## Goals for today

- Understand how Fortran compilers optimize programs
- Make Fortran programs faster using optimization flags
- Get to know the software make
- Use makefiles to control individual compilation and link steps and define dependencies

# Optimization

- Large simulations can take a long time → the speed of the model is important.
- Optimization means making the model as fast as possible, and also minimizing the used memory, code size, and power consumption.
- Optimization stages
  1. Select the fastest algorithm (e.g. multigrid method)
  2. Structure the code efficiently (e.g. outer loops over columns, inner loops over rows)
  3. Use compiler options for optimization (e.g. -O2)
     - This is especially suitable for optimizations that make the code less readable.

# Compiler options

- Optimization levels summarize various optimizations.
  - O0: minimal optimization (gfortran default)
  - O1: low-level optimization
  - O2: moderate optimization (ifort default)
  - O3: strong optimization
  - Ofast: very strong optimization (without consideration of standards)
  - Os: Optimization for code size (similar to O2)
- The optimization levels O0 to Ofast are cumulative, i.e. they include all optimizations of the lower levels.

Speed test for convection model (Exercise 7, Namelist 1)
gfortran, 64bit, HP ZBook-Firefly 14 G7

| Option | Time (s) |
|---|---|
| -fbounds-check | 43.83 |
| -O0 | 26.88 |
| -O1 | 13.25 |
| -O2 | 13.00 |
| -O3 | 11.26 |
| -Ofast | 9.94 |

All individual optimization options for gfortran: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

# Peephole optimization

O0

Compiler looks through a peephole, examines adjacent instructions, and tries to replace them with fewer and/or more efficient instructions.

Examples:

```
a = b + c
d = a + e      →    d = b + c + e

y = x**2       →    y = x*x

y = x*2        →    y = x+x  or   y = ISHFT(x,1)

y = 0          →    y = XOR(y,y)
```

slower

| Operation | Description |
|---|---|
| ** | Power |
| SIN<br>COS<br>TAN | Sine<br>Cosine<br>Tangent |
| SQRT | Square root |
| EXP<br>LOG | Exponential function<br>Logarithm |
| / | Division |
| * | Multiplication |
| ABS | Absolute value |
| +<br>-<br>OR<br>AND<br>XOR<br>ISHFT | Addition<br>Subtraction<br>Disjunction<br>Conjunction<br>Antivalence<br>Bitwise shift |

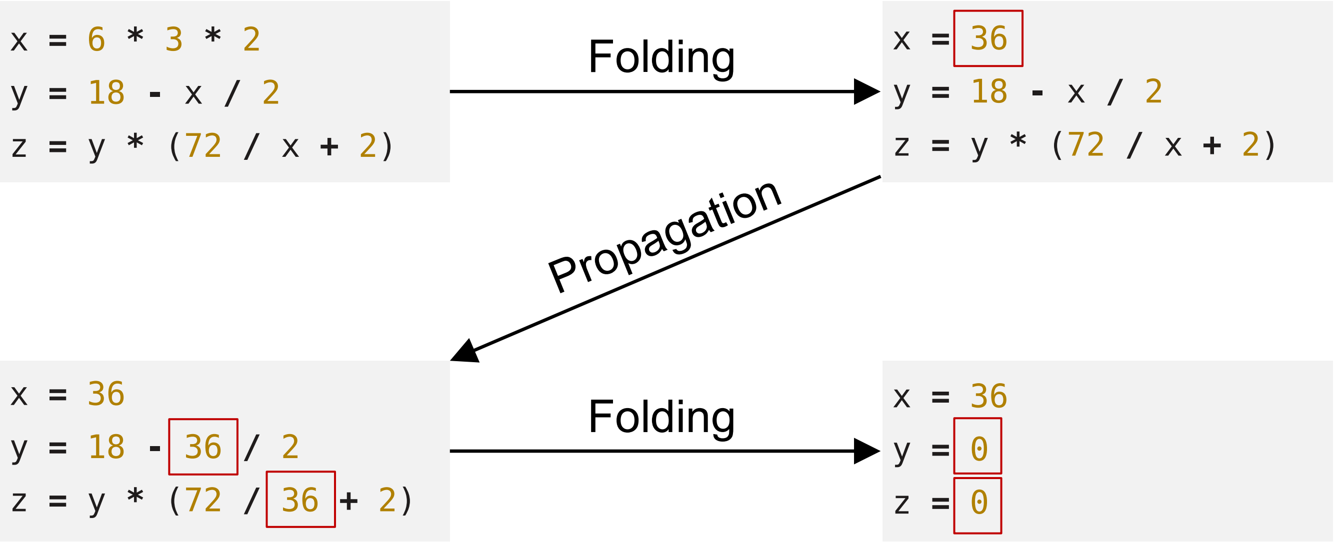
faster

# Constant folding and propagation



Expressions consisting of constants are already calculated during compilation.

The values of constants are propagated, i.e. used in expressions.



# Dead code elimination



Code that does not affect the result of the program is removed.

```
REAL FUNCTION aha(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
REAL :: a, b

a = 10.
b = a / 5.
IF (a > 0) THEN
  PRINT*, 'a is positive'
  aha = x / a
ELSE
  PRINT*, 'a is negative or zero'
  aha = x * a
END IF

END FUNCTION aha
```

is never used

Is never reached

→

```
REAL FUNCTION aha(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
REAL :: a

a = 10.
PRINT*, 'a is positive'
aha = x / a

END FUNCTION aha
```

Although the compiler can do this, it is usually worthwhile to delete dead code yourself to make the program more understandable.

# Inline replacement



Code of called procedures is copied in place of the call.

→ Saves overhead for the call

→ Allows additional optimizations

Works only for non-recursive procedures.

```
MODULE statistics
CONTAINS
  REAL FUNCTION mean(arr)
  IMPLICIT NONE
  REAL, INTENT(IN) :: arr(:)
  mean = SUM(arr)/SIZE(arr)
  END FUNCTION mean
END MODULE statistics
```

```
PROGRAM outline
  USE statistics
  IMPLICIT NONE
  REAL :: array(7)
  CALL RANDOM_NUMBER(array)
  PRINT*, mean(array)
END PROGRAM outline
```

→

```
PROGRAM inline

  IMPLICIT NONE
  REAL :: array(7)
  CALL RANDOM_NUMBER(array)
  PRINT*, SUM(array)/SIZE(array)
END PROGRAM inline
```

# Loop optimization

Often a program spends most of its time in loops, so loop optimization is especially important.

Common loop optimizations:
(see also https://en.wikipedia.org/wiki/Loop_optimization)

- Fission
- Fusion
- Interchange
- Inversion
- Loop invariant code motion
- Parallelization
- Scheduling
- Software pipelining
- Splitting
- Vectorization
- Unrolling
- Unswitching

# Loop invariant code motion



Expressions and statements that do not change in the loop are taken out of loops, and thus need to be calculated only once.

For DO WHILE loops, additional IF query is necessary, otherwise the result could be different (here if n < 0).

```
READ*, n

i = 0
DO WHILE (i < n)
  x = y + z
  a(i) = 6*i + x**2
  i = i + 1
END DO
```

→

```
READ*, n

i = 0
x = y + z
tmp = x**2
DO WHILE (i < n)
  a(i) = 6*i + tmp
  i = i + 1
END DO
```

→

```
READ*, n

i = 0
IF (i < n) THEN
  x = y + z
  tmp = x**2
  DO WHILE (i < n)
    a(i) = 6*i + tmp
    i = i + 1
  END DO
END IF
```

# Unrolling



Loop body is copied such that the loop needs fewer passes.

- Advantages: less overhead for loop, easier to parallelize
- Disadvantage: code becomes larger

Original

```
DO i = 2, 9
  a(i) = b(i+1) - c(i-1)
END DO
```

→

Partially unrolled

```
DO i = 2, 9, 4
  a(i)   = b(i+1) - c(i-1)
  a(i+1) = b(i+2) - c(i)
  a(i+2) = b(i+3) - c(i+1)
  a(i+3) = b(i+4) - c(i+2)
END DO
```

→

Completely unrolled

```
a(2) = b(3)  - c(1)
a(3) = b(4)  - c(2)
a(4) = b(5)  - c(3)
a(5) = b(6)  - c(4)
a(6) = b(7)  - c(5)
a(7) = b(8)  - c(6)
a(8) = b(9)  - c(7)
a(9) = b(10) - c(8)
```

# Fission and fusion



Fission: A loop containing independent statements is split into multiple loops to improve locality of reference.

```
DO i = 1, n
  a(i) = i*3
  b(i) = i+5
END DO
```

↓

```
DO i = 1, n
  a(i) = i*3
END DO
DO i = 1, n
  b(i) = i+5
END DO
```

Fusion: Multiple loops that go over the same area are merged to reduce loop overhead.

```
DO i = 1, n
  x = x*a(i) + b(i)
END DO
DO j = 1, n
  y = y*a(j) + c(j)
END DO
```

↓

```
DO i = 1, n
  x = x*a(i) + b(i)
  y = y*a(i) + c(i)
END DO
```

# Unswitching



Conditional statements are taken out of loops.

→ Fewer tests

→ Easier to parallelize (pipelining)

```
DO i = 2, n
  IF (some_ind > 10) THEN
    y(i) = y(i-1) + a*x(i)
  ELSE
    y(i) = y(i-1) - a*x(i)
  END IF
END DO
```

→

```
IF (some_ind > 10) THEN
  DO i = 2, n
    y(i) = y(i-1) + a*x(i)
  END DO
ELSE
  DO i = 2, n
    y(i) = y(i-1) - a*x(i)
  END DO
END IF
```

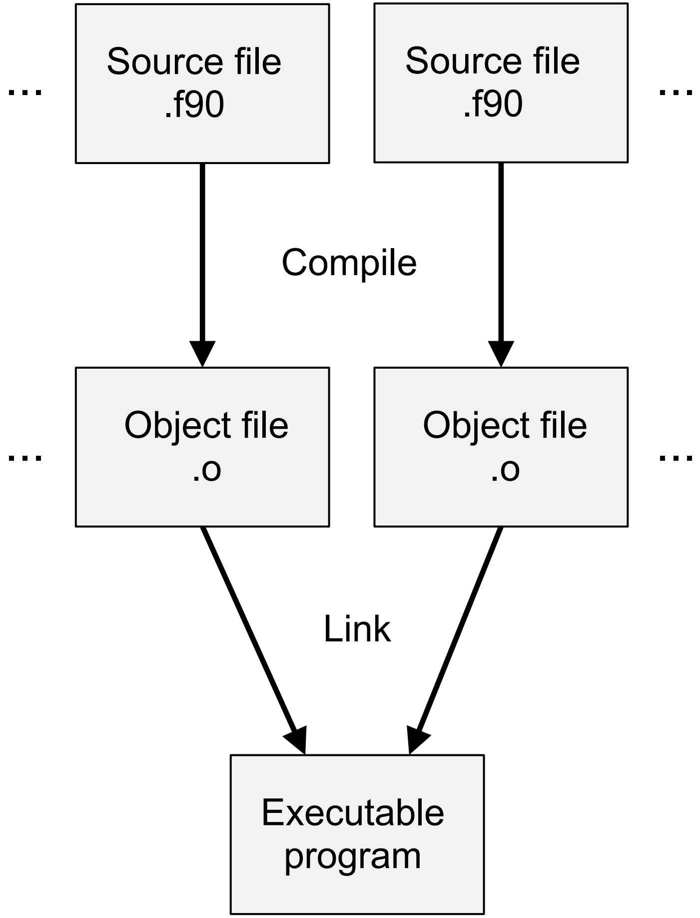
# Compiling and linking

Normally, the compile and link steps are executed together and the object files are deleted. However, they can also be executed separately.

compile only, do not link

```
$ gfortran -c -O2 kinds.f90 findiff.f90 poisson.f90 main.f90
$ gfortran -o conv_model kinds.o findiff.o poisson.o main.o
$ ls
conv_model  findiff.o   kinds.o     poisson.f90
findiff.f90 kinds.f90   main.f90    poisson.mod
findiff.mod kinds.mod   main.o      poisson.o
```

This makes it possible to recompile only the necessary source files. Important: Pay attention to dependencies!



# Makefiles

- Makefiles (and the associated software make) can be used to define dependencies and control individual compile and link steps.

- Makefiles consist of rules.
  - Each rule starts with a dependency line defining a target, followed by a colon and, optionally, a set of components the target depends on.
  - Under each dependency line there can be commands. **The commands must be indented with a Tab**.
  - The first target is always the main target.

- Makefiles are called with the command make [target].

Makefile rule

```
target [target ...]: [component ...]
Tab ↹ [command 1]
          .
          .
          .
Tab ↹ [command n]
```

# Example Makefile

- Variables are defined with = and referenced with $.
- Pattern rule: % stands for any string.
- Not every target must correspond to a file.
- Special variables:
  - $@: Target
  - $<: First component
  - $?: Components newer than the target
  - $*: Text matching % in the pattern

```
FC = gfortran
FFLAGS = -O2 -Wall
SRC = kinds.f90 findiff.f90 poisson.f90 main.f90
OBJ = $(SRC:.f90=.o)
```

```
conv_model: $(OBJ)
	$(FC) $(FFLAGS) -o $@ $(OBJ)
```

Link

```
%.o: %.f90
	$(FC) $(FFLAGS) -c $<
```

Compile

```
findiff.o: kinds.o

poisson.o: kinds.o

main.o: kinds.o findiff.o poisson.o
```

Dependencies between files

```
clean:
	rm -f *.o *.mod conv_model
```

Clean up

# Summary

- Fortran compilers are smart and can make a program much faster through optimization.

- However, optimization takes time. The higher the optimization level, the longer the compilation.

- During development (and debugging) it is best to use little or no optimization, possibly with additional options like -fbounds-check. For the finished program we can use stronger optimization.

- Makefiles save compilation time by specifying dependencies. This is especially helpful for large programs with heavy optimization.