F42: Fortran coding guides

Asis Hallab, "el Bobito"

August 8, 2025

Contents

1 Use latest gfortran compiler version

To ensure optimal performance and compatibility, always use the latest stable version of the gfortran compiler (15+).
We recommend adding a PPA (Personal Package Archive) to your system to easily install the latest version.
If you are using Ubuntu, you can do this by running:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gfortran-<version> # e.g., gfortran-15
```

If gfortran-15 is not available, you can also get the latest version with Docker:

- Install and setup Docker as explained for your operating system in the Docker documentation.
- Use our Dockerfile gfortran.docker:

  ```
  docker build -t arch-gfortran -f gfortran.docker .
  ```

- Then build the project with:

  ```
  docker run -it -v ‘pwd‘:/opt arch-gfortran ’/opt/build.sh’
  ```

2 Use of double precision

Declare in Fortran internal functions and subroutines `real` variables and constants always using the ISO double precision standard (see below for an example). Do make sure that if you assign values at compile time, so constants, you *must use* the `_real64` suffix to guarantee precision (see example below).

```
program double_precision_example
  use, intrinsic :: iso_fortran_env, only: real64  ! Use explicit ISO-standard double precision
  implicit none

  real(real64) :: x, y, result

  ! Assign values
  x = 3.141592653589793_real64
  y = 2.718281828459045_real64

  ! Perform computation
  result = x * y

  ! Print with full precision
  print "(A, F24.16)", "Result = ", result

end program double_precision_example
```

3 Using `test suite framework` for Testing

This framework provides a robust and scalable system for organizing and executing unit tests in Fortran. It allows running individual tests, complete test suites, or all project tests with simple and clear syntax.
Please check this readme file for details.

3.1 Do not test multiple times

The correctness of the calculation, the correctness of the implemented algorithm is *only* tested in Fortran. The API language tests for R, Python, and, if we ever included them, in C should only test that the function can be correctly called and the return values are accessible and of the expected type.
In other words, do not repeat tests, if avoidable.
If your test is created by an AI from the original Fortran code, and you thus do not lose work time on writing the test again in another programming language, please do feel free to do so.

4 Obligatory use of `intent` and `pure`

- Always use `intent` for function or subroutine arguments. Be aware of the kind of data being passed.
  - **intent(in)**: Use when the pointer's contents are only read.
  - **intent(out)**: Use when the contents will be entirely overwritten.
  - **intent(inout)**: Use when the subroutine both reads and writes to the data, and the initial values are important.
- Always use `pure` where possible.

4.1 Optional Arguments in Fortran

4.1.1 Overview

Fortran (ISO standard, since Fortran 90) supports `optional` procedure arguments, which allow the caller to omit certain arguments when invoking a subroutine or function. Their presence must be checked explicitly using the intrinsic function `PRESENT`. Fortran does not support default values in procedure headers. Developers must assign defaults manually.

4.1.2 Syntax and Usage

To declare an optional argument:

```
subroutine example(a, b, c)
  integer, intent(in) :: a
  real, intent(in), optional :: b, c
```

To check whether an optional argument was passed:

```
  if (present(b)) then
    print *, 'B is provided: ', b
  else
    print *, 'B is not provided.'
  end if
```

4.1.3 Example

```
program optional_demo
  call greet("Josef")
  call greet("Josef", "Mexico")
contains
  subroutine greet(name, location)
    character(len=*), intent(in) :: name
    character(len=*), intent(in), optional :: location

    print *, "Hello, ", name
    if (present(location)) then
      print *, "How is the weather in ", location, "?"
    end if
  end subroutine greet
end program optional_demo
```

4.1.4 Default Value Handling

Fortran does not support assigning default values directly in the argument list. Instead, assign them manually after a PRESENT check:

```
subroutine greet(name, title)
  character(len=*), intent(in) :: name
  character(len=*), intent(in), optional :: title
  character(len=20) :: effective_title

  if (present(title)) then
    effective_title = title
  else
    effective_title = "Mr./Ms."
  end if

  print *, "Hello, ", trim(effective_title), " ", trim(name)
end subroutine greet
```

4.1.5 Workaround: Interface Overloading

To emulate default arguments, use multiple procedures with an interface:

```
module greeter_mod
contains
  subroutine greet1(name)
    character(len=*), intent(in) :: name
    call greet2(name, "Mr./Ms.")
  end subroutine

  subroutine greet2(name, title)
    character(len=*), intent(in) :: name
    character(len=*), intent(in) :: title
    print *, "Hello, ", trim(title), " ", trim(name)
  end subroutine
end module

program main
  use greeter_mod
  interface greet
    module procedure greet1, greet2
  end interface

  call greet("Josef")
  call greet("Josef", "Dr.")
end program
```

4.1.6 Best Practices

- Always check optional arguments with PRESENT before accessing them.

- Avoid relying on optional arguments for critical logic without default assignment.
- Use interface overloading to simulate default values if needed for cleaner calling code.
- Do not use optional arguments in `elemental` procedures (not supported).
- Document the behavior and defaults of optional arguments clearly in the procedure's interface documentation.

5 Documenting Fortran Code with FORD

FORD (Fortran Documentation Generator) is a modern tool for generating web and PDF documentation from structured comments in Fortran code. It creates navigable documentation with cross-references, module and subroutine descriptions, argument lists, and usage examples.

5.1 Basic principles

- Use special comments (`!>` and `!|`) to annotate modules, subroutines, functions, and variables.
- Comments must be placed immediately before the entity they document.
- It is recommended to document all arguments, the general purpose, and any relevant side effects.
- Use `!>` for general descriptions of modules, subroutines, and functions.
- Use `!|` to document each argument.
- If relevant, add usage examples and warnings with `!!`.
- Keep documentation up to date when the interface changes.
- Generate documentation with FORD regularly and review the HTML output.

5.2 Minimal example

```
!> Module for calculating normalized tissue (axis) versatility.
module avmod
  use, intrinsic :: iso_fortran_env, only: real64
  implicit none
contains

  !> Computes normalized tissue versatility for selected expression vectors.
  !! The metric is based on the angle between each gene expression vector and the space diagonal.
  pure subroutine compute_tissue_versatility(n_axes, n_vectors, expression_vectors, &
                                             exp_vecs_selection_index, axes_selection, &
                                             tissue_versatilities, tissue_angles_deg)
    !| Number of axes (tissues/dimensions)
    integer, intent(in) :: n_axes
    !| Number of expression vectors (genes)
    integer, intent(in) :: n_vectors
    !| 2D array (n_axes, n_vectors), each column is a gene expression vector
    real(real64), intent(in) :: expression_vectors(n_axes, n_vectors)
    !| Logical array (n_vectors), .TRUE. for vectors to process
    logical, intent(in) :: exp_vecs_selection_index(n_vectors)
    !| Logical array (n_axes), .TRUE. for axes to include in calculation
    logical, intent(in) :: axes_selection(n_axes)
    !| Output, real array, length = count(exp_vecs_selection_index), stores the calculated
```

```
       tissue versatilities
    real(real64), intent(out) :: tissue_versatilities(count(exp_vecs_selection_index))
    !| Output, real array, length = count(exp_vecs_selection_index), stores the calculated
       angles in degrees
    real(real64), intent(out) :: tissue_angles_deg(count(exp_vecs_selection_index))
    ! ...
  end subroutine compute_tissue_versatility
end module avmod
```

5.3 How to generate documentation

1. Run `ford ford.yml` in the project root.
2. Open the generated HTML file (`doc/index.html`) in your browser.

6 Fortran to other programming languages Interface

Ideally, we build our Tensor Omics and F42 code so that it can be included and called from any other programming language. This comprises R, of course, but also Python, Lua, Julia, Matlab, etc. We can easily achieve this by offering two interface functions for every Tensor Omics or F42 function that we implement. Importantly, we should implement such pure Fortran functions in Fortran `modules`. Then add another function of the same name with appended "_R" to expose the module function to R and a `bind(C)` with `use_iso_c_binding` and naming the function with appended "_C". See the below example for clarification.

```
! High-performance module code
module tensor_omics
contains
  subroutine project_vector_core(v, n)
    real(real64), intent(inout) :: v(n)
    integer, intent(in) :: n
    ! core vector projection logic
  end subroutine
end module

! R interface
subroutine project_vector_R(v, n)
  real(real64) :: v(*)
  integer :: n
  use tensor_omics
  call project_vector_core(v, n)
end subroutine

! C / WASM / Python interface
subroutine project_vector_C(v, n) bind(C, name="project_vector_C")
  use iso_c_binding
  real(c_double) :: v(*)
  integer(c_int) :: n
  use tensor_omics
  call project_vector_core(v, n)
end subroutine
```

These distinctions are essential for both performance and correctness. Please document this behavior in all wrapper comments.

6.0.1 Important note on memory and array copies:

- **R wrappers:** When calling Fortran routines from R (using wrappers with the "_R" suffix), the R interface will *always* create a copy of the arrays passed as arguments. This is a fundamental property of the R–Fortran interface and ensures that the original R objects remain unchanged. Therefore, the following statement must be included in the documentation of every wrapper: **When using these R wrapper functions, copies of the arrays will be created. No direct modification of the original R objects occurs.**
- **C wrappers:** When calling Fortran routines from C (using wrappers with the "_C" suffix and `bind(C)`), the interface operates *directly* on the memory provided by the caller. Therefore, the following statement must be included in the documentation of every wrapper: **When using these C wrapper functions, no copies of the arrays will be created. The Fortran routine will operate directly on the memory provided by the caller.**

You may add these legends manually, or for convenience, use the snippets `tox:legend_r` and `tox:legend_c` found in `snippets/tox_snippets.json`.

6.1 When to Write Wrappers for Fortran Functions

Only write wrappers for Fortran functions and subroutines that provide truly new or essential functionality to the API languages (C, R). For example, generic utilities such as sorting, which are already well-supported in most high-level languages, should not be wrapped unless there is a compelling reason. Focus on exposing the core scientific Tensor Omics or F42 functions.

6.2 The _tox API Layer: Safe Data Access and Usage

In addition to the standard wrappers, we will implement another layer of API functions, conventionally named with the suffix `_tox`, that are designed to operate using a pointer to our `txdata` Fortran type. This approach allows for efficient and direct manipulation of complex data structures from API languages, while maintaining safety and clarity.

It is good practice to expose all relevant Fortran routines, but in the user manual and documentation, we will emphasize the use of the `_tox` interface for most users. This interface is safer and more robust, as it manages the internal data structures and memory correctly. Direct access to lower-level routines is possible, but should be used with caution: the user must ensure that arrays or pointers passed to Fortran are not modified elsewhere (e.g., by libraries like matplotlib or numpy) while Fortran code is using them, to avoid undefined behavior or memory corruption.

Documentation guideline: In the manual, show usage examples of the `_tox` interface as the recommended way, and only mention that direct calls are possible for advanced users who understand the risks and memory management requirements.

Example: The `analyze_tox` function takes a pointer to a `txdata` object and performs analysis, ensuring all memory and data integrity is managed by the Fortran backend. Direct calls to lower-level routines should be clearly marked as advanced usage.

6.3 Safe and Efficient Interface Design for `bind(C)` in Tensor Omics

To expose Tensor Omics routines safely and portably, we use minimal `bind(C)` wrappers around internal Fortran procedures. Each wrapper is designed for ABI stability, no-copy interop, and zero semantic loss.

6.3.1 Keep `bind(C)` wrappers minimal

Wrap only, no computation.

```
subroutine project_vector_C(x, n) bind(C, name="project_vector_C")
  use iso_c_binding
  real(c_double) :: x(*)
  integer(c_int) :: n
  call project_vector_core(x, n)
end subroutine
```

6.3.2 Use `iso_c_binding` types

Avoid compiler-specific kinds.

```
real(c_double) :: x(*)
integer(c_int) :: n
```

6.3.3 Accept arguments as C-style pointers

Do not use assumed-shape or allocatable arrays.

```
real(c_double) :: x(*)      ! OK
real(c_double), dimension(:) :: x  ! Not allowed
```

6.3.4 Avoid optional arguments

Optional arguments are not allowed in R and C wrappers because `bind(C)` and `.Fortran()` are not compatible with them. If optional arguments are needed, they should only be used in core subroutines that are called directly from Fortran.

6.3.5 Pass all arguments by reference

Do not use `value`; pass pointers from caller (Python, Lua, etc.).

```
real(c_double) :: x(*)
integer(c_int) :: n
```

6.3.6 Preserve SIMD and optimization pathways

Call computational logic in a separate routine that uses full Fortran features.

```
subroutine project_vector_core(x, n)
  real(real64), intent(inout) :: x(n)
  integer, intent(in) :: n
  ! fast loop, vectorizable
end subroutine
```

6.3.7 Use stable symbol names

Avoid compiler-dependent mangling by specifying names explicitly.

```
subroutine project_vector_C(x, n) bind(C, name="project_vector_C")
```

6.3.8 Ensure ABI compatibility with all consumers

Expose only basic types: scalars and flat arrays.

```
real(c_double) :: vec(*)
integer(c_int) :: n
! No derived types, assumed-shape arrays, or allocatables
```

6.3.9 Verify type equivalence when calling core Fortran routines

Make sure internal types match `iso_c_binding` externally.

```
use iso_c_binding
real(real64) :: x(n)
! Ensure real(real64) == real(c_double) on your compiler
```

6.3.10 Avoid aliasing in calls to `bind(C)` routines

Do not pass the same variable twice under different names.

```
! DON'T - Unsafe aliasing: same memory twice
call project_vector_C(x, x, n)

! OK - Pass unique memory regions only
call project_vector_C(x, y, n)
```

This structure ensures maximum compatibility and reproducibility across calling environments without compromising Fortran-native performance.

6.4 How to pass arguments back and forth to C, and thus Python, without memory copy.

When passing arrays or strings between Fortran and C (or Python), you can achieve zero-copy interoperation using the ISO C binding module and pointer association. Below are canonical examples for both arrays and C strings:

int, real, logical, complex Fortran assumed shape arrays

```
module interop_example
  use iso_c_binding
  implicit none
contains

! Entry point called from C
subroutine process_int_array(ptr, n, f_arr) bind(C)
  use iso_c_binding
  implicit none
  type(c_ptr), value :: ptr         ! Raw C pointer passed from C
  integer(c_int), value :: n        ! Number of elements

  integer(c_int), pointer :: f_arr(:) ! Fortran pointer to wrap the C array

  ! Convert raw C pointer into a Fortran assumed-shape pointer
  call c_f_pointer(ptr, f_arr, [n])

  ! f_arr can now be passed to any subroutine expecting an assumed-shape array
  call do_work_on_array(f_arr)

end subroutine process_int_array

! This subroutine accepts an assumed-shape array and works with it
subroutine do_work_on_array(arr)
  implicit none
  integer(c_int), intent(inout) :: arr(:)
  integer :: i
  do i = 1, size(arr)
    arr(i) = arr(i) + 1
  end do
end subroutine do_work_on_array

end module interop_example
```

C Strings to Fortran Strings

This is for C Strings, but should be treated with caution, because the result **cannot be used as assumed length Fortran chars**.

```
module charptr_demo
  use iso_c_binding
  implicit none
contains

subroutine process_c_string(c_str_ptr, str_len) bind(C)
  use iso_c_binding
  implicit none

  ! Arguments from C
  type(c_ptr), value :: c_str_ptr
  integer(c_int), value :: str_len

  ! Fortran pointer to reshape the C char*
  character(kind=c_char), pointer :: str(:)

  ! Map the C pointer to a Fortran pointer
  call c_f_pointer(c_str_ptr, str, [str_len])

  ! NOTE:
  ! The above Fortran pointer can be passed as an assumed shape Array
  ! to any Fortran subroutine or function expecting an assumed shape
  ! argument. Internally they are converted without (much) overhead.

  ! Print characters one by one (to show contents and no copy)
  print *, 'Received C string (raw char*):'
  print '(a)', str ! Note: prints whole array as string

  ! Optionally: Loop through chars
  ! do i = 1, str_len
  !   write(*,'(a)', advance='no') str(i)
  ! end do
  ! write(*,*)
end subroutine process_c_string

end module charptr_demo
```

Summary Table of Interoperable Array Patterns

Direction	Method	Copy?	Safe?
C → Fortran	`c_f_pointer`	No	Yes
Fortran → C	`c_loc + shape info`	No	Yes
Fortran descriptor to C	Not possible	N/A	No
Allocatable return	Not portable	Yes (implicit)	No

This approach ensures efficient, copy-free interoperation between Fortran and C while preserving the expressiveness of assumed-shape arrays internally in Fortran.
Caller responsibility: The external caller (e.g., from C, Python, Lua) must allocate a 1D array of size `nrow * ncol` and ensure it is stored in Fortran-style column-major order, or reshaped accordingly.
This pattern allows Tensor Omics to retain full multidimensional structure internally while offering safe and portable interfaces to external consumers.

7 Fortran Array Notation Cheat-Sheet

This section summarizes the three common ways to declare array arguments in Fortran procedures, with a focus on their use, safety, and suitability in modern code.

7.1 1. Assumed-Size Arrays (*)

```
subroutine example(a, n)
  real(real64), intent(in) :: a(*)
  integer, intent(in) :: n
  integer :: i
  do i = 1, n
    print *, a(i)
  end do
end subroutine
```

Meaning: `a(*)` is an assumed-size array. The subroutine does not know the array bounds and must be told explicitly via a separate argument (e.g., `n`). This is the lowest-level array form and maps closely to a raw pointer in C.

Limitations:

- No bounds checking
- `size(a)` and `lbound(a)` are illegal
- Cannot use array operations like `matmul` or `sum`
- Unsafe for slicing (`a(2:5)` is undefined)

Use case: C interoperability, legacy code, low-level memory kernels.

7.2 2. Assumed-Shape Arrays (:)

```
subroutine example(a)
  real(real64), intent(in) :: a(:)
  integer :: i
  do i = 1, size(a)
    print *, a(i)
  end do
end subroutine
```

Meaning: `a(:)` is an assumed-shape array. The compiler receives size and bounds information at runtime. Requires an explicit interface (typically via a module or `interface` block).

Advantages:

- Safe: `size(a)` is available
- Supports slicing, broadcasting, and array intrinsics
- Can use `matmul`, `dot_product`, etc.

Use case: Recommended default for most modern Fortran code.

7.3 3. Multidimensional Arrays `(:,:)` and Higher

```
subroutine process_matrix(A)
  real(real64), intent(in) :: A(:, :)
  print *, size(A, 1), size(A, 2)
end subroutine
```

Meaning: Each colon represents a dimension of unknown size. Like 1D assumed-shape arrays, this requires an explicit interface and passes shape info at runtime.

Features:

- Full array operations: `transpose`, `matmul`, etc.
- Safer and clearer semantics for tensors and matrices

Use case: Tensor Omics vector spaces, image/matrix processing, linear algebra routines.

Summary Table

Notation	Bounds Known?	Safe?	Use Case
`(*)`	No	No	C interop, low-level
`(:)`	Yes	Yes	Modern Fortran arrays
`(:,:)`	Yes	Yes	Matrices, tensors

For all assumed-shape forms (`(:)`, `(:,:)`), ensure the subroutine has an explicit interface or is in a module.

8 Memory Passing Behavior in Fortran (F42 Standard for Tensor Omics)

8.1 General Principles

In Fortran (standards 77–2018), arguments to subroutines and functions are, by default, passed **by reference** rather than by copying. This applies to scalars, arrays, and derived types.

Thus:

- Scalars (e.g., `integer`, `real`) are passed by reference.
- Arrays (e.g., `integer, dimension(:)`, `real, dimension(:,:)`) are passed by reference.
- Derived types are passed by reference.

8.2 When Copies Are Made

Despite the default pass-by-reference behavior, hidden copies may be generated by the compiler in the following situations:

- **Passing expressions**: e.g., `call process(a+b)` requires evaluation and temporary storage.
- **Passing strided array sections**: e.g., `array(1:10:2)` is non-contiguous and may trigger copying.
- **Passing non-contiguous slices**: particularly relevant for rows in a 2D array.

8.3 Tensor Omics DataTables: Column and Row Passing

Tensor Omics DataTables store typed data in 2D Fortran arrays of the form:

- `real_array(n_rows, n_real_columns)`
- `int_array(n_rows, n_int_columns)`
- `char_array(n_rows, n_char_columns)`

Accessing Columns:

- Example: `real_array(:, j)`
- Behavior: Passed by reference, **no copy made**.
- Reason: Slicing across the first index in column-major layout yields contiguous memory.

Accessing Rows:

- Example: `real_array(i, :)`
- Behavior: Typically triggers a hidden copy.
- Reason: Slicing across the second index accesses non-contiguous memory addresses, requiring packing into a contiguous temporary.

8.4 Summary Tables

8.4.1 General Fortran Behavior

Action	**Example**	**Copy Made?**
Pass full array	`array`	No
Pass contiguous slice	`array(:, j)`	No
Pass strided slice	`array(1:10:2, j)`	Yes
Pass expression	`array(:, j) * 2.0`	Yes

8.4.2 Tensor Omics DataTables Specifics

Access Type	**Example**	**Copy Made?**
Access full column	`real_array(:, j)`	No
Access full row	`real_array(i, :)`	Yes

8.5 Best Practice Recommendation for Tensor Omics

- Always design core functions to work with columns if possible.
- Avoid passing strided slices or expressions directly to subroutines.
- Treat rows carefully: copy manually if absolutely needed, otherwise prefer per-element processing.

8.6 Conclusion

Efficient memory handling is critical for Tensor Omics. Understanding Fortran's passing conventions allows maximization of speed, minimization of hidden temporaries, and preservation of F42 computational standards.

9 Critical Practices in TOX Interfacing and Control Flow

This section consolidates three key implementation standards for TOX and F42 compliance: correct R–Fortran interfacing, prohibition of `GOTO`, and manual memory cleanup.

9.1 Why `bind(C)` Is Dangerous for R Interfacing

Using `bind(C)` in Fortran changes the function's symbol name and calling convention to conform to the C ABI. This is incompatible with R's native `.Fortran()` interface, which relies on legacy Fortran symbol conventions (e.g., appending underscores, pass-by-address semantics).

Problems Caused by `bind(C)` in R Packages

- **Breaks Symbol Lookup:** R expects mangled names like `mysub_`, which `bind(C)` suppresses.
- **Incorrect Argument Handling:** Changes to calling convention may result in wrong memory alignment or crashes.
- **Unnecessary:** R handles native Fortran linking automatically during package build.

Correct Way to Interface R with Fortran

1. Write your subroutines in pure Fortran *without* `bind(C)`.
2. Place Fortran files in `src/` directory of the package.
3. Register your compiled code using `useDynLib(pkgname)` in `NAMESPACE`.
4. Call Fortran code from R via `.Fortran()`:

```
result <- .Fortran("mysub",
  as.double(x), as.double(y), result = double(1))$result
```

Minimal Example of Package Setup

- `DESCRIPTION`: no special flags needed.
- `NAMESPACE`: `useDynLib(mytoxpkg)`
- `src/mysub.f90`:

```
subroutine mysub(x, y, result)
  real(real64) :: x, y, result
  result = x + y
end subroutine
```

9.2 Why `GOTO` Is Forbidden in F42 and TOX

`GOTO` is a low-level control structure that violates the principles of structured, readable, and verifiable code. While allowed by Fortran, its use is incompatible with both modern software engineering and WebAssembly (WASM) compilation.

Reasons for Prohibition

- **Non-Structured Control Flow:** Makes code hard to reason about and maintain.
- **Incompatible With WASM:** WebAssembly requires structured control graphs.
- **Opaque for Debugging:** Introduces implicit jump logic that hinders traceability.
- **Fails F42 Goals:** Violates statelessness, modularity, and verifiability.

9.3 How to Replace `GOTO` Correctly

1. Use `exit` to break out of a loop

```
do i = 1, n
  if (array(i) < 0.0) then
    exit
  end if
end do
```

2. Use `cycle` to skip to next iteration

```
do i = 1, n
  if (.not. valid(i)) cycle
  call process(i)
end do
```

3. Use `select case` for structured branching

```
select case(code)
case(1)
  call handle_case1()
case(2)
  call handle_case2()
case default
  call handle_default()
end select
```

9.4 Memory Cleanup: Manual Destructors

Since Fortran has no `try/finally` construct and F42 disallows `final ::` destructors due to their opacity, all memory cleanup must be performed manually.

Pattern: Explicit Cleanup Subroutine

```
subroutine deallocate_datatable(dt)
  type(DataTable), intent(inout) :: dt

  if (allocated(dt%real_array)) deallocate(dt%real_array)
  if (allocated(dt%char_array)) deallocate(dt%char_array)
  if (allocated(dt%int_array))  deallocate(dt%int_array)
  if (allocated(dt%column_names)) deallocate(dt%column_names)
end subroutine
```

Pattern: Simulated try-finally via logical block

```
logical :: success
success = .false.

call allocate_datatable(dt, ierr)
if (ierr == 0) then
  call compute(dt)
  success = .true.
end if

call deallocate_datatable(dt)
```

Conclusion

All R–Fortran interfacing, control flow, and memory management in TOX must follow F42 principles:

- Never use `bind(C)` unless wrapping via `.Call()` (not `.Fortran()`).
- Never use `GOTO` — always use structured constructs.
- Always provide manual cleanup subroutines for types containing allocatables.

These principles ensure safety, readability, WASM compatibility, and long-term maintainability.

10 Scientific Kernel and API Design Specification

10.1 Scope

This specification defines the design principles, coding conventions, and interface structure for the Tensor Omics Scientific Kernel (SK) and its associated API helper layers. The aim is to ensure performance, reproducibility, portability, and clarity of responsibility between computational components.

10.2 Scientific Kernel (SK)

10.2.1 Definition

The Scientific Kernel consists exclusively of pure Fortran subroutines and functions implementing the numerical and geometric methods of Tensor Omics.

10.2.2 Core Properties

1. **Purity:** SK routines must have no side effects beyond modifying their output arguments.
2. **No I/O:** SK routines must not perform any file or console I/O.
3. **No Dynamic Allocation:** SK routines must not contain any `allocate` or `deallocate` statements.
4. **Memory Alignment:** All array arguments must be Fortran `allocatable` arrays to ensure alignment and contiguity.
5. **Transient Memory:** Any intermediate (scratch) memory must be preallocated by the caller and passed as arguments.
6. **Determinism:** Given the same inputs, SK routines must always return identical outputs.

10.2.3 Argument Requirements

1. Arrays must be `allocatable` and contiguous.
2. All intent must be explicitly declared (`intent(in)`, `intent(out)`, `intent(inout)`).
3. Kind parameters for floating-point arguments must be specified using `real64` from `iso_fortran_env`.

10.3 Helper Routines

10.3.1 Purpose

Helper routines manage memory allocation, type handling, and API exposure for SK routines. They may have side effects such as allocation or type conversion.

10.3.2 Naming Convention

Helper routines follow suffix-based naming to indicate their role:

- **No suffix:** Pure SK routine.
- **_alloc:** Allocation helper that computes required sizes and allocates memory.
- **_C:** C-facing API wrapper exposing SK routines via `bind(C)` and `iso_c_binding`.
- **_R:** R-facing API wrapper calling SK routines via `.Fortran` or `.Call`.
- **_tox:** Helper operating on the `tox_data` derived type.
- **_tox_C:** C-facing API wrapper for a _tox helper.
- **_helper:** General-purpose orchestration helper not bound to a specific API.

10.3.3 Module Organization

1. All SK and helper routines reside in the same Fortran module for discoverability.
2. Naming conventions are strictly enforced to distinguish pure SK code from helpers.
3. Helpers may call SK routines or other helpers but must not duplicate SK logic.

10.4 API Exposure Layers

10.4.1 C API (_C, _tox_C)

1. Implemented using `bind(C)` and `iso_c_binding`.
2. Accept and return only C-interoperable types.
3. May perform allocation if required.
4. Designed to be callable directly from C, Python (via `ctypes` or `cffi`), and other C-compatible languages.

10.4.2 R API (_R)

1. Implemented as R functions calling `.Fortran` for direct array access.
2. `.Fortran` calls may incur data copying and cannot pass pointers directly.
3. **Note** that we will implement a thin `C glue layer` to expose the above '_tox_C' helpers to R and enable pointer passing.

10.5 tox_data Type

10.5.1 Purpose

A Fortran derived type `tox_data` serves as a container for all related Tensor Omics datasets, ensuring data integrity and preventing unintentional modification from higher-level languages.

10.5.2 Access Policy

1. Access to `tox_data` contents is via getter functions.
2. Setter functions might be provided but discouraged for normal use. As we expose the scientific kernel functions directly, users who want to be able to modify expression-vectors, family-centroids, or similar will be required to manage memory and their data manually.
3. Rich querying is provided via a GraphQL interface. We might implement a 'jq' interface on top of that, too.

10.6 Coarray Considerations

1. SK routines are compatible with coarrays as they operate on preallocated, caller-provided memory.
2. C API wrappers do not expose coarrays; coarray usage requires direct Fortran coding by the user.

10.7 WebAssembly (WASM) Considerations

1. SK routines are compilable to WASM as they are pure and self-contained.
2. API helpers may require adaptation or (most likely) exclusion depending on WASM environment capabilities. This can be done via Fortran precompiler flag usage:

   ```
   #ifndef WASM
   subroutine family_centroid_alloc(...)
     ! ...
   end subroutine family_centroid_alloc
   #endif
   ```

10.8 Enforcement

Naming conventions must be validated during peer revision.

11 High-Performance Fortran Compilation and Memory Alignment

Note that this section is largely deprecated, except the Coarray section and the content on how to optimize for certain compilers and hardware. But we likely want to avoid such manual optimizations.

This section provides practical guidelines for scientific programmers to write memory-efficient and performance-optimized Fortran code using OpenMP, SIMD, and compiler-specific features. It also includes instructions on compiler configuration and memory alignment strategies suitable for AVX2 and AVX-512 systems.

11.1 Precompiler include file for constants

We need a precompiler macros file that needs to be included in all Fortran files that use macros.

File `precompiler_constants.F90`:

```
#ifndef DEFAULT_ALIGNMENT
#define DEFAULT_ALIGNMENT 32
#endif
```

11.2 OpenMP and SIMD: Double Parallelization

Modern Fortran compilers support combined OpenMP threading and SIMD vectorization through a single directive:

```
!$omp parallel do simd schedule(static) private(i)
do i = 1, n
a(i) = b(i) + c(i)
end do
!$omp end parallel do simd
```

This pragma enables:

- **Thread-level parallelism** via OpenMP.
- **Instruction-level parallelism** via SIMD (Single Instruction, Multiple Data).

This combination splits loop iterations across threads, and each thread processes elements in SIMD fashion. It is ideal for dense numerical operations.

11.3 Compiler-Specific Pragmas and Flags

To *ensure* the above OpenMP *and* vectorized parallelization (see 11.2) works with maximum efficiency, we need to use compiler pragmas for Intel and gfortran compilers. See this example that tells both compilers that the Arrays used in the `do`-loop actually are setup correctly for vectorized (`SIMD`) parallelization on the level of the CPU:

```
!DIR$ ASSUME_ALIGNED a:64, b:64, c:64
!$omp parallel do simd schedule(static) private(i) aligned(a, b, c:64)
do i = 1, n
  a(i) = b(i) + c(i)
end do
!$omp end parallel do simd
```

The above ensures most efficient compilation, if and only if the right compiler flags are set (see below sections 11.3.4 and 11.3.5), too.

11.3.1 How to optimize for all CPU architectures?

The problem here is that the above `memory alignment boundary in bytes` set to 64 depends on the CPU vectorized instruction set, i.e. `AVX2`, `AVX512`, etc. So, we need to do some macro magic to squeeze the last bit of maximum efficency out of our code for any hardware it is compiled on:

```
! The below line should likely be the first line
! in any Fortran file that has macros.
#include "precompiler_constants.F90"

! This included module sets the correct memory alignment boundary in bytes
! and ensures that when allocating arrays in Fortran they are correctly aligned.
use config

#if DEFAULT_ALIGNMENT == 16
!DIR$ ASSUME_ALIGNED a:16, b:16, c:16
!$omp parallel do simd schedule(static) private(i) aligned(a, b, c:16)
#elif DEFAULT_ALIGNMENT == 32
!DIR$ ASSUME_ALIGNED a:32, b:32, c:32
!$omp parallel do simd schedule(static) private(i) aligned(a, b, c:32)
#elif DEFAULT_ALIGNMENT == 64
!DIR$ ASSUME_ALIGNED a:64, b:64, c:64
!$omp parallel do simd schedule(static) private(i) aligned(a, b, c:64)
#elif DEFAULT_ALIGNMENT == 128
!DIR$ ASSUME_ALIGNED a:128, b:128, c:128
!$omp parallel do simd schedule(static) private(i) aligned(a, b, c:128)
#else
!$omp parallel do simd schedule(static) private(i)
#endif

do i = 1, n
  a(i) = b(i) + c(i)
```

```
end do
!$omp end parallel do simd
```

Note see section 11.7 for the above `config` module, please.

11.3.2 OpenMP Variable Scoping: `private` and `reduction`

When using OpenMP parallel loops, it is essential to correctly declare the scope of all variables used within the loop body. This ensures thread safety and avoids unintended data races or compiler misoptimizations.

Most importantly:

- **Loop indices** must be declared as `private`, e.g., `private(i)`.
- **Temporary variables or accumulators** introduced inside the loop body must also be marked `private`, unless they are used in a reduction.
- **Reduction variables** (e.g., sums or maxima computed across threads) must be declared using the `reduction(...)` clause.

Example:

```
real(real64) :: tmp
real(real64) :: sum

!$omp parallel do simd private(i, tmp) reduction(+:sum) schedule(static)
do i = 1, n
  tmp = compute(i)
  sum = sum + tmp
end do
!$omp end parallel do simd
```

In this example:

- `i` is the loop counter and must be private.
- `tmp` is a per-iteration local value and must be private to avoid race conditions.
- `sum` is a global accumulator and must be declared as a reduction variable with the `+` operator.

Failing to properly scope these variables may lead to undefined behavior, wrong results, or silent compiler fallbacks to scalar execution. Always carefully inspect variable use when applying OpenMP to numerical loops.

11.3.3 Reduction Operations in OpenMP SIMD Loops

When performing numerical or logical accumulation in parallel regions, OpenMP provides `reduction` clauses that ensure correctness and performance across threads. These clauses combine thread-local results using an associative operation into a final value at the end of the parallel loop.

Supported Reduction Operations in OpenMP:

Operation	**Description**	**Example**
`+`	Sum	`reduction(+:sum)`
`*`	Product	`reduction(*:prod)`
`max`	Maximum value	`reduction(max:maxval)`
`min`	Minimum value	`reduction(min:minval)`
`-`	Subtraction*	`reduction(-:diff)`
`iand`	Bitwise AND (integers)	`reduction(iand:mask)`
`ior`	Bitwise OR (integers)	`reduction(ior:flag)`
`ieor`	Bitwise XOR (integers)	`reduction(ieor:flag)`
`.and.`	Logical AND (booleans)	`reduction(.and.:ok)`
`.or.`	Logical OR (booleans)	`reduction(.or.:status)`

Note:

- Most common are `+`, `max`, and `min` for numerical accumulations.
- The subtraction operator `-` is supported, but caution is advised: subtraction is not associative, so the result may depend on thread order. Use only if mathematically justified.
- Logical and bitwise reductions are especially useful for flags, masks, and status checks.

Best practice: Always declare temporary or accumulator variables as `private(...)` or `reduction(...)` depending on their intended aggregation behavior. This prevents race conditions and ensures correctness.

11.3.4 Intel Compiler (`ifort` or `ifx`)

Recommended flags:

```
-O3 -qopenmp -xHost -align array64byte -qopt-zmm-usage=high -qopt-prefetch=3 -qopt-matmul -fPIC
```

11.3.5 GNU Compiler (`gfortran`)

Recommended flags:

```
-O3 -march=native -mtune=native -fopenmp -ffast-math -funroll-loops -ftree-vectorize
-fassociative-math -fPIC
```

11.4 Configuring `fpm.toml`

```
name = "tensor-omics"
version = "0.1.0"
license = "MIT"

[install]
library = true

[library]
type = "shared"
```

11.5 Compile Script with Hardware-Dependent Defaults

Use a compile script to choose the optimal alignment at runtime and apply the appropriate compiler profile:

```
#!/bin/bash
# build.sh | Optimized build script for FPM with dynamic alignment
# Default fallback alignment for the most likely situation:
ALIGN=32
# Detect capabilities in order of descending priority:
if lscpu | grep -q amx; then
ALIGN=128
elif lscpu | grep -q avx512; then
ALIGN=64
elif lscpu | grep -q avx2; then
ALIGN=32
elif lscpu | grep -q sse2; then
ALIGN=16
fi
# Detect compiler and choose appropriate profile:
if [[ "$FC" == "ifx" || "$FC" == "ifort" ]]; then
```

```
  FLAGS="-O3 -qopenmp -xHost -align array64byte -qopt-zmm-usage=high -qopt-prefetch=3 -qopt-matmul
  -fPIC"
  COMPILER="ifx"
else
  FLAGS="-O3 -march=native -mtune=native -fopenmp -ffast-math -funroll-loops -ftree-vectorize
  -fassociative-math -fPIC"
  COMPILER="gfortran"
fi

# Detect --max-performance flag
MAX_PERF_FLAG=""
for arg in "$@"; do
  if [[ "$arg" == "--max-performance" ]]; then
    MAX_PERF_FLAG="-DMAX_PERFORMANCE"
  fi
done

# Informative output:
# Build with selected profile and alignment parameter:
export FC
fpm build --compiler $COMPILER --flag "$FLAGS" --flag "-DDEFAULT_ALIGNMENT=$ALIGN"
          --flag "$MAX_PERF_FLAG"

echo "Build complete with compiler: $COMPILER, alignment: $ALIGN bytes, optimization flags: $FLAGS,
      max performance: $MAX_PERF_FLAG"

# Find .so file in the build directory
sofile=$(find build -name 'libtensor-omics.so' | head -n 1)

# Create symbolic link in the build directory with relative path
if [[ -n "$sofile" ]]; then
  # Extract the relative path from build/ directory
  relative_path=$(realpath --relative-to=build "$sofile")
  ln -sf "$relative_path" build/libtensor-omics.so
  echo "Created symlink: build/libtensor-omics.so -> $relative_path"
else
  echo "Warning: libtensor-omics.so not found in build directory"
fi
```

11.6 Contiguous and Aligned Arrays in Fortran

A `contiguous` array is one whose elements are stored sequentially in memory. An `aligned` array is one that starts at a memory address that is a multiple of the desired alignment boundary (e.g., 32 or 64 bytes).

All `allocatable` arrays in Fortran are inherently contiguous. Aligned allocation ensures that SIMD instructions can operate efficiently.

11.7 Configuration Module

```
#include "precompiler_constants.F90"

module config
implicit none
#ifdef DEFAULT_ALIGNMENT
integer, parameter :: alignment = DEFAULT_ALIGNMENT
```

```
#endif
end module config
```

11.8 Alignment-Aware Padding Function

This helper computes padded sizes for any array dimension:

```
module alignment_utils
use config
implicit none
contains

integer function padded_size(n)
integer, intent(in) :: n
padded_size = alignment * ceiling(real(n) / alignment)
end function padded_size

end module alignment_utils
```

Use this in your allocation logic:

```
use alignment_utils
integer :: m, padded_m
real(real64), allocatable :: A(:)

m = 1000
padded_m = padded_size(m)
allocate(A(padded_m))
```

This ensures alignment across 1D, 2D, or ND arrays by applying padding logic to each dimension before allocation.

11.9 Alignment-Aware Allocation for ND Arrays

To ensure all dimensions are padded and aligned, wrap padding in a helper procedure or macro-like pattern per array rank. Example for 2D:

```
integer :: nx, ny, px, py
real(real64), allocatable :: A(:,:)

nx = 800
ny = 600
px = padded_size(nx)
py = padded_size(ny)
allocate(A(px, py))
```

Repeat this pattern for higher-dimensional arrays.

11.10 Note on Using Single Helper for Arbitrary Ranks

It is not possible to pass arbitrary-rank assumed-shape arrays into a single routine that allocates them, because Fortran does not support polymorphic allocatable shapes. Instead, compute and apply padding for each dimension prior to allocation, using ‘padded_size‘.

Thus, the best practice is to:

- Store dimension sizes in variables.

- Use `padded_size()` for each.
- Allocate directly in the relevant context.
- Optionally provide `allocate_padded_1D()`, `...2D()` helpers per rank for clarity.

11.11 Best Practice Summary

- Use `!$omp parallel do simd schedule(static)` to enable both thread and instruction-level parallelism.
- Use compiler-specific alignment and vectorization flags in `fpm.toml`.
- Dynamically set alignment to 32 or 64 via a shell script based on CPU capabilities.
- Use `padded_size()` to round up all dimensions to nearest alignment multiple.
- Allocate arrays explicitly using padded sizes to ensure SIMD-safe alignment.
- Maintain a `config` module to centralize the alignment parameter.

11.12 Verifying Alignment at Runtime (Possible but *not* practical)

Intel and GNU compilers do not provide standard runtime alignment checks. For debugging, one may use C interoperability and POSIX memory alignment checks (advanced). For practical purposes, correct use of allocation and compiler flags suffices.

11.13 Conclusion: Thread-Level and CPU-Local Vectorized Parallelization

By combining OpenMP, SIMD, and alignment-aware allocation, scientific Fortran code can achieve modern HPC-level efficiency while maintaining clarity and portability. This setup enables seamless transition to C bindings, WebAssembly targets, and multi-threaded environments such as browser-based Tensor Omics visualizations or clustered batch jobs.

Importantly, the following Fortran code template enables **double parallelization**: first at the **thread level**, where array chunks are distributed across cores using the `!$omp parallel do simd schedule(static)` directive; and second at the **instruction level**, where each thread processes its assigned chunk using vectorized CPU instructions enabled by the `simd` clause.

11.14 Best Practice: SIMD in Leaf Loops, OpenMP in Top-Level Loops

In Tensor Omics, it is crucial to distinguish between two levels of parallelization:

- **Thread-level parallelism**: achieved via `!$omp parallel do` at the top level, typically when iterating over gene families or other large-scale, independent units of computation.
- **Vector-level (SIMD) parallelism**: achieved via `!$omp simd` within inner loops that perform the actual arithmetic operations, e.g., normalization, centroid computation, tensor updates.

Key rule: *Apply `!$omp simd` only at the level of the loop that does the actual vector arithmetic. This is the most nested loop — the "leaf" loop.*

And: *Apply `!$omp parallel do` only at the top level, for instance over gene families or modules.* This should be confirmed and coordinated with the software architect (Vivian or Asis) to ensure it aligns with the design of the specific routine.

11.14.1 Why?

Nested use of `simd` inside a loop that is already marked with `parallel do simd` leads to undefined or inefficient behavior. While technically allowed by OpenMP, compilers may:

- silently ignore the inner `simd` region,
- fail to vectorize at all,
- or generate scalar fallback code.

Hence, we *do not* use both `parallel do simd` and inner `simd` together — unless there is a specific, performance-tested reason.

11.14.2 Incorrect (nested SIMD, should be avoided)

```
subroutine family_centroids(...)
  !$omp simd aligned(a, b, c:64)
  do j = 1, ndim
    ...
  end do
end subroutine

!The below repeated 'simd' is the problem here:
!$omp parallel do simd schedule(static) private(i)
do i = 1, nfam
  call family_centroids(...)
end do
```

Problems:

- The outer loop redundantly specifies both `parallel do` and `simd`, causing potential nested vectorization issues.
- The alignment is hardcoded (64) instead of using the macro defined in our configuration header.

11.14.3 Correct and recommended pattern in Tensor Omics

```
subroutine family_centroids(...)
  !$omp simd aligned(a, b, c:64) private(j)
  do j = 1, ndim
    ...
  end do
end subroutine

!$omp parallel do schedule(static) private(i) aligned(...)
do i = 1, nfam
  call family_centroids(...)
end do
```

Note, see section 11.3.1 for details on how to set the correct memory alignment boundary in bytes, e.g. "`aligned(a, b, c:64`" and "`!DIR$ ASSUME_ALIGNED a:64, b:64, c:64`" to 16, 32, 64, or 128, using precompiler macros.

This is the canonical Tensor Omics pattern:

- The leaf loop that performs arithmetic (e.g., element-wise ops) uses `simd` and the `aligned(...:DEFAULT_ALIGNMENT)` directive. This macro is defined in the precompiler header `precompiler_constants.F90`.
- The outer loop over independent entities (e.g., gene families) uses `parallel do` for thread-level parallelism.

Important: Always use `#include "precompiler_constants.F90"` if alignment is to be ensured. This file defines `DEFAULT_ALIGNMENT`, which is passed at compile time via the build script depending on AVX512, AVX2, or baseline SIMD capabilities. Using this macro ensures correct alignment in both `!DIR$ ASSUME_ALIGNED` and `aligned(...)` clauses.

11.14.4 Summary

- **Use `!$omp simd` only in the leaf loop doing actual vector math.**
- **Use `!$omp parallel do` only in top-level outer loops.**
- **Avoid using `parallel do simd` in Tensor Omics unless explicitly reviewed.**
- **Always include `precompiler_constants.F90` if using alignment directives.**
- **Use correct directives to keep hardware compatibility across compilers and targets:**
  - `aligned(...:<DEFAULT_ALIGNMENT>)` *and*
  - `!DIR$ ASSUME_ALIGNED a:<DEFAULT_ALIGNMENT>, b:<DEFAULT_ALIGNMENT>, c:<DEFAULT_ALIGNMENT>`.

11.15 Distributed Parallelism with Coarrays and OpenMP SIMD Loops

This section describes how Tensor Omics achieves **triple parallelization** on distributed systems by combining:

- **Coarrays** for inter-node (inter-image) parallelism,
- **OpenMP threads** for intra-node (core-level) parallelism,
- and **SIMD vectorization** for CPU-instruction-level efficiency.

This hybrid model enables high-throughput omics analysis on SLURM clusters with minimal programming overhead and near-optimal hardware utilization.

11.15.1 Local Data Strategy in Coarrays

Coarrays in Fortran natively represent distributed memory: each *image* is conceptually an independent process with its own memory space. This makes them ideal for Tensor Omics, where data can be split into families and processed independently.

The key principle is:

> **Each image works on its own local subset of gene families.** Data from other images is only accessed explicitly when needed.

Within each image, we apply the previously described memory-optimized loops with:

- `!$omp parallel do simd` — enabling OpenMP threading and SIMD vectorization.
- Intel pragmas and `aligned(...)` clauses — ensuring memory is aligned to AVX boundaries.

Crucially, **looping over coarrays automatically refers to the local image's data**, without additional indexing or copying, unless explicitly accessing remote images (e.g., `b[2]` to read from image 2).

11.16 Brief summary: Declaring and Accessing Coarrays

A coarray is declared in Fortran by appending square brackets with [*] to the variable declaration. For example:

```
real(real64), allocatable :: a(:)[:], b(:)[:], c(:)[:]
```

The colon : indicates a deferred-shape (allocatable) array, and the brackets [:] declare the variable as a coarray, with one instance per image.

Local image access is performed without brackets:

```
a(i) = b(i) + c(i)        ! Local access: all references are to this_image()
```

Remote image access uses explicit brackets with an image number:

```
a(i) = b(i)[2] + c(i)     ! Reads from image 2 (remote access)
```

Remote access should be minimized for performance and memory locality. Tensor Omics encourages access to remote images only in explicitly synchronized stages.

Memory-aligned OpenMP SIMD loops work identically with coarrays when accessing only local data. The compiler automatically uses the correct memory region without copying:

```
!DIR$ ASSUME_ALIGNED a:64, b:64, c:64
!$omp parallel do simd schedule(static) aligned(a, b, c:64)
do i = 1, n
  a(i) = b(i) + c(i)       ! Local data access on this_image()
end do
!$omp end parallel do simd
```

11.16.1 Data Loading Based on Image ID

The image-specific data loading follows a simple naming convention: each image loads its own chunk from a directory or file pattern like <base>_<image_no>.

Example:

```
integer :: me
me = this_image()

! Allocate coarrays as needed (e.g., expression vectors)
! Each image loads only its own data:
call load_txdata(me, base_file_name, expression_vecs, ...)
```

This pattern supports full scalability with large gene family datasets. For best performance:

- Split data by gene family.
- Assign families evenly to images during pre-processing.
- Avoid inter-image data transfer unless strictly required.

11.16.2 Synchronization and Remote Data Access

Coarray images execute asynchronously by default. To coordinate across images:

- Use `sync all` to wait for all images at the same code line.
- Use `sync images(*)` or `sync images(img)` for partial synchronization.
- Access remote coarray data using bracket syntax (e.g., `vec(1:n)[2]`).

If cross-image data is required for analysis (e.g., inter-family comparisons):

- **Fetch remote data explicitly** using coarray brackets.
- **Copy into aligned local arrays** before processing, preserving SIMD efficiency.
- Perform `sync all` if necessary.

Memory budgeting must ensure that any fetched data fits into local memory.

11.16.3 Compiling for Coarrays

Compilation differs slightly depending on compiler:

With Intel (ifx/ifort):

```
ifx -coarray=distributed -fopenmp -O3 -qopenmp -align array64byte -c file.f90
```

With GNU gfortran + OpenCoarrays:

```
caf -fcoarray=single -fopenmp -O3 -march=native -c file.f90
```

Linking must include coarray runtimes, especially when using OpenCoarrays (`caf` handles this).

11.16.4 SLURM Setup for Coarray Execution

Below is a complete SLURM job script to run Tensor Omics with **3 Coarray images**, each using **40 OpenMP threads**. Adjust resource counts to your system and dataset.

```
#!/bin/bash
#SBATCH --job-name=tensor-omics
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=1           # One Coarray image per node
#SBATCH --cpus-per-task=40            # 40 threads per image
#SBATCH --time=02:00:00
#SBATCH --partition=compute
#SBATCH --exclusive                   # Optional: gets full nodes
#SBATCH --output=slurm-%j.out
#SBATCH --error=slurm-%j.err

# Load modules (adjust to your environment)
module load intel/2023.1
module load openmpi/4.1.5             # or Intel MPI if using ifx/ifort

# Set OpenMP thread count
export OMP_NUM_THREADS=40
export FOR_COARRAY_NUM_IMAGES=3       # Optional: Intel Coarrays only

# Optional: pin threads for reproducibility and performance
export KMP_AFFINITY=granularity=fine,compact,1,0
export KMP_STACKSIZE=2g

# Run executable with MPI wrapper
srun ./tensor_omics_exe
```

Notes:

- Each image automatically processes only its local data.
- Use `this_image()` and `num_images()` to manage indexing.
- Combined with alignment-aware OpenMP SIMD loops, this setup provides maximal throughput.

11.16.5 Conclusion

The coarray-enabled, tripple-parallel architecture of Tensor Omics allows scalable, memory-aligned, and instruction-optimized analysis on both laptops and HPC clusters. Minimal code changes are needed to move from serial to multi-node execution. This enables true geometric tensor-based biology at scale.