Neglected FORTRAN

Better use of f90 in scientific research

Drew McCormack

TABLE OF CONTENTS

1. Course Outline

1.1 Day 1: Basic f90 Constructs

This is the stuff you would find in a book on f90.

We will emphasize the practical aspects, i.e., how you can use the stuff in the books in a useful way.

What is worth using, and what isn't?

What are the reasons for certain constructs?

1.2 Day 2: Abstract Data Types (ADTs)

This is about high-level modular programming.

It is really how f90 was intended to be used.

We will take what we learned on day 1, and apply it in all its glory.

We will also undertake a group project to hopefully show the strengths of the modular programming model.

1.3 Day 3: Parallelism

A crash course in Message Passing Interface (MPI) and OpenMP.

The advantages and disadvantages of each.

The most useful constructs in each.

2. Is f90 an extension of f77?

2.1 A language unto itself

f90 is an abused language. People take little bits of it to improve their f77 codes, but rarely use it how it was intended.

f90 is not just allocatable arrays; it is a modern, powerful language, which when properly used, bears little resemblance to f77.

f90 builds on the strengths of f77 — in particular, high performance for numerical code — by adding tools garnered from other languages to simplify programming, and make programs more structured.

It also adds powerful array facilities unmatched by any other language.

Figure 1 Examples of f77 and f90. Any similarity?

An example of f77

```
c      if(arg.gt.1.) arg=1.
c      if(arg.lt.-1.) arg=-1.
       a=acos(arg)
c      write(6,100) r4,r5,r6,s2,s3,arg,a
       arg=(r1**2+r5**2-r3**2)/(2.*r1*r5)
       if(arg.gt.1.) arg=1.
       if(arg.lt.-1.) arg=-1.
       b=acos(arg)
       arg=(r1**2+r4**2-r2**2)/(2.*r1*r4)
       if(arg.gt.1.) arg=1.
       if(arg.lt.-1.) arg=-1.
       g=acos(arg)
       s2=exp(-gam*(r6i-abs(r2-r3)))
       s3=exp(gam*(r6i-r2-r3))
       r6=r6i+epp*s2-epp*s3
       arg=(r2**2+r3**2-r6**2)/(2.*r2*r3)
c      if(arg.gt.1.) arg=1.
c      if(arg.lt.-1.) arg=-1.
       d=acos(arg)
c      write(6,100) r2,r3,r6,s2,s3,arg,d
  100  format(1x,7e12.4)
       a0=pi*(1.-0.294*exp(-0.6*(r2-1.951)**2)*exp(-0.6*(r1-4.230)**2))
      &*(1.-0.294*exp(-0.6*(r3-1.951)**2)*exp(-0.6*(r1-4.230)**2))
       r6a=sqrt(r4**2+r5**2+2.*r4*r5*cos(a-a0))
       sinaa0=sin(a-a0)
       sina0=sin(a0)
       sina=sin(a)
       sind=sin(d)
```

An example of f90

```
!-------------------------------------------------------------
! Description of CoherentStateFunc here.
!-------------------------------------------------------------
```

```
module CoherentStateFuncClass
implicit none

!-------------------------
! Description of CoherentStateFunc variables.
!-------------------------
type CoherentStateFunc
  private
  ! Variable definitions here
end type

!-------------------------
! Interface statements here.
!-------------------------
interface new
  module procedure newCoherentState
end interface

!-----------------------------------------------------------
! CoherentStateFunc methods.
!-----------------------------------------------------------
contains

  !-------------------------
  ! Constructor
  !-------------------------
  subroutine newCoherentState(this,coordCenter,lowEnergy,highEnergy, &
                              movingPositiveDirection)
  implicit none
  type (CoherentStateFunc)             :: this
  real,intent(in)                      :: coordCenter
  real,intent(in)                      :: lowEnergy,highEnergy
  logical,intent(in)                   :: movingPositiveDirection

  write(6,*)'constructing coherent state func'

  end subroutine newCoherentState

end module
```

3. Form

3.1 Fixed Form

The most obvious change in f90 is form. Though this is probably the least important change, it can make code easier to read, and frees the programmer from worrying about column counts. This can, in some cases, even prevent bugs.

Figure 2 Two very similar pieces of f77. Where could the bug be?

```
      implicit real*8 (a-h,o-z)
      a = b + c * (ax + bx)**2 – ax*bx + (a–c)*a+bx
```

```
      implicit real*8 (a-h,o-z)
      a = b + c * (ax + bx)**2 – ax*bx + (a–c)*a + bx
```

3.2 Free Form

Generally, free form is simpler and neater. It better facilitates indentation.

In free form, you use comments beginning with an exclamation mark (i.e. “!”). They can begin anywhere on the line. You can also add them at the end of a line.

The free form continuation character is “&”. This is put at the end of the continuing line, rather than at the beginning of the next.

Figure 3Comparison of free- and fixed form

```
C
C     we are doing a do loop here, but this
C     comment is not nicely indented, and the
C     continuation character is also out of whack.
C
              do i = 1,n
                a = b + c(i) * (ax + bx)**2 +
```

```
     $              (a–c(i))*a+bx + ax**3 – ax*bx
                enddo
```

```
                !
                ! This comment follows the indentation
                ! So does the continuation character
                !
                do i = 1,n
                  a = b + c(i) * (ax + bx)**2 + &
                     (a–c(i))*a+bx + &
                      ax**3 – ax*bx  ! you can also do
                                     ! this
                enddo
```

3.3 Indentation

Indentation helps make code easier to read.

There is nothing in f77 which prevents indentation, though it is neater with free form.

There are no hard and fast rules about indentation, but indenting by between 2-5 spaces, or with a tab, is good practice.

Figure 4 Indentation makes things more readable. Notice that spacing also helps.

```
! Starting loop
do i = 1,n
a = b + c(i) * (ax + bx)**2 + (a–c(i))*a+bx + &
ax**3 – ax*bx
if ( c(i) < 0 ) then
c(i) = 1
else
c(i) = -1
! extra test
if ( a == b) then
a = b + 1
endif
```

```
endif
enddo
```

```
!
! Starting loop
!
do i = 1,n

  a = b + c(i) * (ax + bx)**2 + (a–c(i))*a+bx + &
      ax**3 – ax*bx

  if ( c(i) < 0 ) then
    c(i) = 1
  else
    c(i) = -1
    !
    ! extra test
    !
    if ( a == b) then
      a = b + 1
    endif
  endif

enddo
```

4. Variables

4.1 Implicit none

This is actually not a f90 construct: it's always been there. It's time to start using it.

DO NOT use implicit typing! Use only 'implicit none', and define all variables.

Figure 5 Reasons not to use implicit typing

You are forced to use a particular form of variable name. For example, 'point' is more to the point than 'ipoint'.
If you misspell a variable, you will generally not get a warning: you will have made a brand new variable.
If you use fixed format, and go past column 72, you will also create a new variable, and may not get a warning.
Using implicit typing doesn't fit well with modular programming, where you have user-defined types.
Someone reading your code can see exactly what everything is in the procedure header if you use explicit typing.
Allows the compiler to find your bugs!

Figure 6 This example, demonstrating why you should use implicit none, really happened. What is wrong with this code? Finding the answer cost the programmer days! Using implicit none, it would have been detected immediately.

```
implicit real*8(a-h,o-z)
...
complex*16 array(numElements)
...
call MPI_SEND(array,numElements,MPI_COMPLEX16, &
              procToSendTo,tag,MPI_COMM_WORLD,error)
```

4.2 Variable names

Try to avoid using meaningless, overly meager variable names. How long you make them is personal choice, but make them so that anyone reading the code knows what variables are.

Two popular naming conventions are underscored and mixed-case.

Figure 7 Examples of underscored and mixed-case variable names

Underscored variables	Mixed-case variable names
wave_function	waveFunction
point_num	pointNum
num_points	numPoints
a_long_variable_name	aLongVariableName
basis_func_num	basisFuncNum

4.3 Variable declarations

In f90 you can define variables like in f77, but I prefer using the "::" operator, because it makes things a bit easier to read.

Figure 8 One way to declare variables

```
real,allocatable                  :: array(:)
real,allocatable,dimension(:)     :: array2, array3
integer,intent(in)                :: num
type(FourierGrid),pointer         :: fourierGrid
type(WaveFunction)                :: waveFunc
```

5. Arrays

5.1 Array Sections

f90 has very powerful array facilities

You can make any array section you can dream of, but be mindful that not everything will be fast.

The ":" operator stipulates ranges in f90 arrays.

Figure 9 Examples and descriptions of array sections

Array Section	Description
`a(:)`	'a' is a rank one array, and we are using all of it.

`a(2:)`	Now we only have from element 2 up.
`a(:5)`	Now we only have up to element 5.
`a(2:5)`	Elements 2 to 5.
`a(2:6:2)`	Elements 2 to 6 in steps of 2. i.e. elements 2, 4, 6
`b(2,:)`	'b' is a rank 2 array, but this is only rank 1. We are taking all the elements in the second dimension of 'b' such that the first dimension has index 2.
`b(1:4,2:2)`	A rank 2 array comprised the matrix formed by allowing the first index to range from 1 to 4, and the second from 2 to 2.

5.2 Allocatable Arrays

Allocatable arrays are an important addition to f90, and one that most people know about.

Allocatable arrays allow the sizing of an array to be postponed until it is known.

Figure 10 Allocatable arrays

Declaration	`real, allocatable :: a(:,:)` `real, dimension(:,:), allocatable :: b`
Allocation	`allocate( a(5,5), b(4,10) )` `allocate( a(2:5,10:2:-2), b(0:jmax) )`
Deallocation	`deallocate(a,b)`
Testing status	`if ( allocated(a) ) then` `...`

5.3 Array Pointers

Pointers have been slower to catch on with the FORTRAN community

Pointers can be used in two basic ways: they can point to something already in existence, or they can be made to point to something new

The pointer assignment operator is "=>". So "a=>b" says "point a at b".

A pointer can point to anything the same type as itself, as long as that thing is a pointer itself, or is declared to have 'target' status.

Array-typed pointers can point to sub-arrays or full-arrays

When a pointer is used to point to something new, the 'allocate' statement is used on it in the same way as if it were an allocatable array

When a pointer points to something, it can be used as if it were a normal variable. Remember though, if you change the pointer variable, you are also changing the thing it points to!

Figure 11 Use of pointers

Declaration	`real,pointer :: a(:,:)`
	`integer,pointer :: b`
	`type(WaveFunc),pointer :: waveFunc`

Pointing at existing object	`real,pointer :: a(:,:)`
	`real :: b(5,10)`
	`a => b(2:4,3:7)`

Using allocate and deallocate	`real,pointer :: a(:,:)`
	`real,pointer :: b`
	
	`allocate( a(2:4,3:7) )`
	`deallocate( a )`
	
	`! doesn't have to be an array`
	`allocate( b )`
	`b = 5`
	`deallocate( b )`

Making pointer point to nothing	`nullify( a )`

5.4 Array Arithmetic

Arrays can be added, subtracted, multiplied etc just like numbers. In the case of an array, the arithmetic occurs in an element-wise manner. *The position of the element is what counts, not the index number.*

Scalars can also be included in array arithmetic. When a scalar is used, it applies to all elements of the array expression.

Figure 12 Examples of array arithmetic

Code	Description
`a = b`	Assign all elements of array a to the corresponding elements of b.
`a = b * c`	Multiply each element in b by the corresponding element in c, and put the result in the corresponding element in a.

`real :: a(10), b(10,10)`
`real :: d(10)`
`a = a + b(2,:) / d`
	Here we take the second dimension of b, corresponding to the first index equal to 2, and use that in the expression. Notice the arrays are still conforming, i.e., they are all rank one.

`waveFunc = waveFunc**2`	Square each element of waveFunc, and put the result back in waveFunc.

5.5 Assumed-Shape Arrays

If the interface between a subroutine and it's caller is explicit (see Section 6), a passed array can be assumed-shape.

This means that the array inside the procedure gets it's dimensions from the array that is passed to the procedure.

You can query the length of any dimension using the 'size' intrinsic function.

Be careful: The lower bound of any assumed-shape array defaults to one, irrespective of the bounds of the array passed in. Only the size of each dimension is passed. You can, however, set a lower or upper bound manually in the called procedure.

Figure 13 Example of an assumed-shape array

```
!
! in main program
!
real        :: a(-1:1)
call sub(a)
call anotherSub(a)

...

!
! subroutine with explicit interface to main
!
subroutine sub(aInSub)
real        :: aInSub(:)   ! indexes are 1,2,3

...

subroutine anotherSub(aInSub)
real        :: aInSub(-1:)  ! indexes are –1,0,1
```

5.6 Assumed-Size Arrays

Sometimes it is desirable to be able to change the rank of an array. This can be done using assumed-size arrays.

If the rank is n, you set the size of the first n-1 dimensions to whatever you want, and the last dimension gets an asterix.

You cannot get the size of an assumed-size array with the 'size' function. You must pass the size of the array in as a variable, or find another way to do it.

An explicit interface is NOT required to use an assumed-size array, though you can of course have one.

Figure 14 Example of an assumed-size array

```
!
! in main program
!
real        :: a(10,10,10)  ! here a is rank 3
integer     :: n = size(a)
call sub(a,n)

...

subroutine sub(aInSub,sizeOfA)
real       :: aInSub(2:4,5,5,5,*) ! now we have rank 5
integer    :: sizeOfA

...
```

5.7 Intrinsic Functions

f90 has many built-in functions which interact nicely with f90 arrays.

These functions can save you a lot of typing, and make for very readable, and less buggy code.

Take the case of doing a matrix-vector multiplication. The f77 way would be to write a double do loop, and introduce a variable to accumulate the results for each column.

In f90 this can be written:
vector = matmul(matrix, vector)

It is nearly impossible to make a mistake doing this without the compiler finding it. It is very easy to make a mistake with the do loops.

Be careful though: I don't recommend using the built-in functions in performance critical pieces of code, because they may not be that fast. If you need to do something only a few times, though, they are invaluable.

Figure 15 Some useful intrinsic functions, and how they can be used

matmul	`array(4,:) = matmul( matrix, array2d(4,:) )`
transpose	`array(:) = matmul( transpose(matrix), &` `matmul( matrix, array ) )`

sum	`real x`
	`x = sum( array(:,:,:) )`
	`x = sum( array * array )`

dot_product	`normWaveFunc = dot_product(waveFunc,waveFunc)`
conjg	`normWaveFunc = sum( conjg(waveFunc) * &` `waveFunc )`
abs	`normWaveFunc = sum( abs(waveFunc)**2 )`
size	`sizeArray = size( a )` `sizeSecondDimOnly = size( a, 2)`
lbound	`lowerBoundSecondDim = lbound(a,2)`
ubound	`upperBoundFirstDim = ubound(a,1)`

6. Interfaces

6.1 Implicit Interface

In f77, all interfaces are implicit. What this basically means is that each program unit (e.g. procedure) does not know anything about the other units in the program.

For example, I can pass a real array to a subroutine, and inside the subroutine I can pretend it's a complex array, and the compiler will not complain. The subroutine doesn't know anything about what it is being

passed to it, other than its memory address.

Unless you deliberately make the interface explicit, in f90 the default is also to have an implicit interface.

I recommend using implicit interfaces as little as possible. Your motto should be “Give the compiler as much information as possible, and it will do the work for you”.

6.2 Explicit Interface

In f90 you have the option of making an interface explicit. This can be achieved in one of two ways.

The obvious way is to use an ‘interface’ statement. This is placed in the ‘caller’ (i.e. the unit calling the subroutine) and tells the caller details about what the ‘callee’ expects.

The second, and in my view easier, way is to put all of your subroutines in modules. Then when you need to call one of the subroutines, you ‘use’ the module in the caller. This automatically makes the interface explicit.

6.3 Interface Statement

Interface statements actually have three uses. The first is making an explicit interface, as just explained.

The second is to facilitate the passing of procedural arguments to procedures (i.e. functions and subroutines). In this case the interface statement tells the callee what the interface of the procedure being passed in is.

The third is to allow procedure name overloading. In this case, an interface statement makes it possible to use the same name to call two different procedures. In order for this to work, the arguments of the two or more overloaded procedures must be different, so the compiler can determine which one should be called.

Figure 16 Uses of the interface statement

Forming an explicit interface	```! ! in caller ! real x integer y(10)
	
	interface
	  subroutine sub(a,b)
	  real :: a
	  integer :: b(:)
	  end subroutine sub
	end interface
	
	call sub(x,y)
	
	...
	
	subroutine sub(a,b)
	real :: a
	integer :: b(:)
	! body of sub here```

Passing a procedure as an argument	```! ! function to pass in ! real function PES(x) real x PES = x end function PES ... ! ! in caller !```

	call receivesFuncArgument(PES)
	
	...
	
	!
	! callee
	!
	subroutine receivesFuncArgument(func)
	real val
	interface
	  real function func(x)
	  real x
	  end function func
	end interface
	val = func(1.0) ! this calls PES
	end subroutine

6.4 Overloading procedure names

Something which is used a lot in modular programming is procedure overloading.

Basically, the same name is given to two or more functions or subroutines. The program distinguishes between them based on the signature of the call made.

For example, I could overload two subroutines to be called ‘setZCut’, with one taking an argument of type ‘real’ and the other taking an argument of type ‘integer’. If I call ‘setZCut’ passing an integer, then the subroutine taking an integer is called.

Obviously, you cannot overload two routines with exactly the same signature.

Figure 17 Overloading procedures

Outside a module	! caller
	!
	real realNum = 2.8
	integer i = 2
	interface setZCut
	  subroutine setZCutVersion1(r)
	  real r
	  end subroutine
	  subroutine setZCutVersion2(i)
	  integer i
	  end subroutine
	end interface
	
	call setZCut(realNum) ! calls version 1
	call setZCut(intNum) ! calls version 2

Inside a module	module modName
	
	interface setZCut
	  module procedure setZCutVersion1
	  module procedure setZCutVersion2
	end interface
	
	contains
	
	  subroutine setZCutVersion1(r)
	  real r
	  ! subroutine code here
	  end subroutine
	
	  subroutine setZCutVersion2(i)
	  integer i
	  ! subroutine code here
	  end subroutine
	
	end module
	
	!

```
! in caller
!
use modName
call setZCut(realNum) ! calls version 1
call setZCut(intNum)  ! calls version 2
```

6.5 Overloading built-in operators

It is also possible to overload built-in operators, such as '*' and '+'. In this way, you can make user-defined types behave more like built in types.

For example, you write a function which adds two wavefunctions together, and returns the resulting wavefunction. If you overload this with the name "operator(+)", whenever you call write "waveFunc1 + waveFunc2", the routine will be called.

Apart from built in operators such as "+" and "*", you can also overload assignment, i.e., "=", and comparison operators like "==".

6.6 Passing functions as arguments

As stated above, the 'interface' statement allows procedures to be passed as arguments to other procedures. Why would this be useful?

It allows for more generic programming. So, for example, you could write a routine which performs a finite difference derivative on any one-dimensional function. This routine can then be used to derivate any function with the signature in the interface block.

7. Modules

7.1 Uses of modules

Modules are an extremely useful addition to f90. For example, they can be used to hold data like common blocks, but much more elegantly and safely.

They can also hold procedures, and make the interfaces explicit automatically.

Modules also make it possible to use abstract data types (ADTs), which will be discussed on Day 2.

Note that a module should be compiled before any code that 'uses' that module. A 'make' file can help with this aspect.

Figure 18 Basic layout of a module

```
module nameOfModule
implicit none
! data is declared here, and interface statements
contains
  ! subroutines and functions are declared here
end module

!
! to use a module
!
program mainProgram
use nameOfModule    ! must be first
implicit none
end program
```

7.2 Modules versus Common Blocks

The simplest application of modules is as a replacement for common blocks.

There are two advantages of modules over common blocks:

Firstly, the declaration of data in the module is located in only one part of the code: the module. In a common block, the data is redeclared

every time the common block is used. If you want to change the common block, you have to change every instance of the common block. To update a module you only change the code in one place.

Modules offer 'strong typing'. A weakness of common blocks is that the number and type of data constructs are not checked, so if, for example, you accidentally leave out a variable, the compiler will not flag it as an error. This could lead to an annoying bug. In modules, however, the type of the data is fixed, and cannot be accidentally changed.

7.3 As 'container' for procedures

A module can also contain procedures. The procedures must be added after a 'contains' statement in the module.

The big advantage of putting procedures in modules is, as already discussed, that the interface to those procedures is automatically explicit.

The procedures in a module can only be called by other parts of the program that 'use' the module.

The ability to add both data and procedures to a module facilitates the modular programming techniques to be discussed on Day 2 of this course.

8. User-defined Types

8.1 Going beyond integer, real and complex

In f77, all data is either an integer, real, complex, or some other simple data type.

Arrays are the most advanced data type possible in f77.

In f90 we have seen f90 arrays, for which sub-arrays can be defined. These are a more powerful data type.

But f90 also allows you to define your own data types. This is achieved via the 'type' statement, and forms the basis for the modular programming to be discussed on Day 2.

A user-defined type is a data structure made up of simple types (e.g. real, integer) and other user-defined types, and can be treated much like the built in types. For example, you can pass a user-defined type to a subroutine.

To use a user-defined type, you first define it. This tells the compiler how the user-defined type is constructed.

Having defined the user-defined type, you can then create 'instances' (i.e. variables) of that type, just like you create instances of reals, integers etc.

The only thing you can't put in a user-defined type is an allocatable array. However, you can use a pointer to achieve exactly the same thing (see Section on pointers).

To access an entry in a user-defined type, you can use the "%" operator.

Figure 19 User-defined types

Declaring	`type typeName`
	`  ! variable declarations here`
	`end type`

Declaring a variable of user-defined type	`type (typeName)   :: typeVar`
Accessing a variable in the user-defined type	`typeVar%var`

Example	`!`
	`! Defining a wavefunction`
	`!`
	`type WaveFunction`
	`  real,pointer    :: coeffs(:,:,:)`
	`  type (BasisSet) :: basis`
	`  integer         :: totalSize`

	type (StopWatch):: sw end type ! ! making a wavefunction ! type (WaveFunction) :: wf wf%totalSize = 10 wf%coeffs(1,1,1) = 5.

9. Intent

9.1 Protecting data

Usually, when you call a procedure, you know how each argument should be treated. For example, you may know that one argument is for input, and another is for output from the procedure.

However, the compiler does not know this, and if for some reason a variable is 'abused', perhaps by someone not familiar with the code, the compiler will be unable to help.

Using the 'intent' keyword, you can tell the compiler (and any other programmers) how an argument should be used. If the variable is used inappropriately, the compiler will not allow it.

Figure 20 Stating the 'intent' of an argument

Preventing an argument from being changed in a procedure	subroutine sub(a) implicit none real,intent(in) :: a ...
Only using a variable for output.	subroutine sub(a) implicit none real,intent(out) :: a ...
Allowing both input and output (the default case)	subroutine sub(a) implicit none real,intent(inout) :: a ...

10. Data hiding

10.1 Public and Private

Something which is very important to modular programming is being able to hide data in one part of a program from another part of the program. This is tied to the concept of 'encapsulation'.

The idea is that if a piece of data is only accessible from a small part of the program, and if there is something wrong with that data or it needs to be modified, only a small and well-defined part of the program needs to be considered.

f90 allows for control of data 'visibility' through the 'private' and 'public' modifiers. These can be added to a variable in a module. 'public' is the default, and means that any part of the program using the module has access to the variable, and can change it. If 'private' is used, the variable is only accessible inside the module itself.

The 'private' modifier can also be added as the first line in a type-definition block, in which case all variables in the user-defined type are only accessible inside the module.

Figure 21 Use of 'private' and 'public'

Controlling visibility of module variables	module modName implicit none integer,private :: i ! only visible in ! module integer,public :: j ! visible wherever ! module is used

	integer :: k ! visible wherever ! module is used end module
Using private in a user-defined type	module modName ! var1 and var2 only visible in module type typeName private integer :: var1 real :: var2 end type end module

11. Control

11.1 Select Case

The 'select case' statement is sometimes neater than an 'if-then-elseif' construct.

Ranges, as used to stipulate sub-arrays, are also allowed. These must not overlap.

Figure 22 A 'select case' example

```
integer i

select case (i)
case (1)
  ! code here for i = 1 case
case (2:10)
  ! code here for if i = 2,10
case (11:)
  ! code here for if i is 11 or greater
case (:-1)
  ! code here for if i is less than -1
case default
  ! code here for none-of-the-above case
end select
```

12. I/O

12.1 Inline formats

Usually in f77 people use a numbered format statement to define input or output format.

An alternative is to put the format in the read/write statement itself.

Figure 23 Format in a write statement

```
write(6,'("the result is ",f12.6)')result
```

12.2 Creating a series of files

Sometimes you need to create a series of numbered files from within a program. This could occur, for example, in a parallel program.

You can do this by creating some numbered strings, and opening files with those names. To create a numbered string, you have to create an internal file. This is just a character variable which can be written to like a file.

If you use an internal file, make sure you use formatted output to write to it.

You can use the 'trim' intrinsic function to discard any blanks at the start or end.

Figure 24 Using an internal file to make a numbered string

```
character(len=20) :: fileName,numAsString
integer           :: i = 44  ! file number is 44
write(numAsString,'(i2)')i
fileName = 'firstPart'//trim(numAsString) ! concatenate
```

13. Day 1 Exercises

13.1 Arrays

i. In one line, evaluate the expectation value of a 2D potential, which is stored in an array 'pot(:,:)' for a normalized wavefunction stored in array 'waveFunc(:,:)'. Try to figure out two different ways to do this. (Hint: use intrinsic functions)

ii. Assume you have set up a 1D DVR representation for a wavefunction. The wavefunction is stored in an array 'waveFunc'. You have also set up 'kineticMatrix', the kinetic energy matrix in the FBR representation. Assume 'transMatrix' transforms from the DVR to the FBR representation. The transpose of 'transMatrix' transforms from the FBR to the DVR. Beginning from the DVR representation, in one line (with continuations if necessary), perform the kinetic energy operation on the wavefunction, and finish with 'waveFunc' in the DVR representation.

13.2 Modules and stuff

i. (a) Write a routine which squares the elements of a rank 1 real array. DO NOT pass the array length as an argument. Use an interface statement to make the interface explicit. Write a main program to test. (b) Now write a module and put the routine you have just written in the module. Remove the interface statement. 'Use' the module in the main program, and see if it works as before.

ii. (a) Write a routine which squares the elements of a real array of any rank (call it 'square'). Do not pass the number of elements as an argument. Instead, put the routine in a module (call it 'ArraySquarer'), introduce a module variable numElements for the number of elements, and set it in the main program before calling the routine. (b) Add another routine to the module called 'setup'. This routine should take a single argument, the number of elements in the array to be squared, and set the module variable with it. Make the module variable 'private'. Now instead of setting the module variable from within the main program, call 'setup' to do it.

iii. (a) Now introduce a user defined type in the module and put numElements in the user defined type. Change the name of the module to 'ArraySquarerClass' and call the user defined type 'ArraySquarer'. Change 'setup' and 'square' so that each takes an extra argument: put an argument of type ArraySquarer as the first argument in each case (call it 'this' in each case). (b) Now in your main program, define two different lengthed real arrays. Also define two ArraySquarer's, and call 'setup' for each with the appropriate arguments corresponding to the array lengths. Square the two arrays using the two different ArraySquarer's. Congrats, you have made your first Abstract Data Type (ADT)!

14. Modular Programming

14.1 Why bother writing 'good' code?

In the eighties, people involved in writing software for commercial interests discovered something disturbing: the 'procedural' approach to programming that they were using didn't scale very well. They found that once a program became bigger than about 100000 lines, updating the code became prohibitively time consuming. Basically, the bigger a 'procedural' program gets, the more complex it gets.

People started to look for programming models which would scale better, and this led to object-oriented programming, which is now the industry-wide standard.

Scientific programming is fast approaching the scenario faced by business a decade ago. Our computers are getting faster, and our codes larger and more complex. If you have ever worked on a f77 code larger than about 50000 lines, you will know the problem.

At the same time, the limiting factor in producing scientific results is being shifted from the computers, to the people programming them. As the computers get faster, the programmers become the bottleneck in the process.

Scientists thus need to adopt practices which will enable them to avoid wasting time. Extra effort to make things clear now could save countless hours later on. Code that is clear can be 'learnt' much quicker by a new researcher than spaghetti code. Code that is easily reusable can save time rewriting code to perform equivalent tasks.

With these goals in mind, today you will be introduced to a new way of programming. The intention is to make your programs more modular. Rather than being one huge interconnected unit, a program gets broken down into a number of much smaller loosely-coupled units. Learning to use this approach will require an initial investment of time, but ultimately it could save you weeks or even months of programming time.

14.2 Procedural approach

Until now, you have been using the procedural programming model, though perhaps you didn't realize it

In procedural programming, data and procedures are kept separate. You pass the data to the procedures, and the procedures modify the data, but the two are not part of the one entity.

The procedure implements an algorithm that performs a certain task on ANY conforming data set.

14.3 Abstract Data Type (ADT)

'Modular' programming makes use of abstract data types (ADT)

An ADT is an entity which has both data and the methods (procedures) that act on that data.

Data and methods are intimately linked in one entity.

We say that an ADT 'encapsulates' data and operations/methods in one entity.

Important to this model is 'data hiding'

Data hiding is ensuring that the data in an ADT is invisible to anything outside the ADT. Other ADTs, for example, should not be able to read or write the data in the ADT. It should be 'private' to the ADT to which it belongs.

Operations or methods are what are used to perform actions. You can send a 'message' to an ADT, requesting some information from it, or requesting it to perform some action. These 'messages' are simply procedure calls. For example, I could 'message' a wave function ADT requesting that it tell me how much memory it is using.

In f90, the module is what makes programming with ADT's possible. A module can encapsulate data and procedures in one unit, and you can use the 'private' and 'public' keywords to hide data.

14.4 Object-Oriented Programming

Object-oriented programming (OOP) is one step further than programming with ADTs.

OOP languages include C++, Delphi, python and Java.

OOP has become the most widely-used programming model in use today.

It has much to offer, particularly for large software systems, but is also more complicated than using ADTs, and probably less useful for scientific programming than it is for, say, commercial programming.

Like ADT-based programming, OOP is centered around encapsulating data and operations in one entity: the class.

However, OOP goes further, by enabling one class to inherit characteristics of other classes. OOP also allows classes to behave 'polymorphically'. That is, they behave differently in different situations.

OOP is beyond the scope of this course, though ADT programming is a good first step to understanding OOP.

14.5 Interface and Implementation

The 'interface' is what the user of an ADT sees. By 'user' I mean anything using the ADT. This can be another ADT, the main program, etc. It is not the person using the computer, though such a person may be writing the 'user' code.

In procedural programming, the data is part of the interface. For example, if I call an FFT, I have to pass it auxiliary arrays. The user has to know how long these arrays must be, their type, etc. The FFT routine is also forever locked in to using auxiliary arrays of that form. If it requires different auxiliary arrays later, all the auxiliary arrays in the program must be changed!

In ADT programming, you try as much as possible to make the interface consist only of procedures, not data. You should try to hide the ADT data, and only allow interaction with the ADT through the methods of the ADT (i.e. procedures).

Using ADTs is about reducing the interface. Changing the interface of a program unit — in any programming model — requires that other sections of the code also be modified. In ADT-based programming, whenever possible you avoid changing the interface. That way, you don't have to make changes to the rest of the program.

You can, however, expand the interface. For example, you can add new methods to the interface, because doing this does not require any changes to be made to the rest of the program. But the interface that already exists should not change.

In ADT programming the implementation of the ADT is independent of the interface. The interface is fixed, but the implementation can change. This is the whole idea: being able to change one part of the code without worrying about other parts. Having a fixed interface facilitates this. The rest of the program just uses the interface, which is unchanging; only the ADT itself knows about the implementation.

As long as the writer of an ADT sticks to the existing interface for that ADT, he/she can implement the ADT any way they like. They can use any data structures (e.g. arrays, linked lists) and algorithms (e.g. different libraries) they choose.

Effectively, an ADT is a 'black box'. But this doesn't mean you can't change what's in it. It only means you can change what's inside independently of what is outside.

A very important goal of ADT programming is "NO CODE DUPLICATION". Rather than copying a particular piece of code several times throughout a program (e.g. code to apply an FFT), code that is used often is encapsulated in units: ADTs. The code for the ADT only occurs once in the program, but you can make multiple 'instances' of the ADT. For example, the code for all FFTs would occur in one ADT, but you might need lots of different FFTs in the one program. Each time you need a new FFT, you simply 'create' an 'instance' of the ADT.

Why is avoiding code duplication good? For one, there is less chance

of programmer error: you get that piece of code right once and forget it. It also makes porting and maintaining the code easier. Porting just requires some very specific ADTs be modified, not a program-wide search. Maintaining the code is easier because if you need to change something, you only need to do this in one place, not throughout your program.

14.6 Connascence

The idea of separating interface from implementation and reducing the scope of the interface is captured well by the idea of 'connascence'.

'Connascence' is a measure of how much a program unit depends on other units, and how much they depend on it. You can ask yourself this to determine the 'connascence': "If I change this aspect of the code, how much of the rest of the code do I need to change?"

The aim is to make the connascence between units, in this case ADTs, as low as possible. This corresponds to reducing the interface.

The connascence between elements inside a single ADT will be very high; this is OK. But the connascence between different ADTs should be kept to a minimum. This can be achieved, for example, by data hiding.

15. ADTs in f90

15.1 Module as Unit of Encapsulation

The f90 module is capable of containing data, and procedures.

A module can also have user-defined types, which are data definitions.

It is possible to hide data in a module from users of the module using 'private' and 'public' keywords.

All of these qualities make the f90 module perfect for implementing an ADT. So how do you do it?

15.2 Data

The ADT data is defined in a user-defined type in the module.

Data hiding is enforced by using the keyword 'private' whenever possible. This can be applied in the user-defined type, or to other data or procedures in the module.

In Figure 25, using 'private' in the user-defined type makes all the data in that type only accessible in the module.

Note that the ADT is called 'MyADT'. The module containing the ADT is, by convention, called 'MyADTClass'.

Figure 25 How to enter data in an ADT in f90

```
module MyADTClass
implicit none

type MyADT
  private
  integer         :: i
  real            :: j
  real            :: array(10)
  complex,pointer :: arrayPointer(:)
end type

end module
```

15.3 Methods

The methods (procedures) for the ADT are also contained in the module, after a 'contains' statement.

These are just like ordinary procedures, except that, by convention, the ADT variable is always passed in first, and is called 'this'. 'this' is the instance of the ADT being 'messaged'.

If a method name is likely to be used in other ADTs as well (e.g.

getNumPoints), it should be overloaded using an interface statement.

You can force users to use the generic name (e.g. getNumPoints), rather than the ADT specific name (e.g. getNumPointsMyADT), by making the ADT procedure name private. This helps make a program more readable: rather than having 5 slightly different names for procedures each doing basically the same thing, you have one generic name.

Figure 26 An ADT with data and methods in f90

```
module MyADTClass
implicit none

type MyADT
  private
  integer :: i
end type

! overload the procedure 'set'
interface set
  module procedure setMyADT
end interface

! make setMyADT private
private :: setMyADT

contains

  !
  ! sets the ADT integer i to num
  !
  subroutine setMyADT(this,num)
  implicit none
  type (MyADT)       :: this
  integer,intent(in) :: num
  this%i = num
  end subroutine setMyADT

end module
```

15.4 Instantiating an ADT

Now that we have defined an ADT, we need to use it. You use an ADT by creating instances of it, or 'instantiating' it.

This is just like declaring a variable. Actually, when you declare a real number, you are really instantiating it; creating an instance of a real number.

Instantiating an ADT in f90 involves first 'using' the ADT module, and then simply declaring a variable of the ADT.

You can then call the procedures of the ADT with the instance you have created. You pass the instance as the first argument.

Figure 27 Instantiating MyADT from Figure 26

```
program MyProgram
use MyADTClass
implicit none
type (MyADT)  :: instanceOfMyADT
type (MyADT)  :: anotherInstance, andAnotherInstance

call set(instanceOfMyADT,5)  ! calling MyADT method

end program
```

15.5 Constructor

Nearly every ADT you write will need some setting up. A 'constructor' is a special procedure that every ADT has to 'setup' and initialize it's data.

By convention, the constructor is always called 'new'. In order that

every ADT can use this generic name, you need to use overloading.

A constructor takes any arguments it needs to setup the ADT. It usually stores values it will need later, and allocates memory.

The constructor of an ADT instance variable should be called before any other methods are called for that instance.

You can have as many constructors as you like. They could each take different parameters, for example.

One commonly used type of constructor is a 'copy constructor'. This takes a second ADT instance as an argument, and copies it into the messaged ADT instance.

Figure 28 Adding a constructor and a copy constructor to MyADT

```
module MyADTClass
implicit none

type MyADT
  private
  integer       :: i
  real,pointer  :: realArray(:)
end type

interface new
  module procedure newMyADT,newCopyMyADT
end interface

interface set
  module procedure setMyADT
end interface

private :: newMyADT,newCopyMyADT,setMyADT

contains

  ! constructor
  subroutine newMyADT(this,num)
  implicit none
  type (MyADT)       :: this
  integer,intent(in) :: num
  this%i = num
  allocate(this%realArray(10))
  end subroutine newMyADT

  ! copy constructor
  subroutine newCopyMyADT(this,toCopy)
  implicit none
  type (MyADT)       :: this,toCopy
  this%i = toCopy%i
  allocate(this%realArray(10))
  this%realArray(:) = toCopy%realArray(:)
  end subroutine newCopyMyADT

  subroutine setMyADT(this,num)
  implicit none
  type (MyADT)       :: this
  integer,intent(in) :: num
  this%i = num
  end subroutine setMyADT

  ...

end module
```

15.6 Destructor

A destructor performs the opposite task to a constructor: it cleans up when the instance is not needed anymore.

A destructor usually deallocates any memory that is associated with a particular instance. The constructor/destructor approach makes memory management easier, because all allocation and deallocation of memory is localized in these procedures.

For every call to a constructor in a program, there should be a corresponding call to a destructor. Otherwise there could be a memory leak, ie, memory could be allocated without being deallocated.

The destructor is called ‘delete’ by convention.

The destructor just takes the instance as an argument. It does not take any other arguments.

Figure 29 Adding a destructor to MyADT

```
module MyADTClass
implicit none

...

interface delete
  module procedure deleteMyADT
end interface

private :: deleteMyADT

contains

  ...

  ! destructor
  subroutine deleteMyADT(this)
  implicit none
  type (MyADT)       :: this
  deallocate(this%realArray)
  end subroutine deleteMyADT

  ...

end module
```

15.7 Getters and setters

Getters and setters are two types of commonly used methods known collectively as ‘accessor methods’, because they offer access to the data in an ADT (albeit through a strictly defined interface).

A ‘getter’ gets some information from an ADT instance, and a ‘setter’ sets some aspect of the ADT.

An example of a getter would be ‘getNumPoints’, which would return the number of points, say, on a Fourier grid.

An example of a setter would be ‘setTimeStep’, which would set the time step, say, in a propagator.

Often setters are not needed, because most of the setting can be done in the constructor. Getters are quite common though.

16. Relationships between ADTs

16.1 Composition: ‘From little things, big things grow’

One of the great strengths of the ADT approach to programming is being able to create hierarchies of interacting instances which mimic relationships in the ‘real world’.

‘Composition’ is about composing one ADT from other ADTs. In this way you can build up layers, beginning with the lowest fundamental ADTs which are made up only of simple types (e.g. real, integer, logical etc), and moving to high level ADTs which are composed mostly of the low-level ADTs.

For example, a low-level ADT would be an FFT. This just transforms a set of data according to some mathematical formulas. Higher than this is a FourierGrid, which is a grid on which to represent a function. The FourierGrid includes several FFTs to perform any transforms appropriate for its data. Higher again is a WaveFunction, which includes FourierGrids in order to represent itself.

Composition is very easy to implement. You simply include a variable of the low-level type in the definition of the high-level type. Of course, in the constructor of the high-level type you must also call the constructor of the low-level type, and equivalently for the destructor.

Figure 30 An example of composition

```
module HighLevelClass
use LowLevelClass
implicit none

type HighLevel
  type (LowLevel) :: low
end type

...

contains

  ! constructor
  subroutine newHighLevel(this)
  implicit none
  type (HighLevel)       :: this
  ! construct low level instance
  call new(this%low)
  end subroutine newHighLevel

  ! destructor
  subroutine deleteHighLevel(this)
  implicit none
  type (HighLevel)       :: this
  call delete(this%low)
  end subroutine deleteHighLevel

  ...

end module
```

16.2 Association

Composition is actually a specialized case of ‘association’.

Association is simply the idea that one class can be associated in some way with another. An Analyzor ADT in a wavepacket program, for example, may be associated with a particular WaveFunction which it is analyzing. I.e., There is a relationship between the two: the Analyzor analyzes the WaveFunction.

An association can either be implemented by composition, i.e., having one ADT as part of the other, or it can be implemented by having separate ADTs which may use ‘references’ to each other.

A ‘reference’ is simply a pointer in f90. The idea is that an ADT stores a pointer to any other ADT which it may need to send messages to. For example, the Analyzor would store a pointer to the particular WaveFunction that it is analyzing. That way it could request information from the WaveFunction. However, the WaveFunction would not need to store a reference to the Analyzor, because it doesn’t need to know anything about the Analyzor.

Usually any ADTs which a particular ADT needs references too are passed in to the constructor. The ADT then simply points pointers, which are part of it’s type-definition, to the passed-in ADTs. It does not need to construct or destruct the passed-in ADTs, because this should already be taken care of.

Figure 31 An example of association

```
module HighLevelClass
use LowLevelClass
implicit none

type HighLevel
  type (LowLevel),pointer :: low
end type
```

```
...

contains

  ! constructor
  subroutine newHighLevel(this,toReference)
  implicit none
  type (HighLevel)          :: this
  type (LowLevel),target    :: toReference
  this%low => toReference
  end subroutine newHighLevel

  ! destructor
  subroutine deleteHighLevel(this)
  implicit none
  type (HighLevel)       :: this
  nullify(low)  ! good practice, but not necessary
  end subroutine deleteHighLevel

  ! a method using the reference
  subroutine method(this)
  implicit none
  type (HighLevel)        :: this
  call lowLevelMethod(this%low) ! call method of low
  end subroutine method

end module
```

17. Design

17.1 Identifying ADTs

At first, modular programming with ADTs can take a bit of getting used to, but ultimately it is a very natural way to structure code.

One thing which may be difficult at first is simply identifying what the different ADTs should be. Often there are many different hierarchy designs which could be used, some better than others.

A simple approach to identifying ADTs is to use language. Write a description of the problem and how it will be solved. The nouns are good candidates for ADTs. It's that simple!

Figure 32 Using language to identify ADTs. The nouns underlined are good candidates for ADTs, though others do exist.

> We are going to write a wavepacket program for one degree of freedom. The wavepacket is represented on a fourier grid, and is initially a coherent state function. It is propagated by the split operator propagator, and analyzed by a flux analyzor. The potential is an Eckart function.

17.2 Identifying ADT methods

Having identified the ADTs, we need to assign them methods. Some of the methods are common to all ADTs, like the constructor and destructor. Others are used to allow limited access to the data: the accessor methods (i.e., getters and setters). These are relatively obvious.

Other good candidates can again be recognized through language. The verbs are potential methods for the ADTs. For example, 'propagate' and 'analyze'.

But basically you should think in terms of messages. What question or request would one ADT have for another? Such questions correspond to methods.

18. An Advanced Example

18.1 The 'MatrixTransform' ADT

The 'MatrixTransform' ADT is an ADT which I developed to fulfil a particular need in quantum wavepacket calculations. These calculations frequently require that a particular dimension of a multidimensional array be transformed by a matrix multiplication, for all possible values of the remaining dimensions. Mathematically, it might look like this

$$x'_{ij'k} = \sum_j A_{j'j} x_{ijk} \quad \text{for all } i \text{ and } k.$$

Here, we are clearly treating a rank 3 array, *x*, and transforming the 2nd dimension by matrix *A*.

Because this occurs so often in wavepacket calculations, I decided it was wise to make this an ADT, to reduce code duplication and make programming easier in the long run. I thus created an ADT called 'MatrixTransform', which transforms any dimension of any complex array using a real matrix.

The creation of the ADT was not that easy, because I had to make sure it could take arrays of different rank, and this led to a lot of code which looks very similar. Unfortunately, this can't be avoided in f90, but at least once someone has done the initial work, the ADT can be used over and over by anyone needing its functionality.

The implementation of MatrixTransform involves representing an array of any rank as a rank 3 array. This is implemented via a second ADT called 'DimensionDescriptor', which describes a dimension in a particular array in terms of its rank 3 equivalent. The array passed in for transforming is first converted to a rank 3 array, with the 2nd dimension the dimension to be transformed. The transform then occurs for the rank 3 array. The reason for doing this is to avoid writing the whole algorithm for every possible rank of array, and every possible dimension to be transformed. You could say that the array is transformed to a 'generic' form first.

Below you can find the source code for 'MatrixTransform'. First look at how the ADT is used. If you wish, you can go on to try to understand the ADT implementation, but it is quite extensive. (Note that there are a few other modules included, which are used by the MatrixTransform ADT.)Try to keep in mind that although writing the ADT could be quite a bit of work initially, you will save lots of time every time you use it. If you have many such matrix transforms in your code, you will presumably benefit greatly from using the ADT, as will anyone else who uses it.

Figure 33 The 'MatrixTransform' ADT and how to use it.

```
use MatrixTransformClass
type (MatrixTransform) :: mt
integer,parameter      :: n=10
real(8)                :: matrix(n,n)
complex(8)             :: coeffs(1000,n,2000)

! setup matrix and coeffs array here
matrix = 1.1d0
coeffs = 1.d0

! setup MatrixTransform
call new(mt,matrix)

! Apply transform to second dimension of coeffs array
call PerformForwardTransform(mt,coeffs,2)

! Transform coeffs back again, assuming matrix is unitary
call PerformReverseTransform(mt,coeffs,2)

call delete(mt)

...
```

```
!-----------------------------------------------------------------------
! Module with the kinds of numbers used in the program. This keeps
! decisions about how large numbers (eg reals) should be localized
! in one module.
!-----------------------------------------------------------------------
```

```
module NumberKinds
implicit none
integer,parameter   :: KIND=8
end module

!-----------------------------------------------------------------
! This class describes an array dimension in terms of a reduced
! representation. The reduced representation is a rank 3 array, where
! the second dimension is the dimension we are describing. All arrays
! can be represented in this form, as far as memory layout
! is concerned.
! So, for example, take a rank 5 array called a(l,m,n,o,p).
! The first dimension would be represented as the reduced array
! red(1,l,m*n*o*p).
! The second dimension as red(l,m,n*o*p).
! The fifth dimension as red(l*m*n*o,p,1).
! The purpose of reducing arrays in this way is to make algorithms
! more generic. Instead of writing a different algorithm to treat
! all possible array sizes, and all dimensions, we simply write
! one algorithm which treats the reduced case.
!-----------------------------------------------------------------
module DimensionDescriptorClass
use NumberKinds
implicit none

!--------------------------------
! Fields
!-----------------
! size1       The size of the first dimension in
!             the reduced representation.
! size2       The size of the second dimension in
!             the reduced representation.
! size3       The size of the third dimension in
!             the reduced representation.
!--------------------------------
type DimensionDescriptor
  integer                :: size1,size2,size3
end type

!--------------------------------
! Private members
!--------------------------------
private :: NewDimensionDescriptor, DeleteDimensionDescriptor

!--------------------------------
! Interfaces
!--------------------------------
interface new
  module procedure NewDimensionDescriptor
  module procedure NewDimensionDescriptor1DC, NewDimensionDescriptor1DR
  module procedure NewDimensionDescriptor2DC, NewDimensionDescriptor2DR
  module procedure NewDimensionDescriptor3DC, NewDimensionDescriptor3DR
  module procedure NewDimensionDescriptor4DC, NewDimensionDescriptor4DR
  module procedure NewDimensionDescriptor5DC, NewDimensionDescriptor5DR
  module procedure NewDimensionDescriptor6DC, NewDimensionDescriptor6DR
end interface

interface delete
  module procedure DeleteDimensionDescriptor
end interface

!--------------------------------
! member functions
!--------------------------------
contains

  !-----------------------------------------------------------------
  ! Constructor
  ! This is a general constructor, taking an array of dimension sizes,
  ! and the dimension to describe.
  !-----------------------------------------------------------------
  subroutine NewDimensionDescriptor(this,sizes,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)   :: this
  integer,intent(in)                         :: sizes(:)
  integer,intent(in)                         :: dim
  integer                                    :: d

  !
  ! tests
  !
  if ( dim < 1 .or. dim > size(sizes) ) then
    write(6,*)'bad dim in DimensionDescriptor constructor'
    stop
  endif

  do d = 1,size(sizes)
    if ( sizes(d) < 1 ) then
      write(6,*)'bad dimension sizes in DimensionDescriptor
constructor'
      stop
    endif
```

```
  enddo

  !
  ! Set reduced dimensions
  !
  this%size1 = 1
  this%size2 = sizes(dim)
  this%size3 = 1

  do d = 1,dim-1
    this%size1 = this%size1 * sizes(d)
  enddo

  do d = size(sizes),dim+1,-1
    this%size3 = this%size3 * sizes(d)
  enddo

  end subroutine NewDimensionDescriptor

  !---------------------------------------------------------------
  ! Constructors
  ! The following constructors take the arrays for which the
  ! descriptor is to be constructed as an argument.
  !---------------------------------------------------------------
  subroutine NewDimensionDescriptor1DC(this,array)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:)

  call new(this,(/size(array)/),1)

  end subroutine NewDimensionDescriptor1DC

  !-------------------------------

  subroutine NewDimensionDescriptor2DC(this,array,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:,:)
  integer,intent(in)                          :: dim

  call new(this,(/size(array,1),size(array,2)/),dim)

  end subroutine NewDimensionDescriptor2DC

  !-------------------------------
```

```
  subroutine NewDimensionDescriptor3DC(this,array,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:,:,:)
  integer,intent(in)                          :: dim

  call new(this,(/size(array,1),size(array,2),size(array,3)/),dim)

  end subroutine NewDimensionDescriptor3DC

  !-------------------------------

  subroutine NewDimensionDescriptor4DC(this,array,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:,:,:,:)
  integer,intent(in)                          :: dim

  call new(this,(/size(array,1),size(array,2), &
                  size(array,3),size(array,4)/), dim)

  end subroutine NewDimensionDescriptor4DC

  !-------------------------------

  subroutine NewDimensionDescriptor5DC(this,array,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:,:,:,:,:)
  integer,intent(in)                          :: dim

  call new(this,(/size(array,1),size(array,2), &
                  size(array,3),size(array,4),size(array,5)/), dim)

  end subroutine NewDimensionDescriptor5DC

  !-------------------------------

  subroutine NewDimensionDescriptor6DC(this,array,dim)
  implicit none
  type (DimensionDescriptor),intent(inout)    :: this
  complex(KIND),intent(in)                    :: array(:,:,:,:,:,:)
  integer,intent(in)                          :: dim

  call new(this,(/size(array,1),size(array,2), &
```

```
                size(array,3),size(array,4), &
                size(array,5),size(array,6)/), dim)

end subroutine NewDimensionDescriptor6DC

!-------------------------------

subroutine NewDimensionDescriptor1DR(this,array)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:)

call new(this,(/size(array)/),1)

end subroutine NewDimensionDescriptor1DR

!-------------------------------

subroutine NewDimensionDescriptor2DR(this,array,dim)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:,:)
integer,intent(in)                          :: dim

call new(this,(/size(array,1),size(array,2)/),dim)

end subroutine NewDimensionDescriptor2DR

!-------------------------------

subroutine NewDimensionDescriptor3DR(this,array,dim)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:,:,:)
integer,intent(in)                          :: dim

call new(this,(/size(array,1),size(array,2),size(array,3)/),dim)

end subroutine NewDimensionDescriptor3DR

!-------------------------------

subroutine NewDimensionDescriptor4DR(this,array,dim)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:,:,:,:)
integer,intent(in)                          :: dim

call new(this,(/size(array,1),size(array,2), &
                size(array,3),size(array,4)/), dim)

end subroutine NewDimensionDescriptor4DR

!-------------------------------

subroutine NewDimensionDescriptor5DR(this,array,dim)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:,:,:,:,:)
integer,intent(in)                          :: dim

call new(this,(/size(array,1),size(array,2), &
                size(array,3),size(array,4),size(array,5)/), dim)

end subroutine NewDimensionDescriptor5DR

!-------------------------------

subroutine NewDimensionDescriptor6DR(this,array,dim)
implicit none
type (DimensionDescriptor),intent(inout)    :: this
real(KIND),intent(in)                       :: array(:,:,:,:,:,:)
integer,intent(in)                          :: dim

call new(this,(/size(array,1),size(array,2), &
                size(array,3),size(array,4), &
                size(array,5),size(array,6)/), dim)

end subroutine NewDimensionDescriptor6DR

!---------------------------------------------------------------
! Destructor
! Currently does nothing.
!---------------------------------------------------------------
subroutine DeleteDimensionDescriptor(this)
implicit none
type (DimensionDescriptor),intent(inout) :: this
end subroutine DeleteDimensionDescriptor

!---------------------------------------------------------------

end module DimensionDescriptorClass
```

```
!-----------------------------------------------------------------
! This represents a transform which involves matrix multiplying
! one dimension of a complex array, where the matrix is real.
! For example:
! if A is a rank 3 array, A(:,:,:), we could construct a matrix
! transform to transform, say, the second dimension. The transform
! transforms the second dimension M(:,:) . A(i,:,j)
! for all values of i and j.
! This transform class uses the dimension descriptor approach, where
! any dimension of any array is reduced to the middle dimension of
! a rank 3 array.
!-----------------------------------------------------------------
module MatrixTransformClass
use NumberKinds
use DimensionDescriptorClass
use OpenMPConfig
implicit none

!--------------------------------
! Fields
!
! forwardMatrix         The matrix for the forward transform.
! reverseMatrix         The matrix for the reverse transform. Assumes
!                       the reverse transform is simply the transpose
!                       of the forward transform matrix.
!--------------------------------
type MatrixTransform
  private
  real(KIND),pointer           :: forwardMatrix(:,:)
  real(KIND),pointer           :: reverseMatrix(:,:)
end type

!--------------------------------
! Enumerators
!--------------------------------
integer,parameter,private :: FORWARD = 0, REVERSE = 1

!--------------------------------
! Private members
!--------------------------------
private :: NewMatrixTransform, DeleteMatrixTransform
private :: PerformForwardMT1D,PerformForwardMT2D,PerformForwardMT3D
private :: PerformForwardMT4D,PerformForwardMT5D,PerformForwardMT6D
private :: PerformReverseMT1D,PerformReverseMT2D,PerformReverseMT3D
private :: PerformReverseMT4D,PerformReverseMT5D,PerformReverseMT6D
private :: PerformMatrixTransformInPlace

!--------------------------------
! Interfaces
!--------------------------------
interface new
  module procedure NewMatrixTransform
end interface

interface delete
  module procedure DeleteMatrixTransform
end interface

interface PerformForwardTransform
  module procedure PerformForwardMT1D
  module procedure PerformForwardMT2D
  module procedure PerformForwardMT3D
  module procedure PerformForwardMT4D
  module procedure PerformForwardMT5D
  module procedure PerformForwardMT6D
end interface

interface PerformReverseTransform
  module procedure PerformReverseMT1D
  module procedure PerformReverseMT2D
  module procedure PerformReverseMT3D
  module procedure PerformReverseMT4D
  module procedure PerformReverseMT5D
  module procedure PerformReverseMT6D
end interface

!--------------------------------
! member functions
!--------------------------------
contains

  !-----------------------------------------------------------------
  ! Constructor
  !-----------------------------------------------------------------
  subroutine NewMatrixTransform(this,matrix)
  implicit none
  type (MatrixTransform)          :: this
  real(KIND),intent(in)           :: matrix(:,:)
  integer                         :: s1,s2
```

```
s1 = size(matrix,1)
s2 = size(matrix,2)

nullify( this%forwardMatrix, this%reverseMatrix )
allocate( this%forwardMatrix(s1,s2) )
allocate( this%reverseMatrix(s2,s1) )

this%forwardMatrix = matrix
this%reverseMatrix = transpose(matrix)

end subroutine NewMatrixTransform

!-----------------------------------------------------------------
! Destructor
!-----------------------------------------------------------------
subroutine DeleteMatrixTransform(this)
implicit none
type (MatrixTransform),intent(inout) :: this

deallocate( this%forwardMatrix, this%reverseMatrix )

end subroutine DeleteMatrixTransform

!-----------------------------------------------------------------
! This scales the reverse transform matrix by a real
! factor.
!-----------------------------------------------------------------
subroutine ScaleReverseTransform(this,scaleFac)
implicit none
type (MatrixTransform),intent(inout) :: this
real(KIND),intent(in)                :: scaleFac

this%reverseMatrix = scaleFac * this%reverseMatrix

end subroutine ScaleReverseTransform

!-----------------------------------------------------------------
! Accessor for the number of rows in the forward transform.
!-----------------------------------------------------------------
integer function GetNumRowsForward(this) result (num)
implicit none
type (MatrixTransform),intent(in) :: this

num = size(this%forwardMatrix,1)

end function GetNumRowsForward
```

```
!-----------------------------------------------------------------
! Accessor for the number of columns in the forward transform.
!-----------------------------------------------------------------
integer function GetNumColsForward(this) result (num)
implicit none
type (MatrixTransform),intent(in) :: this

num = size(this%forwardMatrix,2)

end function GetNumColsForward

!-----------------------------------------------------------------
! Accessor for the number of rows in the reverse transform.
!-----------------------------------------------------------------
integer function GetNumRowsReverse(this) result (num)
implicit none
type (MatrixTransform),intent(in) :: this

num = size(this%reverseMatrix,1)

end function GetNumRowsReverse

!-----------------------------------------------------------------
! Accessor for the number of columns in the reverse transform.
!-----------------------------------------------------------------
integer function GetNumColsReverse(this) result (num)
type (MatrixTransform),intent(in) :: this

num = size(this%reverseMatrix,2)

end function GetNumColsReverse

!-----------------------------------------------------------------
! Accessor which returns the forward matrix.
!-----------------------------------------------------------------
function GetForwardMatrix(this) result (matrix)
type (MatrixTransform),intent(in) :: this
real(KIND),pointer                :: matrix(:,:)

matrix => this%forwardMatrix

end function GetForwardMatrix

!-----------------------------------------------------------------
! Accessor which returns the reverse matrix.
```

```
!---------------------------------------------------------------
function GetReverseMatrix(this) result (matrix)
type (MatrixTransform),intent(in) :: this
real(KIND),pointer                :: matrix(:,:)

matrix => this%reverseMatrix

end function GetReverseMatrix

!---------------------------------------------------------------
! The following routines are for specific ranked arrays. These
! routines reduce the dimension to be transformed under the
! dimension descriptor approach, and then call a general routine
! to actually do the tranformation.
!---------------------------------------------------------------
subroutine PerformForwardMT1D(this,coeffs,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:)
complex(KIND),optional                :: coeffsAfter(:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter

call new(dimDesc,coeffs)
if ( present(coeffsAfter) ) then
  call new(dimDescAfter,coeffsAfter)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,FORWARD)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
endif
call delete(dimDesc)

end subroutine PerformForwardMT1D

!-------------------

subroutine PerformForwardMT2D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:)
complex(KIND),optional                :: coeffsAfter(:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,FORWARD)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
endif
call delete(dimDesc)

end subroutine PerformForwardMT2D

!-------------------

subroutine PerformForwardMT3D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter, &
                              dimDescAfter,FORWARD)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
endif
call delete(dimDesc)

end subroutine PerformForwardMT3D

!-------------------

subroutine PerformForwardMT4D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
```

```
  if ( present(coeffsAfter) ) then
    call new(dimDescAfter,coeffsAfter,dim)
    call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter, &
                                dimDescAfter,FORWARD)
    call delete(dimDescAfter)
  else
    call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
  endif
  call delete(dimDesc)

  end subroutine PerformForwardMT4D

  !-------------------

  subroutine PerformForwardMT5D(this,coeffs,dim,coeffsAfter)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  complex(KIND)                         :: coeffs(:,:,:,:,:)
  complex(KIND),optional                :: coeffsAfter(:,:,:,:,:)
  type (DimensionDescriptor)            :: dimDesc,dimDescAfter
  integer,intent(in)                    :: dim

  call new(dimDesc,coeffs,dim)
  if ( present(coeffsAfter) ) then
    call new(dimDescAfter,coeffsAfter,dim)
    call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
      dimDescAfter,FORWARD)
    call delete(dimDescAfter)
  else
    call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
  endif
  call delete(dimDesc)

  end subroutine PerformForwardMT5D

  !-------------------

  subroutine PerformForwardMT6D(this,coeffs,dim,coeffsAfter)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  complex(KIND)                         :: coeffs(:,:,:,:,:,:)
  complex(KIND),optional                :: coeffsAfter(:,:,:,:,:,:)
  type (DimensionDescriptor)            :: dimDesc,dimDescAfter
  integer,intent(in)                    :: dim

  call new(dimDesc,coeffs,dim)
```

```
  if ( present(coeffsAfter) ) then
    call new(dimDescAfter,coeffsAfter,dim)
    call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
      dimDescAfter,FORWARD)
    call delete(dimDescAfter)
  else
    call PerformMatrixTransformInPlace(this,coeffs,dimDesc,FORWARD)
  endif
  call delete(dimDesc)

  end subroutine PerformForwardMT6D

  !------------------

  subroutine PerformReverseMT1D(this,coeffs,coeffsAfter)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  complex(KIND)                         :: coeffs(:)
  complex(KIND),optional                :: coeffsAfter(:)
  type (DimensionDescriptor)            :: dimDesc,dimDescAfter

  call new(dimDesc,coeffs)
  if ( present(coeffsAfter) ) then
    call new(dimDescAfter,coeffsAfter)
    call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
      dimDescAfter,REVERSE)
    call delete(dimDescAfter)
  else
    call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
  endif
  call delete(dimDesc)

  end subroutine PerformReverseMT1D

  !-------------------

  subroutine PerformReverseMT2D(this,coeffs,dim,coeffsAfter)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  complex(KIND)                         :: coeffs(:,:)
  complex(KIND),optional                :: coeffsAfter(:,:)
  type (DimensionDescriptor)            :: dimDesc,dimDescAfter
  integer,intent(in)                    :: dim

  call new(dimDesc,coeffs,dim)
  if ( present(coeffsAfter) ) then
```

```
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,REVERSE)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
endif
call delete(dimDesc)

end subroutine PerformReverseMT2D

!-------------------

subroutine PerformReverseMT3D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,REVERSE)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
endif
call delete(dimDesc)

end subroutine PerformReverseMT3D

!-------------------

subroutine PerformReverseMT4D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
```

```
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,REVERSE)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
endif
call delete(dimDesc)

end subroutine PerformReverseMT4D

!-------------------

subroutine PerformReverseMT5D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
  call new(dimDescAfter,coeffsAfter,dim)
  call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
    dimDescAfter,REVERSE)
  call delete(dimDescAfter)
else
  call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
endif
call delete(dimDesc)

end subroutine PerformReverseMT5D

!-------------------

subroutine PerformReverseMT6D(this,coeffs,dim,coeffsAfter)
implicit none
type (MatrixTransform),intent(in)     :: this
complex(KIND)                         :: coeffs(:,:,:,:,:,:)
complex(KIND),optional                :: coeffsAfter(:,:,:,:,:,:)
type (DimensionDescriptor)            :: dimDesc,dimDescAfter
integer,intent(in)                    :: dim

call new(dimDesc,coeffs,dim)
if ( present(coeffsAfter) ) then
```

```
    call new(dimDescAfter,coeffsAfter,dim)
    call PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter,&
      dimDescAfter,REVERSE)
    call delete(dimDescAfter)
  else
    call PerformMatrixTransformInPlace(this,coeffs,dimDesc,REVERSE)
  endif
  call delete(dimDesc)

  end subroutine PerformReverseMT6D

  !---------------------------------------------------------------
  ! Transforms a reduced dimension array, where the transform
  ! is not in-place.
  ! dimDesc is a dimension descriptor for the coeffs array.
  !---------------------------------------------------------------
  subroutine PerformMatrixTransform(this,coeffs,dimDesc,coeffsAfter, &
                                    dimDescAfter,direction)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  type (DimensionDescriptor),intent(in) :: dimDesc,dimDescAfter
  complex(KIND),intent(in)              :: &
    coeffs(dimDesc%size1,dimDesc%size2,*)
  complex(KIND),intent(inout)           :: &
    coeffsAfter(dimDescAfter%size1,dimDescAfter%size2,*)
  real(KIND),pointer                    :: transMatrix(:,:)
  integer,intent(in)                    :: direction
  integer                               :: &
    row,col,numCols,numRows,low,high
  integer                               :: s1,s2,s3,s1a,s2a,s3a,i,j,k

  select case (direction)
  case (FORWARD)
    transMatrix => this%forwardMatrix
  case (REVERSE)
    transMatrix => this%reverseMatrix
  case default
    write(6,*)'bad direction in Matrix Transform'
    stop
  end select

  numRows = size(transMatrix,1)
  numCols = size(transMatrix,2)

  s1 = dimDesc%size1
  s2 = dimDesc%size2
  s3 = dimDesc%size3

  s1a = dimDescAfter%size1
  s2a = dimDescAfter%size2
  s3a = dimDescAfter%size3

  !
  ! check dimensions for conformity
  !
  if ( s1 /= s1a .or. s3 /= s3a .or. &
       numRows /= s2a .or. numCols /= s2 ) then
    write(6,*)'non-conforming arrays in MatrixTransform'
    stop
  endif

  if ( s3 >= s1 .and. .not.InParallel() ) then

    !
    ! parallelize last dimension
    !
    !$OMP parallel do private(col,row,k)
    do k = 1,s3
      coeffsAfter(:,:,k) = (0.d0,0.d0)
      do col = 1,numCols
        do row = 1,numRows
          coeffsAfter(:s1,row,k) = coeffsAfter(:s1,row,k) + &
            transMatrix(row,col) * coeffs(:s1,col,k)
        enddo
      enddo
    enddo

  elseif ( .not.InParallel() ) then

    !
    ! parallelize first dimension
    !
    !$OMP parallel private(k,col,row,i,j,low,high)

    low = GlobalIndex(s1,MyThread(),1)
    high = low + GetLocalDim(s1,MyThread()) - 1

    do k = 1,s3
      do j = 1,s2a
        do i = low,high
          coeffsAfter(i,j,k) = (0.d0,0.d0)
        enddo
```

```
      enddo
      do col = 1,numCols
        do row = 1,numRows
          do i = low,high
            coeffsAfter(i,row,k) = coeffsAfter(i,row,k) + &
                transMatrix(row,col) * coeffs(i,col,k)
          enddo
        enddo
      enddo
    enddo

    !$OMP end parallel

  else  ! already in parallel region

    do k = 1,s3
      coeffsAfter(:,:s2a,k) = (0.d0,0.d0)
      do col = 1,numCols
        do row = 1,numRows
          coeffsAfter(:,row,k) = coeffsAfter(:,row,k) + &
            transMatrix(row,col) * coeffs(:,col,k)
        enddo
      enddo
    enddo

  endif

  end subroutine PerformMatrixTransform

  !-----------------------------------------------------------------
  ! Transforms a reduced dimension array, where the transform
  ! is in-place.
  ! dimDesc is a dimension descriptor for the coeffs array.
  !-----------------------------------------------------------------
  subroutine PerformMatrixTransformInPlace(this,coeffs, &
    dimDesc,direction)
  implicit none
  type (MatrixTransform),intent(in)     :: this
  type (DimensionDescriptor),intent(in) :: dimDesc
  complex(KIND),intent(inout)           ::
coeffs(dimDesc%size1,dimDesc%size2,*)
  complex(KIND),allocatable             :: coeffsTemp1(:,:,:)
  complex(KIND),allocatable             :: coeffsTemp2(:,:)
  real(KIND),pointer                    :: transMatrix(:,:)
  integer,intent(in)                    :: direction
  integer                               ::
```

```
row,col,numCols,numRows,low,high
  integer                                 :: s1,s2,s3,i,j,th,k

  select case (direction)
  case (FORWARD)
    transMatrix => this%forwardMatrix
  case (REVERSE)
    transMatrix => this%reverseMatrix
  case default
    write(6,*)'bad direction in Matrix Transform'
    stop
  end select

  numRows = size(transMatrix,1)
  numCols = size(transMatrix,2)

  s1 = dimDesc%size1
  s2 = dimDesc%size2
  s3 = dimDesc%size3

  !
  ! Checks
  !
  if ( numRows /= numCols ) then
    write(6,*)'for in-place matrix transform, matrix must be square'
    stop
  endif
  if ( numRows /= s2) then
    write(6,*)'transform and array non-conforming in MatrixTransform'
    stop
  endif

  if ( s3 >= s1 .and. .not.InParallel() ) then

    !
    ! parallelize last dimension
    !
    !$OMP parallel private(col,row,th,k)

    !$OMP single
    allocate( coeffsTemp1(s1,s2,0:NumThreads()-1) )
    !$OMP end single

    th = MyThread()

    !$OMP do
```

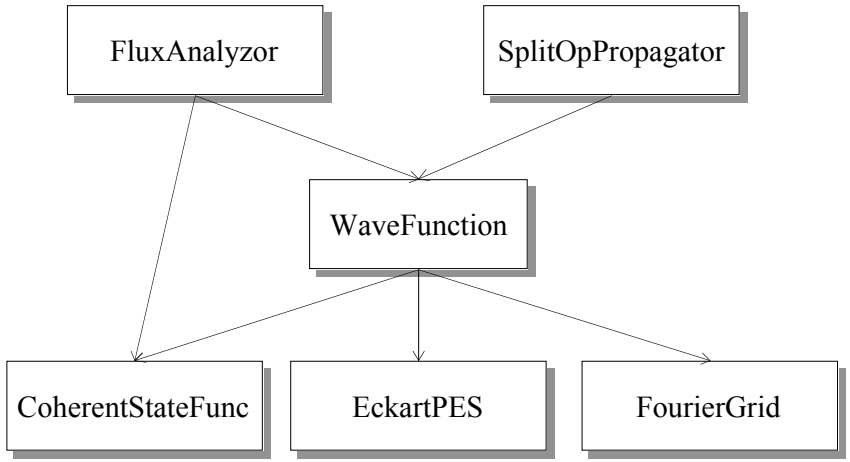
```
    do k = 1,s3
      coeffsTemp1(:,:,th) = (0.d0,0.d0)
      do col = 1,numCols
        do row = 1,numRows
          coeffsTemp1(:s1,row,th) = coeffsTemp1(:s1,row,th) + &
            transMatrix(row,col) * coeffs(:s1,col,k)
        enddo
      enddo
      coeffs(:s1,:s2,k) = coeffsTemp1(:s1,:s2,th)
    enddo

    !$OMP end parallel

    deallocate( coeffsTemp1 )


  elseif ( .not.InParallel() ) then

    !
    ! parallelize first dimension
    !
    allocate( coeffsTemp2(s1,s2) )

    !$OMP parallel private(k,col,row,i,j,low,high)

    low = GlobalIndex(s1,MyThread(),1)
    high = low + GetLocalDim(s1,MyThread()) - 1

    do k = 1,s3

      do j = 1,s2
        do i = low,high
          coeffsTemp2(i,j) = (0.d0,0.d0)
        enddo
      enddo

      do col = 1,numCols
        do row = 1,numRows
          do i = low,high
            coeffsTemp2(i,row) = coeffsTemp2(i,row) + &
                transMatrix(row,col) * coeffs(i,col,k)
          enddo
        enddo
      enddo

      do j = 1,s2
        do i = low,high
          coeffs(i,j,k) = coeffsTemp2(i,j)
        enddo
      enddo

    enddo

    !$OMP end parallel

    deallocate( coeffsTemp2 )

  else

    !
    ! already in a parallel region
    !
    allocate( coeffsTemp2(s1,s2) )
    do k = 1,s3
      coeffsTemp2(:,1:s2) = (0.d0,0.d0)
      do col = 1,numCols
        do row = 1,numRows
          coeffsTemp2(:,row) = coeffsTemp2(:,row) + &
            transMatrix(row,col) * coeffs(:,col,k)
        enddo
      enddo
      coeffs(:,:s2,k) = coeffsTemp2(:,:s2)
    enddo
    deallocate( coeffsTemp2 )

  endif

  end subroutine PerformMatrixTransformInPlace

  !---------------------------------------------------------------------

end module MatrixTransformClass
```

19. Day 2 Exercises

19.1 A group project

We are going to write a 1D wavepacket code to propagate an initially gaussian wavepacket on a 1D Eckart potential energy surface. We are going to analyze with a flux method to determine the probability of crossing the barrier.

I will supply the design document. This is simply the interfaces that everyone must conform to. Each person works independently on implementation. When we are all finished, we will put it together and try to get some results.



ADT	ADT methods
FourierGrid	subroutine new(real lowBound, real highBound, integer numPoints)
	subroutine delete()
	integer function getNumPoints()
	subroutine transToMomenta(complex array(:))
	subroutine transToCoords(complex array(:))
	integer function getCoordStep()
	real array(:) function getMomenta()
	real array(:) function getCoords()
	real function getMomentum(integer index)
	real function getCoord(integer index)
SplitOpPropagator	subroutine new(WaveFunction wfToProp, real timeStep)
	subroutine delete()
	subroutine propagate(integer numTimeSteps)
WaveFunction	subroutine new(FourierGrid fg, CoherentStateFunc initFunc, EckartPES pes, real mass)
	subroutine delete()
	subroutine performExpPotentialOp(real timeStep)
	subroutine performExpKineticOp(real timeStep)
	complex function getValueAtCoord(integer pointNum)
EckartPES	subroutine new(real barrierPosition, real barrierHeight, real widthParam)
	subroutine delete()
	real function getValueAtCoord(real coord)
CoherentStateFunc	subroutine new(real coordCenter, real lowEnergy, real highEnergy, real mass, logical

	movingPositiveDirection) subroutine delete() complex function getValueAtCoord(real coord) complex function getValueAtMomentum(real momentum)
FluxAnalyzor	subroutine new(WaveFunc wf, CoherentStateFunc initGaussFunc, integer cutPointNum, real analysisTimeStep, real mass) subroutine delete() subroutine storeCoeffs() subroutine analyzeAndReport(real lowEnergy, real highEnergy, integer numEnergies)

19.2 Linked list

A linked list, like an array, is a data structure, i.e., it holds data. The nice thing about a linked list is that it can grow and shrink dynamically, allocating and deallocating any memory it needs as it goes. An array, of course, has a fixed size (though see the next exercise for a way to make an array change size).

A linked list is a bit limited because you can't index it like an array. That is, you can't say a(5), for example. Instead you can only move through it in order, moving from one element to the next.

Try to write a linked list ADT to hold integers. You will need two user-defined types in the ADT: one is the list itself, and the other is one 'node' of the list. A list is made up of connected nodes, each node holding one integer.

In the node, you need a pointer to the next node in the list, as well as the integer it holds.

In the list type, you store a pointer to the first node, and a pointer to the current node.

The list is traversed by moving the current node from the first node to the next etc until the last node is reached (it's pointer doesn't point to anything; it is not 'associated').

The list ADT will need to have methods to move the current pointer to the first node; to advance the current pointer one node; to get the integer from the current node; to add a node before the first node; and to add a node before or after the current node.

This is quite a lot of work, but will improve your understanding of pointers and ADTs considerably.

19.3 Dynamic array

A dynamic array is one that can grow itself if it needs more memory.

A dynamic array is useful because you don't need to know how many elements you have in total before you start putting them in the array.

Write an ADT for a 1D dynamic array of reals.

You will need a user defined type holding an array, and an integer for how long the array is at present. You will need methods to set an element of the array; and return the occupied part of the array. For efficiency, you may add a method which enables the user to add multiple elements at a time.

The dynamic array should have two constructors: one takes the size at which the array should initially be constructed, and the other takes no argument and allocates the array with some default initial size (this size should be a parameter in the module).

The idea is that whenever someone attempts to set an element of the array which is outside the current range of indexes, the ADT should allocate a bigger array, copy the data into the bigger array, and deallocate the old small array. This is how it grows. You can decide by how much the array should grow each time, but perhaps a doubling of it's size is good. (You could put a parameter in the module which is the

factor by which the array increases each time. You could even have a constructor which takes this factor from the user.)

20. Parallelization

20.1 Philosophy

One way to make computers go faster is to increase the speed of their processors, memory, etc. This is analogous to having a one very competent person do a project. This is the kind of philosophy behind, for example, the CRAY C90, which uses vectorization to achieve its computational power.

Another approach is to use multiple processors to share the work. This is analogous to having many less competent people carry out the project. There are obvious inefficiencies here because the people have to coordinate their actions; the same is true of parallel computer systems.

Today we are going to talk about the second approach: parallelization.

20.2 Two schemes

Firstly, we will discuss MPI. This can be used on distributed memory machines, like the IBM SP3, and on shared memory machines, like the ORIGIN3000.

Lastly, we will discuss OpenMP, which is only for programming shared-memory machines.

21. Preliminaries of MPI

21.1 Introduction

Message Passing Interface (MPI) is simply a collection of subroutines (in C or FORTRAN) which enable processors to exchange data. You simply call these subroutines from inside your program, meaning you don't have to worry about the nitty-gritty details of memory addresses etc.

21.2 Advantages

Message Passing Interface (MPI) is very portable. It can be used on distributed- and shared-memory computers.

Programs written in MPI are also generally very efficient, though it is always possible to write slow programs if your parallelization strategy is bad.

21.3 Disadvantages

MPI is an all or nothing programming method. You can't really parallelize a part of your program; you must convert your whole program at once.

MPI is probably a bit more difficult to learn than, say, OpenMP, and perhaps more prone to bugs.

21.4 Distributed-Memory architecture

MPI generally runs on distributed-memory machines, though it can also run on shared-memory machines. In the latter case, the shared-memory computer just simulates a distributed-memory computer.

In distributed-memory machines, each processor has its own memory, which only it has access to.

If a processor wants some data that is in another processor's memory, it must send a message (e.g. using MPI) requesting that that data be sent. When it receives the data, it is put somewhere in its own memory, where it can be utilized.

21.5 Writing MPI programs

Writing MPI programs is very similar to writing serial (i.e., one processor) FORTRAN programs. You just have to remember that each processor is running it's own copy of the program.

All the copies are the same, but different processors usually take different paths through the code because of conditional statements (e.g., if then else endif). So even though the code run by each program is the same, each processor will usually do different things.

21.6 Header file

In order to call MPI routines in a FORTRAN program, you must include the 'mpif.h' file in the code unit making the call.

This file contains parameters used in calls to MPI routines.

Figure 34 Including the MPI header file

```
program myProgram
implicit none
include 'mpif.h'
...
```

21.7 Initialization and finalization

Before sending or receiving any data with MPI, it is necessary to call an initialization routine: MPI_INIT.

When all communications are complete, MPI_FINALIZE must be called.

If an error occurs in either case, the integer error flag argument is set by the routine before returning. If you wish, you can check the value of this to determine what happened. However, you can usually ignore this if you are not too worried about robustness of your code.

Figure 35 Calling MPI_INIT and MPI_FINALIZE

```
program myProgram
implicit none
include 'mpif.h'
integer error
call MPI_INIT(error)
...
call MPI_FINALIZE(error)
end program
```

21.8 Communicators

You can think of a communicator as a network linking certain processors.

You can make your own communicators, but this is not usually necessary. Most of the time you will want to communicate through the global communicator MPI_COMM_WORLD. This communicator connects all processors in the 'WORLD', in other words, all processors running your program.

You often need to pass the communicator you are using as an argument to MPI routines. Otherwise, you needn't worry too much about them.

To determine the number of processors in a communicator, you use MPI_COMM_SIZE.

Figure 36 Determining how many processors are in a communicator

```
include 'mpif.h'
integer num_procs,error
call MPI_COMM_SIZE(MPI_COMM_WORLD,num_procs,error)
! num_procs will have been set to the number of
! processors in MPI_COMM_WORLD
```

21.9 Processor identity

Each processor in a particular communicator has an identity, a number between 0 and *one less* than the number of processors in the communicator.

Each processor keeps the same identity throughout the run.

These numbers have no real meaning, they are just 'names'. Processor 0 is no different in MPI from Processor 100, though it is always possible to make a distinction between the processors in you own program, of course. (E.g. You could make processor 0 responsible for certain extra tasks.)

To get the number of a processor, you use the subroutine MPI_COMM_RANK, which stands for the rank of the processor in the particular communicator passed in.

Figure 37 Getting the number (rank) of a processor

```
include 'mpif.h'
integer my_id, error
call MPI_COMM_RANK(MPI_COMM_WORLD,my_id,error)
! after the call, my_id is the number
! of the processor in the MPI_COMM_WORLD communicator
```

22. Point-to-Point Communication

22.1 What is Point-to-Point Communication?

Point-to-Point (PtoP) Communication is communication involving only two processors.

Basically, one processor can send data to another processor, who receives it and places it in its own memory.

In PtoP communication, the two basic operations are to send data and to receive data. That's really all there is to it.

In theory you can accomplish everything in MPI with just PtoP communication, but it is sometimes more convenient to use collective communication, in which more than two processors are involved.

22.2 Sending data

If a processor wants to send some data to another processor, it can call the MPI_SEND routine.

This simply puts in a request to the computer hardware to send a certain amount of memory from an address in the processors memory, to another processor. The other processor must call the MPI_RECV routine in order to receive the sent data.

In an MPI_SEND call you need to pass the following: the variable you would like to send; the number of elements in the variable (1 if scalar; >1 if array); the type of the elements; the identity of the receiving processor; a tag for the message; the communicator; and the usual error flag.

The type of the elements should be an MPI parameter. Each FORTRAN type has a corresponding MPI type. For example, a real is represented by MPI_REAL. (MPI_REAL is simply a parameter in mpif.h)

The tag is simply an integer that you set yourself, which identifies the particular send call you are making. The corresponding receive call should use the same tag.

Requests for sends and receives are carried out in the order that they are made, so usually it is not necessary to use different tags for different calls. You still have to set the tag, but you could always use 0, for example.

Figure 38 Making a request to send data to another processor

```
include 'mpif.h'
integer error, tag, procToSendTo
real*8 array(numElements)
tag = 878
procToSendTo = 5
call MPI_SEND(array,numElements,MPI_REAL8, &
              procToSendTo,tag,MPI_COMM_WORLD,error)
```

22.3 Receiving data

There should be a corresponding receive call made for each send.

The receive routine takes similar parameters to the send routine.

The most significant difference between the MPI_SEND and MPI_RECV calls is the presence of a status array in MPI_RECV. Upon return, the status array contains various information about the call. You can usually ignore it, but sometimes it may have something of interest to you.

Note that you must supply a status array, and its size should be MPI_STATUS_SIZE.

Figure 39 Making a request to receive data from another processor

```
include 'mpif.h'
integer error, status(MPI_STATUS_SIZE)
integer tag, procToRecvFrom
real*8 recvArray(numElements)
tag = 878
procToRecvFrom = 1
call MPI_RECV(recvArray,numElements,MPI_REAL8, &
  procToRecvFrom,tag,MPI_COMM_WORLD,status,error)
```

When receiving, it is possible to stipulate that you don't mind what the tag of a message is, or what the source of the message is.

If you don't care about the tag, you simply pass MPI_ANY_TAG where the tag argument normally goes.

If you are not fussed by which processor is sending the message, use MPI_ANY_SOURCE.

Both the tag and the source of the message can still be determined after the call by examining the status array. The tag is in status(MPI_TAG) and the source is status(MPI_SOURCE).

23. Collective Communication

23.1 What is collective communication?

Collective Communication is communication involving more than two processors at a time.

Usually collective communication routines are just implemented in terms of PtoP calls, but your code will usually be simpler and neater if you use collective communication (though it may not be any faster).

23.2 Broadcast

Broadcasting, as the name suggests, is sending some data from one processor to all other processors in the communicator.

Figure 40 Diagrammatic representation of an MPI broadcast

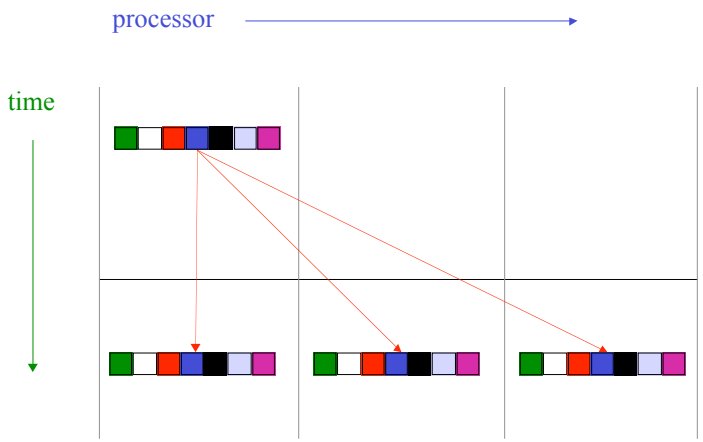


Figure 41 Broadcasting a real array from processor 0.

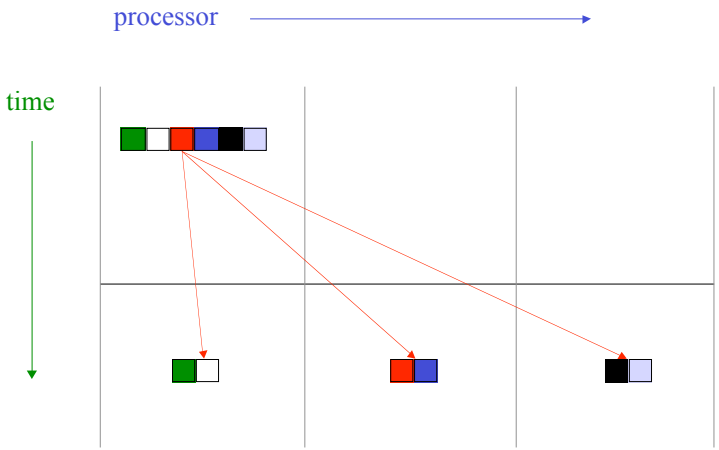
```
include 'mpif.h'
```

```
integer n = 10
integer error
integer bcastProc = 0    ! broadcasting processor
real*8 array(n)
call MPI_BCAST(array,size(array),MPI_REAL8,bcastProc, &
               MPI_COMM_WORLD,error)
```

23.3 Scatter

A scatter involves 'scattering' some data among the processors in the communicator. In other words, one processor divides an array of data up into as many portions as there are processors, and sends each processor one of the portions.

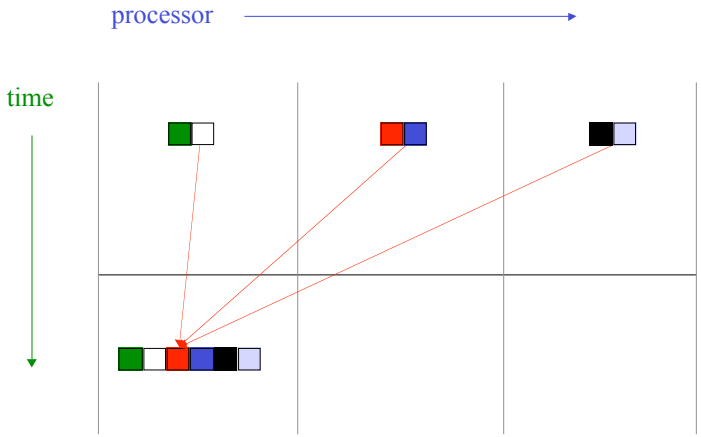
Figure 42 Diagrammatic representation of an MPI scatter



23.4 Gather

Gather is the opposite of scattering. Each processor has an array of data, and all of these are gathered and delivered to one processor.

Figure 43 Diagrammatic representation of an MPI gather



23.5 Send and receive

MPI_SENDRECV allows a processor to send data to one processor while simultaneously receiving from another.

23.6 Reduction

There are many types of reductions possible, but they all involve taking some data from each processor, combining it in some way (i.e. reducing it), and putting the result on one processor.

For example, you can add arrays on all processors, putting the resulting array on one processor.

You can also determine the maximum of all the data, putting the result on one processor.

Note that in each case, the reduction is performed on an element-wise basis. So determining the maximum for arrays does not return a scalar, it returns another array. Each element in the result is the maximum for that array position from the arrays on all processors.

Note also that with MPI_REDUCE only the so-called root processor gets the result. The other processors still have to pass all the arguments, including a result array, but anything not needed will be ignored.

Figure 44 Adding data on all processors, and determining the maximum of the data on all processors. In both cases, the result is only put on processor 2.

```
include 'mpif.h'
integer resultProc = 2  ! root processor
integer error, n = 10
real*8 array(n),resultArray(n)
call MPI_REDUCE(array,resultArray,n,MPI_REAL8, &
                MPI_SUM,resultProc,MPI_COMM_WORLD,error)
```

```
include 'mpif.h'
integer resultProc = 2  ! root processor
integer error, n = 10
real*8 array(n),resultArray(n)
call MPI_REDUCE(array,resultArray,n,MPI_REAL8, &
                MPI_MAX,resultProc,MPI_COMM_WORLD,error)
```

23.7 All-to-all

All-to-all involves each processor sending some distinct data to every other processor. In other words, if you are one processor, you must receive some distinct data from every other processor, and send some distinct data to every other processor.

This can be used if you need to redistribute a multidimensional array. For example, if you have a rank 2 array distributed over the second dimension, and you need to have it distributed over the first dimension, you could use all-to-all.

23.8 Variations

All of the collective communication routines have different variations. To date we have discussed only the most basic cases.

Each type of routine can, for example, be implemented in blocking or non-blocking form (see next section). So far we have only used the blocking form.

It is also possible in many cases to have varying amounts of data on each processor. Such routines have a 'V' attached to their names. For example, the variable data version of MPI_GATHER is called MPI_GATHERV.

Some communication types also have a version with the word 'ALL' included. These versions put the result on all processors, rather than just one processor. For example, MPI_ALLREDUCE does the same thing as MPI_REDUCE, except that the result is sent to all processors.

24. Synchronization

24.1 Blocking Calls

Blocking calls are calls to routines which do not return until all communication is complete. So far we have only seen blocking calls.

For example, a call to MPI_SEND will not return until the receiving processor has called MPI_RECV, and all the transfer of data is complete.

Blocking calls are safe, in that you know that when the program has returned from one, all data has already been transferred, so you can't accidentally change data before it is sent or use it before it has been received.

However, blocking calls may not be that efficient, because usually one processor must wait for the other.

In addition, in some instances blocking calls can cause a program to 'deadlock'. This means that the processors are each waiting for another to do something, and none of them can proceed.

For example, if two processors perform an MPI_SEND to each other, they will deadlock. This is because each will be waiting for the other to make a call to MPI_RECV, and neither will be able to proceed.

Note that you can manually block processors too. One way is to issue a call to MPI_BARRIER. This causes processors reaching the call to wait until all other processors have reached the same point.

24.2 Non-Blocking Calls

A non-blocking call is a call to a routine which is allowed to return before the communication is complete.

Most MPI routines have a non-blocking version. A non-blocking version has an 'I' in it. For example, a non-blocking send is called 'MPI_ISEND', and a non-blocking reduction is 'MPI_IREDUCE'.

Non-blocking calls can lead to more efficient code, because the processors don't have to wait for each other. If one processor issues a request to send some data, and the other processor is not ready to receive, the sending processor can just keep on calculating.

Using non-blocking calls also prevents deadlocks. For example, in the case we discussed above, where two processors send to one another, if they each issue an MPI_ISEND, there is no deadlock. Each exits the routine before the communication has taken place, and both are then able to make calls to MPI_IRECV or MPI_RECV, at which point the communication can proceed.

Non-blocking calls take an extra argument: the request number. This is an integer, which the routine sets (you don't set it yourself). You can use this to check whether the communication is complete later in your program.

The drawback of non-blocking calls is that you have to be more careful when using them. If you use some data that is being communicated before the communication is complete, you will get unpredictable results. You won't get a run-time error, but your results may change between runs of the program.

Basically, if you need to change some data that is involved in a non-blocking communication, you should first make sure that the communication is complete. To do that, you can use a call to MPI_WAIT.

MPI_WAIT takes the request number of the communication request as an argument.

Figure 45 Using a non-blocking send with a wait

```
include 'mpif.h'
integer error, tag, procToSendTo, request
integer status(MPI_STATUS_SIZE)
real*8 array(numElements)

tag = 878
procToSendTo = 5

call MPI_ISEND(array,numElements,MPI_REAL8, &
  procToSendTo,tag,MPI_COMM_WORLD,request,error)

! you can do extra work here, but don't use array!!!

call MPI_WAIT(request,status,error)

! here you may use array again
```

25. OpenMP Preliminaries

25.1 Advantages

A very important advantage of OpenMP is that it is easier to learn than MPI.

Another is that your program can be parallelized incrementally. That is, you don't need to modify the whole program at once, you can just do a few performance-critical sections to begin with, leaving the rest of the program serial. Though, over time you probably will want to parallelize most of your program, and ultimately this could take as long as using MPI.

OpenMP programs look very much like serial programs, with extra comment lines. These comments are ignored by a standard FORTRAN compiler, but are used to parallelize the code by an OpenMP-enabled compiler. Your OpenMP code can be run as a normal serial program, or in parallel!

If used correctly, OpenMP programs are as fast, or faster than MPI programs.

25.2 Disadvantages

The main disadvantage of OpenMP is that it is less portable than MPI, because it can only be used on shared-memory architectures.

25.3 Shared-Memory Architecture

As already stated, OpenMP can only be used on shared-memory computers. Shared memory computers have multiple CPUs, but only one logical address space.

The logical address space is simply the memory addresses that are important to the program. The physical memory may not be in one unit at all, but may be distributed throughout the system; however, the program just thinks there is one big memory.

The memory in SGI Origin computers, like TERAS, is like this: it is physically distributed, with each processor having its own memory, but there is one big address space.

Writing a program for a shared-memory computer is thus more like writing a program for a serial computer. All data is shared, but you have to remember that multiple CPUs could be modifying data at the same time.

25.4 Threads

OpenMP is based on the idea of 'threads'. Threads are 'threads of computation'.

One thread usually runs on one processor, and you can think of it as a processor, except that it doesn't have it's own memory: it shares its memory with other threads.

Unlike in MPI where the number of processors is fixed for a run, in OpenMP the number of threads can change in time. In serial sections, there will be one thread, and parallel regions there could be multiple threads sharing the burden.

The phrase 'work sharing' is used in relation to OpenMP a lot. The idea is that multiple threads share the work that needs to be done. OpenMP is really about indicating to the compiler the way you would like the work to be shared.

26. Work Sharing Constructs

26.1 Parallel Regions

By default, your program is serial, even if it is compiled with an OpenMP compiler. So, if you do nothing but compile a serial program, it will just run as a serial program, on one processor.

In order to use multiple threads, you need to define 'parallel regions'. These are simply regions of your code where multiple threads can run.

Within any one parallel region, the number of threads cannot change, but the number of threads can change between different parallel regions.

To begin a parallel region in f90, you use '!$OMP parallel', and to terminate it you use '!$OMP end parallel'. In f77, you use 'C$OMP parallel' etc.

Notice that each instruction just looks like a comment, so any serial compiler will simply ignore them.

The string !$OMP (or C$OMP for f77) is used in every OpenMP instruction; it tells the compiler that the line is not an ordinary comment, but is OpenMP.

When the program reaches a parallel region, it may form multiple threads. You can't control exactly how many, but it will never be more than is set in your environment.

Throughout that parallel region, the threads work as though they are each running a separate copy of the code. If there are no more OpenMP instructions, each thread will simply do the same thing, to the same data.

26.2 Do loops

Having multiple threads do the same thing is not generally very useful. To get any benefit from OpenMP, you have to have the threads do different things: they have to 'work share'.

The most widely applicable way to do this is to divide up the iterations in a loop between the threads. You do this by including '!$OMP do' before a do loop, and '!$OMP enddo' directly after the loop.

Doing this will cause the compiler to divide the iterations of the loop as evenly as possible between the threads. Each thread then only does its share of the iterations, rather than all of them.

Figure 46 Parallelizing a do loop with OpenMP

```
!$OMP parallel
...
!$OMP do
do i = 1,10
  a(i) = i*0.2
enddo
!$OMP enddo
...
!$OMP end parallel
```

26.3 Parallelizing a single loop

If in some part of your code you only have a single do loop to parallelize, and no other parallel constructs, it is more efficient to combine the '!$OMP parallel' and '!$OMP do' instructions into the one '!$OMP parallel do' instruction.

Note also that an OpenMP enddo is not necessary.

Figure 47 Parallelizing a single do loop

```
!$OMP parallel do
do i = 1,10
  a(i) = i*0.2
enddo  ! parallel loop ends here
```

26.4 Avoiding synchronization

Just as in MPI, in OpenMP you can get synchronization. Synchronization is where one processor has to wait for another. This is inefficient.

In the do loop examples above any thread finished its iterations has to

wait at the end of the loop for all the other threads before continuing.

If you know that it is safe for the thread to continue without waiting (i.e., the results will not be affected), you can use the 'nowait' keyword to indicate this to the compiler.

Figure 48 Using 'nowait' to avoid synchronization

```
!$OMP parallel
...
!$OMP do
do i = 1,10
  a(i) = i*0.2
enddo
!$OMP enddo nowait
...
!$OMP end parallel
```

26.5 Scheduling iterations

In the default case, the compiler just assigns iterations of a parallel do loop as evenly as possible to the different threads. This is known as 'static scheduling'.

Sometimes, this may be undesirable. For example, if some iterations involve much more calculation than others, some threads may finish much earlier than others, and may have to wait.

You can override the default scheduling of iterations in OpenMP using the 'schedule' keyword.

You can choose between 'static', 'dynamic', and 'guided'.

'static' is the default, where iterations are distributed as evenly as possible.

'dynamic' distributes a chunk of iterations to each thread, and when a thread is finished, it goes back to ask for another chunk. This is good if iterations are not evenly-balanced in terms of computation time.

'guided' is like 'dynamic', except the chunksize gets exponentially smaller.

Figure 49 An example of scheduling a do loop

```
!$OMP parallel

...

! In the following loop, we use dynamic scheduling
! with a chunk size of 2. The chunk size parameter
! is optional.

!$OMP do schedule(dynamic,2)
do i = 1,10
  a(i) = i*0.2
enddo
!$OMP enddo

...

!$OMP end parallel
```

26.6 What is a race condition?

Because we have multiple threads which are all accessing the same memory, it is possible, if you are not careful, that these threads will try to change the same data at the same time.

This does not lead to a run-time error, but may lead to different results each time the program is run. The results may depend on the order in which the threads change the data, and because the speed of each thread may be different in each run, the results may change between runs.

This is called a 'race condition', because the results depend on a 'race' between threads.

Note that there is no problem if more than one thread 'reads' a memory location. *A problem can only arise if a memory location is being 'written' by one or more threads*.

OpenMP has several ways of avoiding race conditions.

Figure 50 Example of a race condition. Each thread attempts to update the same j.

```
!$OMP parallel do
do i = 1,10
  do j = 1,5
    a(i) = i*0.2 + j*0.5
  enddo
enddo
```

26.7 Private and Shared

One way to avoid certain race conditions is to duplicate data. This is the idea behind 'private' variables.

By default, all variables in a parallel region are 'shared'. That means there is only one copy of them, with all threads using that copy.

When you enter a parallel region, you can direct the compiler to make some variables 'private'. What this does is make a separate copy of those variables for each thread.

These private variables are, by default, uninitialized upon entry to the parallel region, and have an undefined value after the parallel region has been exited.

Note that loop variables for parallel loops are automatically private. (Otherwise, you would get a race condition in our earlier examples.)

If you do not explicitly give a variable the private or shared attribute, it will take on the default. Usually this is 'shared'.

You can change the default from 'shared' to 'private' by simply including 'default(private)' at the start of the parallel region.

You can also choose 'default(none)'. This assumes nothing, and you are then forced to explicitly make every variable shared or private. This can be useful for debugging.

Figure 51 Making variables private to avoid race conditions

```
integer i,j
real a(10),priv,sharedConst

priv = 0.0
sharedConst = 1.1
j = 10

!$OMP parallel private(j,priv) shared(sharedConst)

! priv has no definite value here
! j also has no definite value here
! sharedConst is still 1.1

!$OMP do
do i = 1,10   ! i is automatically private
  priv = i + 5.0
  do j = 1,5
    a(i) = i*0.2 + j*0.5 – priv + sharedConst
  enddo
enddo
!$OMP enddo

!$OMP end parallel

! priv has no definite value here
! neither does i or j
! sharedConst is still 1.1
```

26.8 Allowing variables to retain their values

We have just seen that if variables are made private in a parallel region, they have no definite value upon entry to the region, or on exiting the region. You can override this behavior.

If you want your private variables to all be initialized with the value of the variable before entering the parallel region, you can use 'firstprivate'.

If you want your variable after the parallel region to take on the value that the variable would have had if the program were serial, you can use 'lastprivate'.

Figure 52 Using 'firstprivate' and 'lastprivate'

```
integer i,j
real fpriv,lpriv,a(10)

fpriv = 5.0
lpriv = 1.0

!$OMP parallel private(j) firstprivate(fpriv) &
!$OMP lastprivate(lpriv)

! fpriv is still 5.0 here
! lpriv has no definite value here

!$OMP do
do i = 1,10
  do j = 1,5
    a(i) = priv + 1.
  enddo
  lpriv = i
enddo
!$OMP enddo

!$OMP end parallel

! fpriv has no definite value here
! lpriv is 10.0 here
```

26.9 Using locks

Sometimes a race condition can be more subtle than we have seen so far.

For example, if you are using a variable 'sum' to accumulate a sum in a loop, you may think you are safe.

Figure 53 A subtle race condition

```
sum = 0.0
!$OMP do
do i = 1,100
  sum = sum + a(i)
enddo
!$OMP enddo
```

Actually, when you run this code, you will have a race condition. The reason is that the threads can race even within a single statement!

Say thread 0 got the value of 'sum' and 'a(i)' from memory, but before it could add these and put the result back in 'sum', thread 1 updated 'sum' itself. Hopefully you can see that this will lead to the wrong result for 'sum'.

One way to avoid this is to use a 'lock'. The basic idea is that once one thread is in a 'locked' region, no other thread can enter until the first thread has left it.

The most basic directive for getting a lock is 'atomic'. When used, this applies to one statement only, and allows only one thread at a time to perform that statement.

Figure 54 Using atomic to prevent race condition

```
sum = 0.0
!$OMP do
do i = 1,100
  !$OMP atomic
  sum = sum + a(i)
enddo
!$OMP enddo
```

If you have more than one statement, you can use the 'critical' keyword.

Figure 55 Using 'critical' to 'lock' more than one statement

```
sum = 0.0
!$OMP do
do i = 1,100
  !$OMP critical
  sum = sum + a(i)
  sum = sum * 2.0
  !$OMP end critical
enddo
!$OMP enddo
```

Note that neither of the above examples will be efficient, because the threads will have to take it in turn to go through the 'locked' code, but the results will be correct.

This hopefully shows you how you can use locks, but, in practice, you would use reduction to do the above examples (see next section).

26.10 Reduction

Often you will come across cases where you are summing something in a loop, and you have a variable (e.g. 'sum') which you are using to accumulate the result.

You would like to make this fast, and so avoid using the 'lock' approach above. Basically, you want each thread to be able to run independently, rather than having to wait for other threads.

So you decide it would be good to have separate 'sum' variables for each thread, i.e., you decide to use 'private'. But wait... the private variables lose their value before you exit the parallel section, so how could you add them all up to get the total sum.

You can probably think of a way to do it using an extra shared variable, but the most elegant way is to use a 'reduction'.

This is quite a lot like the reductions we saw in MPI. You accumulate some different results for each thread, and at the end you combine these.

You can apply any arithmetic operation in the reduction. For example, you can add things (i.e., sum), or multiply them, etc

The compiler creates private variables for you, initializing them, accumulates the result for each thread, and at the end of the loop combines them into one variable again.

Figure 56 Performing a sum with reduction

```
sum = 0.0
!$OMP parallel do reduction(+:sum)
do i = 1,100
  sum = sum + a(i)
enddo
```

Reductions only work for scalar variables; you can't do it like this with an array.

If you want accumulate multiple sums in an array, you will have to do that manually. The next example shows you how you could perform a matrix transformation of the second dimension of a rank 3 array.

There are some things in the example which have not been covered yet. For example, the 'single' directive, and the OMP functions. Maybe it is better to go forward and review these before you really try to understand this example.

Figure 57 Using a temporary array for a matrix transform.

```
!$OMP parallel private(row,col,threadNum)

!$OMP single
allocate( sumArray(s1,s2,OMP_GET_NUM_THREADS()) )
!$OMP end single

threadNum = OMP_GET_THREAD_NUM()

!$OMP do
do j = 1,s1
  sumArray(:,:,threadNum) = (0._KIND,0._KIND)
  do row = 1,s2
    do col = 1,s2
      sumArray(:,row,threadNum) = sumArray(:,row) + &
        matrix(row,col) * coeffs(:,col,j)
    enddo
  enddo
  coeffs(:,:,j) = sumArray(:,:,threadNum)
enddo
!$OMP enddo

!$OMP single
deallocate(sumArray)
!$OMP end single

!$OMP end parallel
```

26.11 Calling procedures from parallel regions

What happens if you call a procedure from inside a parallel region?

If you think of each thread calling it's own copy of the procedure, you can pretty much guess what will happen.

If you pass a shared variable to a procedure inside a parallel region, it will also be shared in the procedure itself, i.e., all procedures will be using the same variable.

Any global variables, such as module variables, are also shared, because there is only one copy of them in the program.

Variables declared inside the procedure (i.e. local variables) are created inside each thread's copy of the procedure, so these are not shared: they are private.

So basically you can call a procedure as you normally would, but be careful if there are some global variables being written in the procedure, or you are passing shared data to a procedure, because then you could get a race condition!

26.12 Serial code in parallel sections

Sometimes in a parallel section a situation arises where you would like only one thread to perform a certain block of code. For example, you may need to allocate an array, and then you don't want all threads to allocate the array, because this would lead to an error.

OpenMP has a couple of ways of indicating that code should only be performed by one processor. Note that these *cannot* be used in a parallel do loop, they are for use in a parallel region but outside a parallel do loop.

The 'single' directive indicates that the first thread that arrives should carry out the code, but that all others should ignore the code and wait at the end of the code block. You can allow the other threads to continue without waiting by using 'nowait'.

Figure 58 Using the 'single' directive

```
!$OMP parallel

!$OMP single
allocate( sumArray(10,10) )
!$OMP end single

...
```

```
!$OMP single
deallocate(sumArray)
!$OMP end single nowait

!$OMP end parallel
```

A very similar directive is the 'master' directive. Only the master thread (i.e., thread number 0) will perform the code in a master block. The other threads go on without waiting.

Figure 59 Using the 'master' directive

```
!$OMP parallel

!$OMP master
call startTimer()
!$OMP end master

...

!$OMP master
call stopTimer()
!$OMP end master

!$OMP end parallel
```

26.13 OpenMP routines

OpenMP has several procedures which can be called to get certain information, or to set an aspect of the environment. We'll cover a few important ones here, but there are routines for just about all aspects of OpenMP.

One useful routine is OMP_GET_NUM_THREADS. This returns the current number of threads in a parallel region.

Similar to this is OMP_GET_MAX_THREADS. This returns the maximum number of threads that can be used, as set in the environment. It can be useful if you want to declare an array for the whole program, and one dimension depends on the number of threads.

You can get the current thread number using OMP_GET_THREAD_NUM. This returns a number between 0 (the master) and OMP_GET_NUM_THREADS() –1.

Note that if you use these functions, and then try to compile your code with a serial compiler, you will get link errors. You can easily get around this by including wrapper routines for the OpenMP procedures. You don't have to write these yourself; they are given in an appendix of the OpenMP documentation. The documentation is available at the OpenMP web site (see next section).

26.14 More information on OpenMP

We haven't covered all OpenMP constructs here, but we have certainly covered all the ones that I have found useful in my work.

If you would like to know more, check out the OpenMP web site: www.openmp.org

27. Day 3 Exercises

27.1 Array redistribution in MPI

Assume we have a rank 2 array of data. In an MPI program, we decide to distribute this data over all processors, and we decide that we will do this by evenly distributing the first dimension of the array. So, if our array is a(1:n,1:m), then processor 0 has a(1:n/nprocs,1:m), processor 1 has a(n/nprocs+1:2*n/nprocs, 1:m), etc. (We will assume that n is exactly divisible by the number of processors, nprocs.)

Now assume that we need the data distributed not over the first dimension, but over the second dimension of the array. Write MPI

code to accomplish this redistribution. You will need an extra array which represents the redistributed 'a' array. You can do this any number of ways. Consider using MPI_ALL_TO_ALL, or perhaps non-blocking sends and receives, or even MPI_SENDRECV.

27.2 Reduction in MPI

(a) Determine the sum of all elements in the array 'a' in the above exercise. Do this first using MPI_GATHER and MPI_BCAST, such that all processors get the result of the sum. You should calculate the sum of the elements on each processor separately, then gather them onto one processor, sum the partial sums, and then broadcast the result.

(b) Next do it with MPI_ALLGATHER, avoiding the broadcast.

(c) Finally, do it with MPI_ALLREDUCE.

27.3 Do Loops with OpenMP

Add OpenMP statements to parallelize the following loops. Make sure that you identify the variables that should be private. Try to reduce synchronization as much as possible. (Hint: you may need 'nowait')

```
do i = 1,1000
  do j = 1,10
     const = dble(i)/dble(j)
     a(i,j) = const * b(j)
  enddo
enddo

do i = 1,1000
  do j = 1,5
    a(i,j) = a(i,j) * 0.1d0
  enddo
enddo
```

27.4 Avoiding race conditions

(a) The following code accumulates the sums of the second dimension of an array for each index of the first dimension. The result is added to array 'sum'. Use OpenMP to parallelize the outer loop of this code. When the program is run, the values in 'sum' must end up being the same in the OpenMP code as in the serial code. (Hint: To avoid a race condition, you will need to add a dimension to the temporary array.)

```
real,allocatable  :: temp(:)
integer,parameter :: n = 1000, m = 2000
integer           :: i,j
real              :: array(n,m),sum(n)

sum = 1.155d0
allocate(temp(n))

temp = 0.d0
do j = 1,m
  do i = 1,n
    temp(i) = temp(i) + array(i,j)
  enddo
enddo

sum(:) = sum(:) + temp(:)
deallocate(temp)
```

(b) Use OpenMP to parallelize the inner loop of the above example. To make the code efficient, you will need to use 'nowait'. You will need to restructure the code quite a bit.

28. Implementation Specifics

28.1 Make files

A makefile keeps track of the dependencies in your program. By using a makefile, you can avoid recompiling your whole program when only a small part of it really needs to be recompiled.

A makefile includes various default rules for compiling your code at the beginning.

It then has a list of 'dependencies'. This simply tells the 'make' program what to update if changes are made to a particular file.

You can also use macros in makefiles. These can be assigned values, and used throughout the makefile. By using macros, you only have to change a variable in one place, rather than all occurrences in your makefile.

To compile your program you simply type 'make'. Your makefile should be called 'makefile' or 'Makefile'.

By default, make just tries to compile the first dependency it comes to. You should make this your standard executable.

A dependency line consists of a file name, followed by a colon, and then the names of the files it depends on. For example, 'main.o: main.f another.o' means that the file main.o depends on main.f and another.o. If either main.f or another.o change, main.o will be recompiled.

The line continuation character in a make file is a backslash.

You can add shell commands after any dependency line. These are the commands used to update the file. Note, *you must put a TAB before any command*.

You can also add 'phony targets'. These can be invoked by typing 'make phony_target_name'. For example, it is standard to have one phony target called 'clean', which removes any executables and object files (see example). A phony target does not depend on any other files; it just has shell commands.

Figure 60 A sample makefile.

```
# this is a macro
OBJECTS = WaveFunction.o FluxAnalyzor.o CoherentStateFunc.o \
          EckartPES.o SplitOpPropagator.o FourierGrid.o


#
# this is the default rule for creating a .o file from a .f
#
# The $< means to substitute the name of the first file depended
# upon (files that are depended upon are called 'prerequisites').
#
.f.o:
        f90 -64 -freeform -O3 -c $<


onedim.x: $(OBJECTS) main.o
        f90 -64 -o onedim.x $(OBJECTS) main.o -lscs

main.o:                 main.f $(OBJECTS)
FourierGrid.o:          FourierGrid.f
EckartPES.o:            EckartPES.f
CoherentStateFunc.o:    CoherentStateFunc.f
WaveFunction.o:         WaveFunction.f FourierGrid.o \
                        CoherentStateFunc.o \
                        EckartPES.o
SplitOpPropagator.o:    SplitOpPropagator.f WaveFunction.o
FluxAnalyzor.o:         FluxAnalyzor.f WaveFunction.o \
                        CoherentStateFunc.o

clean:
        rm *.o *.mod *.x
```

28.2 IBM SP2 Considerations

In order to compile MPI code on the SP2, you need to use the mpxlf90 command. This is simply a shell script which, after linking in the appropriate MPI libraries, calls the standard xlf90 compiler.

If you want more information about MPI, or the IBM parallel environment, there is lots of documentation in the directory: /usr/lpp/poe.pedocs

The scientific library to use on the IBM SP2 is the ESSL library. Documentation on this can be found in: /usr/lpp/essl

Figure 61 Recommended compile options for the IBM SP2

For debugging	`mpxlf90 -qflttrap=ov:zero:inv:en -qsigtrap -qfixed -g -C -qarch=pwr2 -lessl`
Fully optimized	`mpxlf90 -qfixed -O3 -qarch=pwr2 -lessl`
Free format	`Replace –qfixed with -qfree`

To run a program on the SP2, you can either run it interactively using the 'poe' command, or submit it to the queuing system: loadleveller. Either way you need to use 'poe', the 'parallel operating environment' to run the MPI program.

To use 'loadleveller', you need to make a .cmd file. This has various details about setting up the parallel environment, such as the number of processors to use, if the program is parallel or serial, etc. To learn more about writing a .cmd file, read the poe guides in the documentation directories on the SP2.

Figure 62 A sample .cmd file for using loadleveller

```
#@ executable = /usr/bin/poe
#@ arguments = executable_file -euilib us -pulse 0
#@ job_type = parallel
#@ requirements = (Adapter == "hps_user")
#@ preferences = (Machine == {"sp2n13" "sp2n14" "sp2n15" "sp2n16"})
#@ class = plong
#@ output = job.o
#@ error = job.e
#@ file_limit = unlimited,unlimited
#@ core_limit = unlimited,unlimited
#@ data_limit = unlimited,unlimited
#@ stack_limit = unlimited,unlimited
#@ min_processors = 6
#@ max_processors = 6
#@ queue
```

Having written a command file, submit it using 'llsubmit file_name.cmd'

'llq' is a command to view the current loadleveller queue, and 'llstatus' shows the status of the various processors in the system.

28.3 SGI Origin Considerations

To compile an MPI program on the SGI, you need to link in the MPI library, using '-lmpi'. Note that you may need to load the mpt module first. If that is the case, you should put 'module load mpt' somewhere in your shell resource file (e.g., .cshrc, .profile).

To compile an OpenMP program, you need to include the flag '-mp', both for compilation and linking.

The SGI scientific library is SCSL. You can link this in using '-lscs'. For OpenMP programs, use '-lscs_mp'. Note that you need to ensure that the appropriate module is loaded first. You may need to add 'load module scsl' to your resource files. The NAG library is also available on the SGI system.

Figure 63 Recommended compile options for the SGI Origin computers

Debug	`f90 -DEBUG:trap_uninitialized=ON -g -C`
Fully optimized	`f90 -O3`
MPI	`Link in –lmpi`
OpenMP	`Add –mp option`

To run an MPI program on the SGI Origin systems, it is necessary to use 'mpirun'. This command takes the number of processors as an option. Set this with '-nt num_procs'.

There are many environment variables on the SGI. You set these in

your jobs to control how your parallel programs run. To learn more about the environment variables, type ‘man ENVIRONMENT’.

Figure 64 Some important environment variables

The maximum number of OpenMP threads.	`OMP_NUM_THREADS`
Whether or not the number of threads can change in time (TRUE) or is fixed (FALSE).	`OMP_DYNAMIC`