Special Homework: Performant Fortran Matrix Code

Harrison Reisinger

November 17, 2025

1 Introduction

This homework is a special assignment as part of the Tools and Techniques for Computational Science at the University of Texas at Austin. In this homework, I was challenged to make an object oriented performance focused compiled code that can solve a specific matrix operation. The language I chose for this task was Fortran, as I know of its good performance, and I was familiar with the language from previous classes. The algorithm that needed to be implemented consisted of the necessity to create an input array object, an output array object, and a method to compute the output array.

The algorithm that needed to be implemented involved the initialization of an input array and solving to give an output array. There are three main input variables n, m, and k, where n $\geq$ m and n $\geq$ k. There are two input arrays x and b, where x is an n-by-n array of single precision and b is a vector of length m. The array x is initialized to have all entries equal to 2.0, and b is initialized so that all entries are equal to 1.0. The output array takes in the input array object and initializes an array y whose all its entries are set to 1.0. The array y is operated on based on Equation 1, then on the k-th column (column because Fortran is a column major code), Equation 2 is applied. The algorithm produces an n-by-n array filled with 1.0, except for column k, where, for the first m rows, it is set to 2.0.

$$y_{i,j} = \frac{y_{i,j} + 2x_{i,j}}{5} \tag{1}$$

$$y_{:,k} = y_{:,k} + b \tag{2}$$

2 Methodology

The code was organized into a directory system that organized the Fortran code. The main Fortran file was placed in a directory called app, and the modules that this main file used were placed in src. The main Fortran file's main tasks are to take in input parameters, validate the input parameters, initialize the input and output objects, call the compute method, and produce a summary output file. Besides the parameters m, n, and k, two other parameters were created to input into the main Fortran app row and print_array. The parameter row defined whether the operation on the k-th matrix line was done on the row or column of the matrix. The column was always chosen due to Fortran's column-major architecture. The parameter print_array is a toggle for whether the code should print the output array y to a file. This last parameter was only used for small n-value cases because the production of large text files significantly reduces performance. The input parameters were read into the main Fortran file via Fortran's built-in namespace file. This allowed Fortran to assign variables easily and without any third-party packages, which is useful since this code needed to be compiled on TACC.

Python was used to act as an auto-tool for the project. The Python file first compiles the Fortran code using system commands to first produce intermediate .o files and then compile the binary. The -O3 parameter was used so that the code was compiled with the highest level of stable optimization. The Fortran code was also compiled using f2018 due to a feature used to input a matrix to a module in a concise way. The Python code does the compilation in a very simple way and essentially does the complete compilation of every file every time. If I were to rewrite this, I would implement the use of a Makefile so that compilation can be done more dynamically during development; however, the code compiles very quickly, so this was not an issue during development of the code. The code for this Python file is shown at the end of the document in Appendix A.

2.1 Modules

The input array module can be seen in Listing 1 where the InputArrays object is defined and the initialization method new_input_arrays that takes n and m as input parameters. This module allocates x and b and then fills the arrays using a parallel workshare, which reduces the real time to allocate the two arrays. After this, the size in memory of the two arrays is calculated based on the size of the arrays and the type of variable that fills the array.

Listing 1: Input Arrays Module

```fortran
module input_arrays
    ! file: input_arrays.f90
    use, intrinsic :: iso_fortran_env, only: int64, real32
    implicit none
    private
    public :: InputArrays, new_input_arrays

    type, public :: InputArrays
        integer :: n, m
        real(real32), allocatable :: x(:,:)
        real(real32), allocatable :: b(:)
        integer(int64) :: x_bytes, b_bytes
    end type InputArrays

    contains
    subroutine new_input_arrays(self, n, m)
        class(InputArrays), intent(out) :: self
        integer, intent(in) :: n, m
        integer(int64) :: x_bytes, b_bytes
        integer :: i, j

        self%n = n
        self%m = m

        allocate(self%x(n,n))
        !$omp parallel workshare
        self%x = 2.0_real32
        !$omp end parallel workshare

        allocate(self%b(m))
        !$omp parallel workshare
        self%b = 1.0_real32
        !$omp end parallel workshare

        self%x_bytes = int(self%n, int64) * int(self%n, int64) *
           storage_size(self%x) / 8_int64

        self%b_bytes = int(self%m, int64) * storage_size(self%b) / 8
           _int64
    end subroutine new_input_arrays
end module input_arrays
```

The output array module is shown in Listing 2. This module is similar to the input_array module in that it has a public-facing object and a way to initialize that object. The output_array initializes the matrix y and finds the matrix's size in memory. This calculation could be further improved by taking in the size in memory of x from input_array and passing that in as an argument to output_array. This improvement would require the input_array object to be initialized first, whereas in the current implementation, it does not matter the order the array objects are initialized.

This module also has the compute method, which modifies y based on the Equations 1 and 2. The compute method takes in an InputArray object, k, and whether the operation should occur on a row or column. The operations for the computation are parallelized for each column of the array. This means that each OpenMP process takes in a column to operate on at any given time. This is optimal because Fortran is column-major, meaning that the process only does not needs to jump around in the memory to find the next entry until it reaches the end of the column. Furthermore, the omp simd allows for the iteration to be vectorized, which allows for further performance by allowing the compiler to create machine code that processes multiple array elements simultaneously with a single instruction. This is a combination of both thread and instruction parallelism.

Listing 2: Output Arrays Module

```fortran
module output_array
    ! file: output_array.f90
    use input_arrays
    use, intrinsic :: iso_fortran_env, only: int64, real32
    implicit none
    private

    public :: OutputArray

    type, public :: OutputArray
        integer :: n
        real(real32), allocatable :: y(:,:)
        integer(int64) :: y_bytes
        contains
            procedure :: compute
    end type OutputArray

    public :: new_output_array

    contains
    subroutine new_output_array(self, n)
        class(OutputArray), intent(out) :: self
        integer, intent(in) :: n
        integer(int64) :: y_bytes
        integer :: i, j

        self%n = n

        allocate(self%y(n,n))
        !$omp parallel workshare
        self%y = 1.0_real32
        !$omp end parallel workshare

        self%y_bytes = int(self%n, int64) * int(self%n, int64) *
           storage_size(self%y) / 8_int64
    end subroutine new_output_array

    subroutine compute(self, inputs, k, row)
        class(OutputArray), intent(inout) :: self
        class(InputArrays), intent(in) :: inputs
        integer, intent(in) :: k
        logical, intent(in) :: row
        integer :: i, j

        !$omp parallel do
        do j = 1, self%n
            !$omp simd
            do i = 1, self%n
                self%y(i,j) = 0.2_real32 * self%y(i,j) + 0.4_real32
                   * inputs%x(i,j)
            end do
        end do
        !$omp end parallel do

        if (row) then
            !$omp parallel do
            do i = 1, inputs%m
                self%y(k,i) = self%y(k,i) + inputs%b(i)
            end do
            !$omp end parallel do
        else ! column major (Fortran)
            !$omp parallel do
            do i = 1, inputs%m
                self%y(i,k) = self%y(i,k) + inputs%b(i)
            end do
            !$omp end parallel do
        end if
    end subroutine compute
end module output_array
```

The mem_util (Listing 3) and path_util (Listing 4) are modules that have functions that assist in the input management and output formatting. The mem_util module has the pbytes (pretty bytes), which takes in an integer number of bytes and returns a string. This string is nicely formatted to give the number of bytes with proper units (B, KiB, MiB, or GiB). This is used in the summary creation and formats the size of each of the arrays. The path_util module has a function that is called join_path that takes in a directory and a filename, then returns the full path. This can be further modified so that it just combines the front half of a path and the end half of a path, and formats it in such a way that it is always formatted correctly. This is used when calling the input namespace and when producing the output files. This is similar to a Python function (os.path.join), which does a similar operation.

Listing 3: Memory Utility Module

```fortran
module mem_util
    ! file: mem_util.f90
    use, intrinsic :: iso_fortran_env, only : int64, real64
    implicit none

    contains
    pure function pbytes(bytes) result(s)
        integer(int64), intent(in) :: bytes
        character(:), allocatable :: s
        real(real64) :: value
        integer :: index
        character(len=3), parameter :: units(4) = ['B  ', 'KiB', '
            MiB', 'GiB']
        character(len=32) :: buf

        value = real(bytes, real64) ! int32 -> real64
        index = 1

        if (value >= 1024.0_real64) then
            value = value / 1024.0_real64
            index = 2

            if (value >= 1024.0_real64) then
                value = value / 1024.0_real64
                index = 3

                if (value >= 1024.0_real64) then
                    value = value / 1024.0_real64
                    index = 4

                end if
            end if
        end if

        write(buf, '(F0.1,1X,A3)') value, units(index)

        s = trim(adjustl(buf))
    end function pbytes
end module mem_util
```

Listing 4: Path Utility Module

```fortran
module path_util
    ! file: path_util.f90
    implicit none

    contains
    function join_path(directory, filename) result(full_path)
        character(len=*), intent(in) :: directory, filename
        character(len=256) :: full_path

        if (len_trim(directory) > 0) then
            if (directory(len_trim(directory):len_trim(directory))
               == '/') then
                full_path = trim(directory) // trim(filename)
            else
                full_path = trim(directory) // '/' // trim(filename)
            end if
        else
            full_path = filename
        end if
    end function join_path
end module path_util
```

2.2 Main Program

The main Fortran file that is file, which depends on the previously shown files, holds the pain program and compiles into the binary that is invoked. The definitions of the program file are shown in Listing 5, where all of the modules are imported, the input integers, the input and output array objects, indices, summary variables, and file string designations and units are all defined. This code snippet also shows the namelist definitions where the namelist is unfolded into the input parameters.

Listing 5: Module Imports and Variable Definitions

```fortran
use, intrinsic :: iso_fortran_env, only: real32, output_unit
use mem_util, only : pbytes
use path_util, only : join_path
use input_arrays
use output_array
implicit none

integer :: n, m, k
type(InputArrays) :: inputs
type(OutputArray) :: output
logical :: row, print_summary, print_array
integer :: i, j
real(real32) :: sum_first_row, sum_kth_row, sum_first_col,
   sum_kth_col
character(len=256) :: nml_file, output_dir, output_summary_dst,
   output_array_dst

integer :: nml_unit, output_summary_unit, output_array_unit

integer :: nargs

namelist /params/ n, m, k, row, print_summary, print_array
```

These input parameters are then set to a default if the namelist is not given. Listing 6 shows the default values for the input parameters and the output units. The input unit is determined by the nml_unit built in.

Listing 6: Default parameters

```fortran
n = 100
m = 50
k = 25
output_dir = '.'

row = .false. ! Fortran is column major
print_array = .false.

output_summary_unit = 10
output_array_unit = 11
```

The binary takes in a number of arguments. The first argument is the input namelist file. The second is the direction that all output files should be placed. This is managed in Listing 7 where the number of arguments is returned, and then if there are any arguments, the namelist file is read. Then the code checks for a second argument, and then the output directory is set if it is given.

Listing 7: Argument mangament

```fortran
nargs = command_argument_count()

if (nargs > 0) then
    call get_command_argument(1, nml_file)
    open(newunit=nml_unit, file=trim(nml_file), status='old')
    read(nml_unit, nml=params)
    close(nml_unit)
end if

if (nargs > 1) then
    call get_command_argument(2, output_dir)
end if
```

After the input parameters are read into the code and set to the variables, the variables are validated. This ensures that the code does not execute incorrect parameters and ends the execution early. There are five different fail conditions:

1. $n < 0$
2. $m < 0$
3. $k < 0$
4. $m > n$
5. $k > n$

The code validation can be seen in Listing 8.

Listing 8: Input Validation

```fortran
! 3e Safeguard/validation implementation. These if/else statements
! check to make sure all input parameters are valid.
! These checks ensure that there are no index out of bounds
! errors. These checks are placed here to prevent slow failing
! of memory allocation or during computation

if (n <= 0) then ! fail 1
    write(*, '(A,I0)') 'Error(1) n must be positive, n=', n
    stop 1
end if

if (m <= 0) then ! fail 2
```

```fortran
    write(*, '(A,I0)') 'Error(2) m must be positive, m=', m
    stop 1
end if

if (k <= 0) then ! fail 3
    write(*, '(A,I0)') 'Error(3) k must be positive, k=', k
    stop 1
end if

if (m > n) then ! fail 4
    write(*, '(A,I0, A,I0)') 'Error(4) m must be less than n, m=', m
        , ' n=', n
    stop 1
end if

if (k > n) then ! fail 5
    write(*, '(A,I0, A,I0)') 'Error(5) k must be less than n, k=', k
        , ' n=', n
    stop 1
end if
```

Next, the code initializes and runs the compute. This is also where the code comments for question 3 are placed. The new_input_arrays() function initializes and sets the inputs variable to an InputArrays object. The new_output_arrays() function initializes and sets the output variable to an OutputArray object. The compute invocation takes in these input variables and then sets the y attribute in the output variable. Listing 10 shows the object initialization, computation invocation, and the code comments for question 3.

Listing 9: Object Initialization and Computation

```fortran
! 3a I placed the class initializations here because it is after the
!    definition of the n, m, and k variables but before the
!    invocation of the computation method

! 3a initializing the input object
call new_input_arrays(self=inputs, n=n, m=m)

! 3a initializing the output object
call new_output_array(self=output, n=n)

! 3b The arrys are dynamically allocated on the heap at runtime
!    this also makes senst to allocate to the heap because if
!    the size of some of the arrays being 90k x 90k it would
!    likely overflow the stack if allocated there

! 3c I used the standard dynamic allocation built into Fortran.
!    This was done so that the size of the arrays could be
!    defined at runtime based on the input parameters

! 3d I expect the code to spend the most time on the in the
!     method call below. This is because this is where the matrix
!     operations take place, even though I am using the built in
!     Fortran matrix operations, the underlying machine code needs
!     still loop over every entry
!                                                      --- or ---
!     (if it is enabled) the array save to a file at the bottom
!     of this code, this takes forever because it has to loop over
!     every entry then write to storage (not cache or ram) which
!     is very slow
!                                                    --- actually ---
!    What I found out is that a large amount of time was spend on
!    the initialization of the matricies which is the result of
!    high system time. Depending on the methodology the
!    intialization of the object could could take longer than the
!    actual computation because it required the intialization of 2
!    matricies and 1 vector. Where as the computaiton only operates
!    on 1 matrix that is already in memory. The result of the high
!    system time is beleived to be due to minor page faults as the
!    code attempts to grab a peice of memory that is not yet
!    initialized.
call output%compute(inputs=inputs, k=k, row=row)
```

After this is the post-processing of the computed y array. The first and kth columns and the first and kth rows are summed. Then the summary output file begins to be created. The size in memory of each array is formatted with pbytes, then the sums are added to the file. After this the summary file is closed and saved. If print_array is toggled in the input parameters, the output array is also saved to an output file. After this, the program terminates.

Listing 10: Object Initialization and Computation

```fortran
sum_first_col = sum(output%y(:, 1))
sum_kth_col = sum(output%y(:, k))

sum_first_row = sum(output%y(1, :))
sum_kth_row = sum(output%y(k, :))

output_summary_dst = join_path(output_dir, 'summary.txt')

open(newunit=output_summary_unit, file=output_summary_dst, status='
   replace', action='write')
write(output_summary_unit, '(A, A)') 'x: ', pbytes(inputs%x_bytes)
write(output_summary_unit, '(A, A)') 'b: ', pbytes(inputs%b_bytes)
write(output_summary_unit, '(A, A)') 'y: ', pbytes(output%y_bytes)
write(output_summary_unit, '(A)') ''

write(output_summary_unit,'(A,F12.0)') 'Sum of first row: ',
   sum_first_row
```

```fortran
write(output_summary_unit,'(A,F12.0)') 'Sum of kth row:   ',
   sum_kth_row
write(output_summary_unit,'(A,F12.0)') 'Sum of first col: ',
   sum_first_col
write(output_summary_unit,'(A,F12.0)') 'Sum of kth col:   ',
   sum_kth_col
write(output_summary_unit, '(A)') ''

close(output_summary_unit)

if (print_array) then
    output_array_dst = join_path(output_dir, 'array.txt')
    print *, "Saving: ", output_array_dst

    open(newunit=output_array_unit, file=output_array_dst, status='
       replace', action='write')
    do i = 1, output%n
        write(output_array_unit,'(*(i1))') (int(output%y(i,j)), j=1,
             output%n)
    end do
    close(output_array_unit)
end if
```

3 Results

3.1 Performance Measurments

Using the Python file, multiple cases were created to test the performance of the Fortran binary. These cases were run on Stampede3 with their timing and memory usage reported in time.txt and summary.txt. The cases that were run are shown below. It should be noted that case_d could only be run on Stampede3 due to the large memory requirements to store the matrices. Using TACC was also useful because I could access nodes that had up to 48 cores, which is much more than on my personal machine.

The following sections show the results for each of the given cases. All cases were run on Stampede3 in an idev session with 48 available cores. What can be seen is that as the size of n increases, the size in memory of the x and y arrays also increases. By case D, the total amount of memory needed to run is on the order of 60 GiB. The timing for each case can be seen to increase as well, which makes sense because more operations are happening on the larger matrices. What can be seen is that the user and system time do not add up to the real time. This is because the binary is run with multiple threads, and the user and system time are summed over all threads. Moreover, there is a jump in the system time in case C, and it is significant for case D. For these two cases, the system time appears to be half of the user time, which means that around one third of the compute time is related to system operations. This can be reasoned as the result of memory allocations causing minor page faults. Page faulting causes is disruption in the operations, which reduces the performance.

This occurs during the first time an element is being filled with a number. Options such as increasing the page size and modifying the allocation method were looked into. However, the page limit is determined by root, which I do not have access to on TACC, and the allocation method used had the best performance for the real time. Overall, I believe that the code is fairly well optimized for real time as it can process 60 GiB of data in 1.7 seconds.

Listing 11: Case A (n=100, m=50, k=44)

```
x: 39.1 KiB
b: 200.0 B
y: 39.1 KiB

real 0.01
user 0.24
sys 0.04
Subprocess time: 3.83E-02
```

Listing 12: Case B (n=1000, m=50, k=88)

```
x: 3.8 MiB
b: 200.0 B
y: 3.8 MiB

real 0.01
user 0.32
sys 0.04
Subprocess time: 2.39E-02
```

Listing 13: Case C (n=25000, m=12345, k=12346)

```
x: 2.3 GiB
b: 48.2 KiB
y: 2.3 GiB

real 0.13
user 3.46
sys 1.87
Subprocess time: 1.40E-01
```

Listing 14: Case D (n=90000, m=12345, k=12346)

```
x: 30.2 GiB
b: 48.2 KiB
y: 30.2 GiB

real 1.66
user 40.88
sys 23.72
Subprocess time: 1.67E+00
```

3.2 Matrix Validation

For each of the test cases, the summation was used to validate that the correct structure of the matrix was formed. It should be noted that we expect all entries to be equal to 1.0 except for the first m rows of column k, in which case those entries are 2.0. By looking at case A, we can confirm that the matrix is creating the correct structure. For the Sum of the first row, we expect to get a result of n+1, which we get with 101. For the kth row, we expect to also get n+1 because the operated matrix line is in column k. The sum of the first column should be equal to n because the operated column is not k=1 but k=44. For the kth column, we expect to get a result that is n+m, which we get. By looking at the other 3 cases, we can confirm that these checks have passed.

Listing 15: Case A (n=100, m=50, k=44)

```
Sum of first row:        101.
Sum of kth row:          101.
Sum of first col:        100.
Sum of kth col:          150.
```

Listing 16: Case B (n=1000, m=50, k=88)

```
Sum of first row:       1001.
Sum of kth row:         1000.
Sum of first col:       1000.
Sum of kth col:         1050.
```

Listing 17: Case C (n=25000, m=12345, k=12346)

```
Sum of first row:      25001.
Sum of kth row:        25000.
Sum of first col:      25000.
Sum of kth col:        37345.
```

Listing 18: Case D (n=90000, m=12345, k=12346)

```
Sum of first row:      90001.
Sum of kth row:        90000.
Sum of first col:      90000.
Sum of kth col:       102345.
```

3.3 Matrix Visualization

Based on question 6, we were able to output the matrix to a file as seen in Listing 19. To recreate the given matrix, the setting row needed to be set. This is suboptimal as Fortran is a column-major code, but the case is only 5 by 5, leading to the slowdown being negligible.

Listing 19: Question 6 Row Matrix (n=5, m=3, k=2, row=True)

```
11111
22211
11111
11111
11111
```

3.4 Fail Checks

Each failed case was also tested. This could be used for continuous integration as an expected fail test. Each case failed as expected with the given output message relaying to the user the reason why the code failed to run. Listing 20 shows the console output from running a case with the first fail condition. In this case, n was set to -1, which does not make sense for the problem, and thus promptly ended the execution during the validation step.

Listing 20: Question 6 Row Matrix (n=5, m=3, k=2, row=True)

```
 Starting reading nml file
 Ending reading nml file
 Starting validation
Error(1) n must be positive, n=-1
STOP 1
```

4 Discussion

The Fortran code was developed with performance in mind. The Fortran code utilizes OpenMP to parallelize as much as possible in the code. The initialization of the arrays and the computation are both done in parallel. The system time can be attributed to the filling of the allocated arrays. The user time can be attributed to the computation of the output y array. The reason why the system time is so high is due to page faults during the initialization of the arrays in the objects. All cases successfully gave good results in that their columns and rows summed to the expected results. The code appears to be memory-limited as the largest case, which used around 60 GiB of RAM, was completed in 1.7 seconds using 48 cores. This means that the code scales well with access to large amounts of cores, but does not scale well with large arrays.

A Python Code

In this appendix, the Python code that orchestrated the compilation, case creation, and execution of the Fortran code. In Listnig 21 we can see the Python code in full. There are a few parts to this code: the case object, the compilation function, and the main function.

The case object is initialized using variables for a given case. No checks are done in this object because input validation is done in the Fortran code. This object has a method called make_nml that utilizes the f90nml Python package. This package translates a Python dictionary to a nml file. This allows the translation between the Python cases to the Fortran input parameters.

The compile function compiles the Fortran code into the binary. The Python code takes in an argument for whether the Fortran code should be compiled with OpenMP parallelism. In the future, this could be changed to an invocation of a Makefile. This would improve the compilation as it would allow for more sophisticated dependency mapping and prevent unnesary recompilation of modules. At the moment, whenever this function is called, the entire build and bin directories are cleaned, and the entire Fortran code is recompiled.

Finally, the main function executes the entire code discussed in this document. First, if toggled, the Fortran code is compiled using the compile function. Then the cases are created by creating directories based on the case name. Then the nml is placed into their respective directories. After this, the binary is called for each case, and its outputs are placed in the case directory.

Listing 21: Python Code

```python
import os
import f90nml
import time
import subprocess
import argparse

class Case:
    def __init__(self,
                 name: str, n: int, m: int, k: int,
                 row: bool=False, print_summary: bool=False,
                    print_array: bool=False):
        self.name: str = name
        self.n: int = n
        self.m: int = m
        self.k: int = k
        self.row: bool = row
        self.print_summary: bool = print_summary
        self.print_array: bool = print_array

    def make_nml(self, dst:str, group_name: str = "params"):
        print(f'\nSaving case={self.name} -> {dst}')
        nml = {
            "params": {
                "n": self.n,
                "m": self.m,
                "k": self.k,
                "row": self.row,
                "print_summary": self.print_summary,
                "print_array": self.print_array
            }
        }

        os.makedirs(os.path.dirname(dst), exist_ok=True)

        if os.path.exists(dst):
            os.remove(dst)

        f90nml.write(nml, dst)

def compile(clean: bool = True, enable_par:bool=False):
    print('Compiling FORTRAN code')

    gfortran_base = ['gfortran', '-O3', '-std=f2018'] # base gfortran compiling settings

    app = 'app'
    src = 'src'
    build = 'build'
    bin = 'bin'

    binary_name = 'hw-special'
    binary_path = os.path.join(bin, binary_name)

    os.makedirs(build, exist_ok=True)
    os.makedirs(bin, exist_ok=True)

    if clean: # then clear out files in build, src, and bin
        print(f'Cleaning: {build} {src} {bin}')
        for file in os.listdir(build):
           if file.endswith('.mod') or file.endswith('.o'):
               os.remove(os.path.join(build, file))

        for file in os.listdir(src):
            if file.endswith('.mod') or file.endswith('.o'):
                os.remove(os.path.join(src, file))

        if os.path.exists(binary_path):
            os.remove(binary_path)

    print('Compiling modules')
    # list of modules in /src/
    modules = ['input_arrays.f90', 'output_array.f90', 'mem_util.f90', 'path_util.f90']

    # compile modules into intermediate files
    o_list = []
    for mod in modules:
        in_path = os.path.join(src, mod)
        out_path = os.path.join(build, mod.replace('f90', 'o'))
        o_list.append(out_path)
        # print(f'    {in_path} -> {out_path}')
        if enable_par:
            r = subprocess.run(gfortran_base + ['-c', in_path, '-J', 'build', '-o', out_path, '-fopenmp'], check=True)
        else:
            r = subprocess.run(gfortran_base + ['-c', in_path, '-J', 'build', '-o', out_path], check=True)
        print(' '.join(r.args))

    # compile main app into intermediate file
    print('Compiling main app')
    main_in = 'main.f90'
    main_out = main_in.replace('f90', 'o')

    in_path = os.path.join(app, main_in)
    out_path = os.path.join(build, main_out)
    o_list = [out_path] + o_list

    if enable_par:
        r = subprocess.run(gfortran_base + ['-c', in_path, '-I',
           build, '-fopenmp', '-o', out_path], check=True)
    else:
        r = subprocess.run(gfortran_base + ['-c', in_path, '-I',
           build, '-o', out_path], check=True)
    print(' '.join(r.args))

    print('Compiling binary')
    if enable_par:
        r = subprocess.run(gfortran_base + o_list + ['-o',
           binary_path, '-fopenmp'], check=True)
    else:
        r = subprocess.run(gfortran_base + o_list + ['-o',
           binary_path], check=True)
    print(' '.join(r.args))

    print('Done compiling \n')
    return binary_path

def main():
    # toggle if to compile
    # usage python hw.py --no-compile -> set run_compilation=False
    p = argparse.ArgumentParser()
    p.add_argument('--no-compile', action='store_true')
    p.add_argument('--enable-par', action='store_true')

    args = p.parse_args()
    if not args.no_compile:
        if args.enable_par:
            print('Building with Parallel')
            binary_path = compile(enable_par=True)
        else:
            print('Building with Single')
            binary_path = compile(enable_par=False)
    else:
        binary_path = './bin/hw1'

    # hw cases
    case_a = Case(name="a", n=100,   m=50,    k=44,    row=False,
        print_summary=True, print_array=False)
    case_b = Case(name="b", n=1000,  m=50,    k=88,    row=False,
        print_summary=True, print_array=False)
    case_c = Case(name="c", n=25000, m=12345, k=12346, row=False,
        print_summary=True, print_array=False)
    case_d = Case(name="d", n=90000, m=12345, k=12346, row=False,
        print_summary=True, print_array=False)

    case_q6_row = Case(name="q6_row", n=5, m=3, k=2, row=True,
        print_summary=True, print_array=True)
    case_q6_col = Case(name="q6_col", n=5, m=3, k=2, row=False,
        print_summary=True, print_array=True)

    # expected fails
    fail_1 = Case(name="fail_1", n=-1,  m=10, k=10)
    fail_2 = Case(name="fail_2", n=10,  m=-1, k=10)
    fail_3 = Case(name="fail_3", n=10,  m=10, k=-1)
    fail_4 = Case(name="fail_4", n=10,  m=11, k=10)
    fail_5 = Case(name="fail_5", n=10,  m=10, k=11)

    cases = [
        case_a,
        case_b,
        case_c,
        case_d, # this case requires a lot of ram

        case_q6_row,
        case_q6_col,

        # expected fails
        fail_1,
        fail_2,
        fail_3,
        fail_4,
        fail_5,
        ]

    for case in cases:
        directory = f"./cases/{case.name}/"
        params_dst = os.path.join(directory, "params.nml")

        case.make_nml(dst=params_dst)
        time_file = os.path.join(directory, "time.txt")

        # Run with enhanced timing and append to summary.txt
        cmd = f'/usr/bin/time -p -o {time_file} -- {binary_path} {
            params_dst} {directory}'

        print(f"Running: {cmd}\n")
        start = time.perf_counter() # start time
        os.system(cmd)
        end = time.perf_counter() # end time
        elapsed = end - start # elapsed time for subprocess

        # append the timed subprocess execution to the time.txt file
        with open(time_file, 'a') as f:
            f.write(f'\nSubprocess time: {elapsed:.2E} [s]')


if __name__ == "__main__":
    main()
```