`

This pattern efficiently exchanges data between immediate neighbors using the single, atomic `MPI_Sendrecv` call.

```
program     mpi_introduction

 use mpi
 implicit none
 integer, parameter :: dp = 8

 integer  :: ierr, nprocs, myrank, tag
 integer  :: send_to, recv_from
 real(dp) :: a(2), b(2)
 integer  :: status(MPI_STATUS_SIZE)

 call mpi_init(ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 !Determine communication partners:
 !Exchange with immediate neighbor
 send_to    = mod(myrank + 1         , nprocs)
 recv_from  = mod(myrank - 1 + nprocs, nprocs)
 tag        = 10

 a = [ myrank + 1.1_dp,myrank + 1.2_dp]
 b = 0.0_dp

 print '(A,I3,A,2F5.1)','00 Initial: Rank ',myrank,': A= ',a

 call mpi_sendrecv(                    &
      ! Send    arguments (Sending        my 'a' array)
      a, 2, MPI_REAL8, send_to  , tag, &
      ! Receive arguments (Receiving into my 'b' array)
      b, 2, MPI_REAL8, recv_from, tag, &
      MPI_COMM_WORLD, status, ierr)

 print '(A,I3,A,2F5.1)','01 Final:   Rank ',myrank,': B= ',b

 call mpi_finalize(ierr)

end program mpi_introduction
```

Running the program using 4 processes produces the (sorted) output:

```
00 Initial: Rank   0: A=   1.1  1.2
00 Initial: Rank   1: A=   2.1  2.2
00 Initial: Rank   2: A=   3.1  3.2
00 Initial: Rank   3: A=   4.1  4.2

01 Final:   Rank   0: B=   4.1  4.2
01 Final:   Rank   1: B=   1.1  1.2
01 Final:   Rank   2: B=   2.1  2.2
01 Final:   Rank   3: B=   3.1  3.2
```

### 6.2.2 Timing your Program

Timing is crucial for analyzing program performance and for adhering to strict time limits common in supercomputing environments. The relevant metric for performance analysis is the wall time, which is the elapsed physical time of the program's execution on each individual process.

MPI provides a convenient and portable mechanism for obtaining this measurement through the function `MPI_Wtime`.

The function `MPI_Wtime` measures the elapsed physical time since an arbitrary reference point. It is important to remember that each process maintains its own independent timer, meaning only time differences calculated within the same process are meaningful. The function returns a `real(8)` value.

The standard usage pattern for measuring an elapsed interval is:

```
real(8) t_start, t_now
...
t_start = MPI_Wtime()
... work done here ...
t_now   = MPI_Wtime()
print *,'Time elapsed on',myrank,' is ',t_now-t_start,' sec'
...
```

The time resolution (or precision) of the timer can be determined by calling the companion function, `MPI_Wtick`.

The program `mpi_03_Timing.f90`, demonstrates how to implement a graceful, timed shutdown in a parallel job, a feature particularly useful when running on resource-constrained HPC queues.

The objective is to allow all processes to work until a single designated process (rank 0) determines that the maximum allowed time (`wtime_run`) has been exceeded, at which point all processes must terminate simultaneously:

- **Time Initialization**: Each process sets the maximum execution limit (`wtime_run=5.0_dp`) and records its individual start time (`wtime_start`) using `MPI_Wtime()`.
- **Work Loop**: The program enters an infinite do loop where work is simulated by a one-second pause (`call sleep(1)`).
- **Timeout Check (Rank 0)**: Only Process 0 checks the elapsed time (`wtime_now - wtime_start`) against the time limit (`wtime_run`) and sets the logical variable `timeout` to `.true.` if the limit is exceeded.
- **Collective Broadcast and Synchronization**: The key to synchronized termination is the `MPI_Bcast` call:

```
call mpi_bcast(timeout, 1, MPI_LOGICAL, 0,      &
               MPI_COMM_WORLD, ierr)
```

This routine broadcasts the `timeout` status (1 item of type `MPI_LOGICAL`) from the root process (rank 0) to all other processes in `MPI_COMM_WORLD`. `MPI_Bcast` acts as an implicit barrier; all processes stop and wait until the broadcast is complete. After the call, every process has the same termination status.

- **Graceful Exit**: If `timeout` is `.true.`, all processes simultaneously hit the `exit` statement and proceed to `MPI_Finalize`.

The program's time accuracy in stopping depends on the duration of the work done in the loop. In this demonstration, the work duration is 1 second (`call sleep(1)`). If the simulated work load were longer, the time accuracy of the forced shutdown would be correspondingly worse. Additionally, since each process measures time independently, the elapsed time reported by each process upon exit can vary slightly, even though they exit simultaneously due to the synchronization.

```
program     mpi_introduction

 use mpi
 implicit none

 integer , parameter :: dp = 8
 integer             :: ierr, nprocs, myrank
 real(dp)            :: wtime_start, wtime_now,wtime_run
 integer             :: step_counter
 logical             :: timeout

 call mpi_init(ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 wtime_run    = 5.0_dp  !time limit

 wtime_start  = mpi_wtime()

 step_counter = 0
 do             !Infinite loop

  call sleep(1)  !Simulates the real work of each process
  step_counter = step_counter + 1

  if(myrank == 0) then
   wtime_now = mpi_wtime()
   !Have we exceeded the time limit on process 0?
   timeout   = (wtime_now - wtime_start) > wtime_run
```

```
  end if

  !. timeout's value is broadcasted from process 0 to
  !  all processes. After the call, all processes have the
  !  same value stored in timeout
  !. All processes synchronize here: mpi_bcast is an
  !  effective barrier where all processes stop and wait.
  !. This is why step_counter is the same on all processes
  call mpi_bcast(timeout, 1, MPI_LOGICAL, 0,    &
   MPI_COMM_WORLD, ierr)

  if(timeout)then
   !Print the elapsed time on each process: can be different!
   print *,myrank,step_counter,                  &
    mpi_wtime()-wtime_start,mpi_wtick()
   exit ! finish work, exit the loop
  end if

 end do

 call mpi_finalize(ierr)

end program mpi_introduction
```

### 6.2.3 Collective Operations

While the elementary functions `MPI_Send` and `MPI_Recv` are sufficient to program any communication pattern, complex tasks –such as distributing an entire matrix or calculating global sums (known as collective operations)– involve multiple point-to-point operations that are both tedious to program and prone to errors. MPI provides a highly efficient suite of collective operation functions.

These specialized functions offer several advantages over implementing communication using only elementary routines. They allow the MPI implementation to select highly-optimized algorithms, often leveraging non-trivial network topologies, which would otherwise be impossible. And they simplify complex programming patterns.

A critical requirement is that all processes within the specified communicator must call the collective function, and they must use matching arguments for the operation to complete successfully.

The essential collective operations are typically categorized into three main types: Synchronization (like `MPI_Barrier`), data movement (like `MPI_Bcast`), and collective computation ("reduction") (like `MPI_Reduce`).

The `MPI_Bcast` (Broadcast) collective operation is the optimized method for distributing identical data from one specific process (the root) to all other processes within a communicator. Figure 6.10 shows an example of the flow of data from the root process to all other processes.

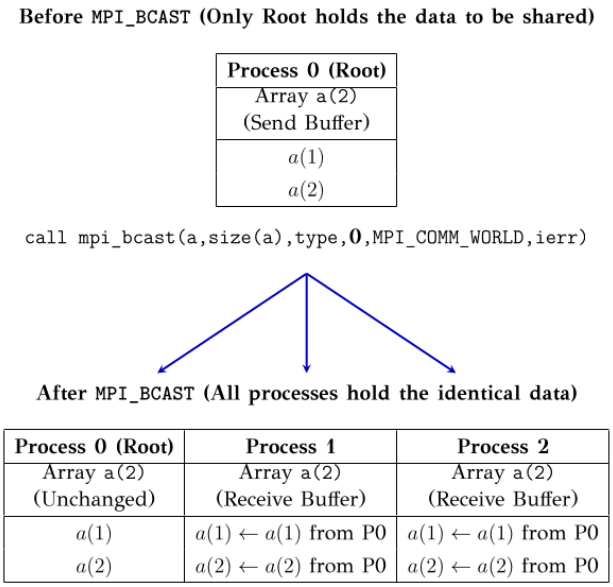



Figure 6.10: Illustration of the `MPI_Bcast` (Broadcast) collective operation. The array `a(2)` is sent from the designated root process (here, Rank 0) to all other processes within the communicator. The array `a` on the root acts as the source buffer, and on all non-root processes, it acts as the destination buffer, receiving an identical copy of the data.

This task, while achievable with multiple point-to-point calls, is made highly efficient by `MPI_Bcast`, which employs optimized communication patterns (e.g., binary trees) to significantly reduce latency and increase scalability. This makes it essential for initializing large-scale simulations. Its primary usage is to transmit global parameters, such as the matrix dimension `nmat` or a logical control flag like `timeout` (as seen in `mpi_03_timeout.f90`), ensuring every parallel task operates with a consistent set of initial values. When executed, the root process acts as both the source and the destination buffer.

The full Fortran calling sequence for the routine is:

```
call mpi_bcast(buffer,count,datatype,root,comm,ierr)
```

The argument `buffer` is a scalar, array, or any other MPI type. `count` is an integer specifying the number of elements of `datatype` contained in `buffer` to be sent from the `root` process to all other processes in the communicator `comm`. `root` is an integer specifying the root process that is sending data to all other processes. This value must be identical on all processes calling the subroutine, and can be any of the processes in `comm`.

An example demonstrating the usage of `MPI_Bcast` is in the file `mpi_04_Bcast.f90`,

and is listed below:

```
program     mpi_introduction

 use mpi
 implicit none

 integer , parameter :: dp = 8
 integer             :: ierr, nprocs, myrank
 integer             :: nmat
 real(dp)            :: a(2)

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat  = 8   ; a = -1.0_dp  !initialize on all processes
 if(myrank == 0)then
  nmat = 128 ; a = [-99.0_dp,-98.0_dp]
 end if

 print *,'01 Before Bcast: ',myrank,nmat,a

 call mpi_bcast(nmat,1       ,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
 call mpi_bcast(a   ,size(a),MPI_REAL8   ,0,MPI_COMM_WORLD,ierr)

 print *,'02 After  Bcast: ',myrank,nmat,a

 call mpi_finalize(ierr)

end program mpi_introduction
```

The output of the program, when run on 4 processes, is:

```
01 Before Bcast: 0 128  -99.0 -98.0
01 Before Bcast: 1   8  -1.00 -1.00
01 Before Bcast: 2   8  -1.00 -1.00
01 Before Bcast: 3   8  -1.00 -1.00
02 After  Bcast: 0 128  -99.0 -98.0
02 After  Bcast: 1 128  -99.0 -98.0
02 After  Bcast: 2 128  -99.0 -98.0
02 After  Bcast: 3 128  -99.0 -98.0
```

The `MPI_Scatter` collective operation is the canonical method for initiating data decomposition. Its core function is to take a global array of data residing entirely on the root process and distribute contiguous, equal-sized chunks of that data to every process in the communicator.

`MPI_Scatter` is a symmetric collective call, meaning every process must execute it, but the role of the arguments changes significantly depending on whether the process is the root (sender) or a receiver. The Fortran

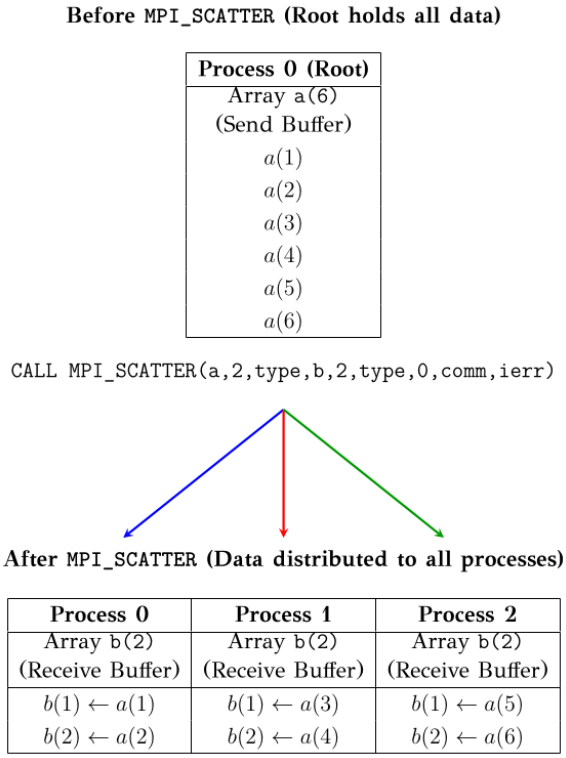



Figure 6.11: Illustration of the `MPI_Scatter` collective operation. The root process (Rank 0) takes its initial global array `a(6)` and distributes it in contiguous blocks of equal size (two elements each). Each of the three participating processes receives one of these two-element blocks, storing it in its local array `b(2)`. The total number of elements sent must equal the sum of elements received by all processes (in this example, $6 = 3 \times 2$).

calling sequence for this routine is:

```
call mpi_scatter(                &
  sendbuffer,sendcount,sendtype, &
  recvbuffer,recvcount,recvtype, &
  root,comm,ierr)
```

`sendbuffer` is the global array, defined on the `root` process, containing the data to be distributed to all processes. `sendcount` is the number of elements of type `sendtype` that will be sent to each individual process. This value is typically equal to `sendcount` = `size(sendbuffer)/nprocs` and is not the total size of the global array. The size of `sendbuffer` must be at least `sendcount` × `nprocs`. `sendbuffer` is ignored on the non-root processes (therefore it is not necessary to allocate and/or initialize it).

`recvbuffer` is the local array on each process in the communicator

comm where the received chunk of sendbuffer will be stored. The size of recvbuffer must be at least recvcount[2].

An example demonstrating the usage of MPI_Scatter is in the file mpi_05_Scatter.f90, and is listed below:

```
program     mpi_introduction

 use mpi
 implicit none

 integer , parameter :: dp = 8
 integer             :: ierr, nprocs, myrank
 integer             :: nmat_loc,nmat, i
 real(dp),allocatable :: a(:),b(:)

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat_loc = 2;
 nmat     = nmat_loc*nprocs

 allocate(a(nmat),b(nmat_loc))

 a = -99.0_dp ; b = -99.0_dp

 if(myrank == 0)then
  a = [(10.0_dp *i, i = 1,nmat)]
 end if

 print '(A,I2,20F6.1)','00 Before scatter: a= ',myrank,a
 print '(A,I2,20F6.1)','02 Before scatter: b= ',myrank,b

 call mpi_scatter(          &
      a,nmat_loc,MPI_REAL8, & !send nmat_loc to EACH process
      b,nmat_loc,MPI_REAL8, & !receive nmat_loc elements
      0,                    & !root process
      MPI_COMM_WORLD,ierr)

 print '(A,I2,20F6.1)','01 After  scatter: a= ',myrank,a
 print '(A,I2,20F6.1)','03 After  scatter: b= ',myrank,b

 call mpi_finalize(ierr)

end program mpi_introduction
```

[2] sendcount = recvcount, so technically one of the arguments is redundant. But the MPI standard keeps this syntax for all MPI_Sendrecv, MPI_Scatter, and MPI_Gather, for consistency and clarity. There are other variable count collective operations, like MPI_Scatterv and MPI_Gatherv, where the root process sends/receives blocks of data of different sizes from the other processes.

When you run the above program on 3 processes, the (sorted) output is:

```
00 Before scatter: a=  0  10.0  20.0  30.0  40.0  50.0  60.0
00 Before scatter: a=  1 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
00 Before scatter: a=  2 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0

01 After  scatter: a=  0  10.0  20.0  30.0  40.0  50.0  60.0
01 After  scatter: a=  1 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
01 After  scatter: a=  2 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0

02 Before scatter: b=  0 -99.0 -99.0
02 Before scatter: b=  1 -99.0 -99.0
02 Before scatter: b=  2 -99.0 -99.0

03 After  scatter: b=  0  10.0  20.0
03 After  scatter: b=  1  30.0  40.0
03 After  scatter: b=  2  50.0  60.0
```

The `MPI_Gather` subroutine implements the collective communication that is the reverse of `MPI_Scatter`. Its function is to reassemble distributed data back into a single, cohesive array on the designated root process. After each parallel process completes its local computation on its chunk of data, `MPI_Gather` collects the local results from all participating ranks and concatenates them sequentially into the large global array on the root. This data flow is illustrated in Figure 6.12.

Unlike `MPI_Scatter`, where the global array is the send buffer, in `MPI_Gather` the global array acts as the receive buffer. As with `MPI_Scatter`, the resulting global array is meaningful only on the root process. The Fortran calling sequence for this routine is:

```
call mpi_gather(                  &
  sendbuffer,sendcount,sendtype, &
  recvbuffer,recvcount,recvtype, &
  root,comm,ierr)
```

`sendbuffer` is the *local* array, defined on all processes in the communicator `comm`, containing the data to be sent to the `root` process. The size of `sendbuffer` must be at least `sendcount`. `sendcount` is the number of elements of type `sendtype` that will be sent from each process to the root.

`recvbuffer` is the global array, which contains useful data only on the `root` process after the call. The size of `recvbuffer` must be at least `recvcount` $\times$ `nprocs`. `recvcount` is the number of elements that the `root` process expects to receive from each process. This is typically equal to `recvcount` = `size(recvbuffer)/nprocs`.

An example demonstrating the usage of `MPI_Gather` is in the file `mpi_06_Gather.f90`, and is listed below:

**Before `MPI_GATHER` (All processes hold local data)**

| Process 0 | Process 1 | Process 2 |
|---|---|---|
| Array `b(2)` (Send Buffer) | Array `b(2)` (Send Buffer) | Array `b(2)` (Send Buffer) |
| $b(1)$ | $b(1)$ | $b(1)$ |
| $b(2)$ | $b(2)$ | $b(2)$ |

`CALL MPI_GATHER(b,2,type,a,2,type,0,comm,ierr)`

**After `MPI_GATHER` (Root holds all aggregated data)**

| Process 0 (Root) |
|---|
| Array `a(6)` (Receive Buffer) |
| $a(1) \leftarrow b(1)$ from P0 |
| $a(2) \leftarrow b(2)$ from P0 |
| $a(3) \leftarrow b(1)$ from P1 |
| $a(4) \leftarrow b(2)$ from P1 |
| $a(5) \leftarrow b(1)$ from P2 |
| $a(6) \leftarrow b(2)$ from P2 |

Figure 6.12: Illustration of the `MPI_Gather` collective operation. Every participating process sends its local array `b(2)` (the send buffer) to the root process (Rank 0). The root collects and concatenates these blocks sequentially, arranged by the rank of the sender (P0, P1, P2), to assemble the complete global array `a(6)` (the receive buffer). The total number of elements received by the root must equal the sum of the elements sent by all processes (in this example, $6 = 3 \times 2$).

```
program     mpi_introduction

 use mpi
 implicit none

 integer , parameter :: dp = 8
 integer             :: ierr, nprocs, myrank
 integer             :: nmat_loc,nmat, i
 real(dp),allocatable :: a(:),b(:)

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat_loc = 2;
```

```
 nmat     = nmat_loc*nprocs

 allocate(a(nmat),b(nmat_loc))

 a = -99.0_dp ;

 b = [((myrank+1) * 10.0_dp + i, i = 1, nmat_loc)]

 print '(A,I2,20F6.1)','02 Before gather: a= ',myrank,a
 print '(A,I2,20F6.1)','00 Before gather: b= ',myrank,b

 call mpi_gather(            &
      b,nmat_loc,MPI_REAL8,  & !send nmat_loc to root
      a,nmat_loc,MPI_REAL8,  & !recv nmat_loc from all procs
      0,                     & !root process
      MPI_COMM_WORLD,ierr)

 print '(A,I2,20F6.1)','03 After  gather: a= ',myrank,a
 print '(A,I2,20F6.1)','01 After  gather: b= ',myrank,b

 call mpi_finalize(ierr)

end program mpi_introduction
```

When you run the above program on 3 processes, the (sorted) output is:

```
00 Before gather: b=  0  11.0  12.0
00 Before gather: b=  1  21.0  22.0
00 Before gather: b=  2  31.0  32.0

01 After  gather: b=  0  11.0  12.0
01 After  gather: b=  1  21.0  22.0
01 After  gather: b=  2  31.0  32.0

02 Before gather: a=  0 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
02 Before gather: a=  1 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
02 Before gather: a=  2 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0

03 After  gather: a=  0  11.0  12.0  21.0  22.0  31.0  32.0
03 After  gather: a=  1 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
03 After  gather: a=  2 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
```

When the globally assembled data from an `MPI_Gather` operation is required by all processes, it would typically necessitate a second collective call, specifically an `MPI_Bcast`, to distribute the final result from the root.

The `MPI_Allgather` routine efficiently combines these two steps by performing the aggregation and distribution simultaneously. It computes a global result and then efficiently deposits the final, complete result into the receive buffer of every process in the communicator. This is equivalent to an `MPI_Gather` followed by an `MPI_Bcast`, but often executed far more efficiently. The operation is depicted in Figure 6.13.

The Fortran calling sequence for `MPI_Allgather` is nearly identical to `MPI_Gather`, but with the key exception that the root argument is no longer present:

```
call mpi_allgather(              &
  sendbuffer,sendcount,sendtype, &
  recvbuffer,recvcount,recvtype, &
  comm,ierr)
```

An example demonstrating the usage of `MPI_Gather` is in the file `mpi_07_Allgather.f90`. The only difference from the program in `mpi_06_Gather.f90` is the name of the subroutine `call mpi_allgather(...)`, and that the `root` argument has been removed. Running the program on 3 processes produces the (sorted) output:

```
00 Before gather: b=  0  11.0  12.0
00 Before gather: b=  1  21.0  22.0
00 Before gather: b=  2  31.0  32.0

01 After  gather: b=  0  11.0  12.0
01 After  gather: b=  1  21.0  22.0
01 After  gather: b=  2  31.0  32.0

02 Before gather: a=  0 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
02 Before gather: a=  1 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0
02 Before gather: a=  2 -99.0 -99.0 -99.0 -99.0 -99.0 -99.0

03 After  gather: a=  0  11.0  12.0  21.0  22.0  31.0  32.0
03 After  gather: a=  1  11.0  12.0  21.0  22.0  31.0  32.0
03 After  gather: a=  2  11.0  12.0  21.0  22.0  31.0  32.0
```

We see that the global array `a(nmat)` has the same data on all processes.

The `MPI_Reduce` collective operation is fundamental for aggregating distributed data to compute a single, global result (a reduction) and making that result available only on the designated root process. It is indispensable for tasks such as calculating the global sum of a scalar or array elements, or finding the global maximum/minimum value across all parallel processes. The operation simultaneously applies a specified associative and commutative mathematical operation (`OP`) element-wise to the local data (`sendbuffer`) contributed by every process. An example, applying the operation `MPI_SUM` on a scalar variable, is depicted in Figure 6.14.

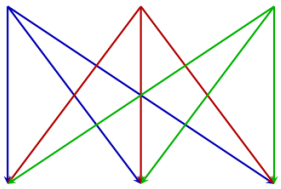
The Fortran calling sequence for this routine is:

```
call mpi_reduce(        &
 sendbuffer,recvbuffer, &
 count,     datatype,   &
 OP,                    & ! reduction operation e.g. MPI_SUM
```

**Before `MPI_ALLGATHER` (All processes hold local data)**

| Process 0 | Process 1 | Process 2 |
|---|---|---|
| Array `b(2)` (Send Buffer) | Array `b(2)` (Send Buffer) | Array `b(2)` (Send Buffer) |
| $b(1)$ | $b(1)$ | $b(1)$ |
| $b(2)$ | $b(2)$ | $b(2)$ |

`CALL MPI_ALLGATHER(b,2,type,a,2,type,comm,ierr)`



**After `MPI_ALLGATHER`**
**(All processes hold the complete aggregated data)**

| Process 0 | Process 1 | Process 2 |
|---|---|---|
| Array `a(6)` (Receive Buffer) | Array `a(6)` (Receive Buffer) | Array `a(6)` (Receive Buffer) |
| $a(1) \leftarrow b(1)$ from P0 | $a(1) \leftarrow b(1)$ from P0 | $a(1) \leftarrow b(1)$ from P0 |
| $a(2) \leftarrow b(2)$ from P0 | $a(2) \leftarrow b(2)$ from P0 | $a(2) \leftarrow b(2)$ from P0 |
| $a(3) \leftarrow b(1)$ from P1 | $a(3) \leftarrow b(1)$ from P1 | $a(3) \leftarrow b(1)$ from P1 |
| $a(4) \leftarrow b(2)$ from P1 | $a(4) \leftarrow b(2)$ from P1 | $a(4) \leftarrow b(2)$ from P1 |
| $a(5) \leftarrow b(1)$ from P2 | $a(5) \leftarrow b(1)$ from P2 | $a(5) \leftarrow b(1)$ from P2 |
| $a(6) \leftarrow b(2)$ from P2 | $a(6) \leftarrow b(2)$ from P2 | $a(6) \leftarrow b(2)$ from P2 |

Figure 6.13: Illustration of the `MPI_Allgather` collective operation. Every process sends its local array `b(2)` to all other processes (including itself). As a result, all processes receive the identical, complete, and globally assembled array `a(6)`, with the contributions ordered by the rank of the sending process (P0, P1, P2). This operation is functionally equivalent to performing an `MPI_Gather` followed by an `MPI_Bcast` of the result, but is executed in a single, optimized step.

```
root,                    &
comm, ierr)
```

`sendbuffer` is a local scalar variable or an array upon which the operation `OP` will be applied. It must be identical in type and shape on every process; if it is an array, `OP` is applied element-wise (elemental procedure). `recvbuffer` is an object similar in kind to `sendbuffer` and, on the `root` process, it receives the final result of the collective operation. The contents of `sendbuffer` are undefined on non-root processes. Examples of reduction operators `OP` include `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, and `MPI_MIN`, described in Table 6.3.

An example demonstrating the usage of `MPI_Reduce` is in the file `mpi_08_Reduce.f90`, and is listed below:

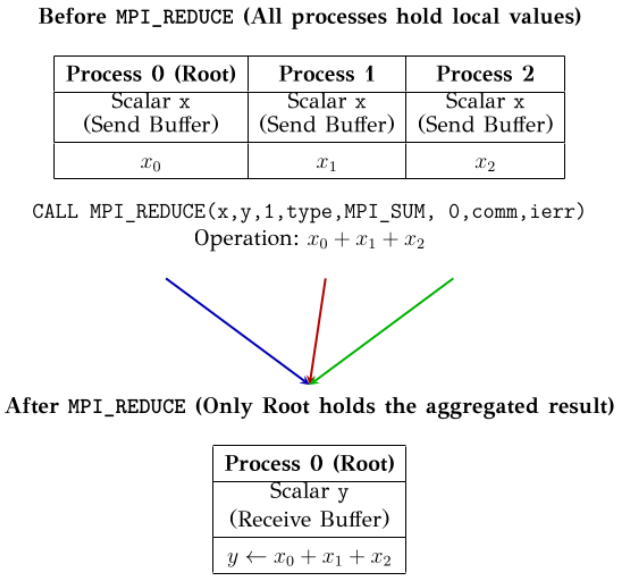



Figure 6.14: Illustration of the `MPI_Reduce` collective operation with `MPI_SUM`. Each participating process sends its local data (`x`, e.g., $x_i$) to the designated root process (Rank 0). The MPI library performs the specified associative and commutative operation ($\sum x_i$) on these values, and the final aggregated result is stored exclusively in the root's designated receive buffer (`y`).

```
program     mpi_introduction

 use mpi
 implicit none

 integer , parameter :: dp = 8, n=2
 integer             :: ierr, nprocs, myrank
 integer             :: nmat_loc,nmat, i
 real(dp)            :: x    ,x_sum
 real(dp)            :: a(n),a_sum(n)

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 x =     (myrank+1) * 10.0_dp
 a = [ ( (myrank+1) * 10.0_dp + i  , i = 1,n) ]

 print '(A,I2,10F6.1)','00     x= ',myrank,x
 print '(A,I2,10F6.1)','02     a= ',myrank,a

 call mpi_reduce(    &
      x,x_sum,       & !send+recv buffers
```

| Operator | Description |
|---|---|
| MPI_SUM | Sums the element values across all processes. |
| MPI_PROD | Multiplies the element values across all processes. |
| MPI_MAX | Returns the element-wise maximum value across all processes. |
| MPI_MIN | Returns the element-wise minimum value across all processes. |

Table 6.2: Some predefined MPI reduction operators for collective computations (OP).

```
      1,MPI_REAL8,    & !count and type
      MPI_SUM,        & !operation on data
      0,              & !root process: receives result
      MPI_COMM_WORLD,ierr)

 call mpi_reduce(     &
      a,a_sum,        & !send+recv buffers
      n,MPI_REAL8,    & !count and type
      MPI_SUM,        & !operation on data
      0,              & !root process: receives result
      MPI_COMM_WORLD,ierr)

 if(myrank == 0)then
  print '(A,I2,10F6.1)','01 sum x= ',myrank,x_sum
  print '(A,I2,10F6.1)','03 sum a= ',myrank,a_sum
 end if

 call mpi_finalize(ierr)

end program mpi_introduction
```

Running the program on 4 processes produces the (sorted) output:

```
00      x=  0  10.0
00      x=  1  20.0
00      x=  2  30.0
00      x=  3  40.0
01 sum x=  0 100.0

02      a=  0  11.0  12.0
02      a=  1  21.0  22.0
02      a=  2  31.0  32.0
02      a=  3  41.0  42.0
03 sum a=  0 104.0 108.0
```

The MPI_Allreduce collective operation efficiently combines the aggregation function of MPI_Reduce with the distribution function of MPI_Bcast in a single, highly optimized call. It is used to simultaneously compute a global result across all contributing processes and then efficiently de-

posit the final, identical result into the receive buffer of every process in the communicator. An example, applying the `MPI_SUM` operation to scalar variables, is depicted in Figure 6.15.

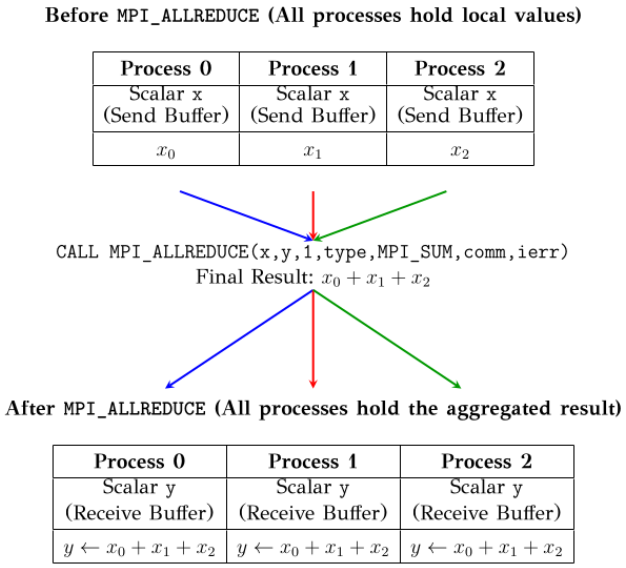



Figure 6.15: Illustration of the `MPI_Allreduce` collective operation with the `MPI_SUM` operator. This routine simultaneously performs a reduction (e.g., summation) across all processes and stores the final aggregated result in the receive buffer (`y`) of every participating process. Functionally, it achieves the same result as an `MPI_Reduce` followed by an `MPI_Bcast`, but it is typically executed much more efficiently.

The Fortran calling sequence for this routine is:

```
call mpi_allreduce(      &
 sendbuffer,recvbuffer, &
 count,      datatype,   &
 OP,                     & ! reduction operation e.g. MPI_SUM
 comm, ierr)
```

which is identical to the `MPI_Reduce`, except that `root` is not an argument, since it makes no sense in this case. An example demonstrating the usage of `MPI_Allreduce` is in the file `mpi_09_Allreduce.f90`.

### 6.2.4 Broadcasting an Array

As a practical exercise in utilizing collective operations, we will implement the two main data management tasks for an $N \times N$ matrix $\mathbf{A}$ distributed into $N_{\text{loc}} \times N_{\text{loc}}$ blocks $\mathbf{A}_{IJ}$ on each process (see Figure 6.2, and Eq.(6.5)).

We will start by programming the construction process in the subroutine `loc2glob`, mirroring the function of its Coarrays counterpart. This subroutine accepts the local matrix block `a(N_loc,N_loc)` (which holds $\mathbf{A}_{IJ}$) as input, and computes and returns the complete global matrix `a_glob(N,N)` on all processes, using `MPI_Allgather` for the data transfer.

The primary subtle point is that `MPI_Allgather` packs data linearly, meaning the elements of `a(N_loc,N_loc)` must be rearranged from their linear, packed order in the intermediate receiving buffer (`recvbuf`) back into the correct 2D block positions within the final `a_glob(N,N)` array. Note the use of the Fortran `reshape` function.

The implementation of `loc2glob` is designed to be as modular as possible, relying only on the exposed MPI routines:

```
subroutine          loc2glob(a, a_glob)
 real(dp), allocatable, intent(in) :: a      (:,:)
 real(dp), allocatable             :: a_glob (:,:)
 real(dp), allocatable             :: recvbuf(:  )
 integer                           :: nprocs, myrank
 integer                           :: N,N_loc,N_loc2,N_p
 integer                           :: myp, myi, myj
 integer                           :: istart, jstart

 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 N     = size(a_glob,1)
 N_loc = size(a,1)        ; N_loc2 = N_loc * N_loc
 N_p   = N/N_loc

 !Buffer that will receive the gathered data
 allocate(recvbuf(nprocs * N_loc * N_loc))

 !Gather all local matrices into recvbuffer
 !on each process:
 call mpi_allgather(                  &
      a          , N_loc2, MPI_REAL8, &
      recvbuf    , N_loc2, MPI_REAL8, &
      MPI_COMM_WORLD, ierr)

 !Redistribute the data as intended in the global matrix:
 do myp   = 0, nprocs - 1
   !Grid coordinates (1-based) for the process myp
   myi    = mod(myp , N_p) + 1 ! Row    index (1 to N_p)
   myj    =    (myp / N_p) + 1 ! Column index (1 to N_p)

   ! Calculate the 1-based indices for the global matrix block
   istart = (myi - 1) * N_loc + 1
   jstart = (myj - 1) * N_loc + 1
```

```
   a_glob(istart:istart+N_loc-1,jstart:jstart+N_loc-1) = &
    reshape(recvbuf(myp*N_loc2+1:(myp+1)*N_loc2), [N_loc,N_loc])

  end do

 end subroutine     loc2glob
```

The subroutine `loc2glob` functions similarly to the function `loc2glob`, shown on page 201. The difference is that it distributes `a_glob(N,N)` to all processes. An example of its usage can be found in the file `mpi_10_loc2glob.f90`:

```
program     mpi_introduction

 use, intrinsic        :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p
 integer               :: ierr, nprocs, myrank, myi, myj
 integer               :: imat, jmat, imat_glob, jmat_glob
 real(dp), allocatable :: amat(:,:),amat_glob(:,:)

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 !Global matrix dimension
 nmat     = 4
 !Number of processes per row/col
 nmat_p   = int(sqrt(nprocs+1.0e-6_dp))
 nmat_loc = nmat/nmat_p              !Local matrix dimension
 myi      = mod(myrank , nmat_p) + 1 !myrank coordinates
 myj      =     myrank / nmat_p  + 1 !in process grid

 if(nmat_p * nmat_p /= nprocs) &
  call abort('Error: nmat_p * nmat_p /= nprocs')
 if(mod(nmat,nmat_p)/= 0     ) &
  call abort('Error: mod(nmat,nmat_p)/= 0'     )

 if(myrank == 0) print '(A,I0.2,A,3I4)',&
  '00_Nmat:      ',myrank,' :: ',nmat,nmat_loc,nmat_p
 print                '(A,I0.2,A,8I4)',&
  '01_Proc:      ',myrank,' :: ',myi,myj,nmat,nmat_loc,nmat_p

 allocate(amat(nmat_loc,nmat_loc))

!Construct the local matrix amat:
 do concurrent(jmat=1:nmat_loc,imat=1:nmat_loc)
  imat_glob       = (myi-1)*nmat_loc + imat
  jmat_glob       = (myj-1)*nmat_loc + jmat
  amat(imat,jmat) = imat_glob        + jmat_glob/10.0_dp
```

```
 end do

 do imat           = 1, nmat_loc
  print '(A,I0.2,A,100F4.1)',&
   '02_amat:       ',myrank,' :: ',amat(imat,:)
 end do

 allocate(amat_glob(nmat,nmat))
 !Gather amat -> amat_glob on all processes
 call loc2glob(amat,amat_glob)

 do imat_glob     = 1, nmat
  print '(A,I0.2,A,100F4.1)',&
   '03_amat_glob: ',myrank,' :: ', amat_glob(imat_glob,:)
 end do

 call mpi_finalize(ierr)

!-----------------------------------------------------------
contains
!-----------------------------------------------------------
 subroutine         loc2glob(a, a_glob)
 subroutine         abort(errmes)

end program mpi_introduction
```

Running the above program on 4 processes produces output, part of which is shown below (compare with Figure 6.4):

```
02_amat:       00 ::   1.1 1.2
02_amat:       00 ::   2.1 2.2

02_amat:       01 ::   3.1 3.2
02_amat:       01 ::   4.1 4.2

02_amat:       02 ::   1.3 1.4
02_amat:       02 ::   2.3 2.4

02_amat:       03 ::   3.3 3.4
02_amat:       03 ::   4.3 4.4

03_amat_glob: 00 ::   1.1 1.2 1.3 1.4
03_amat_glob: 00 ::   2.1 2.2 2.3 2.4
03_amat_glob: 00 ::   3.1 3.2 3.3 3.4
03_amat_glob: 00 ::   4.1 4.2 4.3 4.4
```

The data transfer can be reversed using the subroutine `glob2loc` to distribute the global array to the local matrix blocks. Here, we assume the complete $N \times N$ global array, `a_glob(N,N)`, is available only on process 0 (the root), and we use the `MPI_Scatter` routine to distribute its elements into the local $N_{\text{loc}} \times N_{\text{loc}}$ blocks, `a(N_loc,N_loc)`, on every process:

```
subroutine         glob2loc(a_glob, a)
 real(dp), allocatable, intent(in) :: a_glob (:,:)
 real(dp), allocatable             :: a      (:,:)
 real(dp), allocatable             :: sendbuf(:  )
 integer                           :: nprocs, myrank
 integer                           :: N,N_loc,N_loc2,N_p
 integer                           :: istart,jstart,myp,myi,myj

 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 if(myrank == 0) then
  N     = size(a_glob,1) ! a_glob defined only on myrank = 0
 end if
 call mpi_bcast(N,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
 N_loc = size(a,1)        ; N_loc2 = N_loc * N_loc
 N_p   = N/N_loc

 if(myrank == 0)then
  !buffer that will store a_glob as a message in MPI_Scatter
  !(irrelevant for other processes)
  allocate(sendbuf(nprocs * N_loc * N_loc))

  do myp  = 0, nprocs - 1
   !Grid coordinates (1-based) for the process myp
   myi    = mod(myp , N_p) + 1 ! Row    index (1 to N_p)
   myj    =    (myp / N_p) + 1 ! Column index (1 to N_p)

   ! Calculate the indices for the global matrix block
   istart = (myi - 1) * N_loc + 1
   jstart = (myj - 1) * N_loc + 1

   ! Copy the 2D block (column-major in memory)
   ! into the 1D send buffer (sequential)
   sendbuf(myp*N_loc2+1:(myp+1)*N_loc2) = &
    reshape(a_glob(istart:istart+N_loc-1, &
                   jstart:jstart+N_loc-1), [N_loc2])

  end do

 end if

 call mpi_scatter(                &
      sendbuf, N_loc2, MPI_REAL8, & ! sendbuf defined on root
      a      , N_loc2, MPI_REAL8, & ! a is defined on all ranks
      0, MPI_COMM_WORLD, ierr)      ! root is 0

end subroutine     glob2loc
```

The basic steps in the subroutine are:

1. **Global Size Broadcast**: Since `a_glob` is only guaranteed to be available on rank 0, the dimension $N$ must first be broadcast from rank

0 using `MPI_Bcast` to all other processes to ensure they can correctly compute $N_\mathrm{p}$ and $N_\mathrm{loc}$.

2. **Packing (Rank 0 Only)**: Process 0 constructs the one-dimensional `sendbuf` by iterating through the grid blocks, calculates the correct global index slices, and uses `reshape` to map the 2D block slice of `a_glob` into the linear `sendbuf`.

3. **Scattering (All Processes)**: The `MPI_Scatter` routine is called by all processes. Process 0 uses `sendbuf` as the source, and every process receives $N_\mathrm{loc}^2$ elements into its local array `a`.

A complete demonstration program utilizing `glob2loc` subroutine can be found in the file `mpi_11_glob2loc.f90`.

### 6.2.5 Matrix Multiplication

In this section, we develop an MPI-based algorithm for matrix multiplication that is structurally similar to the approach presented in the Coarrays section. We emphasize the practical application of MPI communication primitives, rather than focusing on the most efficient communication scheme, which is covered later by Fox's algorithm.

We assume that three $N \times N$ square matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, are distributed across a square grid of $N_\mathrm{procs} = N_\mathrm{p} \times N_\mathrm{p}$ processes (see Figure 6.2)

Each process, identified by its rank `myrank` $= 0, \ldots, N_\mathrm{procs}$, holds a local $N_\mathrm{loc} \times N_\mathrm{loc}$ block ($\mathbf{A}_{IJ}$, $\mathbf{B}_{IJ}$, $\mathbf{C}_{IJ}$) of each matrix. The process rank (`myrank`) is mapped to the 1-based grid coordinates $(I, J)$ = (`myi`,`myj`) using a column major ordering:

$$I \equiv \texttt{myi} = \mod(\texttt{myrank}, N_p) + 1 \qquad I, J = 1, \ldots, N_p \tag{6.6}$$

$$J \equiv \texttt{myj} = (\texttt{myrank}/N_p) + 1\,, \tag{6.7}$$

The goal of this algorithm is to compute the resulting block $\mathbf{C}_{IJ}$ of the matrix product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ using the summation over matrix blocks (see Eq.(6.1)):

$$\mathbf{C}_{IJ} = \sum_{K=1}^{N_\mathrm{p}} \mathbf{A}_{IK} \cdot \mathbf{B}_{KJ} \quad I, J = 1, \ldots, N_\mathrm{p}\,. \tag{6.8}$$

For a fixed block $\mathbf{C}_{IJ}$, the process $(I, J)$ must execute $N_\mathrm{p}$ steps, each requiring blocks $\mathbf{A}_{IK}$ and $\mathbf{B}_{KJ}$ for $K = 1, \ldots, N_\mathrm{p}$. These blocks must be received from processes $(I, K)$ and $(K, J)$, respectively. The process $(I, J)$ then performs the local block matrix multiplication $\mathbf{A}_{IK} \cdot \mathbf{B}_{KJ}$ and accumulates the result in $\mathbf{C}_{IJ}$.

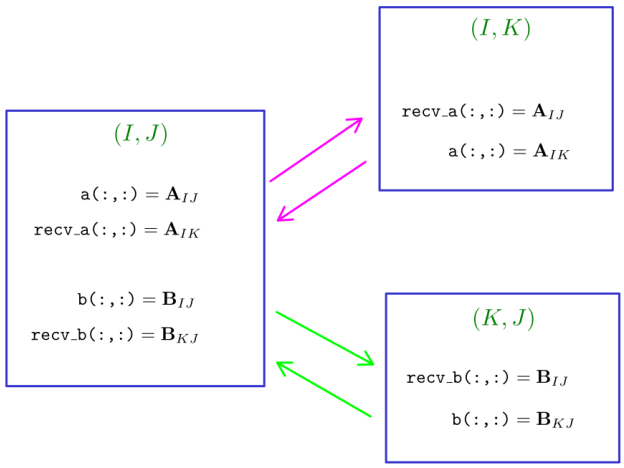



Figure 6.16: Inter-Process Data Exchange during the parallel block matrix multiplication, $\mathbf{C}_{IJ} = \sum_{K=1}^{N_\mathrm{p}} \mathbf{A}_{IK} \cdot \mathbf{B}_{KJ}$ (Eq.(6.8)). The figure shows the communication required when the summation index is fixed at $K$. The process $(I,J)$ sends its block $\mathbf{A}_{IJ}$ to $(I,K)$, while simultaneously receiving the required block $\mathbf{A}_{IK}$ from $(I,K)$. The incoming $\mathbf{A}_{IK}$ is stored locally in `recv_a(:,:)`. Process $(I,J)$ sends its block $\mathbf{B}_{IJ}$ to $(K,J)$ while simultaneously receiving the required block $\mathbf{B}_{KJ}$ from $(K,J)$. The incoming $\mathbf{B}_{KJ}$ is stored locally in `recv_b(:,:)`. This two-way exchange ensures that all participating processes receive the necessary matrix blocks for their respective local computations in a single step.

To optimize data communication, the protocol exploits the fact that at the same time process $(I,J)$ is receiving data, the processes $(I,K)$ and $(K,J)$ need blocks from $(I,J)$ for their own respective computations. Specifically, at each step, a one-to-one exchange occurs:

- Process $(I,J)$ needs $\mathbf{A}_{IK}$ and $\mathbf{B}_{KJ}$.
- Process $(I,K)$ needs $\mathbf{A}_{IJ}$ for computing $\mathbf{C}_{IK}$.
- Process $(K,J)$ needs $\mathbf{B}_{IJ}$ for computing $\mathbf{C}_{KJ}$.

This simultaneous send/receive pattern, illustrated in Figure 6.16 and Figure 6.17, is efficiently implemented using the collective point-to-point subroutine `MPI_Sendrecv`, as shown in the following listing:

```
function           pmmult(a, b, tag) result(c)
 integer              , intent(in) :: tag(2)  !MPI message tags
 real(dp), allocatable , intent(in) :: a(:,:), b(:,:)
 real(dp), allocatable              :: c(:,:)
 integer                            :: nprocs, myrank
```

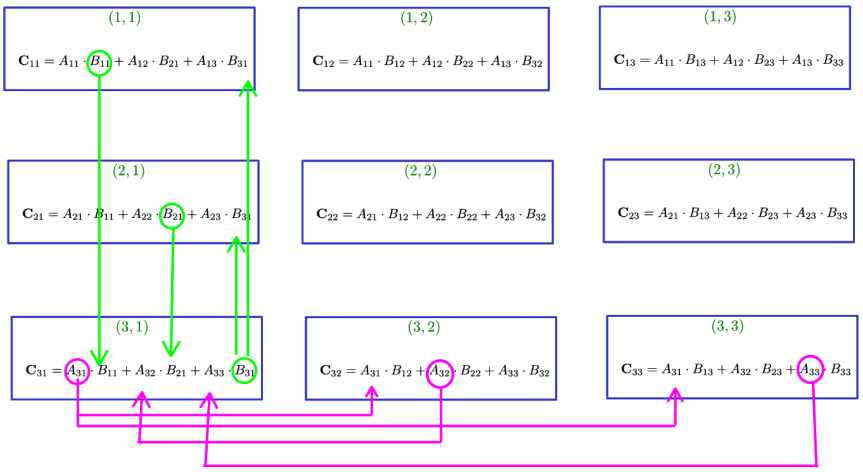



Figure 6.17: Example of Block Exchange for Process $(3, 1)$ during the parallel matrix multiplication (Section 6.2.5). The figure illustrates the data exchanges required for computing the block $\mathbf{C}_{31}$. The circled blocks $\mathbf{A}_{31}$ and $\mathbf{B}_{31}$ represent the local data available on process $(3, 1)$ at the start. The magenta arrows show the block exchanges along the row $I = 3$, and the green arrows show the block exchanges along the column $J = 1$. Specifically, there is exactly one send/receive exchange between process $(3, 1)$ and every process on its row, $(3, K)$ for $K = 1, 2, 3$, and exactly one exchange between process $(3, 1)$ and every process on its column, $(K, 1)$ for $K = 1, 2, 3$.

```
integer                                 :: N_loc,N_loc2,N_p
integer                                 :: myi, myj, myk
integer                                 :: rank_ik, rank_kj
integer                                 :: ierr
real(dp), allocatable                   :: recv_a(:,:), recv_b(:,:)

call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

N_loc = size(a,1) ; N_loc2 = N_loc * N_loc
N_p   = int(sqrt(nprocs+1.0e-6_dp))

!I am process (I,J) = (myi,myj)
myi   = mod(myrank , N_p) + 1       ! Row    index (1 to N_p)
myj   =    (myrank / N_p) + 1       ! Column index (1 to N_p)

! Allocate receive buffers
allocate(recv_a(N_loc, N_loc))
allocate(recv_b(N_loc, N_loc))

! Initialize the local result block
allocate(c(N_loc,N_loc))
```

```
  c = 0.0_dp

  !Loop through myk to perform the block matrix multiplication
  do myk = 1, N_p
   !Rank of the process holding the (myi,myk) block of a_glob:
   !Assumes column major distribution of processes.
   rank_ik = (myi - 1) + (myk - 1) * N_p   !Process (I,K)

   !Rank of the process holding the (myk,myj) block of b_glob
   rank_kj = (myk - 1) + (myj - 1) * N_p   !Process (K,J)

   ! Send and receive a block
   call mpi_sendrecv(                                  &
        a        , N_loc2, MPI_REAL8, rank_ik, tag(1), &
        recv_a   , N_loc2, MPI_REAL8, rank_ik, tag(1), &
        MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
   !                    MPI_STATUS_IGNORE is a predefined
   !constant that you can pass to the receive functions
   !when you don't need to examine this status information.

   ! Send and receive b block
   call mpi_sendrecv(                                  &
        b        , N_loc2, MPI_REAL8, rank_kj, tag(2), &
        recv_b   , N_loc2, MPI_REAL8, rank_kj, tag(2), &
        MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)

   ! Perform local matrix multiplication
   c = c + matmul(recv_a, recv_b)

  end do

  deallocate(recv_a)
  deallocate(recv_b)

 end function       pmmult
```

An example on how to use the function `pmmult` is in the file `mpi_12_mmult.f90`, and it is shown below:

```
program     mpi_introduction

 use, intrinsic        :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p
 integer               :: ierr, nprocs, myrank
 integer               :: imat, jmat, imat_glob, jmat_glob
 real(dp), allocatable :: amat(:,:),amat_glob(:,:)
 real(dp), allocatable :: bmat(:,:),bmat_glob(:,:)
 real(dp), allocatable :: cmat(:,:),cmat_glob(:,:)
 integer               :: mtags(2)
 real(dp)              :: mult_error
```

```
!-------------------------------------------------------
 call mpi_init      (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)
!-------------------------------------------------------
 nmat     = 64

 nmat_p   = int(sqrt(nprocs+1.0e-6_dp))
 nmat_loc = nmat/nmat_p

 !unique tags needed by the pmmult function
 !for MPI communication messages:
 mtags    = [10,20]

 if(myrank == 0) print '(A,I0.2,A,3I4    )'      , &
  '00_Nmat:       ',myrank,' :: N,N_loc,N_p = ' , &
  nmat,nmat_loc,nmat_p

 allocate(amat(nmat_loc,nmat_loc),amat_glob(nmat,nmat))
 allocate(bmat(nmat_loc,nmat_loc),bmat_glob(nmat,nmat))
 allocate(cmat(nmat_loc,nmat_loc),cmat_glob(nmat,nmat))
!-------------------------------------------------------
 call random_number(amat) ; call random_number(bmat)

 cmat = pmmult(amat,bmat,mtags)

!Check the results:
 call loc2glob(amat,amat_glob)
 call loc2glob(bmat,bmat_glob)
 call loc2glob(cmat,cmat_glob)
 if(myrank == 0)then
  mult_error = &
   sum(abs(cmat_glob-matmul(amat_glob,bmat_glob)))/(nmat*nmat)
  print '(A,I0.2,A,G26.16)', &
   '01_Mmult:      ',myrank,' :: ||C_g-A_g.B_g|| = ',mult_error
 end if

 call mpi_finalize(ierr)
!-------------------------------------------------------
contains
!-------------------------------------------------------
 function           pmmult(a, b, tag) result(c)
 subroutine         loc2glob(a, a_glob)
 subroutine         abort(errmes)
end program mpi_introduction
```

Compiling and running the program on 4 processes produces the output:

```
00_Nmat:       00 :: N,N_loc,N_p      =   64   32    2
01_Mmult:      00 :: ||C_g-A_g.B_g|| =   0.1032594149075194E-14
```

The first line shows that $N = 64$, $N_{\text{loc}} = 32$, and $N_{\text{p}} = 64/32 = 2$. The

last line displays a measure of the numerical discrepancy between the computation of the product using the global matrices versus the parallel computation using the local block matrices.

### 6.2.6 Transpose and Traces

This section details how to compute the matrix transpose, $\mathbf{A}^T$, for a matrix distributed across a process grid (as shown in Figure 6.2).

For a matrix distributed in blocks, the transpose is computed block-wise using the relation $\mathbf{A}^T_{IJ} = (\mathbf{A}_{JI})^T$ (Eq.(6.3)). This means the block held by process $(I, J)$ is simply the transpose of the block held by process $(J, I)$. This required data exchange between processes $(I, J)$ and $(J, I)$ is efficiently performed using a single `MPI_Sendrecv` call.

The function `ptranspose` computes the local block of the transpose, `at(N_loc,N_loc)` $= \mathbf{A}^T_{IJ}$:

```
function           ptranspose(a,tag)     result(at)
 integer              , intent(in) :: tag
 real(dp)             , intent(in) :: a (:,:)
 real(dp), allocatable             :: at(:,:)
 real(dp), allocatable             :: recv_a(:,:)
 integer                           :: nprocs, myrank
 integer                           :: N_loc,N_loc2,N_p
 integer                           :: myi, myj, myk
 integer                           :: rank_ji

 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 N_loc = size(a,1) ; N_loc2 = N_loc * N_loc
 N_p   = int(sqrt(nprocs+1.0e-6_dp))

 !Calculate the grid coordinates for the process myrank
 myi   = mod(myrank , N_p) + 1       ! Row    index (1 to N_p)
 myj   =    (myrank / N_p) + 1       ! Column index (1 to N_p)

 !Initialize the local result block
 allocate(at(N_loc,N_loc))

 !Rank of process holding the (myj,mi) block:
 rank_ji =  (myj - 1) + (myi - 1) * N_p

 if(myi == myj) then
  !No communication needed, just local transpose
  at = transpose(a)
 else
  allocate(recv_a(N_loc,N_loc))

  call mpi_sendrecv(                              &
       a      , N_loc2, MPI_REAL8, rank_ji, tag, &
```

```
        recv_a, N_loc2, MPI_REAL8, rank_ji, tag, &
        MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)

   at = transpose(recv_a)
  end if

 end function       ptranspose
```

The calling program must supply a unique integer `tag` for the `MPI_Sendrecv` call. By using this optimized primitive, processes $(I, J)$ and $(J, I)$ execute a simultaneous exchange of blocks $\mathbf{A}_{IJ}$ and $\mathbf{A}_{JI}$ in one step. The process then locally transposes the received block to produce $\mathbf{A}_{IJ}^T$. For diagonal processes ($I = J$), only a local transpose is performed.

The program, `mpi_14_trace2.f90`, serves as a comprehensive example demonstrating the parallel computation and verification of the matrix transpose ($\mathbf{A}^T$), the trace squared ($\mathrm{tr}\,\mathbf{A}^2$), and the trace ($\mathrm{tr}\,\mathbf{A}$) using `MPI_Sendrecv` and collective reduction routines (`MPI_Allreduce`):

```
program     mpi_introduction

 use, intrinsic        :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p
 integer               :: ierr, nprocs, myrank, myi, myj, tag
 integer               :: imat, imat_glob
 real(dp), allocatable ::  amat(:,:), amat_glob(:,:)
 real(dp), allocatable :: atmat(:,:),atmat_glob(:,:)
 real(dp)              :: terror
 real(dp)              :: tra2,tra2_loc,tra2_glob
 real(dp)              :: tra , tra_loc, tra_glob

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat     = 64

 nmat_p   = int(sqrt(nprocs+1.0e-6_dp))
 nmat_loc =       nmat / nmat_p
 myi      = mod(myrank , nmat_p) + 1 ! myrank coordinates
 myj      =     myrank / nmat_p  + 1

 !Unique tag for ptranspose
 tag      = 10

 if(myrank == 0) print '(A,I0.3,A,3I4    )',       &
  '00_Nmat:       ',myrank,                         &
  ' :: N,N_loc,N_p     = ',nmat,nmat_loc,nmat_p
```

```
 allocate( amat(nmat_loc,nmat_loc), amat_glob(nmat,nmat))
 allocate(atmat(nmat_loc,nmat_loc),atmat_glob(nmat,nmat))
!-------------------------------------------------------------
 call random_number(amat)

 atmat = ptranspose(amat,tag)

!Check the results:
 call loc2glob( amat, amat_glob)
 call loc2glob(atmat,atmat_glob)

 if(myrank == 0)then
  terror = sum(abs(atmat_glob - transpose(amat_glob)))/nmat**2
  print '(A,I0.3,A,G28.16)',                          &
   '01 Transpose: ',myrank,                           &
   ' || at_glob - (a_glob)T || = ',terror
 end if
!-------------------------------------------------------------
!trA2:
 tra2_loc = sum(amat*atmat)
 call mpi_reduce(tra2_loc,tra2,1,                     &
  MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,ierr)
!Check the results:
 if(myrank == 0)then
  tra2_glob = sum(amat_glob * transpose(amat_glob))
  terror    = abs(tra2_glob-tra2)/nmat**2
  print '(A,I0.3,A,G28.16)',                          &
   '02 TrA2:      ',myrank,                           &
   ' |TrA2_g - TrA2|          = ',terror
 end if
!-------------------------------------------------------------
!trA:
 trA_loc    = 0.0_dp
 if(myi    == myj)then
  do imat   = 1, nmat_loc
   tra_loc  = tra_loc + amat(imat,imat)
  end do
 end if
 call mpi_allreduce(tra_loc,tra,1,                    &
  MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,ierr)
 print  '(A,I0.3,A,G28.16)',                          &
  '03 TrA:       ',myrank,                           &
  ' TrA2                    = ',tra
!check results:
 if(myrank == 0)then
  tra_glob  = 0.0_dp
  do imat   = 1, nmat
   tra_glob = tra_glob + amat_glob(imat,imat)
  end do
  print '(A,I0.3,A,G28.16)',                          &
   '04 δTrA:      ',myrank,                           &
   ' |TrA_glob-TrA|          = ',abs(tra-tra_glob)
 end if
```

```
 call mpi_finalize(ierr)

!-----------------------------------------------------------------
contains
 function           ptranspose(a,tag)     result(at)
 subroutine         loc2glob(a, a_glob)
 subroutine         abort(errmes)
!-----------------------------------------------------------------
end program mpi_introduction
```

Using this program, `tra2` is available only on the process with rank 0, whereas `tra` is available on all processes.

### 6.2.7 Communicators

So far, all parallel operations have been confined to the entire set of available processes, defined by the global communicator, `MPI_COMM_WORLD`. The MPI library provides functionality to define communicators for subsets of these processes.

A *communicator* defines a *communication domain*, which is a context that binds an ordered collection of processes, referred to as its *group* of processes. Similar to `MPI_COMM_WORLD`, each process within a new communicator is assigned a unique, local, non-negative integer rank, starting from 0 up to $N_{\text{procs}}-1$, where $N_{\text{procs}}$ is the total number of processes in that specific group. All MPI functions must be passed a communicator argument to specify the exact set of processes involved.

Custom communicators, created as subsets of `MPI_COMM_WORLD`, offer encapsulation and safe communication contexts; a message sent on one communicator cannot be received on another. This ensures modularity and prevents accidental message interference in complex applications. They allow users to logically partition and organize tasks, defining smaller, specialized groups to run collective operations without involving irrelevant processes. Correct use can enable structured parallelism and optimize collective operations, resulting in reduced communication overhead and improved scalability. Communicators enable topology awareness. This means they can create virtual topologies that align with the underlying real hardware layout, ensuring that heavy inter-process communication utilizes the fastest available links.

#### 6.2.7.1 The Diagonal Communicator

A basic example of creating a custom communicator within `MPI_COMM_WORLD` is to isolate the processes responsible for the diagonal blocks of a distributed $N \times N$ matrix $\mathbf{A}$.

The matrix blocks $\mathbf{A}_{IJ}$ are distributed across $N_{\text{procs}} = N_{\text{p}} \times N_{\text{p}}$ processes, with each process holding an $N_{\text{loc}} \times N_{\text{loc}}$ block. The processes are

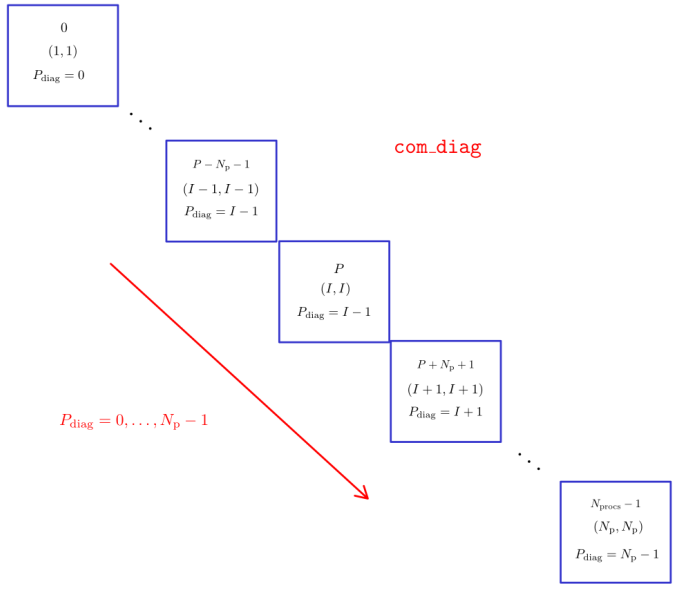



Figure 6.18: The Diagonal Communicator `com_diag`. The figure illustrates $N_{\text{procs}}$ processes, with global rank `myrank` $= 0, \ldots, N_{\text{procs}} - 1$, distributed across an $N_{\text{p}} \times N_{\text{p}}$ grid, where coordinates $(I, J)$ are determined by the Eqs.(6.9)–(6.11). The `com_diag` communicator consists exclusively of the $N_{\text{p}}$ processes located on the diagonal of this grid (where $I = J$). Within this communicator, each process is reassigned a contiguous local `myrank_diag` $\equiv P_{\text{diag}}$, ranging from 0 to $N_{\text{p}} - 1$. Their corresponding global rank is $P_{\text{diag}}(N_{\text{p}} + 1)$.

assigned blocks in a column-major ordering based on their global rank $P$:

$$P = (I-1) + (J-1)N_{\text{p}}\,, \quad I, J = 1, \ldots, N_{\text{p}}\,. \tag{6.9}$$

The process's grid coordinates $(I, J)$ are calculated from the rank $P$ as:

$$I = \texttt{myi} = \mod(P, N_{\text{p}}) + 1\,, \quad P = 0, \ldots, N_{\text{procs}} - 1\,, \tag{6.10}$$

$$J = \texttt{myj} = P/N_{\text{p}} + 1\,. \tag{6.11}$$

This structure defines a virtual grid where each process, labeled by $(\texttt{myi}, \texttt{myj})$, holds the block $\mathbf{A}_{IJ}$. The diagonal blocks, $\mathbf{A}_{II}$, are defined by the condition $I = J$. This group of diagonal processes has non-contiguous ranks in `MPI_COMM_WORLD`.

The objective is to create a new communicator, `com_diag`, that contains only this subset of $N_{\text{p}}$ processes.

The new `com_diag` contains `nprocs_diag` $= N_{\text{p}}$ processes. Each process is assigned a contiguous local rank `myrank_diag` $= I - 1$, ranging

from 0 to $N_p - 1$. Within this specialized communicator, the diagonal blocks $\mathbf{A}_{II}$ are defined on processes identified by `myrank_diag` $= I - 1$. This structure is visualized in Figure 6.18.

Understanding the logical construction of `com_diag` involves these four low-level steps:

1. **Group Extraction**: Determine the group of processes (`grp_world`) within the existing communicator (`com_world` = `MPI_COMM_WORLD`).
2. **Rank Selection**: Identify and store the `com_world` ranks of the diagonal processes into an array (`ranks(nmat_p)`).
3. **New Group Creation**:Use the ranks collected in `ranks` to create a new group (`grp_diag`) that contains only the diagonal processes.
4. **Communicator Creation**:Use the new process group (`grp_diag`) to construct the final communicator (`com_diag`).

Implementing these steps requires the fundamental MPI routines `MPI_Comm_group`, `MPI_Group_incl`, and `MPI_Comm_create`.

Creating a custom communicator from scratch involves operating directly on the underlying process groups. The following routines are used for group and communicator manipulation:

- `MPI_Comm_group`: Extracts the process group handle from an existing communicator.

```
call mpi_comm_group(comm,group,ierr)
```

  Here, `comm` is the input communicator handle, and `group` is the returned handle representing the ordered set of processes within `comm`.

- `MPI_Group_incl`: Creates a new group by including a subset of ranks from a current group.

```
call mpi_group_incl(group,n,ranks,newgroup,ierr)
```

  The subroutine takes the current `group` handle and an integer array `ranks(:)` containing the `n` ranks of processes to be included in the `newgroup`.

- `MPI_Comm_create`: Creates a new communicator from a defined process group.

```
call mpi_comm_create(comm,newgroup,newcomm,ierr)
```

This routine takes the parent communicator `comm` and the defined `newgroup` handle to return the handle for the `newcomm`.

The full calling sequence to construct the diagonal communicator (`com_diag`) from the global communicator (`com_world`) is:

```
!1. compute the group of processes within comm_world:
!   stored in grp_world
 com_world = MPI_COMM_WORLD
 call mpi_comm_group(com_world,grp_world,ierr)
!2. compute the ranks of processes to be included in com_diag
!   Their ranks are (I-1) (N_p -1)
 ranks = [ ( (i-1) * (nmat_p+1),i=1,nmat_p) ]
!3. use ranks(:) to create the new group grp_diag,
!   selected from grp_world:
 call mpi_group_incl (grp_world,nmat_p,ranks,grp_diag,ierr)
!4. using the newly created group grp_diag,
!   create its communicator com_diag
 call mpi_comm_create(com_world,grp_diag,com_diag,ierr)
```

These routines are called by all processes in `MPI_COMM_WORLD`. Processes belonging to the diagonal group receive a valid `com_diag` handle. Off-diagonal processes, which are not members of the new group, receive the predefined invalid value, `com_diag` = `MPI_COMM_NULL`.

The value of `com_diag` must be used on every process to check its membership before attempting to communicate within the new group:

```
 if(com_diag /= MPI_COMM_NULL)then
  ! do stuff within  the communicator com_diag
 else
  ! do stuff outside the communicator com_diag
 end if
```

As an example, to compute the trace of the matrix $\mathbf{A}$, which depends only on the diagonal blocks $\mathbf{A}_{II}$, we use `MPI_Allreduce` exclusively within the diagonal group:

```
if(com_diag /= MPI_COMM_NULL)then
 !Compute the trace of the local block A_II:
 tra_loc = sum( [ (amat(imat,imat),imat=1,nmat_loc) ] )
 !Compute tra as the sum of all tra_loc, only on com_diag:
 call mpi_allreduce(tra_loc ,tra ,1,MPI_REAL8,MPI_SUM,          &
                    com_diag,ierr)
end if
```

Since collective calls like `MPI_Allreduce` must be executed only by members of the specified communicator, the `if(com_diag /= MPI_COMM_NULL)` check is mandatory. Passing `MPI_COMM_NULL` to a collective routine will result in a runtime error.

The full program that implements the procedures described above is in the file `mpi_15_communicators.f90` and is listed below:

```
program     mpi_introduction

 use, intrinsic        :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter   :: dp = real64
 integer               :: nmat, nmat_loc,nmat_p
 integer               :: ierr, nprocs, myrank, myi, myj
 integer               :: com_world,com_diag
 integer               :: grp_world,grp_diag
 integer               :: nprocs_diag,myrank_diag
 integer , allocatable :: ranks(:)
 integer               :: imat, i
 real(dp), allocatable :: amat(:,:)
 real(dp)              :: tra,tra_loc

 call mpi_init     (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat     = 128

 nmat_p   = int(sqrt(nprocs+1.0e-6_dp))
 nmat_loc = nmat/nmat_p
 myi      = mod(myrank , nmat_p) + 1
 myj      =     myrank / nmat_p  + 1

 allocate(amat(nmat,nmat))
 call random_number(amat)

!Create the group of processes in MPI_COMM_WORLD::
 com_world = MPI_COMM_WORLD
 call mpi_comm_group(com_world,grp_world,ierr)
!Select the ranks that you want to include in the
!new communicator. Select ranks s.t. myi = myj
 ranks = [ ( (i-1) * (nmat_p+1),i=1,nmat_p) ]
!Create the new group of processes:
 call mpi_group_incl (grp_world,nmat_p,ranks,grp_diag,ierr)
!Create a communicator for the processes on the diagonal grid:
 call mpi_comm_create(com_world,grp_diag,com_diag,ierr)
!com_diag is set to MPI_COMM_NULL for all non-diagonal processes

 if(com_diag /= MPI_COMM_NULL)then
  !Find my position in the new communicator: myrank_diag
  call mpi_comm_size(com_diag,nprocs_diag,ierr)
  call mpi_comm_rank(com_diag,myrank_diag,ierr)
  print  '(A,I0.3,A,9I4    )',                          &
   '02 Diag Comm  ',myrank,                              &
   ' :: ',myrank_diag,nprocs_diag,myi,myj
```

```
  tra_loc = sum( [ (amat(imat,imat),imat=1,nmat_loc) ] )
  call mpi_allreduce(tra_loc,tra,                          &
   1,MPI_REAL8,MPI_SUM,com_diag,ierr)
  print  '(A,I0.3,A,9G28.16)',                             &
   '03 trA         ',myrank,                               &
   ' :: ',tra,tra_loc
 end if

 call mpi_finalize(ierr)

end program mpi_introduction
```

Running the program using 16 processes ($N_\mathrm{p} = 4$), produces the (sorted) output:

```
02 Diag Comm  000 ::    0   4   1   1
02 Diag Comm  005 ::    1   4   2   2
02 Diag Comm  010 ::    2   4   3   3
02 Diag Comm  015 ::    3   4   4   4

03 trA        000 ::  59.41128310437657  14.69920497387400
03 trA        005 ::  59.41128310437657  16.49692208847727
03 trA        010 ::  59.41128310437657  14.15881810433120
03 trA        015 ::  59.41128310437657  14.05633793769410
```

We see that processes with `myrank` $= 0, 5, 10,$ and $15$ belong to the `com_diag` communicator, with `myrank_diag` $= 0, 1, 2, 3$, respectively.

While creating communicators via explicit group manipulation aids in understanding the underlying logic, the easiest and most common way to partition a communicator is using the subroutine `MPI_Comm_split`. To use `MPI_Comm_split`, every process must first be assigned a *color*. All processes sharing the same color are subsequently assigned to the same new communicator. In Fortran, the `color` variable is an integer. If a process should not be included in any resulting communicator, its color must be set to `MPI_UNDEFINED`.

The Fortran calling sequence for the subroutine `MPI_Comm_split` is:

```
call mpi_comm_split(comm,color,key,newcomm,ierr)
```

`comm` is the handle for the parent communicator. `color` is the non-negative integer color of the calling process. `key` is a non-negative integer that determines the internal rank order within the new communicator. Processes with the same `color` are assigned ranks in ascending order based on their `key` values (with ties broken by their rank in the parent `comm`). `newcomm` is the handle returned for the newly created communicator. Processes for which `color = MPI_UNDEFINED` receive `newcom = MPI_COMM_NULL`. An example demonstrating the use of color and key for partitioning is shown in Figure 6.19.

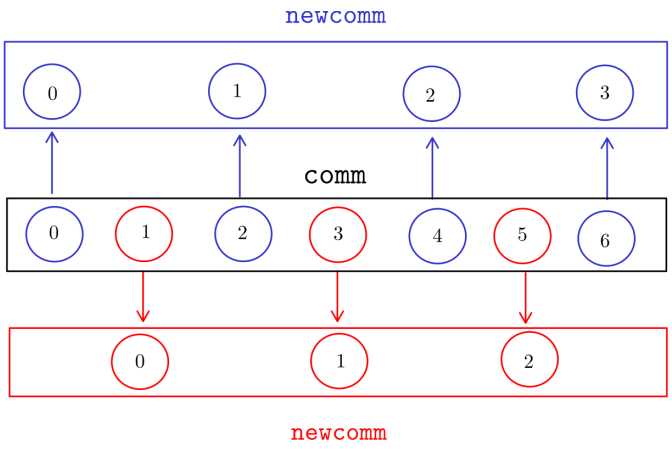



Figure 6.19: Process Partitioning using `MPI_Comm_split`. This example shows how an initial communicator (`comm`) containing 7 processes is split into two smaller, distinct communicators (`newcomm`). The partitioning is based on the `color` assigned by `color` = mod (`myrank`, 2), separating the original processes into even (color 0, blue) and odd (color 1, red) groups. The call `call mpi_comm_split(comm,mod(myrank,2),myrank,newcomm,ierr)` groups processes of the same color into a unique `newcomm` context. Since `key` = `myrank`, the processes within each new group are ranked in ascending order based on their original rank. Note that the resulting communicators have different sizes: 4 processes in the blue group and 3 in the red group.

The entire process of creating the diagonal communicator (`com_diag`) is accomplished with a single call to `MPI_Comm_split`:

```
mycolor   = MPI_UNDEFINED
if(myi   == myj ) mycolor = 0
call mpi_comm_split(com_world,mycolor,myrank,com_diag,ierr)
```

In this code, only processes satisfying the diagonal condition (`myi == myj`) receive a color and a valid communicator; all others are assigned `com_diag` = `MPI_COMM_NULL`. By setting the ordering key to the original global rank (`key` = `myrank`), the new local ranks (`myrank_diag`) are ordered in ascending fashion, aligning with the structure shown in Figure 6.18. The program utilizing this efficient method is provided in the file `mpi_16_communicators.f90`.

#### 6.2.7.2 Row and Column Communicators

Another set of communicators that play a role in matrix multiplication[3], are the row communicator `com_row` and column communicator `com_col` of each process. The block matrices $\mathbf{A}_{IJ}$, $J = 1, \ldots, N_{\rm p}$ of each row of the global matrix $\mathbf{A}$ belong to the group of the same row communicator, and the block matrices $\mathbf{A}_{IJ}$, $I = 1, \ldots, N_{\rm p}$ of each column of the global matrix $\mathbf{A}$ belong to the group of the same column communicator. The processes of a row communicator have the same *color* $I$, and they are ranked using the *key* $J$. The processes of a column communicator have the same *color* $J$, and they are ranked using the *key* $I$. This grouping is depicted in Figure 6.20, which shows the `com_row` and the `com_col` of a process with global `myrank` $= P$, and grid coordinates $(I, J)$ given by Eqs.(6.9)–(6.11). The rank of each process in the `com_row` of the process $(I, J)$ is `myrank_row` $\equiv P_{\rm row} = J - 1$. The rank of each process in the `com_col` of the process $(I, J)$ is `myrank_col` $\equiv P_{\rm col} = I - 1$.

The calls that create the two communicators are shown below:

```
!                       parent comm     ,color ,key ,nrcomm
 call mpi_comm_split(MPI_COMM_WORLD, myi   ,myj ,com_row ,ierr)
 call mpi_comm_split(MPI_COMM_WORLD, myj   ,myi ,com_col ,ierr)
```

For `com_row` the color is `myi` $\equiv I$, therefore all processes with the same `myi` are placed in the same communicator `com_row`. For `com_col` the color is `myj` $\equiv J$, therefore all processes with the same `myj` are placed in the same communicator `com_col`. The keys are `myj` and `myi`, respectively, and their ranks in the respective communicators are computed by the calls:

```
 call mpi_comm_rank(com_row ,myrank_row ,ierr)
 call mpi_comm_rank(com_col ,myrank_col ,ierr)
```

and found to be `myj-1` and `myi-1` respectively.

To compute the processes that belong to the same communicator we do the following trick. First we define the array entry

```
amat(1,1) = myi + myj/10.0_dp
```

on each process, so that its value is $I.J$ (assuming $J < 10$, of course). Then we collect those values from all processes in the same communicator in an array `iamgroup(nmat_p)` using `MPI_Allgather` (see Figure 6.13):

```
call mpi_allgather(amat(1,1) ,1 ,MPI_REAL8, iamgroup ,1 ,MPI_REAL8, &
 com_row ,ierr)
```

[3] See e.g., Fox's algorithm discussed in Section 6.2.7.3

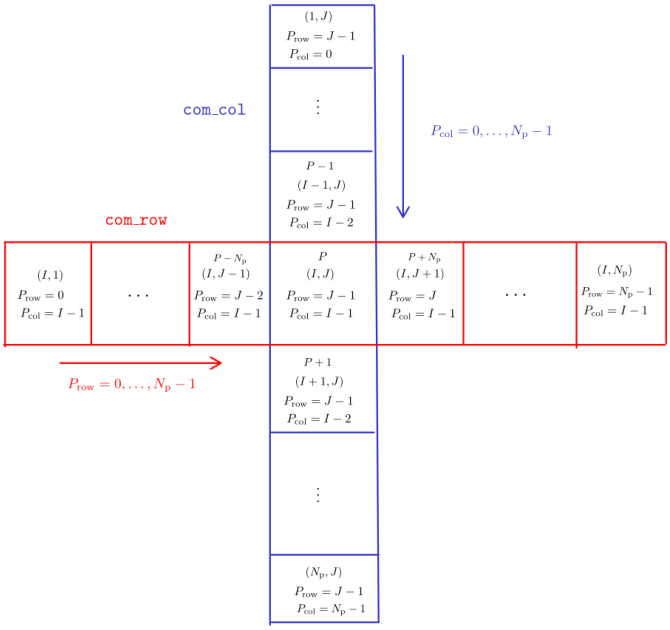



Figure 6.20: Row and Column Communicators. The figure illustrates the custom communicators, `com_row` (red) and `com_col` (blue), for a reference process $P$ located at grid coordinates $(I, J)$. These communicators are subsets of `MPI_COMM_WORLD` that partition the entire grid into rows and columns. The Row Communicator (`com_row`) groups all processes with the same row index $I$. The local rank within this group is defined by the column index $P_{\rm row} = J - 1$. The Column Communicator (`com_col`) groups all processes with the same column index $J$. The local rank within this group is defined by the row index $P_{\rm col} = I - 1$.

Printing the array `iamgroup(:)` from each process, displays the $I.J$ values in their `com_row` communicator. Then the printing command

```
print '(I0.3,A,900F6.1)',myrank,' :: ',iamgroup
```

when run with 4 processes, produces the output

```
000 ::     1.1    1.2
001 ::     2.1    2.2
002 ::     1.1    1.2
003 ::     2.1    2.2
```

showing that processes $P = 0, 2$ belong to the same `com_row` communicator and $P = 1, 3$ to the same `com_row` communicator. This is consistent with the column major construction of the process grid.

Repeating the same exercise with `com_col`, we obtain the output

```
000 ::     1.1    2.1
001 ::     1.1    2.1
002 ::     1.2    2.2
003 ::     1.2    2.2
```

showing that processes $P = 0, 1$ belong to the same `com_col` communicator and $P = 2, 3$ to the same `com_col` communicator.

The full program implementing this exercise is listed below, and can be found in the file `mpi_17_communicators.f90`:

```
program     mpi_introduction

 use, intrinsic         :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter    :: dp = real64
 integer                :: nmat, nmat_loc,nmat_p
 integer                :: ierr, nprocs, myrank, myi, myj
 integer                :: com_world,com_row,com_col
 integer                :: nprocs_row,myrank_row
 integer                :: nprocs_col,myrank_col
 integer                :: imat, i
 real(dp), allocatable  :: amat(:,:), iamgroup(:)

 call mpi_init      (ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nmat     = 64

 nmat_p   = int(sqrt(nprocs+1.0e-6_dp))
 nmat_loc = nmat/nmat_p
 myi      = mod(myrank , nmat_p) + 1
 myj      =     myrank / nmat_p  + 1

 call mpi_comm_split(MPI_COMM_WORLD,myi,myj,com_row,ierr)

 call mpi_comm_split(MPI_COMM_WORLD,myj,myi,com_col,ierr)

 !Test results:
 call mpi_comm_size(com_row,nprocs_row,ierr)
 call mpi_comm_rank(com_row,myrank_row,ierr)
 print  '(A,I0.3,A,3I4     )',                                  &
  '02 com_row     ',myrank,' :: ',                              &
 myrank_row,myrank_row-myj+1,nprocs_row-nmat_p

 !Test results:
 call mpi_comm_size(com_col,nprocs_col,ierr)
 call mpi_comm_rank(com_col,myrank_col,ierr)
 print  '(A,I0.3,A,3I4     )',                                  &
  '03 com_col     ',myrank,' :: ',                              &
```

```
  myrank_col,myrank_col-myi+1,nprocs_col-nmat_p


 allocate(amat(nmat,nmat)) ; allocate(iamgroup(nmat_p))
 amat(1,1) = myi + myj/10.0_dp

 call mpi_allgather(amat(1,1),1,MPI_REAL8,iamgroup,1,MPI_REAL8,&
  com_row,ierr)
 print  '(A,I0.3,A,900F6.1)',                                    &
  '04 grp_row     ',myrank,' :: ',iamgroup

 call mpi_allgather(amat(1,1),1,MPI_REAL8,iamgroup,1,MPI_REAL8,&
  com_col,ierr)
 print  '(A,I0.3,A,900F6.1)',                                    &
  '05 grp_col     ',myrank,' :: ',iamgroup

 call mpi_finalize(ierr)

end program mpi_introduction
```

When run using 4 processes, the (sorted) output produced by the run is:

```
02 com_row     000 ::     0    0    0
02 com_row     001 ::     0    0    0
02 com_row     002 ::     1    0    0
02 com_row     003 ::     1    0    0

03 com_col     000 ::     0    0    0
03 com_col     001 ::     1    0    0
03 com_col     002 ::     0    0    0
03 com_col     003 ::     1    0    0

04 grp_row     000 ::     1.1    1.2
04 grp_row     001 ::     2.1    2.2
04 grp_row     002 ::     1.1    1.2
04 grp_row     003 ::     2.1    2.2

05 grp_col     000 ::     1.1    2.1
05 grp_col     001 ::     1.1    2.1
05 grp_col     002 ::     1.2    2.2
05 grp_col     003 ::     1.2    2.2
```

#### 6.2.7.3 Matrix Multiplication: Fox's Algorithm

In Subsection 6.2.5 we described an algorithm that performs the matrix multiplication of two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$, distributed on a grid of $N_{\text{procs}} = N_{\text{p}} \times N_{\text{p}}$ processes as shown in Figure 6.2. Each process is labeled by a pair of integers $(I, J)$, $I, J = 1, \ldots, N_{\text{p}}$, and holds a $N_{\text{loc}} \times N_{\text{loc}}$ block $\mathbf{A}_{IJ}$, $\mathbf{B}_{IJ}$ of each matrix, where $N = N_{\text{p}} \times N_{\text{loc}}$. The result is computed using Eq.(6.1), and, after the computation, each process holds the block

$\mathbf{C}_{IJ}$ of $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$.

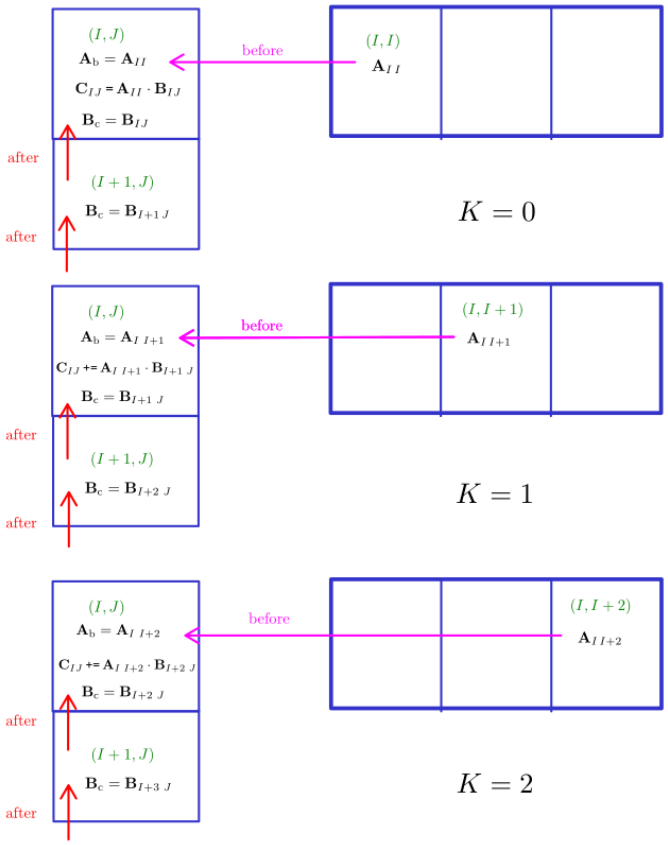



Figure 6.21: The first 3 steps of Fox's algorithm described on page 257. Process $(I, J)$ initializes `B_current` $\equiv \mathbf{B}_c \leftarrow \mathbf{B}_{IJ}$. For step $K = 0$, it identifies process $(I, I)$ holding $\mathbf{A}_{II}$, and receives `A_broadcasted` $\equiv \mathbf{A}_b \leftarrow \mathbf{A}_{II}$ from the broadcast initiated on all processes on the `com_row` communicator. Then, it computes the matrix product $\mathbf{A}_b \cdot \mathbf{B}_c = \mathbf{A}_{II} \cdot \mathbf{B}_{IJ}$, and stores it in $\mathbf{C}_{IJ}$. Then, the matrix $\mathbf{B}_c \leftarrow \mathbf{B}_{I+1\,J}$ is received from process $(I + 1, J)$, and $\mathbf{B}_{IJ}$ is sent over to process $(I - 1, J)$. For step $K = 1$, $\mathbf{A}_{I\,I+1}$ is broadcasted from process $(I, I + 1)$ along `com_row`, and received by $(I, J)$ in $\mathbf{A}_b \leftarrow \mathbf{A}_{I\,I+1}$. Then, $\mathbf{A}_b \cdot \mathbf{B}_c = \mathbf{A}_{I\,I+1} \cdot \mathbf{B}_{I+1\,J}$ is computed, and added to $\mathbf{C}_{IJ}$. Then $\mathbf{B}_c \leftarrow \mathbf{B}_{I+2\,J}$ is received from process $(I + 1, J)$, and $\mathbf{B}_{I+1\,J}$ is sent over to process $(I - 1, J)$. And so on... The "before" arrows mark messages broadcasted before the local matrix multiplication along a process row, and the "after" arrows mark messages exchanged between neighboring processes in the same column after the local matrix multiplication.

In this section we describe Fox's algorithm, which performs the same

computation, but it handles interprocess communication more effectively by avoiding unnecessary latency cost and the lack of message concurrency.

The grid of processes is divided in row and column communicators, as described in the previous section. Each process $(I, J)$ belongs to its own `com_row` and `com_col` communicators, as shown in Figure 6.20.

Consider the sum in Eq.(6.1) written as $\mathbf{C}_{IJ} = \sum_{K=0}^{N_\mathrm{p}-1} \mathbf{A}_{I\,I+K}\mathbf{B}_{I+K\,J}$, where $I+K$ is computed $\mod N_\mathrm{p}$. Then, the processes execute $N_\mathrm{p}$ steps, each one of which consists of:

1. Broadcasting matrix $\mathbf{A}$: All processes on row $I$ identify the process holding the block $\mathbf{A}_{I\,I+K}$ and broadcast it across the entire row.

2. Performing a local matrix multiplication: Each process $(I, J)$ computes the matrix product $\mathbf{A}_{I\,I+K} \cdot \mathbf{B}_{I+K\,J}$, and adds the result to $\mathbf{C}_{IJ}$.

3. Shifting matrix $\mathbf{B}$: All processes in column $J$ shift the blocks of the matrix $\mathbf{B}$ cyclically upwards within their `com_col` column communicator. This way, the process $(I, J)$ receives $\mathbf{B}_{I+K+1\,J}$ from the process $(I + 1, J)$, to be used in the next step.

The first 3 steps of this algorithm are depicted in Figure 6.21.

The source code implementing the above algorithm in the function `pmmult_fox` is listed below. A test program using this function is in the file `mpi_13_mmultFox.f90`.

```
! Function implementing Fox's Parallel
! Matrix Multiplication Algorithm
 function           pmmult_fox(a, b) result(c)
  real(dp), intent(in)  :: a(:,:), b(:,:)
  real(dp), allocatable :: c(:,:)
  integer               :: nprocs, myrank, myi, myj
  integer               :: N_p, N_loc, N_loc2
  integer               :: k, ierr
  integer               :: root_rank_a
  integer               :: source_rank_b, dest_rank_b
  integer               :: myrank_row, myrank_col
  integer               :: com_row, com_col
  real(dp)              :: A_broadcasted(size(a,1), size(a,2))
  real(dp)              :: B_current    (size(b,1), size(b,2))

  call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

  ! --- 1. Initialization and Setup ---
  N_loc  = size(a,1) ; N_loc2 = N_loc * N_loc
  N_p    = int(sqrt(nprocs+1.0e-6_dp))
```

```
myi    = mod(myrank , N_p)  + 1
myj    =    (myrank / N_p)  + 1

allocate(c(N_loc,N_loc)); c = 0.0_dp

!Initial B_current block (B_ij)
B_current = b

call mpi_comm_split(MPI_COMM_WORLD, myi, myj, com_row, ierr)
myrank_row = myj - 1
call mpi_comm_split(MPI_COMM_WORLD, myj, myi, com_col, ierr)
myrank_col = myi - 1

! --- 2. The Main Fox's Algorithm Loop (N_p steps) ---
fox: do k = 0, N_p-1

 ! A) Broadcast Phase (A matrix)

 !The A block needed in step k is A_i, (i+k) mod N_p. This
 !block is initially held by the process at
 !(myi, (myi + k) mod N_p). This process becomes the root
 !for the broadcast along row myi.

 !Determine the local rank (within the com_row)
 !of the broadcast root.
 root_rank_a = mod(myrank_col + k, N_p)

 ! Only the root provides its A block to the broadcast buffer
 if (myrank_row  == root_rank_a) then
  A_broadcasted = a
 end if

 ! Broadcast A_i, (i+k) mod N_p along the current row
 call mpi_bcast(A_broadcasted, N_loc2, MPI_REAL8,          &
  root_rank_a, com_row, ierr)

 ! B) Local Matrix Multiplication

 c = c + matmul(A_broadcasted, B_current)

 ! C) Cyclic Shift Phase (B matrix)

 !Shift B upward in the column.
 !B_i,j goes to process ((i-1)mod N_p, j).

 !Notice the column major ordering of processes,
 !making them nearest neighbors!

 !Source      rank (B comes from the process below):
 source_rank_b = mod(myrank_col + 1      , N_p)

 !Destination rank (B goes  to   the process above):
 dest_rank_b   = mod(myrank_col - 1 + N_p, N_p)
```

```
    !Use MPI_Sendrecv_replace for in-place
    !shifting within the column

    !Note that B_current is the block being sent/received and
    !it is overwritten by the block from source_rank_b.
    call mpi_sendrecv_replace(B_current, N_loc2, MPI_REAL8,    &
      dest_rank_b, 0, source_rank_b, 0, com_col,               &
      MPI_STATUS_IGNORE, ierr)

  end do fox

  ! --- 3. Cleanup ---
  call mpi_comm_free(com_row, ierr)
  call mpi_comm_free(com_col, ierr)

end function       pmmult_fox
```

### 6.2.8 Scalability

Scalability analysis is the estimation of how computational and communication requirements depend on the size of the problem ($N$, the matrix size) and the amount of parallelization ($N_{\text{procs}}$, the number of processes). Project goals often dictate the optimization criteria: determining $N_{\text{procs}}$ that either minimizes the wall time or maximizes computational efficiency. Maximizing efficiency is crucial when minimizing the cost (monetary or energy) of the computation is the primary goal.

We analyze the scalability of Fox's matrix multiplication algorithm, introduced in Section 6.2.7.3.

#### Computational Cost ($t_{\text{comp}}$)

1. **Serial Cost**: Multiplying two $N \times N$ matrices requires $\mathcal{O}(N^3)$ floating point operations (flop). Assuming the time per flop is $f$, the serial computation time is:

$$t_{\text{serial}} \sim N^3 f\,. \tag{6.12}$$

2. **Parallel Computation Cost**: When the computation is distributed across $N_{\text{procs}} = N_{\text{p}} \times N_{\text{p}}$ processes, Fox's algorithm executes in $N_{\text{p}}$ steps. At each step, all processes concurrently perform a multiplication of two local $N_{\text{loc}} \times N_{\text{loc}}$ blocks. The total parallel wall time for computation is:

$$t_{\text{comp}} \sim N_{\text{p}}\, N_{\text{loc}}^3\, f = N_{\text{p}} \left(\frac{N}{N_{\text{p}}}\right)^3 f = \frac{N^3}{N_{\text{p}}^2}\, f = N^3\, (N_{\text{procs}})^{-1}\, f\,. \tag{6.13}$$

   If communication overhead is negligible, this yields maximal efficiency. The monetary cost (cost $\sim N_{\text{procs}}\, t_{\text{comp}} = N^3\, f$) is the same

as the serial cost, meaning the wall time is reduced by a factor of $1/N_{\text{procs}}$ without increasing expense.

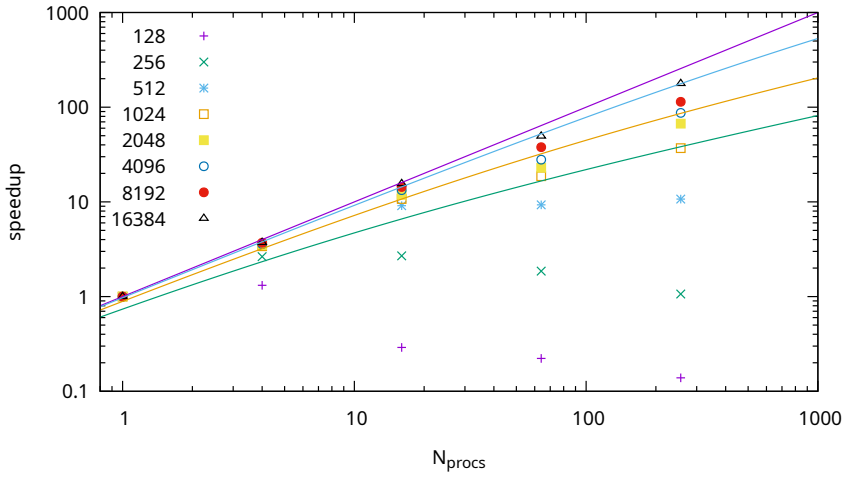



Figure 6.22: The speedup of Fox's algorithm, as a function of the number of processes $N_{\text{procs}}$. The speedup is defined as the ratio of $t_{\text{serial}}/t_{\text{parallel}}(N_{\text{procs}})$, where $t_{\text{serial,parallel}}$ are the wall times per matrix multiplication. The purple straight line is the line of perfect parallelization speedup $= N_{\text{procs}}$. The other lines are fits to Eq.(6.17), providing estimates for the ratio $f/r \approx 2 - 3 \times 10^{-3}$. $s'/f$ cannot be determined on the particular supercomputer, due to dominance of other factors of the underlying hardware.

**Communication Cost ($t_{\text{comm}}$)** In reality, the total runtime is increased by communication overhead, which is determined by two primary factors: the data volume (bandwidth) and latency.

1. **Volume Cost ($t_{\text{comm vol}}$)**: This cost is proportional to the volume of data exchanged, which consumes network bandwidth. At each of the $N_{\text{p}}$ steps, processes exchange two matrix blocks of size $N_{\text{loc}}^2$. If $r$ is the time required to communicate a unit of information[4] (e.g., a byte), the total volume time is:

$$t_{\text{comm vol}} \sim N_{\text{p}}\, N_{\text{loc}}^2\, r = N_{\text{p}} \left(\frac{N}{N_{\text{p}}}\right)^2 r = \frac{N^2}{N_{\text{p}}}\, r = N^2\, (N_{\text{procs}})^{-1/2}\, r \tag{6.14}$$

[4] If the bandwidth is e.g. 10 TB/sec, then $r = 1/(10^{13}\,\text{bytes/sec}) = 10^{-13}\text{sec/byte}$.

2. **Latency Cost ($t_{\text{comm lat}}$)**: This cost is the time required to initiate a message.

   - For cyclically shifting matrix **B** across the column communicator, the cost over $N_\text{p}$ steps is proportional to $N_\text{p}\, s$, where $s$ is the latency per message.
   - For broadcasting matrix **A** using the optimized `MPI_Bcast`, the latency cost is $\mathcal{O}(\ln N_\text{p}\, s')$.
   - The total latency time is estimated as:

$$t_{\text{comm lat}} \sim N_\text{p}\, s + N_\text{p} \ln(N_\text{p})\, s' = (N_\text{procs})^{1/2}\, s + \frac{1}{2}\,(N_\text{procs})^{1/2} \ln(N_\text{procs})\, s' \,. \tag{6.15}$$

3. **Total Communication Time**: The total communication time is dominated by:

$$t_\text{comm} = t_{\text{comm vol}} + t_{\text{comm lat}} \sim \frac{N^2}{(N_\text{procs})^{1/2}}\, r + (N_\text{procs})^{1/2} \ln(N_\text{procs})\, s' \,. \tag{6.16}$$

   For a very large number of processes, this latency cost can become the dominant factor limiting the algorithm's performance.

**Speedup and Efficiency** The speedup of the parallel program is defined as:

$$\text{speedup} = \frac{t_\text{serial}}{t_\text{comp} + t_\text{comm}} = \frac{N_\text{procs}}{1 + \frac{N_\text{procs}^{1/2}}{N^2}\,\frac{r}{f} + \frac{N_\text{procs}^{3/2}}{N^3}\,\frac{s'}{f}} \,. \tag{6.17}$$

When the matrix size is much larger than the number of processes ($N \gg N_\text{procs}$) the speedup $\sim N_\text{procs}$, indicating perfect parallelization. However, as $N_\text{procs}$ increases, the communication terms dominate[5] because the message passing time ($r$) and latency ($s'$) are typically orders of magnitude larger than the flop time ($f$), making parallelization expensive and ineffective.

The efficiency of the computation, a measure of the cost, is defined as:

$$\text{efficiency} \sim \frac{1}{N_\text{procs}} \text{speedup} = \frac{1}{1 + \frac{N_\text{procs}^{1/2}}{N^2}\,\frac{r}{f} + \frac{N_\text{procs}^{3/2}}{N^3}\,\frac{s'}{f}} \,. \tag{6.18}$$

For the same reasons, efficiency can deteriorate rapidly as $N_\text{procs}$ increases.

[5] Typical values of the ratios in 2025 are: $f/r \approx 10^{-2} - 10^{-4}$, and $f/s' \approx 10^{-6} - 10^{-9}$.

Figure 6.22 displays the empirical speedup of Fox's algorithm on the ARIS supercomputer in Greece. The plot shows the ratio of serial time ($t_{\text{serial}}$) to parallel time ($t_{\text{parallel}}$) as a function of $N_{\text{procs}}$ for $N_{\text{procs}} \leq 256$. For small matrices, the speedup is negligible, and can even drop below 1 for large $N_{\text{procs}}$. For large matrices, the algorithm shows good scalability and the measurements approach the line of perfect parallelization (speedup $= N_{\text{procs}}$).

By fitting the data to the speedup Eq.(6.17), the ratio of floating point time to message passing time is estimated to be $f/r \approx 2 - 3 \times 10^{-3}$. The latency ratio ($s'/f$) could not be reliably determined and was set to zero in the fits. This suggests the simple model's estimate of $t_{\text{comm lat}}$ may not fully account for other factors in the underlying hardware architecture, such as memory hierarchy or varied communication speeds between nodes and cores.

## 6.3 ScaLAPACK

In this book, we are primarily concerned with numerical problems in dense linear algebra (such as matrix multiplications, eigenvalues, and eigenvectors, etc.). The Scalable Linear Algebra PACKage (ScaLAPACK) [5, 6] is a library explicitly designed to perform such computations efficiently on distributed-memory parallel computers. Built upon the fundamental routines of the sequential LAPACK and BLAS libraries, ScaLAPACK extends this functionality to the parallel domain. Specifically, ScaLAPACK and the underlying Parallel BLAS (PBLAS) provide a standardized, portable, and high-performance framework for solving distributed-memory dense linear algebra problems, including matrix multiplication, solving linear systems, and computing eigenvalues and eigenvectors.

A primary feature of ScaLAPACK is its common interface, which is consistent with that of LAPACK and BLAS. This consistency greatly enhances the clarity and modularity of parallel codes, enabling researchers to transition from sequential to distributed-memory paradigms with minimal conceptual overhead. This design choice directly contributes to the portability of applications across various high-performance computing (HPC) platforms.

While ScaLAPACK remains a powerful and well-respected tool, its dominance in contemporary HPC environments has evolved. It is widely regarded as the foundational standard for distributed-memory dense linear algebra, but its architectural assumptions limit its performance relative to modern libraries. ScaLAPACK is actively maintained and remains a standard, reliable solution for its specific domain—dense linear algebra. However, for certain applications and in the context of emerging hardware architectures, several alternatives have gained significant traction. SLATE

[7, 8, 9] is considered a successor to ScaLAPACK[6], aiming for portable performance on accelerated architectures (GPUs) using dynamic scheduling and communication-avoiding algorithms. For GPU-accelerated implementations of BLAS and LAPACK-like features, NVIDIA's cuBLAS and cuSOLVER [10, 11] are specialized alternatives. ELPA [12] often outperforms ScaLAPACK for large-scale eigenvalue problems by implementing significantly faster parallel solution steps. MAGMA [13] focuses on utilizing both CPUs and GPUs within a single node, providing hybrid implementations of LAPACK routines. DPLASMA / Chameleon [14, 15] utilize a Directed Acyclic Graph (DAG) engine to dynamically schedule tasks, allowing for the overlap of communication and computation, thereby avoiding the bulk-synchronous design of ScaLAPACK. They provide efficient procedures for distributed dense linear algebra operations like Cholesky, LU, and QR factorizations.

In terms of scalability, ScaLAPACK and PBLAS exhibit generally good performance and efficiency on medium-to-large supercomputers, particularly when utilizing a few hundred to several thousand parallel processes for problems with sufficiently large matrix dimensions. The library is highly optimized to minimize the communication overhead associated with dense matrix operations, making it suitable for many scientific codes. However, as the number of processes enters the tens of thousands, the inherent latency and communication volume resulting from the classical block-cyclic data distribution can begin to limit scalability. Its parallelism model is Bulk-Synchronous Parallel (BSP), where all processes synchronize explicitly (fork-join) after large computational phases. Most modern libraries (SLATE, DPLASMA, MAGMA, ELPA) utilize the Task-Based/Dynamic Scheduling paradigm, which employs dependency graphs to break algorithms into smaller, asynchronous tasks, allowing for overlap of communication with computation.

### 6.3.1 BLACS

ScaLAPACK is built upon a fundamental communication layer known as the Basic Linear Algebra Communication Subprograms (BLACS). BLACS serves as the standard communication layer for linear algebra within ScaLAPACK and is specifically designed to manage the complexity of parallel execution on distributed-memory computers. The primary function of BLACS is to provide a 2D processor grid abstraction for parallel matrix operations. It defines the high-level communication primitives (such as point-to-point and collective operations) necessary for distributing and

[6]SLATE provides a ScaLAPACK Compatibility Layer. This ensures that existing programs that utilize compatible ScaLAPACK routines can run without significant modification, allowing researchers and legacy codes to leverage SLATE's modern optimizations while maintaining their original API calls.

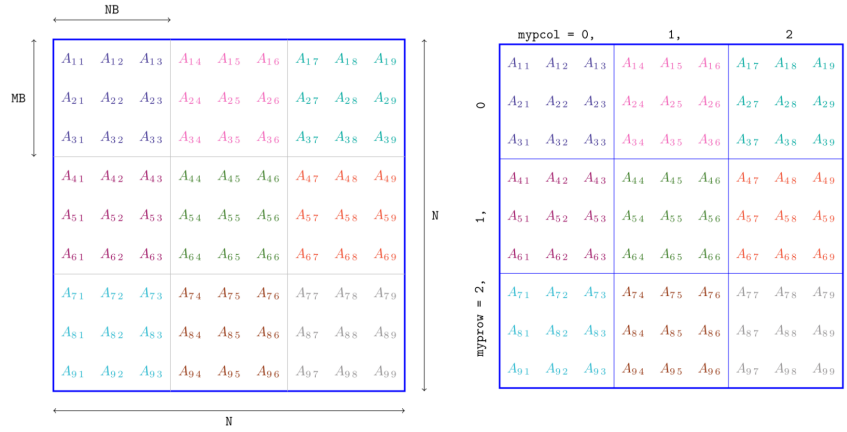



Figure 6.23: The block-cyclic data distribution for a $9 \times 9$ matrix, $\mathbf{A}$, mapped onto a $3 \times 3$ process grid. In this specific configuration, the global matrix size is $N = 9$, block sizes are $\texttt{MB} = \texttt{NB} = 3$, and the grid dimensions are $\texttt{nprow} = \texttt{npcol} = 3$. Since the block size equals the matrix dimension divided by the number of processes per dimension ($N_{\text{loc}} = N/\texttt{nprow} = 3$), the distribution is purely block-wise. Each process receives a single $N_{\text{loc}} \times N_{\text{loc}}$ block $\mathbf{A}_{IJ}$, where the process grid coordinates $(\texttt{myprow}, \texttt{mypcol})$ correspond to the block indices $(I-1, J-1)$. Matrix elements assigned to the same process are visually grouped by color.

manipulating matrices across an array of processes. BLACS itself does not implement these communication protocols; rather, it functions as a portability layer built on top of MPI.

BLACS effectively isolates the core ScaLAPACK routines from the specific details of direct MPI calls, thereby offering a consistent interface for communication. Each BLACS processor grid is associated with an existing MPI communicator. For instance, the core BLACS function `BLACS_Gridinit` accepts an existing communicator and logically decomposes it into a 2D grid of $N_{\text{prow}} \times N_{\text{pcol}}$ processes. This resulting grid is represented by an integer handle known as a *context*. This context is mandatory for all subsequent BLACS and ScaLAPACK calls, as it rigorously defines the communication domain and the process geometry.

Each process within this grid is assigned a unique row index, `myprow`, and column index, `mypcol`, which are indexed from zero:

$$\texttt{myprow} = 0, \ldots, \texttt{nprow} - 1\,, \quad \texttt{nprow} \equiv N_{\text{prow}} \tag{6.19}$$

$$\texttt{mypcol} = 0, \ldots, \texttt{npcol} - 1\,, \quad \texttt{npcol} \equiv N_{\text{pcol}} \tag{6.20}$$

The total number of processes in the grid is given by:

$$\texttt{nprocs} = \texttt{nprow} \times \texttt{npcol}\,. \tag{6.21}$$

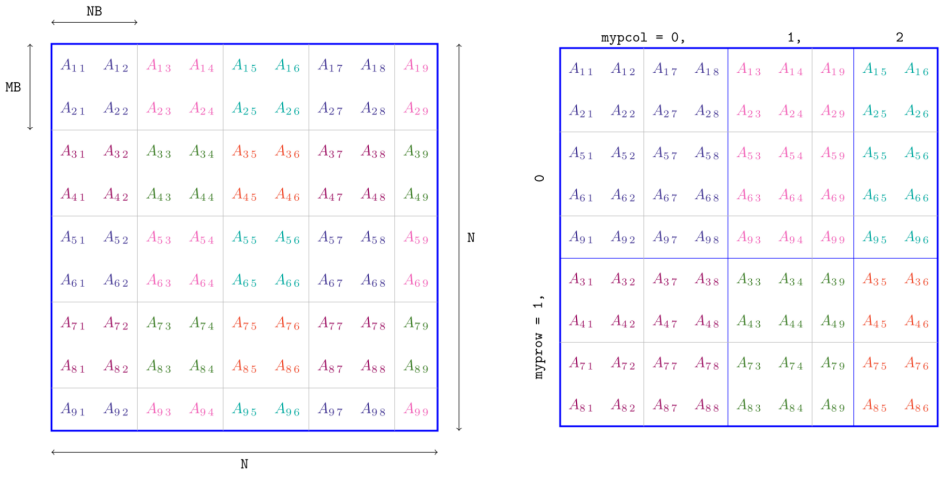



Figure 6.24: The block-cyclic data distribution for a $9 \times 9$ matrix, $\mathbf{A}$, mapped onto a rectangular $2 \times 3$ process grid. The global matrix size is $N = 9$, block sizes are $\texttt{MB} = \texttt{NB} = 2$, and the grid dimensions are $\texttt{nprow} = 2$ and $\texttt{npcol} = 3$. The overall matrix is partitioned into $M_{\text{loc}} \times N_{\text{loc}}$ blocks, and these blocks are distributed cyclically among the processes. Consequently, the local matrix stored on each process has a size of $M_{\text{loc}} \times N_{\text{loc}}$, which varies depending on the process's coordinates $(\texttt{myprow}, \texttt{mypcol})$ within the grid. Matrix elements assigned to the same process are visually grouped by color.

ScaLAPACK employs the *block-cyclic data distribution* scheme to partition a global matrix A onto the 2D processor grid. This method is the default and most versatile distribution for dense linear algebra because it ensures load balance and minimizes communication overhead.

The global $N \times N$ matrix[7] is conceptually partitioned into smaller, uniform blocks of size $\texttt{MB} \times \texttt{NB}$, where MB is the row block size and NB is the column block size. These blocks are then distributed among the $N_{\text{prow}} \times N_{\text{pcol}}$ process grid. The dimensions of the local matrix depend on the global matrix size $N$, the block sizes (MB, NB), the grid dimensions (nprow, npcol), and the coordinates (myprow, mypcol) of the owning process.

The core of the block-cyclic scheme is that it positions each $\texttt{MB} \times \texttt{NB}$ block contiguously in the local memory of the assigned process. However, neighboring blocks of the global matrix are distributed to different processors in a round-robin (cyclic) fashion. When we set $\texttt{MB} = \texttt{NB} = N/N_{\text{p}}$, the block-cyclic data distribution scheme simplifies to the purely block-wise distribution used throughout this chapter, as depicted in Figure 6.23. In this case, the local matrices $\mathbf{A}_{IJ}$ are $N_{\text{loc}} \times N_{\text{loc}}$ matrices, where $N_{\text{loc}} = \texttt{MB} = \texttt{NB} = N/N_{\text{p}}$, and $I = \texttt{myprow} + 1$, $J = \texttt{mypcol} + 1$. For

[7]Note that BLACS and ScaLAPACK are also capable of handling rectangular $M \times N$ matrices. For simplicity, we limit our discussion to square matrices.

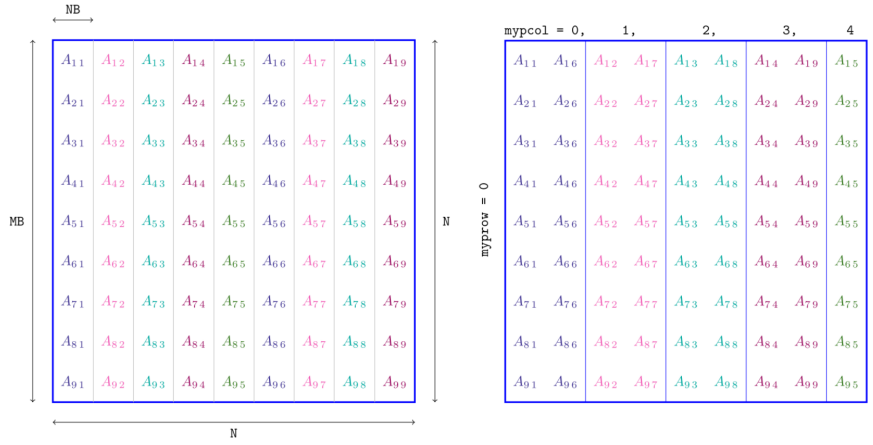



Figure 6.25: The block-cyclic data distribution for a $9 \times 9$ matrix, $\mathbf{A}$, mapped onto a thin, row–dominant $1 \times 5$ grid. The global matrix size is $N = 9$, block sizes are $\texttt{MB} = 9$ and $\texttt{NB} = 1$. The grid dimensions are $\texttt{nprow} = 1$ and $\texttt{npcol} = 5$. This configuration is often chosen to enhance load balance for specific algorithms, such as LU decomposition. The $M_{\text{loc}} \times N_{\text{loc}}$ blocks are distributed cyclically among the processes, resulting in local matrices whose size, $M_{\text{loc}} \times N_{\text{loc}}$, depends on the position of the process within the grid. Matrix elements assigned to the same process are visually grouped by color.

example, this is shown in Figure 6.23, where $N = 9$, and $\texttt{MB} = \texttt{NB} = 3$.

Such an equal-sized, block-wise distribution provides good load balance for operations like matrix multiplication, but it is often suboptimal for other algorithms, such as Gaussian elimination for solving linear systems or LU decomposition. BLACS offers a flexible interface to handle more complex data distributions. By setting the blocking factors MB and NB to other, potentially unequal values, or by constructing non-square process grids, more general distributions are obtained. For example, in the case of LU decomposition, a tall and thin process grid (e.g., $\texttt{nprow} = \texttt{nprocs}$, $\texttt{npcol} = 1$, or $\texttt{nprow} = 2\,\texttt{npcol} = \sqrt{2\,\texttt{nprocs}}$) may yield a more balanced work distribution among processes. The ScaLAPACK and BLACS libraries are specifically engineered to manage these non-square decompositions to enable a more finely tuned load balance appropriate for the needs of specific algorithms.

Examples of the block-cyclic data distribution scheme in more general cases are shown in Figures 6.24 and 6.25. Figure 6.24 depicts the distribution of a square matrix $\mathbf{A}$ ($N = 9$) on a process grid with $N_{\text{prow}} = 2$, and $N_{\text{pcol}} = 3$. The distribution applies the block-cyclic scheme by dividing the matrix into $\texttt{MB} \times \texttt{NB}$ ($\texttt{MB} = \texttt{NB} = 2$) blocks, and distributing these blocks cyclically among the grid processes. The total $N_{\text{procs}} = N_{\text{prow}} \times N_{\text{pcol}} = 6$

processes receive a matrix block $\mathbf{A}_{IJ}$, whose size is $M_{\text{loc}} \times N_{\text{loc}}$, varies among the processes.

Similarly, Figure 6.25 depicts the distribution of a square matrix $\mathbf{A}$ ($N = 9$) on a very thin process grid with $N_{\text{prow}} = 1$, and $N_{\text{pcol}} = 5$.

The local dimensions $M_{\text{loc}}$ and $N_{\text{loc}}$ depend on the global matrix size $N$, the block sizes (`MB`, `NB`), the process grid geometry, and the coordinates $(I, J) = (\texttt{myprow} + 1, \texttt{mypcol} + 1)$ of the receiving process. The algorithms used to compute the relationships between $M_{\text{loc}}$, $N$, `MB`, and $N_{\text{prow}}$, and the analogous relationship for the column dimension, can be found in the ScaLAPACK User's Guide [6]. The significant advantage of the BLACS library is that the programmer is not required to know these precise low-level relationships, but can instead use specialized BLACS functions to compute these dimensions.

The execution of a program utilizing ScaLAPACK and BLACS requires a distinct initialization sequence to structure the parallel environment. This process is sequential, building upon the established MPI environment:

1. **MPI Initialization**: First, the MPI environment must be initialized to determine the total number of processes (`nprocs`) and the rank of each process (`myrank`).

2. **Grid Geometry**: The desired process grid geometry must be specified by setting the row and column dimensions, `nprow` $= N_{\text{prow}}$ and `npcol` $= N_{\text{pcol}}$.

3. **BLACS Context**: BLACS is initialized by a call to `BLACS_Get`:

```
call blacs_get(-1,0,context)
```

   This obtains an integer handle, `context`, which represents the global communication domain for all subsequent ScaLAPACK and BLACS operations.

4. **Grid Creation**: This context is then used to define the process grid geometry via `BLACS_Gridinit`:

```
call blacs_gridinit(context,'Col-major', nprow, npcol)
```

   This routine assigns the `nprocs` processes to a virtual $N_{\text{prow}} \times N_{\text{pcol}}$ 2D process grid. The `'Col-major'` option assigns the grid coordinates in a column-major fashion (i.e., $0 \to (0,0)$, $1 \to (1,0)$, $2 \to (2,0)$, ..., $N_{\text{prow}} \to (0,1)$, ...), which is the convention maintained throughout this chapter for consistency with Coarray Fortran (CAF). A `'Row-major'` option is also available.

5. **Process Coordinates**: Finally, each process identifies its position within the newly created grid using `BLACS_Gridinfo`:

```
call blacs_gridinfo(context,nprow,npcol,myprow,mypcol)
```

This returns the local row coordinate, `myprow` $= I - 1$, and column coordinate, `mypcol` $= J - 1$, for the calling process.

After all parallel computation phases are complete, the environment must be properly shut down using either the BLACS finalization routine:

```
call blacs_exit(0)
```

or the standard MPI finalization routine:

```
call mpi_finalize(ierr)
```

Only one of these routines must be called, as calling both will result in a runtime error. The program listing below, found in the file `sca_01_hello.f90`, executes this minimal sequence:

```
program         scalapack_intro
 use, intrinsic           :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter      :: dp = real64
!-----------------------------------------------------------------
 integer                  :: nprocs, myrank, ierr
!-----------------------------------------------------------------
 integer                  :: context
 integer                  ::  nprow,  npcol
 integer                  :: myprow, mypcol, myid
 logical                  :: iamingrid
!-----------------------------------------------------------------
!Init mpi:
 call mpi_init     (                      ierr)
 call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

 nprow = int(sqrt(nprocs + 1.0e-6_dp)) ; npcol = nprocs/nprow
 if(myrank == 0 ) print '(A,3I5)','00 Init: ',nprocs,nprow,npcol
!-----------------------------------------------------------------
!Init BLACS:                        !nprocs same as from MPI call
 call blacs_pinfo   (myid,nprocs ) !myid = myrank
 call blacs_get     (-1,0,context)
 call blacs_gridinit(context,'Col-major', nprow, npcol)
 call blacs_gridinfo(context,nprow,npcol,myprow,mypcol)
!-----------------------------------------------------------------
```

```
!Do I belong to the grid? If not, then exit graciously:
 iamingrid    =                                                  &
 (myprow >= 0) .and. (myprow < nprow)                            &
 .and.                                                           &
 (mypcol >= 0) .and. (mypcol < npcol)                            &
!----------------------------------------------------------------
 if( .not. iamingrid ) then
  call blacs_exit(0)
  stop
 end if
!----------------------------------------------------------------
 print '(A,I0.2,2I5)','01 Rank: ',                               &
  myrank,myid,myprow,mypcol,nprocs,nprow,npcol
!----------------------------------------------------------------
!Sync everything: use the A,R, or C scope
!'R' for row, 'C' for column, 'A' for all
 call blacs_barrier(context,'A')
!----------------------------------------------------------------
!*alternatively* you may use call mpi_finalize(ierr)
 call blacs_exit(0)
!----------------------------------------------------------------
end program     scalapack_intro
```

It is important to note that when running with a number of processes greater than $N_{\text{prow}} \times N_{\text{pcol}}$, some processes will not be assigned to the grid. This condition is typically computed and tracked using a logical variable, such as `iamingrid`.

The synchronization call `call blacs_barrier(context,'A')` is executed by all processes associated with the given `context`. All processes wait at this barrier until all active processes in the grid have reached it. Using the standard `mpi_barrier(MPI_COMM_WORLD,ierr)` can cause a deadlock if non-grid processes exit prematurely. It is often best practice to define a separate communicator, `comm_grid`, that contains only the processes assigned to the BLACS grid.

When compiling a program that uses ScaLAPACK, the library must be explicitly linked to your executable. For the packages recommended in this book, this typically involves linking the OpenMPI-compatible shared library, often named `libscalapack-openmpi.so`.

The compilation and execution are performed using the MPI wrappers:

```
mpifort sca_01_hello.f90 -lscalapack-openmpi -o m

mpirun -n 4 ./m
mpirun -n 5 ./m
```

The `mpifort` command handles the necessary linking. The `mpirun` command then launches the executable, `m`, on the specified number of processes, as is standard for any MPI program. Running the program with 5

processes (as opposed to 4, assuming a $2 \times 2$ grid) demonstrates the behavior of processes that are not successfully assigned to the BLACS process grid.

### 6.3.2 Naming Conventions

It is highly recommended that one invests time in understanding the naming and calling sequence conventions of BLACS, PBLAS, and ScaLAPACK. Gaining sufficient familiarity significantly simplifies the process of searching for the appropriate subroutine or function, correctly providing input parameters, and utilizing the programming interface efficiently. Programmers already familiar with the use of the sequential BLAS and LAPACK libraries will find the interface with the corresponding PBLAS and ScaLAPACK subroutines conceptually straightforward.

To search for available procedures, the reader is encouraged to browse the comprehensive online reference guides for BLACKS, PBLAS, and ScaLAPACK [16, 17, 18]. These resources provide a concise description of each procedure's function and a detailed breakdown of its calling sequence.

The naming of most BLACS routines is distinguished by the prefix `BLACS_`. We have already encountered basic initialization and shutdown subroutines such as `BLACS_Pinfo`, `BLACS_Get`, `BLACS_Gridinit`, `BLACS_Gridinfo`, `BLACS_Barrier`, and `BLACS_Exit`. Additionally, there are communication and combine operation procedures that adhere to the PBLAS/ScaLAPACK conventions detailed below.

The naming scheme for virtually all ScaLAPACK data type and precision, is directly inherited from BLAS/LAPACK conventions. The typical structure for a procedure name follows the form:

`[P][T][YY][ZZZ]`,

This sequence is broken down into four key components:

1. **Parallel Prefix (`P`)**: The routine name begins with `P` to signify that it is a Parallel routine (PBLAS or ScaLAPACK).

2. **Type and Precision (`T`)**: The second character specifies the data type and precision, directly corresponding to BLAS/LAPACK conventions:

   - `S`: Single precision real.
   - `D`: Double precision real.
   - `C`: Single precision complex.
   - `Z`: Double precision complex.
   - `I`: Integer data, used primarily in BLACS procedures.

3. **Matrix Type Suffix (YY)**: This two-character sequence identifies the properties of the matrix being operated upon, which guides the selection of the internal algorithm. This is consistent with LAPACK conventions:
    - GE: General matrix.
    - HE: Hermitian matrix.
    - SY: Symmetric matrix.
    - TR: Triangular matrix.
    - UN: Unitary matrix.
    - OR: Orthogonal matrix.
4. **Driver/Computational Routine Suffix (ZZZ)**: The remaining characters denote the specific linear algebra function to be performed, often aligning directly with LAPACK and BLAS functions:
    - MV: Matrix-Vector multiplication.
    - MM: Matrix-Matrix multiplication.
    - SV: Linear Systems solution.
    - QRF: QR Factorization.
    - EV: Eigenvalues and eigenvectors.
    - LS: Least Squares problem solution.
    - SVD: Singular Value Decomposition.
    - TRI: Inverse of a matrix.

The fundamental difference in the API between the sequential and parallel libraries lies in how the matrix data is referenced. While sequential routines primarily pass local memory addresses and dimensions, parallel routines must additionally pass extensive metadata describing the matrix's complete distribution across the process grid.

We now present an example using the BLAS matrix multiplication subroutine DGEMM and its corresponding PBLAS routine, PDGEMM:

- The primary distinction is the initial P in PDGEMM, which indicates its parallel function.
- The letter D signifies that the routines operate on matrices with elements that are double-precision real ( real(8) ) scalar variables.
- The characters GE confirm that the matrices are general (without special structure, such as symmetry, orthogonality, or triangular form).
- The suffix MM denotes the function of the subroutine: to perform Matrix–Matrix multiplication.

The Fortran calling sequences for these two routines are:

```
call  dgemm(ta,tb,M,N,K,alpha,A,       LDA   ,&
                               B       ,LDB   ,&
                         beta ,C       ,LDC    )
call pdgemm(ta,tb,M,N,K,alpha,A,IA,JA,DESCA,&
                               B,IB,JB,DESCB,&
                         beta ,C,IC,JC,DESCC )
```

Assuming `ta=tb='N'` (no transpose), both routines compute the operation $\mathbf{C} = \texttt{alpha}\,\mathbf{A}\cdot\mathbf{B} + \texttt{beta}\,\mathbf{C}$. The global matrices **A**, **B**, and **C**, have dimensions $\texttt{M}\times\texttt{K}$, $\texttt{K}\times\texttt{N}$, and $\texttt{M}\times\texttt{N}$, respectively, and are distributed onto the local arrays `A`, `B`, and `C`.

For the serial `DGEMM`, the arrays are described by their *leading dimensions* (`LDA`, `LDB`, and `LDC`), which, for column-major Fortran arrays, equal the number of their rows. The leading dimension may be larger than the corresponding matrix dimension (`M`,`K`,`M`) if the matrix being multiplied is a section of a larger array.

The PBLAS routine `PDGEMM` requires supplementary information to describe the distribution of data across the process grid:

- The integers `IA`,`JA` specify the global row and column indices that mark the beginning of matrix **A**.
- The array `DESCA` is the *descriptor* of the array `A`. It provides a rigorous definition of how the global matrix is partitioned, sliced, and mapped onto the 2D process grid. This detail is necessary because ScaLAPACK routines need to know how the elements of the global matrix **A** are placed within the local array `A` on each process. See Appendix 6.A for more details.

Similar distributional information (`IB`, `JB`, `DESCB` and `IC`, `JC`, `DESCc`) is provided for matrices **B** and **C**.

Another example is the comparison between the LAPACK subroutine `ZHEEV` and its ScaLAPACK counterpart, `PZHEEV`. Both procedures are designed to compute the eigenvalues and, optionally, the eigenvectors of a Hermitian matrix **A**.

The name `ZHEEV` conveys the function and matrix properties:

- The letter `Z` signifies that the matrix is constructed using double-precision complex ( `complex(8)` ) variables.
- The characters `HE` indicate that the matrix is Hermitian.
- The final characters `EV` denote that the subroutine's function is to solve an eigenproblem.

Their Fortran calling sequences are:

```
call  zheev(JOBZ,UPLO,N,A       ,LDA   ,W,&
            WORK,LWORK,RWORK,LRWORK,INFO)
call pzheev(JOBZ,UPLO,N,A,IA,JA,DESCA,W,&
                        Z,IZ,JZ,DESCZ,  &
            WORK,LWORK,RWORK,LRWORK,INFO)
```

The routines compute the real eigenvalues of the N × N matrix **A** and place them in the array W. The key difference in the output is that for ZHEEV, the eigenvectors (if requested) are returned in the columns of the input array A, whereas for PZHEEV, they are returned in the columns of the separate array Z.

Aside from this slight difference and the extra arguments necessary to specify the global array distribution (IA,JA,DESCA, etc.), the two calling sequences remain conceptually quite similar. The purpose of the remaining arguments will be detailed in a subsequent discussion (see Section 6.3.6).

### 6.3.3 Distributing Arrays

Assume an $N \times N$ matrix **A** is to be distributed across an $N_{\text{prow}} \times N_{\text{pcol}}$ process grid. Following the initialization steps in Section 6.3.1, we obtain a BLACS context context defined by $N_{\text{prow}}$ (nprow) and $N_{\text{pcol}}$ (npcol). The choice of row and column block sizes (MB and NB) determines the dimensions of the local array, amat(mloc,nloc), that stores **A**'s elements on each process (e.g., Figures 6.23 – 6.25). The local array dimensions (mloc and nloc) are calculated using ScaLAPACK tools.

We start by describing the construction of the descriptor desca of the array amat. The array descriptor is a fixed-size integer array defined as:

```
integer, parameter       :: DLEN = 9
integer, dimension(DLEN) :: desca
```

The descriptor is built using the ScaLAPACK subroutine DESCINIT:

```
call descinit(desc,M,N,MB,NB,irsrc,icsrc,ictxt,LLD,info)
```

Here, desc(DLEN) is the computed descriptor. The inputs required are the dimensions of the global matrix (M × N), the block sizes ( MB × NB), and the leading dimension (LLD) of the local array amat. The integers irsrc and icsrc are the row and column process coordinates, respectively, that hold the first row and column of the global matrix.

For our $N \times N$ matrix, where M = N = nmat, we typically choose the process at grid coordinates (irsrc, icsrc) = (0, 0) to hold the first global matrix element, and we set the local array's leading dimension LLD to mloc. The array descriptor desca is computed by the call:

```
call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)
```

The actual shape of the local matrix (i.e., its dimensions `mloc` and `nloc`) on a process with coordinates (`myprow`, `mypcol`) is computed using the ScaLAPACK function `NUMROC`:

```
integer :: numroc
mloc = numroc(nmat,MB,myprow,0,nprow)
nloc = numroc(nmat,NB,mypcol,0,npcol)
```

The general calling sequence of `NUMROC` is:

```
NLOC = numroc(N,NB,iproc,isrcproc,nprocs)
```

In this sequence, `N` represents the size of the corresponding dimension of the global matrix, and `NB` is the row or column block size. The parameter `iproc` is the coordinate of the process whose local array dimension is being determined, while `isrcproc` is the coordinate of the process that possesses the first row or column of the distributed matrix. Finally, `nprocs` is the number of row or column processes over which that dimension of the matrix is distributed.

Once the array descriptor (`desca`) and the shape of the local array (`amat(mloc,nloc)`) are determined, the local matrix elements can be computed using the element values of the global matrix $\mathbf{A}$, a process managed by the ScaLAPACK subroutine `PDELSET`.

The code listing below summarizes the sequence of necessary steps:

```
mloc = numroc(nmat,MB,myprow,0,nprow)
nloc = numroc(nmat,NB,mypcol,0,npcol)
call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)

do  imat = 1,nmat ;  do jmat = 1, nmat
 call pdelset(amat,imat,jmat,desca,amat_glob(imat,jmat))
end do            ;  end do
```

It is important to note that the explicit construction of the full global array `amat_glob(nmat,nmat)` is not strictly necessary. The final argument of `PDELSET` is simply a real scalar value that can be generated or computed locally on the process making the call.

The calling sequence of `PDELSET`[8] is:

[8]For other data types, the corresponding routines are PSELSET, PCELSET, and PZELSET for real(4), complex(4), and complex(8), respectively, following ScaLAPACK's naming conventions. Similarly, for the related PDELGET routine the corresponding routines are PSELGET, PCELGET, and PZELGET.

```
call pdelset( A, IA, JA, DESCA, ALPHA )
```

where:

- A is the real(8) local array on the calling process.
- IA and JA are the row and column indices of the global matrix **A**.
- DESCA is the descriptor of the local array A.
- ALPHA is the scalar real(8) value corresponding to $\mathbf{A}_{\mathtt{IA\,JA}}$.

The use of PDELSET abstracts away the local implementation details: it determines if and where in A the matrix element $\mathbf{A}_{\mathtt{IA\,JA}}$ should be stored based on the array distribution scheme. Consequently, many pairs of global indices (IA, JA) passed to this routine will be irrelevant to the memory space of the process making the call.

The inverse procedure –constructing the global array from the distributed local array– is achieved using the ScaLAPACK subroutine PDELGET. Given a local array amat(mloc,nloc) and its descriptor desca, we can construct the global array amat_glob containing all matrix elements of **A** as follows:

```
do imat = 1, nmat ; do jmat = 1, nmat
 call pdelget('A',' ',amat_glob(imat,jmat),amat,imat,jmat,desca)
end do            ; end do
```

The first character 'A' signifies that amat_glob(imat,jmat) is updated on all processes of the grid, and the second character signifies the topology to be used for broadcasting (here it is left empty).

The general calling sequence of PDELGET is:

```
call pdelget(scope,topology,ALPHA,A,IA,JA,DESCA)
```

In this routine:

- A, IA, JA, and DESCA maintain the same meaning as defined for PDELSET.
- The real(8) scalar ALPHA is assigned the value of the matrix element $\mathbf{A}_{\mathtt{IA\,JA}}$

The key control is provided by the scope argument, which dictates the scope of the update:

- If scope is set to 'A', ALPHA (and thus $\mathbf{A}_{\mathtt{IA\,JA}}$) is updated on all processes of the grid. This is how amat_glob is constructed globally in the example above.

- If `scope` is set to `'R'`, `ALPHA` is updated only on the process row containing $\mathbf{A}_{\mathtt{IA\,JA}}$.
- If `scope` is set to `'C'`, `ALPHA` is updated only on the process column containing $\mathbf{A}_{\mathtt{IA\,JA}}$.
- If `scope` is set to any other value, `ALPHA` is updated only on the process that owns $\mathbf{A}_{\mathtt{IA\,JA}}$.

The `topology` argument determines the specific communication pattern (e.g., ring, tree) to be used if broadcasting is required; otherwise, it is set to a space character (`' '`).

The file `sca_02_array.f90` contains an example of a code programming the distribution of a particular matrix $\mathbf{A}$ on a process grid.

### 6.3.4 Matrix Multiplication

For matrix-matrix multiplication involving `real(8)` matrices, ScaLAPACK provides the routine `PDGEMM`, which is the parallel counterpart to the sequential BLAS routine `DGEMM`. The underlying `DGEMM` is used internally for the local matrix multiplications, taking advantage of vendor-specific optimizations in their BLAS implementations.

The primary advantage of using `PDGEMM` over custom parallel implementations (like Fox's algorithm discussed earlier in this chapter) is its enhanced performance and simplified user interface. The `PDGEMM` routine utilizes highly tuned algorithms for inter-process communication, which are managed by the BLACS layer. These algorithms incorporate sophisticated block-cyclic data distribution and optimized collective communication patterns designed to minimize latency and maximize network bandwidth. While contemporary libraries favor task-based scheduling, ScaLAPACK's Bulk-Synchronous Parallel (BSP) model, when applied to a dense, regular operation like matrix multiplication, often provides a high-efficiency implementation by guaranteeing that compute time is maximized between explicit synchronization points.

As we have seen in the discussion on page 272, a significant benefit of ScaLAPACK is its consistency with the established LAPACK and BLAS naming and calling conventions. The general Fortran calling sequence for `PDGEMM` is shown on page 272.

To ensure coding transparency and simplicity in the implementation of parallel routines, we assume that the program uses the following global data structure, which defines the environment and array properties on every process:

```
integer          :: nmat          ! global matrix size
integer          :: mloc, nloc    ! local  matrix size
```

```
 integer             :: MB  , NB       ! block sizes
 integer             :: nprocs, myrank ! MPI process info
 integer             :: comm_grid      ! Grid MPI communicator
 integer             :: context        ! global BLACS context
 integer             ::  nprow,  npcol ! grid size
 integer             :: myprow, mypcol ! and coordinates
 integer, parameter:: DLEN = 9
 integer             :: desca(DLEN)    ! universal descriptor
```

These variables are expected to be properly initialized prior to calling the parallel linear algebra routines.

These global data variables simplify the user interface by being accessible to all internal procedures. Matrix multiplication is encapsulated and computed by the subroutine `sca_mmult`:

```
 subroutine          sca_mmult   (a, b, c )
  real(dp), intent(in) :: a(mloc,nloc), b(mloc,nloc)
  real(dp)             :: c(mloc,nloc)
  real(dp)             :: alpha, beta
  !-----------------------------------------------
  alpha = 1.0_dp ; beta = 0.0_dp
  call pdgemm(                          &
       'N','N',nmat,nmat,nmat,alpha, &
       a,1,1,desca,                     &
       b,1,1,desca,           beta , &
       c,1,1,desca)
  call blacs_barrier(context,'A')
  !-----------------------------------------------
 end subroutine      sca_mmult
```

The subroutine assumes that variables such as `nmat`, `mloc`, `nloc`, and the universal matrix descriptor `desca` have been correctly initialized elsewhere. The `PDGEMM` routine is called to compute $\mathbf{C} = \texttt{alpha}\,\mathbf{A}\cdot\mathbf{B} + \texttt{beta}\,\mathbf{C}$, where `alpha` $= 1$ and `beta` $= 0$. Then, the usage of the subroutine has a very simple calling sequence:

```
 call sca_init ! initialize everything
!Initialize arrays, and define their values, e.g.:
 allocate(amat(mloc,nloc))
 allocate(bmat(mloc,nloc))
 allocate(cmat(mloc,nloc))
 call random_number(amat)
 call random_number(bmat)
!compute the matrix multiplication of amat with bmat
!store the result in cmat:
 call sca_mmult(amat,bmat,cmat)
```

The complete implementation required to perform this parallel matrix-matrix multiplication is contained in the file `sca_03_mmult.f90`. This file includes the necessary array distribution subroutines, `sca_loc2glob`

and sca_glob2loc, and a comprehensive initialization routine, sca_init, which executes the entire sequence of MPI and BLACS setup, including the computation of mloc, nloc, and desca, along with consistency checks. The sca_loc2glob subroutine is used to transfer data from the distributed local arrays to a centralized global array, which allows the program to compare the result of the parallel PDGEMM operation against a standard serial MATMUL on the global matrix for verification.

### 6.3.5 Transpose and Traces

In this section, we show how to compute the matrix transpose $\mathbf{A}^T$ of an $N \times N$ matrix distributed over a process grid. Subsequently, we show how to use $\mathbf{A}^T$ to compute the trace squared, $\mathrm{tr}\,\mathbf{A}^2$, according to Eq.(6.4). Finally, we demonstrate the efficient computation of the matrix trace, $\mathrm{tr}\,\mathbf{A}$.

The ScaLAPACK subroutine used to compute the transpose $\mathbf{A}^T$ for matrices with real(8) elements is PDTRAN. Its full calling sequence is:

```
call pdtran(M,N,ALPHA,A,IA,JA,DESCA,BETA,C,IC,JC,DESCC)
```

This subroutine computes the matrix operation C = alpha $\mathtt{A}^T$ + beta C, where alpha and beta are real(8) scalar coefficients. The arrays A and C represent M × N global matrices. Their respective starting elements are indexed by (IA, JA) and (IC, JC) within their context.

With the conventions established on page 276, the parallel matrix transpose of $\mathbf{A}$ (stored in a) to produce $\mathbf{A}^T$ (stored in at) can be encapsulated within the subroutine sca_transpose:

```
subroutine          sca_transpose(a,at)
 real(dp), intent(in) :: a (mloc,nloc)
 real(dp)             :: at(mloc,nloc)
 real(dp)             :: alpha, beta
 !-------------------------------------
 alpha = 1.0_dp ; beta = 0.0_dp
 call pdtran(                          &
      nmat, nmat,  alpha,              &
      a ,1,1,desca, beta,              &
      at,1,1,desca)
end subroutine      sca_transpose
```

The calculation of $\mathrm{tr}\,\mathbf{A}^2$ s straightforward once the transpose is available, by using Eq.(6.4). This procedure is encapsulated in the subroutine sca_trace2, which includes a final collective summation step:

```
subroutine          sca_trace2   (a, tra2 )
 real(dp), intent(in) :: a (mloc,nloc)
 real(dp)             :: tra2
```

```
 real(dp)              :: at(mloc,nloc)
 real(dp)              :: tra2_loc

 call sca_transpose(a,at)

 tra2_loc = sum(a * at)

 call mpi_allreduce(tra2_loc,tra2,1,MPI_REAL8,MPI_SUM,&
  comm_grid,ierr)

end subroutine      sca_trace2
```

Computing the matrix trace, $\mathrm{tr}\,\mathbf{A}$, requires careful attention to avoid unnecessary communication overhead. Although it is fundamentally an $\mathcal{O}(N)$ computation, a naive approach using `PDELGET` (see page 292) to retrieve each diagonal element globally would be inefficient:

```
 tra     = 0.0_dp
 do imat = 1, nmat
  !Large communication overhead: avoid!
  call pdelget('A',' ',aii,a,imat,imat,desca)
  tra    = tra + aii
 end do
```

This method generates a large number of communication calls. Since the diagonal elements of the global matrix $\mathbf{A}$ are generally distributed across various memory locations, they are not necessarily mapped to the diagonal elements of the local array `a` (as illustrated, for example, in Figure 6.24).

A more efficient method is to locally identify and sum the diagonal elements present on each process, followed by a single collective summation. This is achieved using the ScaLAPACK function `INDXL2G` (INDeX–Local–to-Global), which maps a local array index to its corresponding global matrix index. The `INDXL2G` function computes the global row or column index (`indxglob`) corresponding to a local index (`indxloc`). Its general calling sequence is:

```
indxglob = indxl2g(indxloc,NB,iproc,isrcproc,nprocs)
```

Here, `indxglob` is the returned global index. The inputs are the local array index (`indxloc`), the blocking factor (`NB` or `MB`), the process coordinate (`iproc`), the source process coordinate (`isrcproc`), and the number of processes in that dimension (`nprocs`).

Using the established conventions, the global indices (`imat_glob` $\equiv a$, `jmat_glob` $\equiv b$) corresponding to the local array element `a(imat,jmat)` are calculated as:

```
imat_glob = indxl2g(imat, MB, myprow, 0, nprow)
jmat_glob = indxl2g(jmat, NB, mypcol, 0, npcol)
```

The final computation of the trace is encapsulated in the subroutine `sca_trace`. It iterates through all local array elements, checks if their corresponding global indices satisfy the diagonal condition ($a = b$), sums these local contributions, and performs a single collective reduction:

```
subroutine         sca_trace    (a, tra  )
 real(dp), intent(in) :: a (mloc,nloc)
 real(dp)             :: tra
 real(dp)             :: tra_loc
 integer              :: imat     , jmat
 integer              :: imat_glob, jmat_glob
 integer , external   :: indxl2g

 tra_loc     = 0.0_dp
 do  jmat    = 1, nloc ; do imat    = 1, mloc

   imat_glob = indxl2g(imat, MB, myprow, 0, nprow)
   jmat_glob = indxl2g(jmat, NB, mypcol, 0, npcol)

   if( imat_glob == jmat_glob) tra_loc = tra_loc + a(imat,jmat)

  end do               ; end do

 call mpi_allreduce(tra_loc, tra, 1, MPI_REAL8, MPI_SUM,       &
                    comm_grid, ierr)
end subroutine     sca_trace
```

The file `sca_04_trace.f90` contains the complete testing code necessary to compute $\mathbf{A}^T$, $\mathrm{tr}\,\mathbf{A}^2$, $\mathrm{tr}\,\mathbf{A}$, and verify the accuracy of the results.

### 6.3.6 Eigenvalues and Eigenvectors

Computing eigenvalues and eigenvectors is a critical operation in numerical computations. In this section, we discuss the computation of the real eigenvalues and eigenvectors of a real symmetric matrix, where $\mathbf{A} = \mathbf{A}^T$.

The defining equation for the eigenproblem is:

$$\mathbf{A} \cdot \mathbf{v} = \lambda \mathbf{v} \,. \tag{6.22}$$

In this equation, $\mathbf{A}$ is the $N \times N$ matrix, $\lambda$ is the eigenvalue, and $\mathbf{v}$ the non–zero eigenvector. For a real symmetric matrix $\mathbf{A}$, all eigenvalues are real, and the corresponding eigenvectors can be chosen to have real components and to form an orthonormal basis

$$\mathbf{v}_i \cdot \mathbf{v}_j = \delta_{ij} \,. \tag{6.23}$$

The computational complexity of solving the dense symmetric eigenproblem is determined by the theoretical parallel wall time scaling $\mathcal{O}(N^3/N_{\text{procs}})$. While the problem scales similarly with the matrix size in both cases, the computational effort for calculating the eigenvectors is higher than that for calculating only the eigenvalues:

1. **Reduction**: For eigenvalues only, the primary step involves reducing the dense matrix to a simpler form, typically a tridiagonal matrix, using Householder transformations.

2. **Iterative Solution**: This is followed by an iterative method (such as the QR algorithm or bisection/inverse iteration) to find the eigenvalues of the resulting tridiagonal matrix.

3. **Back Transformation**: Computing the eigenvectors requires performing the tridiagonal-to-dense back transformation for all $N$ eigenvectors. This process involves multiplying the eigenvectors of the tridiagonal matrix by the product of the Householder reflectors generated during the initial reduction.

The accuracy of the computed eigenpairs $(\lambda_i, \mathbf{v}_i)$ is checked by calculating the associated residuals. The residual error of the defining equation Eq.(6.22) is given by:

$$\mathbf{r}_i = \frac{1}{N} \left( \mathbf{A} \cdot \mathbf{v}_i - \lambda_i \, \mathbf{v}_i \right) . \tag{6.24}$$

Accuracy is confirmed when the magnitude of the residual, $|\mathbf{r}_i|$, is small (i.e., near machine epsilon). Similarly, the normalization and the orthogonality errors can be defined by:

$$r = |\mathbf{v}_i \cdot \mathbf{v}_i - 1| \, , \tag{6.25}$$

$$r_{ij} = |\mathbf{v}_i \cdot \mathbf{v}_j| \, , \quad i \neq j \, . \tag{6.26}$$

The standard methodology for solving the dense symmetric eigenproblem on distributed-memory architectures is provided by the ScaLAPACK library through the routine `PDSYEV`. This routine is the parallel analogue of the LAPACK routine `DSYEV` and is designed to compute all eigenvalues and, optionally, the corresponding eigenvectors of a distributed *real symmetric matrix* $\mathbf{A}$. The `PDSYEV` routine's name follows the ScaLAPACK conventions: `D` for `real(8)` arrays, `SY` for symmetric matrices, and `EV` for eigenvalues/eigenvectors. Note that this library provides similar routines for other types (e.g., `PSSYEV` for `real(4)` symmetric arrays, and `PZHEEV`/`PCHEEV` for complex Hermitian matrices), but no routines for general non-symmetric matrices (`PDGEEV`/`PZGEEV`) are included.

The calling sequence of `PDSYEV` is:

```
call pdsyev(JOBZ,UPLO, N,      &
    A    ,IA,JA,DESCA,         &
    EVS  ,                     &
    EVEC,IE,JE,DESCE,          &
    WORK,LWORK,INFO)
```

The character variable JOBZ is an option for the required output: JOBZ='N' requests only the eigenvalues, while JOBZ='V' requests both eigenvalues and eigenvectors.

The character variable UPLO indicates whether the upper ( UPLO='U' ) of lower ( UPLO='L' ) triangular part of the symmetric matrix is supplied. Its value is not important if the entire matrix has been computed and stored in A.

The real(8) array A contains the local portion of the matrix to be diagonalized. The global matrix's starting element $\mathbf{A}_{11}$ has indices (IA, JA) in the global array. The input array A *is overwritten* (partially destroyed) upon completion of the PDSYEV routine; a copy must be retained if the original matrix is needed.

The ScaLAPACK routine imposes constraints on the matrix distribution, requiring MB = NB and necessitating that the starting global indices satisfy mod (IA−1, MB) = mod (JA−1, NB) = 0. The ScaLAPACK online manual [6] should be consulted for full details.

The eigenvalues are placed in the real(8) array section EVS(1:N).

If JOBZ='V', the real(8) array EVEC contains the eigenvectors. For the global eigenvector matrix EVEC_GLOB(N,N), the $i$–th eigenvector is $\mathbf{v}_i$ = EVEC_GLOB(1:N,$i$). The descriptor for EVEC is the integer array DESCE, with its global start defined by (IE, JE).

The real(8) array WORK serves as a memory buffer for PDSYEV's internal calculations and must have a length of at least LWORK. The optimal value of LWORK is calculated by the routine and returned in WORK(1). The calculation of the optimal workspace size and the subsequent computation is therefore performed in a two-step calling procedure:

1. **Query**: The first call sets LWORK = −1. The routine then calculates and returns the optimal LWORK value in WORK(1), without actually solving the eigenproblem.

2. **Compute**: The WORK array is dynamically allocated based on the returned optimal LWORK value. The second call to PDSYEV then proceeds to solve the eigenproblem, returning the eigenvalues and (optionally) the eigenvectors.

The calculation can be encapsulated in the subroutine sca_evs listed below:

```
subroutine         sca_evs      (ain, evec, evs)
 real(dp), intent(in) :: ain (mloc,nloc)
 real(dp)             :: evec(mloc,nloc), evs(nmat)
 real(dp)             :: a   (mloc,nloc)
 real(dp), allocatable:: work(:)
 real(dp)             ::qwork(1)
 integer              ::lwork, info
 character(100)       :: errmes
!-----------------------------------------------------------------
 if(MB /= NB) call sca_abort("pdsyev requires MB = NB")
!-----------------------------------------------------------------
 a = ain  ! local copy, keep ain intact
!-----------------------------------------------------------------
!Query size of work(lwork):
 call pdsyev(                          &
      'V' ,'U',nmat,                   &
      a   ,1,1,desca,                  &
      evs ,                            &
      evec,1,1,desca,                  &
      qwork,-1,info)
! take care of truncation errors with a small epsilon:
 lwork = int(qwork(1)+1.0e-4_dp)
!-----------------------------------------------------------------
 allocate(work(lwork))
!-----------------------------------------------------------------
 call pdsyev(                          &
      'V' ,'U',nmat,                   &
      a   ,1,1,desca,                  &
      evs ,                            &
      evec,1,1,desca,                  &
      work,lwork,info)
!-----------------------------------------------------------------
 if(info /= 0)then
  write(errmes,'(A,I4)')               &
   'Error: sca_evs: pdsyev failure, info = ',info
  call sca_abort(errmes)
 end if
end subroutine     sca_evs
```

The test calculation program is contained in the file `sca_05_evs.f90` and is listed below. It is designed to demonstrate the complete workflow for solving a parallel dense symmetric eigenproblem:

1. **Initialization**: The program begins by defining a random symmetric array, `amat_glob(nmat,nmat)`, exclusively on the root process (where `iamroot` is true, as set in `sca_init`).

2. **Distribution**: This global array is then broadcast to all processes using `MPI_Bcast`. Subsequently, the data is distributed into the local arrays, `amat(mloc,nloc)`, using the internal subroutine `sca_glob2loc`.

3. **Computation**: The eigenvalues (`evs(nmat)`) and eigenvectors (`evec(mloc,nloc)`)

are calculated and stored using the internal subroutine `sca_evs`.

4. **Reassembly**: To verify the results, the local eigenvector arrays are gathered and the complete global eigenvector matrix, `evec_glob(nmat,nmat)`, is constructed using a call to `sca_loc2glob`.

5. **Validation**: Finally, the program calculates and prints the residual errors given by Eqs.(6.24)–(6.26).

The complete code is listed below:

```
program         scalapack_intro
 use, intrinsic           :: iso_fortran_env
 use mpi
 implicit none
 integer , parameter      :: dp = real64
!MPI:-------------------------------------------------------------
 integer                  :: nprocs, myrank, ierr
 integer                  :: comm_grid
!BLACS:-----------------------------------------------------------
 integer                  :: context
 integer                  ::  nprow,  npcol
 integer                  :: myprow, mypcol, myid, myproc
 integer                  :: info
 logical                  :: iamingrid, iamroot
!-----------------------------------------------------------------
 integer, parameter       :: DLEN = 9
 integer, dimension(DLEN) :: desca
 integer, external        :: numroc
 integer                  :: nmat,MB,NB,mloc,nloc
 integer                  :: imat,jmat
 real(dp),allocatable     :: amat     (:,:),evec     (:,:)
 real(dp),allocatable     :: amat_glob(:,:),evec_glob(:,:)
 real(dp),allocatable     :: evs(:)
!-----------------------------------------------------------------
 nmat  = 9
 nprow = 2    ; npcol  = 3
 MB    = 2    ; NB     = 2
!-----------------------------------------------------------------
 call sca_init
!-----------------------------------------------------------------
!Construct local arrays:
 allocate(amat     (mloc,nloc),evec     (mloc,nloc))
 allocate(amat_glob(nmat,nmat),evec_glob(nmat,nmat),evs(nmat))
!Construct global array: must be symmetric:
 if(iamroot)then
  call random_number(amat_glob) ;
  amat_glob = 0.5_dp * (amat_glob + transpose(amat_glob))
 end if
 call mpi_bcast(amat_glob,nmat*nmat,MPI_REAL8,0,comm_grid,ierr)
!-----------------------------------------------------------------
 call sca_glob2loc(amat_glob,amat)
```

```
!-----------------------------------------------------------------------
 call sca_evs(amat,evec,evs)
!-----------------------------------------------------------------------
 call sca_loc2glob(evec,evec_glob)
!-----------------------------------------------------------------------
 print   '(A,I0.2,2I3,A     ,1000G10.2)','01 evs : ',          &
  myrank,myprow,mypcol,' :: ',evs
!-----------------------------------------------------------------------
 print   '(A,I0.2,2I3,A     ,1000G10.2)','02 evec: ',          &
  myrank,myprow,mypcol,' :: ',                                 &
  (sum(abs(                                                    &
            matmul(amat_glob,evec_glob(:,imat))                &
                  -evs(imat)*evec_glob(:,imat)                 &
  ))/nmat,imat=1,nmat)
!-----------------------------------------------------------------------
 print   '(A,I0.2,2I3,A     ,1000G10.2)','03 norm: ',          &
  myrank,myprow,mypcol,' :: ',                                 &
  (abs(                                                        &
  dot_product(evec_glob(:,imat),evec_glob(:,imat)) - 1.0_dp    &
  ),imat=1,nmat)
!-----------------------------------------------------------------------
 if(iamroot)then
  do imat = 1, nmat
   print '(A,I0.2,2I3,A,I0.2,1000G10.2)','04 orth: ',          &
    myrank,myprow,mypcol,' :: ', imat,&
    (abs(dot_product(evec_glob(:,imat),evec_glob(:,jmat)))     &
    ,jmat=1,nmat)
  end do
 end if
!-----------------------------------------------------------------------
 call blacs_exit(0)
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
contains
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
 subroutine          sca_evs     (ain, evec, evs)
 subroutine          sca_glob2loc(a_glob,a)
 subroutine          sca_loc2glob(a,a_glob)
 subroutine          sca_init
 subroutine          sca_abort(errmes)
end program     scalapack_intro
```

To check the results, compile and run using the commands:

```
mpifort sca_05_evs.f90 -lscalapack-openmpi -o m
mpirun -n 6 ./m | sort
```

## 6.4 Working Together: Coarrays, MPI, ScaLAPACK

The inherent complexity of high-performance parallel programming often necessitates leveraging the specialized strengths of multiple parallel programming environments simultaneously. While ScaLAPACK offers highly optimized routines for dense linear algebra essential for tasks like matrix multiplication and diagonalization, and Fortran Coarrays (CAF) provide elegant, language-integrated syntax for rapid communication, MPI remains the universal and foundational standard for low-level communication primitives.

The basic principle that enables these seemingly disparate paradigms to coexist and interoperate seamlessly is layering and unification. Both ScaLAPACK (through its underlying BLACS layer) and most modern CAF implementations (such as OpenCoarrays) are built upon the Message Passing Interface (MPI) standard. By ensuring that the parallel environment is correctly initialized by the appropriate foundational layer, all three interfaces can operate within the same unified context.

This mixed-environment approach offers several architectural advantages:

- **Performance**: Expensive, regular operations, such as matrix multiplication and diagonalization, can be delegated to the highly optimized ScaLAPACK library, maximizing parallel speed and ensuring efficient load balancing.

- **Productivity**: Coarrays, due to their ease of use and direct language integration, can be employed for lightweight tasks, debugging, and simple, intuitive array indexing. Simple collective operations are easily handled using the elegant Coarray syntax, which simplifies code development and improves modularity compared to verbose, explicit MPI calls.

- **Stability**: The core process management, environment setup, and foundational communication context rely on the stability and universal standardization provided by the MPI layer.

### 6.4.1 Coarrays & ScaLAPACK Integration

This section introduces a simple program that integrates Fortran Coarray programming with ScaLAPACK routines. This hybrid approach builds upon the program for computing eigenvalues and eigenvectors of a real symmetric matrix of Section 6.3.6.

To ensure interoperability, the array distribution is configured such that:

- Coarrays must have the same type and shape on every image (process).
- Arrays are distributed using a block-wise scheme with `MB` = `NB` = $N_{\text{loc}}$.
- The processes form an $N_{\text{p}} \times N_{\text{p}}$ grid, similar to Figure 6.2 or Figure 6.23.
- The total number of processes is $N_{\text{procs}} = N_{\text{p}} \times N_{\text{p}}$.
- This setup ensures that the local dimensions are identical across all processes (`mloc` = `nloc` = $N_{\text{loc}}$).

The demonstration code `sca_06_CAF.f90`, uses CAF for distributed array indexing and collective communication, and it delegates the core computational task (eigenvalue/eigenvector calculation) to the highly optimized ScaLAPACK routine `PDSYEV`.

To correctly compile and link a hybrid program mixing CAF and ScaLAPACK (which implicitly relies on MPI), specific compiler flags must be used to manage both the Coarray runtime and the ScaLAPACK library:

```
mpifort -fcoarray=lib         sca_06_CAF.f90                    \
        -lscalapack-openmpi                                     \
        -L/usr/lib/x86_64-linux-gnu/open-coarrays/openmpi/lib   \
        -lcaf_mpi
```

1. The `mpifort` compiler wrapper automatically includes the necessary MPI header files and linking flags.
2. The `-fcoarray=lib` option and the linking option `-lcaf_mpi` instruct the compiler to use the external CAF library, which handles the MPI environment initialization implicitly at runtime.
3. The option `-lscalapack-openmpi` links the specific ScaLAPACK library required for the computation.

The path specified by `-L` must be updated to the location of the `libcaf_mpi.so` file on your system.

The hybrid `sca_06_CAF.f90` program, which integrates Coarrays with ScaLAPACK, differs from the pure MPI version (`sca_05_evs.f90`) in four key areas:

- **MPI Initialization**: The CAF library handles the parallel environment setup implicitly at runtime, meaning `MPI_Init` *must not be called*.

- **MPI Finalizing**: Similarly, the CAF library manages the shutdown at the program's conclusion, so neither `BLACS_Exit` nor `MPI_Finalize` should be called for a normal program exit.
- **Data Structures**: The program requires specific Coarray declarations and allocation.
- **Communication**: It incorporates the use of Coarray collective subroutines, such as `co_broadcast`.

The streamlined MPI and BLACS initialization sequence, found in the `sca_init` subroutine, reflects this integration by omitting the explicit MPI initialization call:

```
!call mpi_init       (                          ierr) !commented!
 call mpi_comm_size (MPI_COMM_WORLD,nprocs,ierr)
 call mpi_comm_rank (MPI_COMM_WORLD,myrank,ierr)

 call blacs_pinfo    (myid,nprocs )
 call blacs_get      (-1,0,context)
 call blacs_gridinit(context,'Col-major', nprow, npcol)
 call blacs_gridinfo(context,nprow,npcol,myprow,mypcol)

 mloc = numroc(nmat,MB,myprow,0,nprow)
 nloc = numroc(nmat,NB,mypcol,0,npcol)

 call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)
```

In all other respects, the calls to query MPI ranks and perform BLACS setup (getting the context, initializing the grid, querying grid information) remain the same. The sequence concludes with the computation of local dimensions (`mloc`,`nloc`) and the definition of the array descriptor (`desca`).

The following variables are declared as Coarrays–objects accessible across multiple images–primarily for demonstration purposes in this hybrid example:

```
integer                  :: mloc[*], nloc[*]
integer ,allocatable     :: myrank   [:,:]
real(dp),allocatable     :: evec(:,:)[:,:], evs(:) [:,:]
```

The initialization of the Coarray environment and the determination of the necessary ScaLAPACK parameters are performed using the following steps:

1. **Determine Grid and Local Sizes**: The global matrix size (`nmat`) is set. The number of images (`nimg`) is queried, and the dimensions of the square process grid (`nmat_p`) and the local block size (`nmat_loc`) are calculated:

```
nmat     = 4

nimg     = num_images() ;
nmat_p   = int(sqrt(nimg+1.0e-6_dp));
nmat_loc = nmat/nmat_p
```

2. **Map to ScaLAPACK Parameters**: The calculated grid and block sizes are explicitly assigned to their corresponding ScaLAPACK parameters:

```
nprow = nmat_p   ; npcol  = nmat_p
MB    = nmat_loc ; NB     = nmat_loc
```

3. **Initialize Environments**: With these essential parameters defined, the `sca_init` subroutine is called to construct the underlying BLACS and MPI environment:

```
call sca_init
```

The coarrays required for the distributed calculation are allocated using the calculated grid dimensions (`nmat_p` and `nmat_loc`).

```
allocate(myrank[nmat_p,*])
allocate(evec(nmat_loc,nmat_loc)[nmat_p,*])
allocate(evs (nmat)             [nmat_p,*])
!Then we can define the coarray grid environment variables:
myimg  = this_image()          ! = myrank + 1
myi    = this_image(evs,1)     ! = myprow + 1
myj    = this_image(evs,2)     ! = mypcol + 1
```

The intrinsic function `this_image` is used to determine the 1-based coordinates (`myi`, `myj`) of the current image within the Coarray grid. The relationship shown between the Coarray and BLACS coordinates (`myi` = `myprow`+1, etc.) holds true only because `BLACS_Gridinit` was called using the `'Col-major'` option, ensuring consistency in the mapping order.

The local array dimension values, calculated by `NUMROC` on each process, are retrieved into standard arrays for diagnostics using the coindexing feature:

```
allocate(mlocs(nimg),nlocs(nimg))
do     img  = 1,  nimg
 mlocs(img) = mloc[img]
 nlocs(img) = nloc[img]
end do
```

The construction and distribution of the random real symmetric matrix **A** across all images relies on the Coarray collective subroutine `co_broadcast`:

```
if(iamroot)then
 call random_number(amat_glob) ;
 amat_glob = 0.5_dp * (amat_glob + transpose(amat_glob))
end if
call co_broadcast(amat_glob,1)
```

The `amat_glob` is randomized and symmetrized exclusively on the root image (`iamroot`), and then `co_broadcast` efficiently transfers an identical copy of the matrix to all other images/processes. The use of `co_broadcast` in this context is equivalent to calling `MPI_Bcast`.

The global output, such as the calculated eigenvalues, can be managed from the root process by accessing the coindexed data from all other images:

```
if(iamroot)then
 do ip = 1, nmat_p ; do jp = 1, nmat_p
  print *,myrank[ip,jp], evs(:)[ip,jp]
 end do            ; end do
end if
```

To explore the complete code, compile and run the `sca_06_CAF.f90` file using the following commands:

```
mpifort -fcoarray=lib sca_06_CAF.f90 -lscalapack-openmpi  \
 -L/usr/lib/x86_64-linux-gnu/open-coarrays/openmpi/lib    \
 -lcaf_mpi -o m

mpirun -n 4 ./m | sort
```

Enjoy!

### 6.4.2 Independent "Simulations" and ScaLAPACK Contexts

In Monte Carlo simulations, one often seeks to increase statistical accuracy by running $N_{\text{sim}}$ multiple, independent replicas of a program simultaneously. Within each of these simulations, it may be necessary to perform parallel dense linear algebra computations, using ScaLAPACK's optimizations.

BLACS provides the mechanism to define entirely separate computational domains, effectively isolating each simulation to its own group of processes. This strategy leverages the hierarchical management of parallelism afforded by MPI and BLACS to logically partition the global proces-

sor pool, thereby managing the concurrent execution of $N_{\text{sim}}$ independent jobs, as illustrated in Figure 6.26.

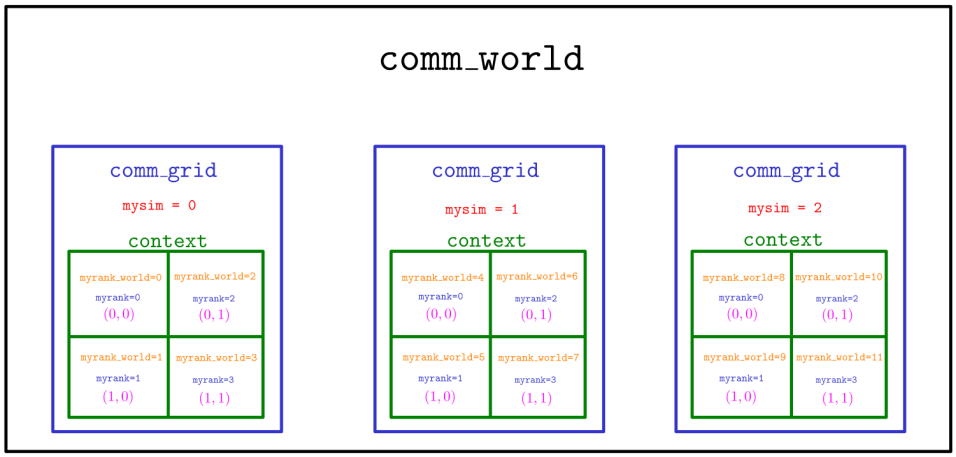



Figure 6.26: Schematic diagram illustrating the partitioning of the global `comm_world` communicator into $N_{\text{sim}} \equiv$ `nsim` $= 3$ independent sub-communicators (`comm_grid`). This division facilitates concurrent execution of $N_{\text{sim}}$ simulations. Each `comm_grid` defines a local BLACS context for one simulation, running on four processes. Within each simulation's grid, processes are reassigned local ranks (`myrank` $= (0, 1, 2, 3)$) and unique BLACS grid coordinates (e.g., $(0, 0)$ to $(1, 1)$). Note how the original, non-contiguous global ranks (`myrank_world`) are logically grouped into the new, ordered, and independent grids.

This partitioning is achieved through two critical steps:

First, by partitioning the MPI Communicator. The global communication pool, `comm_world` $\equiv$ `MPI_COMM_WORLD`, is logically partitioned into $N_{\text{sim}}$ maller, mutually exclusive communicators, each designated `comm_grid`. Each `comm_grid` defines a self-contained environment, representing the parallel resource pool for a single simulation.

This partitioning is executed using the `MPI_Comm_split` routine, which uses the computed simulation ID (`mysim`) as the coloring parameter:

```
call mpi_comm_split(comm_world,mysim,myrank_world,comm_grid, &
                    ierr)
```

Here, processes sharing the same `mysim` $=$ `myrank_world/nprocs` value are grouped into a new `comm_grid`. In this setup, `myrank_world` is the rank of the process within `comm_world`, `nsim` $\equiv N_{\text{sim}}$ is the number of simulations, and `nprocs` $=$ `nprocs_world/nsim` is the number of processes allocated to each `comm_grid`.

Second, within each newly created `comm_grid`, a unique BLACS context (`context`) must be established. This `context` ensures that all subsequent ScaLAPACK operations are logically confined only to the processes within that specific `comm_grid`.

Running independent contexts requires explicitly mapping the global ranks (`myrank_world`) of the processes belonging to the `comm_grid` to the desired BLACS grid coordinates via the `BLACS_Gridmap` routine and a pre-computed user map array (`usermap`):

```
allocate(comm_grid_procs(0:nprocs-1))

call mpi_allgather(                                          &
        myrank_world    ,1,MPI_INTEGER,                      &
        comm_grid_procs,1,MPI_INTEGER,                       &
        comm_grid,ierr)

allocate(usermap(0:nprow-1,0:npcol-1))

iproc    = 0
do jpcol = 0, npcol - 1 ; do iprow = 0, nprow - 1
 usermap(iprow,jpcol) = comm_grid_procs(iproc)
 iproc                = iproc + 1
end do                  ; end do

call blacs_get     (0,0,context)
call blacs_gridmap (context,usermap    , nprow, nprow,npcol)
call blacs_gridinfo(context,nprow,npcol,myprow,mypcol)
```

The array `comm_grid_procs(0:nprocs-1)` gathers the `myrank_world` values of all processes active within the `comm_grid` (using `MPI_Allgather` with the `comm_grid` communicator). These ranks are stored in a column-major fashion into the 2D usermap array, creating the mapping (`iprow`, `jpcol`) $\rightarrow$ `myrank_world`. This explicit mapping allows `BLACS_Gridmap` to construct the correct process grid within the newly created context.

Using these conventions, ScaLAPACK routines can be executed efficiently on their dedicated processor resources. Variables such as `comm_grid`, `context`, `nprocs`, `nprow`, `npcol`, `myprow` and `mypcol` are used uniformly across the main program and all internal subroutines.

For instance, the determination of local array dimensions and the descriptor setup relies on these context-specific variables:

```
mloc = numroc(nmat,MB,myprow,0,nprow)
nloc = numroc(nmat,NB,mypcol,0,npcol)
call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)
```

The file `sca_07_MPI.f90` provides a working test program of these ideas. The custom `sca_init` subroutine, listed below, has been modified

to handle the complex, multi-context initialization discussed previously:

```
 subroutine         sca_init
  integer , allocatable:: usermap(:,:), comm_grid_procs(:)
  integer              :: iprow,jpcol,iproc
  character(500)       :: errmes
!-----------------------------------------------------------------
!Init mpi: This communicator contains all simulations
  call mpi_init     (                              ierr)
  call mpi_comm_size(MPI_COMM_WORLD,nprocs_world,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,myrank_world,ierr)
  comm_world =       MPI_COMM_WORLD
!-----------------------------------------------------------------
  if( nsim * npcol * nprow /= nprocs_world )                    &
   call sca_abort("Error: nsim * npcol * nprow /= nprocs")
!-----------------------------------------------------------------
!Create the communicator of each simulation, comm_grid.
!Each simulation will hold its own BLACS grid
  nprocs = nprocs_world/nsim
  mysim  = myrank_world/nprocs
!Create comm_grid using mysim as "color".
!All processes with same mysim are included:
  call mpi_comm_split(comm_world,mysim,myrank_world,            &
                      comm_grid ,ierr)
  call mpi_comm_size (comm_grid ,nprocs,ierr)
  call mpi_comm_rank (comm_grid ,myrank,ierr)
!The comm_world ranks of processes in comm_grid:
  allocate(comm_grid_procs(0:nprocs-1))
  call mpi_allgather (                                          &
           myrank_world    ,1,MPI_INTEGER,                      &
           comm_grid_procs,1,MPI_INTEGER,                       &
           comm_grid       ,ierr)
!-----------------------------------------------------------------
!Init BLACS: We create a context using
!the processes only in comm_grid:
  allocate(usermap(0:nprow-1,0:npcol-1))
  iproc    = 0
!we define a column major grid (use row-major if you prefer!)
  do jpcol = 0, npcol - 1 ; do iprow = 0, nprow - 1
   usermap(iprow,jpcol) = comm_grid_procs(iproc)
   iproc                = iproc + 1
  end do                  ; end do
  call blacs_get     (0,0,context)
!Use   blacs_gridmap, not blacs_gridinit:
  call blacs_gridmap (context,usermap    , nprow, nprow,npcol)
  call blacs_gridinfo(context,nprow,npcol,myprow,mypcol)
!-----------------------------------------------------------------
!Local array:
!Size of local array is mloc x nloc
  mloc = numroc(nmat,MB,myprow,0,nprow)
  nloc = numroc(nmat,NB,mypcol,0,npcol)
!Universal array descriptor:
  call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)
```

```
!-----------------------------------------------------------------
 end subroutine      sca_init
```

In the main program, the global matrix $\mathbf{A}$ is initialized using a specific formula: $\mathbf{A}_{ab} = 100(\texttt{mysim}+1) + a + b/100$. This assignment ensures that the simulation ID (`mysim`) is visibly encoded into the matrix elements, allowing for easy visualization of the distinct matrix associated with each independent simulation.

The setup parameters are defined as:

```
nsim  = 3
nmat  = 4
nprow = 2     ; npcol  = 2
MB    = 2     ; NB     = 2
!......
call sca_init
!......
do imat = 1, nmat ; do jmat = 1, nmat
 amat_glob(imat,jmat) = (mysim+1)*100 + imat + jmat / 100.0_dp
end do            ; end do
call sca_glob2loc(amat_glob,amat)
!......
```

This configuration corresponds to $N_{\text{sim}} = 3$ parallel simulations, each handling an $N = 4$ matrix problem on an $N_{\text{p}} \times N_{\text{p}} = 2 \times 2$ grid. This requires a total of $N_{\text{procs\_world}} = 12$ processes (with $N_{\text{procs}} = 4$ processes per simulation), as schematically depicted in Figure 6.26. The global matrix `amat_glob` is then properly partitioned and distributed to the local matrices `amat(mloc,nloc)` within each dedicated context using the `sca_glob2loc` subroutine.

If you wish to compute the eigenvalues of $N_{\text{sim}}$ different matrices, you can do so as we did in the program `sca_05_evs.f90`

```
allocate(evec(mloc,nloc),evec_glob(nmat,nmat),evs(nmat))
!Define a random symmetric matrix, different for each comm_grid:
!iamroot is the (0,0) process of context, which is a different
!process of comm_world for each simulation:
if(iamroot)then
 call random_number   (amat_glob)
 amat_glob = 0.5_dp * (amat_glob + transpose(amat_glob))
end if
!Broadcast the global matrix only in      comm_grid:
call mpi_bcast(amat_glob,nmat*nmat,MPI_REAL8,0,comm_grid,ierr)
!Distribute the global matrix to the local amat:
call sca_glob2loc(amat_glob,amat)
!Compute the eigenvalues and eigenvectors:
call sca_evs     (amat,evec,evs)
```

For $N_{\text{sim}} = 3$ and $N_{\text{procs}} = 4$, the program must be run on $N_{\text{procs\_world}} = 12$ processes:

```
mpifort sca_07_MPI.f90 -lscalapack-openmpi -o m
n=12
mpirun -n $n --host $(hostname):$n ./m | sort
```

## 6.5 A Guide to Further Reading

For a pedagogical introduction to Fortran Coarrays and Teams, see [2]. For a more in-depth exposition see [3].

For an introduction to MPI using Fortran, see [4]

ScaLAPACK is taught by going through the online documentation system[16, 17, 18], but you can also browse the online user guide [6].

## 6.6 Problems

6.1 Write code to compute the product of two complex matrices using the PBLAS subroutine `PZGEMM`. Your program should verify the accuracy of the calculation by constructing the global arrays, and computing their matrix product using `MATMUL`.

6.2 Write code to compute the eigenvalues and eigenvectors of a Hermitian matrix, using the ScaLAPACK subroutine `PZHEEV`. Your program should test the accuracy of the calculation using Eqs.(6.24)–(6.26)

## 6.A Appendix: Array Descriptors and Index Mapping in ScaLAPACK

The Array Descriptor is the standardized mechanism ScaLAPACK procedures use to manage how a global matrix is partitioned and distributed across a 2D process grid. It provides the necessary metadata for PBLAS and ScaLAPACK routines to understand the geometry of the global matrix, the size of its distributed blocks, and its placement within the BLACS context.

The Array Descriptor is a fixed-size integer array, `DESCA(DLEN)`, where `DLEN=9`.

The array descriptor, whose structure is detailed in Table 6.3, must be initialized for each distributed array using the ScaLAPACK routine `DESCINIT`.

The general Fortran calling sequence for `DESCINIT` is:

| Index | Value | Description |
|---|---|---|
| DTYPE_ | 1 | Descriptor Type (1 for dense matrices) |
| CTXT_ | 2 | BLACS context handle |
| M_ | 3 | Global number of rows of the matrix, M |
| N_ | 4 | Global number of columns of the matrix, N |
| MB_ | 5 | Row blocking factor, MB |
| NB_ | 6 | Column blocking factor, NB |
| RSRC_ | 7 | BLACS row index of the process that holds the first row |
| CSRC_ | 8 | BLACS column index of the process that holds the first column |
| LLD_ | 9 | Leading dimension of the local array |

Table 6.3: The entries of the integer array DESCA(DLEN), where DLEN=9. Typically in this Chapter, $N$ = nmat = M = N, CTXT_ = context, RSRC_ = CSRC_ = 0, and LLD_ = mloc.

```
call descinit(desca,M,N,MB,NB,RSRC,CSRC,CTXT,LLD,info)
```

This call sets the corresponding indices of the integer array descriptor, desca, with the provided parameters (Table 6.3).

For a square matrix $\mathbf{A}$ of size $N \times N$ (where M = N = nmat) being distributed using MB $\times$ NB blocks within the BLACS context context, the array descriptor is set by the following call:

```
call descinit(desca,nmat,nmat,MB,NB,0,0,context,mloc,info)
```

In this common case, the call assumes:

- The dimensions are M = nmat and N = nmat
- The Leading Dimension (LLD) is set to the local row size, mloc.
- The starting block containing the matrix element $\mathbf{A}_{11}$ is located in the process with grid coordinates (RSRC, CSRC) = $(0, 0)$.

The local row dimension (mloc) and column dimension (nloc) are calculated as follows:

```
mloc = numroc(nmat,MB,myprow,0,nprow)
nloc = numroc(nmat,NB,mypcol,0,npcol)
```

The starting problem in distributed linear algebra is mapping between three types of indices: the global index of a matrix element $\mathbf{A}_{ab}$, the local

array index where that element is stored, and the process that owns the element. The BLACS/ScaLAPACK environment provides a comprehensive set of tools to perform these essential mappings.

The function `INDXG2P` (INDeX–Global–to–Process) is used to determine the grid coordinates (`iprow`, `jpcol`) of the process that owns a specific matrix element $\mathbf{A}_{ab}$, given its global indices `imat_glob` $\equiv a$ and `jmat_glob` $\equiv b$.

The calling sequence for retrieving the owner process coordinates is:

```
iprow = indxg2p(imat_glob,MB,0,desca(RSRC_),nprow)
jpcol = indxg2p(jmat_glob,NB,0,desca(CSRC_),npcol)
```

To determine the local indices (`imat`, `jmat`) where a matrix element $\mathbf{A}_{ab}$ is stored within the local array `amat(imat,jmat)`, given the global indices `imat_glob` $\equiv a$ and `jmat_glob` $\equiv b$, you use the function `INDXG2L` (INDeX–Global–to–Local):

```
imat = indxg2l(imat_glob,MB,0,desca(RSRC_),nprow)
jmat = indxg2l(jmat_glob,NB,0,desca(CSRC_),npcol)
```

The inverse operation, computing the global indices (`imat_glob`, `jmat_glob`) from the local array indices (`imat`, `jmat`), is achieved using the function `INDXL2G` (INDeX–Local–to–Global):

```
imat_glob = indxl2g(imat,MB,myprow,desca(RSRC_),nprow)
jmat_glob = indxl2g(jmat,NB,mypcol,desca(CSRC_),npcol)
```

The BLACS process number, `myproc`, retrieved by `BLACS_PNUM`,

```
myproc = blacs_pnum(context,myprow,mypcol)
```

is not necessarily the same as the process's MPI rank (`myrank`) within the MPI communicator of the `context`. This equality only holds if the grid was constructed using the `'Row–major'` option.

However, the `myproc` value can be used safely to construct the mapping from grid coordinates (`myprow`, `mypcol`) to the true MPI rank (`myrank`):

```
myproc   = blacs_pnum(context,myprow,mypcol)

allocate(myprocs(0:nprocs-1))

call mpi_allgather(                                    &
  myproc ,1,MPI_INTEGER,                               &
  myprocs,1,MPI_INTEGER,                               &
  comm_grid,ierr)
```

```
allocate(myranks(0:nprocs-1))

do irank = 0, nprocs - 1
 myranks(myprocs(irank)) = irank
end do

allocate(prowpcol2rank(0:nprow-1,0:npcol-1))

do  iprow = 0, nprow -1
 do jpcol = 0 , npcol - 1
  iproc   = blacs_pnum(context,iprow,jpcol)
  irank   = myranks(iproc)
  prowpcol2rank(iprow,jpcol) = irank
 end do
end do
```

The resulting array `prowpcol2rank` can then be used to map any grid coordinate (`myprow`, `mypcol`) to its corresponding MPI rank `myrank`. The inverse map, `myrank` $\rightarrow$ (`myprow`, `mypcol`), is left as an exercise for the reader.

Finally, we remind the reader of the utility functions `PDELSET` and `PDELGET`, which were discussed in Section 6.3 (pages 274 and 292). These routines offer a simple, direct interface for setting and retrieving local and global matrix elements, respectively. However, users should be cautious: while convenient, relying heavily on these point-to-point operations for iterative tasks can incur significant communication overhead.

An example program using the various functions introduced throughout this Appendix can be found in the file `sca_08_DescIndx.f90`.

## References

[1] *OpenCoarrays*. URL: http://www.opencoarrays.org/.

[2] Milan Curcic. *Modern Fortran: Building efficient parallel applications*. Manning Publications, 2020.

[3] Michael Metcalf, John Reid, Malcolm Cohen, and Reinhold Bader. *Modern Fortran Explained: Incorporating Fortran 2023*. Oxford University Press, 2024.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Pasing Interface - Third Edition*. The MIT Press, 2014. ISBN: 9780262527392. URL: https://mitpress.mit.edu/9780262527392/using-mpi/.

[5] *ScaLAPACK - Scalable Linear Algebra PACKage*. URL: https://www.netlib.org/scalapack/.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), 1999. ISBN: 978-0-898719-64-2. URL: https://www.netlib.org/scalapack/slug/.

[7] Mark Gates, Ali Charara, Jakub Kurzak, Asim YarKhan, Mohammed Al Farhan, Dalal Sukkari, and Jack Dongarra. *SLATE Users' Guide*. SLATE Working Notes 10, ICL-UT-19-01. Innovative Computing Laboratory, University of Tennessee, 2020-07 2020. URL: https://icl.utk.edu/publications/swan-010.

[8] *Software for Linear Algebra Targeting Exascale (SLATE)*. URL: https://icl.utk.edu/slate/.

[9] *SLATE Git Repository*. URL: https://github.com/icl-utk-edu/slate.

[10] *cuBLAS: Basic Linear Algebra on NVIDIA GPUs*. URL: https://developer.nvidia.com/cublas.

[11] *cuSOLVER: Basic Linear Algebra on NVIDIA GPUs*. URL: https://developer.nvidia.com/cusolver.

[12] *ELPA: Eigenvalue Solvers for Petaflop Applications*. URL: https://elpa.mpcdf.mpg.de/.

[13] *MAGMA: Matrix Algebra on GPU and Multi-core Architectures*. URL: https://elpa.mpcdf.mpg.de/.

[14] *DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures*. URL: https://icl.utk.edu/dplasma/.

[15] *CHAMELEON: A Task-based Programming Environment for Developing Reactive HPC Applications*. URL: https://github.com/chameleon-hpc/chameleon.

[16] *BLACS Online Reference Guide*. URL: https://www.netlib.org/blacs/BLACS/QRef.html.

[17] *PBLAS Online Reference Guide*. URL: https://www.netlib.org/scalapack/pblas_qref.html.

[18] *ScaLAPACK Online Documentation*. URL: https://www.netlib.org/scalapack/explore-html/.