1. Лабораторная работа №0

1.1. Вводные замечания

Каждая лабораторная состоит из нескольких заданий. Выполнение каждого задания заключается в написании одной или нескольких функций или подпрограмм на языке Fortran. Перед началом выполнения заданий прочтите теоретическую часть, где приводятся необходимые формулы и даются дополнительные пояснения. Всю информацию по языку Fortran и OpenMP можно найти в лекциях и методическом пособии.

При написании программы следует организовать код определенным образом. Для примера смотрите тренировочную лабораторную № 0.

В каждом задании необходимо провести замеры времени работы функций и добиться того, чтобы параллельная версия работала быстрее последовательной. Кроме того, при увеличении количества потоков время выполнения программы должно уменьшаться. Как производить замеры и как нарисовать графики времени выполнения, ускорения и эффективности смотрите в нижеследующем описании лабораторной №0. Продемонстрируем образец выполнения лабораторной работы на примере двух простейших заданий. Основное внимание уделим тому, как правильно оформлять код и тестировать созданную программу. Принципы оформления и тестирования в своей основе не зависят от выбранного языка программирования и их понимание пригодится при написании программ на любом другом языке программирования.

1.2. Задание №1

Написать функцию или подпрограмму, которая решает квадратное уравнение $ax^2 + bx + c = 0$ с действительными коэффициентами a, b и c. Протестировать программу на корректность работы.

1.3. Задание №2

Написать функцию, находящую сумму элементов одномерного массива с использованием OpenMP. Протестировать функцию на корректность и на производительность, то есть удостоверится, что при увеличении количества потоков время вычислений уменьшается.

1.4. Общие пояснения

Вышеприведенные задания крайне просты и обычно даются учащимся в самом начале обучения программирования. Однако мы подойдем к выполнению данных заданий используя методику написания крупных программ.

- В первую очередь мы организуем программу следуя принципам структурного программирования: разобьем ее на функции, подпрограммы и исходные файлы.
- Во вторую очередь мы напишем тесты, автоматически проверяющие наш код, что позволит отловить большинство потенциальных ошибок.

Начнем с того, что создадим каталог `lab00` и все исходные файлы, касающиеся лабораторной №0 будем хранить исключительно в нем. Далее мысленно представим из каких частей будет состоять наша первая программа, решающая квадратное уравнение.

- Непосредственное решение уравнения. Можно оформить его в виде главной программы, но так как мы хотим далее протестировать этот код, указывая разные коэффициенты a, b и c, то выделить его в виде отдельной подпрограммы или функции.
- Проверка работы нашей программы (тестирование). На первый взгляд проверять в такой простой программе нечего, но далее мы увидим, что это не так.

Для второго задания программа кроме вышеперечисленных двух частей будет содержать еще две.

- Проверка эффективности работы параллельной версии.

- Визуализация результатов замеров быстродействия параллельной версии программы, то есть построение графиков времени выполнения, ускорения и эффективности.

Кроме исходного кода в процессе сборки и компиляции программы могут появится временные служебные файлы `.mod` и `.o`, непосредственно исполняемые файлы программ, а также файлы с изображениями графиков. Даже в такой небольшой программе как наша, будет неудобно, если все эти файлы будут находиться в одном каталоге. Поэтому создадим следующую иерархию каталогов.

- `bin` — в данный каталог будем сохранять созданные исполняемые файлы; изначально пуст.
- `imgs`— в данный каталог будем сохранять графики времени выполнения, ускорения и эффективности; изначально пуст.
- `mod` — в данный каталог сохраняются созданные компилятором `mod` файлы; изначально пуст.
- `src` — в данном каталоге сохраним файлы с исходным кодом, который непосредственно ответственен за выполнение заданий.
- `test` — в данном каталоге создадим файлы с исходным кодом тестирующих программ.

Так как наши программы будут сопровождаться проверками на корректность выполнения и на производительность, разумно команды компиляции и запуска не вводить каждый раз с консоли, а автоматизировать. Существует масса утилит для такой автоматизации (системы сборки). Для Unix (GNU Linux, macOS) простейшей является утилита Make, а для Windows можно обойтись bat или powershell скриптом. Поэтому у нас в каталоге будут присутствовать еще два файла.

- `Makefile.bat` — сценарий для запуска компиляции и тестирующих программ под Windows.
- `Makefile` — `make` файл для сборки программ и запуска тестирования под GNU Linux и macOS.

Незабудем также, что для второго задания следует нарисовать ряд графиков. Будем использовать язык Python с библиотекой Matplotlib. Скрипт назовем `plot.py` и напишем так, чтобы с минимальными изменениями он работал и для других заданий.

Скриптами для сборки можно и не пользоваться, однако они существенно облегчат работу.

1.5. Выполнение задания №1

Начнем выполнение задания №1. Для вычисления корней квадратного уравнения $ax^2+bx+c=0$ необходимо вычислить дискриминант $D=b^2-4ac$ и далее корни по формуле

$$x_{1,2}=\frac{-b\pm\sqrt{D}}{2a}.$$

Следует отдельно учесть случаи $D>0$, $D<0$ и $D=0$. Так как Fortran поддерживает комплексные числа, то наша программа может выдать корректный результат и при $D<0$, а не сообщение об отсутствии действительных корней. Кроме того, с вычислительной точки зрения эффективней будет поделить уравнение на коэффициент a и решать его в виде $x^2+px+q=0$, где $p=b/a$ и $q=c/a$. В результате подпрограмма решающая квадратное уравнение может выглядить следующим образом.

```
31 subroutine quadroots(a, b, c, real_roots, complex_roots, is_real)
32   implicit none
33   real(real64), intent(in) :: a, b, c
34   real(real64), intent(out), dimension(1:2) :: real_roots
35   complex(real64), intent(out), dimension(1:2) :: complex_roots
36   logical, intent(out) :: is_real
37
38   real(real64) :: p, q
39   real(real64) :: D
40
```

```
  p = b/a
  q = c/a
  D = p*p - 4.0*q
  is_real = .true.
  if (D > 0) then
    real_roots(1) = 0.5*(-p - sqrt(D))
    reaL_roots(2) = 0.5*(-p + sqrt(D))
  else if (D < 0) then
    is_real = .false.
    complex_roots(1) = 0.5*(-p - sqrt(cmplx(D, kind=8)))
    complex_roots(2) = 0.5*(-p + sqrt(cmplx(D, kind=8)))
  else if (abs(D) < epsilon(1.0d0)) then
    real_roots(1) = -0.5*p
    real_roots(2) = -0.5*p
  end if

end subroutine quadroots
```

Рассмотрим какие аргументы принимает данная подпрограмма.

- Аргументы `a`, `b`, `c` соответствуют коэффициентами уравнения a, b и c и имеют тип `real(real64)` или иначе `double precision`. Подпрограмма не должна в процессе своей работы каким либо образом модифицировать аргументы `a`, `b`, `c`, на что указывает атрибут `intent(in)`.
- Аргумент `real_roots` является одномерным массивом из двух элементов типа `double precision`. В этот аргумент подпрограмма должна записать результат вычисления корней уравнения, в случае если корни действительные.
- Аргумент `complex_roots` является одномерным массивом из двух элементов типа `complex(real64)`. В этот аргумент подпрограмма должна записать результат вычисления корней уравнения, в случае если корни комплексные или чисто мнимые.
- Аргумент `is_real` имеет тип `logical(kind=1)`. Ему присваивается значение `.true.` в случае если корни уравнения действительные и `.false.` в случае комплексных корней.

Обратите внимание, что проверка на равенство нулю дискриминанта сделана как

```
else if (abs(D) < epsilon(1.0d0)) then
```

так как для действительных чисел необходимо учитывать погрешность представления с помощью чисел с плавающей запятой.

Дополним нашу подпрограмму еще двумя функциями, вычисляющими значение квадратного трехчлена. Эти функции понадобятся при тестировании результатов вычисления.

```
! Вычисление квадратного трехчлена
pure function real_p2(a, b, c, x)
  implicit none
  real(real64), intent(in) :: a, b, c
  real(real64), intent(in) :: x
  real(real64) :: real_p2
  ! Оптимизируем число операций
  real_p2 = (x + (b/a)) * x + (c/a)
end function real_p2

! Вычисление квадратного трехчлена от комплексных чисел
pure function complex_p2(a, b, c, x)
  implicit none
```

```
  real(real64), intent(in) :: a, b, c
  complex(real64), intent(in) :: x
  real(real64) :: complex_p2
  ! оптимизируем число операций
  complex_p2 = dble((x + (b/a)) * x + (c/a))
end function complex_p2
```

Сохраним все процедуры в файле `quadequation.f90`, заключив их в модуль `quadequation` для последующего удобного вызова. В результате получим следующий исходный код.

```
module quadequation
  use iso_fortran_env, only: int32, int64, real32, real64
  implicit none
  private

! Доступные вовне функции и подпрограммы
  public :: real_p2, complex_p2, quadroots

  contains
  ! Вычисление квадратного трехчлена
  pure function real_p2(a, b, c, x)
    implicit none
    real(real64), intent(in) :: a, b, c
    real(real64), intent(in) :: x
    real(real64) :: real_p2
    ! Оптимизируем число операций
    real_p2 = (x + (b/a)) * x + (c/a)
  end function real_p2

  ! Вычисление квадратного трехчлена от комплексных чисел
  pure function complex_p2(a, b, c, x)
    implicit none
    real(real64), intent(in) :: a, b, c
    complex(real64), intent(in) :: x
    real(real64) :: complex_p2
    ! оптимизируем число операций
    complex_p2 = dble((x + (b/a)) * x + (c/a))
  end function complex_p2

  ! Вычисление корней квадратного уравнения
  subroutine quadroots(a, b, c, real_roots, complex_roots, is_real)
    implicit none
    real(real64), intent(in) :: a, b, c
    real(real64), intent(out), dimension(1:2) :: real_roots
    complex(real64), intent(out), dimension(1:2) :: complex_roots
    logical, intent(out) :: is_real

    real(real64) :: p, q
    real(real64) :: D

    p = b/a
    q = c/a
    D = p*p - 4.0*q
    is_real = .true.
    if (D > 0) then
```

```
      real_roots(1) = 0.5*(-p - sqrt(D))
      reaL_roots(2) = 0.5*(-p + sqrt(D))
    else if (D < 0) then
      is_real = .false.
      complex_roots(1) = 0.5*(-p - sqrt(cmplx(D, kind=8)))
      complex_roots(2) = 0.5*(-p + sqrt(cmplx(D, kind=8)))
    else if (abs(D) < epsilon(1.0d0)) then
      real_roots(1) = -0.5*p
      real_roots(2) = -0.5*p
    end if

  end subroutine quadroots
end module quadequation
```

После того, как написали процедуру можно приступать к ее проверке. Необходима подавать на вход процедуре `quadroots` разные коэффициенты a, b и c и проверять правильность вычисленных корней. Даже для такой простой программы как наша можно придумать большое количество проверок. Проверяющая программа может выглядеть следующим образом.

```
include "../src/quadequation.f90"

! Тестирование решений квадратного уравнения
subroutine test(a, b, c)
  use iso_fortran_env, only: int32, int64, real32, real64
  use quadequation
  implicit none

  real(real64), intent(in) :: a, b, c

  real(real64), dimension(1:2) :: x
  complex(real64), dimension(1:2) :: cx
  logical :: is_real

  is_real = .true.

  call quadroots(a, b, c, x, cx, is_real)

  ! действительные корни
  if (is_real) then
    ! проверяем корни
    print *, "test 01: ", abs(real_p2(a, b, c, x(1))) <= epsilon(1.0d0)
    print *, "test 02: ", abs(real_p2(a, b, c, x(2))) <= epsilon(1.0d0)
    ! теорема Виета
    print *, "test 03: ", abs(x(1) + x(2) + b/a) <= epsilon(1.0d0)
    print *, "test 04: ", abs(x(1) * x(2) - c/a) <= epsilon(1.0d0)
  ! комплексные корни
  else if (.not. is_real) then
    ! проверяем корни
    print *, "test 01: ", abs(complex_p2(a, b, c, cx(1))) <= epsilon(1.0d0)
    print *, "test 02: ", abs(complex_p2(a, b, c, cx(2))) <= epsilon(1.0d0)
    ! теорема Виета
    print *, "test 03: ", abs(cx(1) + cx(2) + b/a) <= epsilon(1.0d0)
    print *, "test 04: ", abs(cx(1) * cx(2) - c/a) <= epsilon(1.0d0)
  end if
end subroutine test
```

```
37
38 program test_quad
39   use iso_fortran_env, only: int32, int64, real32, real64
40   implicit none
41   real(real64) :: a, b, c
42
43   print *, "Целые коэффициенты"
44   a = 1; b = 2; c = 1
45   call test(a, b, c)
46
47   print *, "Рациональные коэффициенты"
48   a = 1.2; b = 2.5; c = 0.5
49   call test(a, b, c)
50
51   print *, "Иррациональные коэффициенты"
52   a = 1; b = sqrt(2.0d0); c = 0.5
53   call test(a, b, c)
54
55   print *, "Комплексные корни"
56   a = 4.0; b = 2.0; c = 1.0
57   call test(a, b, c)
58
59   print *, "Большие числа"
60   a = 45678907.0; b = 20987654.0; c = 109876.0
61   call test(a, b, c)
62
63   print *, "Большие отрицательные числа"
64   a = -45678907.0; b = 20987654.0; c = -109876.0
65   call test(a, b, c)
66 end program test_quad
```

Мы едва ли перебрали все возможные проверки. Например, наша программа даст сбой, при $a = 0$, так как в теле самой программы происходит деление на a. Для более крупных программ перебрать все возможные случаи и протестировать все ветки бывает зачастую невозможно.

Для запуска программы выполним следующую команду

```
1 gfortran -Wall ./test/test_quad.f90 -o ./bin/test_quad.exe -J mod
```

Так как в файле `test_quad.f90` в первой строке содержится директива

```
1 include "../src/quadequation.f90"
```

то компилятор автоматически подключит текст модуля `quadequation` и все переменные и процедуры доступные вовне модуля станет возможно вызывать и использовать в основной программе. Опция `-J` указывает, что файл `mod`, который компилятор генерирует автоматически, надо сохранить в каталог `mod`, а не создавать непосредственно в корневом каталоге.

1.6. Выполнение задания №2

Во втором задании следует используя директивы OpenMP реализовать итеративное суммирование элементов массива. Организуем код точно также в виде файла с кодом модуля и заключенных в нем процедур. Хотя в данном случае можно обойтись всего одной функцией, но модуль все равно нужен, чтобы компилятор автоматически создал интерфейс для вызова функции.

```
1 module summation
2   use iso_fortran_env, only: int32, int64, real32, real64
3   implicit none
```

```
  private
  public :: iterative_sum
  contains

  function iterative_sum(array, threads_num) result(res)
    implicit none
    real(real64), dimension(1:), intent(in) :: array
    integer(int32), intent(in) :: threads_num
    real(real64) :: res
    integer(int32) :: length, i

    length = size(array, 1)
    res = 0.0
    !$omp parallel shared(array) num_threads(threads_num)
    !$omp do reduction(+ : res)
      do i = 1,length,1
        res = res + array(i)
      end do
    !$omp end do
    !$omp end parallel
  end function iterative_sum

end module summation
```

- Аргумент `array` — массив типа `double precision`, сумму элементов которого необходимо найти.
- Аргумент `threads_num` — число потоков, которые функция может создать для проведения вычислений. Имеет тип `integer`.

В отличие от предыдущего задания, в данном задании есть две тестирующие программы. Первая программа проверяет функцию на корректность, а вторая замеряет время работы, ускорение и эффективность в зависимости от числа потоков. Приведем исходный код этих программ.

```
include "../src/summation.f90"

program test_sum
  use iso_fortran_env, only: int32, int64, real32, real64
  use summation
  implicit none

  ! Количество испытаний
  integer(int32), parameter :: iterations_num = 100
  ! Длина массива
  integer(int64) :: array_length
  ! Количество нитей
  ! integer(int32) :: threads_num

  ! Динамический массив для проверки функции
  real(real64), dimension(:), allocatable :: test_array
  real(real64) :: correct_sum
  real(real64) :: test_res
  integer(int64) :: i
  ! Вначале проверяем на корректность вычислений


```

```
    ! --------  Тест 1 ---------
    array_length = 10
    allocate(test_array(1:array_length))

    ! Массив натуральных чисел от 1 до конца массива
    test_array = [(i, i=1,array_length)]
    correct_sum = 0.5 * (test_array(1) + test_array(array_length)) * array_length
    test_res = iterative_sum(test_array, 1)

    print *, test_res == correct_sum

    deallocate(test_array)
    ! -----------------

    ! --------  Тест 2 ---------
    array_length = 1000
    allocate(test_array(1:array_length))

    ! Массив отрицательных целых чисел
    test_array = [(-i, i=1,array_length)]
    correct_sum = 0.5 * (test_array(1) + test_array(array_length)) * array_length
    test_res = iterative_sum(test_array, 1)
    print *, test_res == correct_sum

    deallocate(test_array)
    ! -----------------

    ! --------  Тест 3 ---------
    array_length = 1000
    allocate(test_array(1:array_length))

    ! Наполняем случайными числами
    call random_number(test_array)

    correct_sum = sum(test_array)
    test_res = iterative_sum(test_array, 1)
    print *, abs(test_res - correct_sum) <= epsilon(1.0d0)

    deallocate(test_array)
    ! -----------------
end program test_sum
```

```
include "../src/summation.f90"

program test_sum
  use iso_fortran_env, only: int32, int64, real32, real64
  use summation
  implicit none
  include "omp_lib.h"

  ! Количество испытаний
  integer(int32), parameter :: iterations_num = 1000
  ! Максимальное количество нитей
  integer(int32), parameter :: max_threads_num = 8
```

```
    ! Длина массива
    integer(int64) :: array_length
    ! Количество нитей
    integer(int32) :: threads_num
    ! Для замера времени
    real(real64) :: t1, t2
    real(real64), dimension(iterations_num) :: T

    ! Динамический массив для проверки функции
    real(real64), dimension(:), allocatable :: test_array
    real(real64) :: test_res
    integer(int64) :: i

    ! Создаем большой массив для тестирования производительности
    array_length = 1000 * 1000
    allocate(test_array(1:array_length))

    call random_number(test_array)

    do threads_num = 1,max_threads_num,1
      do i = 1,iterations_num,1
        t1 = omp_get_wtime()
        test_res = iterative_sum(test_array, threads_num)
        t2 = omp_get_wtime()
        T(i) = t2 - t1
      end do
      print '(i2,",",G0)', threads_num, sum(T) / dble(iterations_num)
    end do

    deallocate(test_array)
end program test_sum
```

Для компиляции проверяющих программ и сборки исполняемых файлов следует выполнить следующие команды:

```
gfortran -fopenmp -Wall test_sum.f90 -o test_sum  -J mod
gfortran -fopenmp -Wall test_sum_omp.f90 -o test_sum_omp  -J mod
```

В результате компиляции будет созданы два исполняемых файла: `test_sum` и `test_sum_omp`. Для визуализации результатов замеров времени выполняем следующую команду

```
./test_sum_omp | python plot.py
```

Пример получившихся графиков можно видеть на рисунках 1, 2 и 3. Обратите внимание, что вам необходимо выяснить сколько параллельных потоков поддерживает ваш процессор и указать в исходном коде программы `test_sum_omp` соответствующее число. Код программы `plot.py` специально разбирать не будем. При желании за пояснениями можете обратиться к преподавателю.

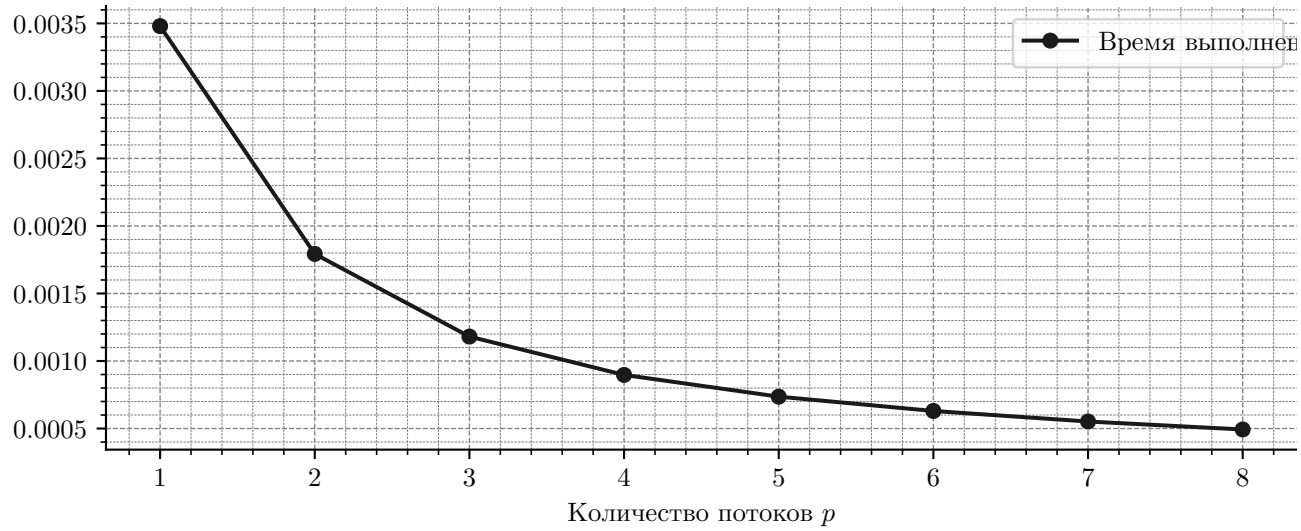

Рис. 1: Время, затраченное на вычисление суммы элементов массива в зависимости от количества порожденных потоков

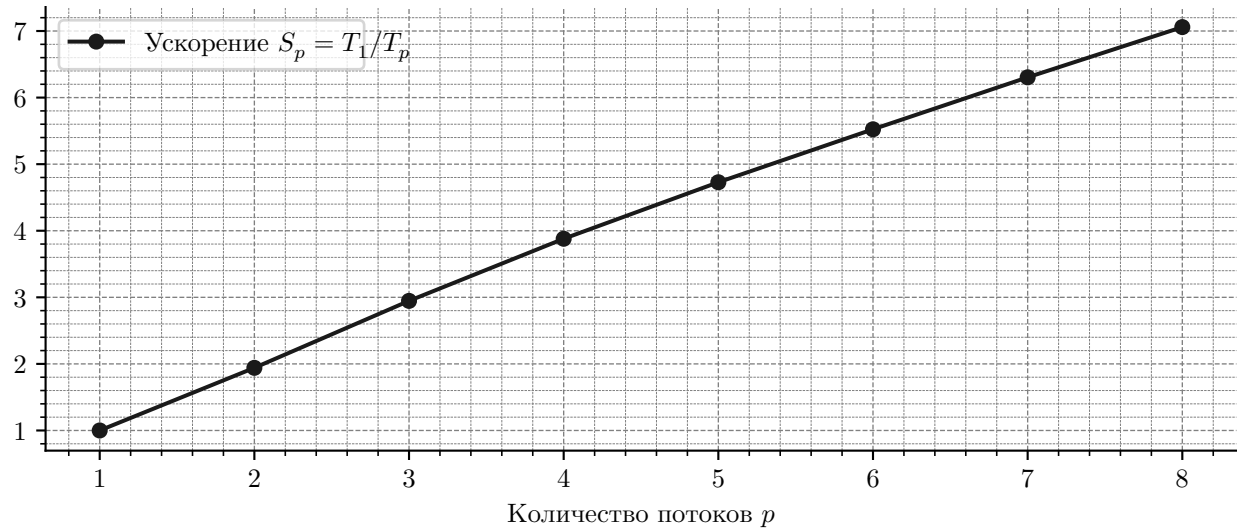

Рис. 2: Время, затраченное на вычисление суммы элементов массива в зависимости от количества порожденных потоков

Рис. 3: Время, затраченное на вычисление суммы элементов массива в зависимости от количества порожденных потоков