РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

Отчет по лабораторной работе № 1

Дисциплина: Параллельное программирование

Студент: Логинов Сергей Андреевич

Группа: НФИбд-01-18

МОСКВА 2021г

Задание

Задание №1

- Напишите простейшую программу, которая ждет некоторое время, например 1 секунду. Для ожидания используйте встроенную процедуру ***sleep*** с аргументом 1.
- Замерьте время ее работы N раз с помощью ***omp_get_wtime( )***. Распечатайте результаты замеров времени, например для N = 10. Убедитесь, что почти всегда полученное время отличается от 1.
- Проведем N = 10^6 замеров времени. Вместо процедуры sleep необходимо использовать произвольные вычисления в цикле.
- Замерьте время работы новой подпрограммы для N = 10^6.
- Вычислить среднее значение и выборочную дисперсию от полученных замеров.

Сперва напишем подпрограмму wait, которая реализует функцию sleep с переданным в подпрограмму аргументом:

```
subroutine wait(i) ! подпрограмма, реализующая "сон" на 1 секунду

    implicit none

    integer, intent(in) :: i

    call sleep(i)

end subroutine wait
```

Далее в теле основной программы main проведем замеры времени работы подпрограммы в цикле

```
! замеряем время работы подпрограммы wait
    do i=1,10,1
        t1 = omp_get_wtime()
        call wait(1)
        t2 = omp_get_wtime()
        print *,i, "замер : ", t2-t1
    end do
```

Результат выполнения:

```
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % gfortran -fopenmp lab
01_1.f90
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % ./a.out
           1 замер :    1.0025540003553033
           2 замер :    1.0048519996926188
           3 замер :    1.0004489999264479
           4 замер :    1.0040889997035265
           5 замер :    1.0019970005378127
           6 замер :    1.0050290003418922
           7 замер :    1.0050550000742078
           8 замер :    1.0007750000804663
           9 замер :    1.0050249993801117
          10 замер :    1.0010899994522333
```

Действительно, результаты во всех десяти замерах отличаются от единицы.

Далее проводим N = 10^6 замеров времени, функцию sleep заменим на вычисление экспоненты и натурального логарифма от переменной, в которую записывается системное время. Все значения наших измерений запишем в массив arr (понадобится для дальнейшего задания). Не забудем в самом начале выделить память под массив.

```
! проведем большее число замеров (10^6), результаты будем записывать
! в массив для дальнейшей работы с его элементами
    N = 10**6
    if( .not. allocated(arr) ) allocate(arr(1:N)) ! выделяем память(с
проверкой выделения) под массив
    do i=1, N
        t1 = omp_get_wtime()
        summ = exp(t1) + log(t1) ! произвольные вычисления
        t2 = omp_get_wtime()
        arr(i) = t2-t1  ! запись в массив
    end do
```

От полученных замеров требуется найти среднее значение и выборочную дисперсию.

Найдем среднее значение в массиве замеров:

```
med = sum(arr) / size(arr) ! находим среднее значение массива
```

Формула выборочной дисперсии:

$$S_n^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \dot{x})^2, \text{ где } \dot{x} - \text{выборочное среднее}$$

Разобьем формулу на части. Для начала заполним массив значениями под знаком суммы. Далее проведем суммирование и поделим на N.

Заполняем массив, не забудем выдлить под него память:

```
if( .not. allocated(arr_disp) ) allocate(arr_disp(1:N)) ! второй массив для поиска дисперсии, в нем будут элементы вида
    !(Xi - X`)^2(часть формулы дисперсии), где Xi - замер времени выполнения вычислений из прошлого цикла а X` - найденное среднее значение
    do i=1, N
        arr_disp(i) = (arr(i) - med)**2
    end do
```

Далее вычисляем дисперсию. Просуммируем элементы прошлого массива и поделим эту сумму на N, после чего вычислим корень от полученного значения:

```
disp = sqrt(sum(arr_disp)/N) ! дополним формулу и найдем дисперсию
```

Осталось распечатать полученные значения. Результат:

```
Median =   4.7870090231299399E-008
Disp =   2.1879373013270343E-007
```

Задание №2

- Создайте программу с параллельной областью. Распечатайте количество созданных потоков. С помощью какой функции это делается?
- Реализовать динамическое изменение потоков в двух вариантах: переменная окружения и функция omp_set_num_threads
- Узнать и создать оптимальное количество потоков для своего процессора
- Создать многопоточную программу с четырьмя потоками, которая будет суммировать входящий массив.
- Протестировать программу из пункта выше.

Для начала воспользуемся функцией omp_get_num_threads чтобы узнать, сколько потоков создается по умолчанию:

```
 !$omp parallel ! начало параллельной области
        n = omp_get_num_threads() ! смотрим, сколько потоков создается
по умолчанию
    !$omp end parallel
    write(*, *) "Количество созданных потоков: ", n
```

Вывод:

```
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % gfortran -fopenmp lab
01_2.f90
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % ./a.out
 Количество созданных потоков:           8
```

Далее проверим максимальное количество потоков для моего процессора:

```
program max_threads
    use iso_fortran_env
    use omp_lib
    implicit none
    write(*, *) omp_get_max_threads()
end program max_threads
```

Вывод:

```
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % gfortran -fopenmp tma
x.f90
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % ./a.out
           8
```

Теперь изменим переменную окружения и запустим программу заново (без компиляции)

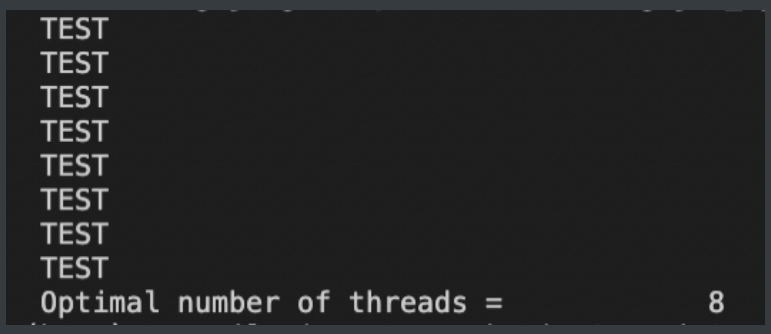
```
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % export OMP_NUM_THREAD
S=4
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % ./a.out
           4
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % unset OMP_NUM_THREADS

(base) sergejloginov@MacBook-Air-Sergej 2_openmp %
```

Через функцию реализовать не удалось

Далее напишем программу, в которой создается оптимальное количество потоков:

```
program optimal
    use iso_fortran_env
    use omp_lib
    implicit none
    integer :: n
    n = omp_get_num_procs()
!$omp parallel num_threads(n)
    write(*, *) 'TEST'
!$omp end parallel
    write(*, *) "Optimal number of threads = ", n
end program optimal
```

Вывод:



Создадим модуль, включающий нашу функцию для суммирования через потоки:

```
module subs ! отдельный модуль для функции суммы через потоки
    implicit none
    contains
    integer function summ_by_tr(intarray, n)
        include "omp_lib.h"

        integer, intent(in) :: n ! размер массива
```

```
        integer, intent(in), dimension(1:n) :: intarray ! входной
массив
        integer :: thread, i
        integer :: base(4) ! вспомогательный массив, в которой свое
значение суммы будет записывать каждый поток

        base = [0, 0, 0, 0] ! изначально элементы равны нулю
        !$omp parallel num_threads(4) ! начало параллельной области
        thread = omp_get_thread_num() + 1 ! тк нумерация потоков
начинается с нуля, нам необходимо увеличить номер на единицу для
корректной
                                          ! работы с массивом

        do i = thread, n, 4 ! шаг 4 тк 4 потока
            base(thread) = intarray(i) + base(thread) ! суммируем
элементы
        end do
        !$omp end parallel ! выход из параллельной области
        summ_by_tr = sum(base) ! сумма через потоки
    end function
end module subs
```

Далее напишем подпрограмму-тестер, в которой будем вызывать функцию *summ_by_tr* и сравнивать результат с библиотечной функцией *sum*:

```
subroutine test() ! подпрограмма для тестирования
    use iso_fortran_env
    use subs    ! подключили модуль с нашей функцией
    implicit none
    integer :: i, n
    integer, allocatable :: arr(:) ! динамический массив со значениями
    n = 100
    if( .not. allocated(arr) ) allocate(arr(1:n)) ! выделяем память(с
проверкой выделения) под массив
```

```
    do i = 1, n
        arr(i) = i
    end do

    if (summ_by_tr(arr, n) == sum(arr)) then ! проверяем, равна ли
сумма через потоки сумме, найденной через библиотечну функцию sum
        write(*, *) "True"
    else
        write(*, *) "False"
    end if

    write(*, *) "Сумма через потоки = ", summ_by_tr(arr, n)
    write(*, *) "Сумма через библиотечную функцию = ", sum(arr)

    deallocate(arr) ! освободим память
end subroutine
```

Вывод:

```
True
Сумма через потоки =          5050
Сумма через библиотечную функцию =          5050
```

Задание № 3

- Реализовать вычисление произвольной функции от массива двумя способами: поэлеметно в цикле и с помощью передачи массива-аргумента в функцию.
- Провести замеры времени выполнения. Второй способ должен дать выигрыш в производительности.

Вычислять будем следующую функцию:

$$\sqrt{e^{sin(x)}}$$

Для начала заполним два массива значениями типа *real*, чтобы в дальнейшем не было проблем с вычислением нашей функции:

```
program main
    use iso_fortran_env
    implicit none
    include "omp_lib.h"
    real, allocatable :: args(:), fullarr(:) ! объявим два динамических массива
    real(Real64) :: t1, t2, time, time1 ! переменные для замера времени
    integer :: i, n

    n = 100

    if( .not. allocated(args) ) allocate(args(1:n)) ! выделяем память(с проверкой выделения) под массив
    if( .not. allocated(fullarr) ) allocate(fullarr(1:n)) ! выделяем память(с проверкой выделения) под массив

    do i = 1, n                    ! заполняем начальные массивы значениями типа real
        args(i) = i + 1.0
        fullarr(i) = i + 1.0
    end do
```

Далее поэлеметно применяем функцию к элементам первого массива, замеряя время на выполнение:

```
t1 = omp_get_wtime() ! первый замер времени
    do i = 1, n
        args(i) = sqrt(exp(sin(args(i))))  ! поэлементно вычислили
значения в первом массиве
    end do
    t2 = omp_get_wtime()
    time = t2 - t1 !    временные затраты на поэлементное вычисление
    write(*, *) "Поэлементно :", time
```

Затем в нашу функцию передаем второй массив в качестве аргумента и замеряем время на выполнение:

```
t1 = omp_get_wtime() ! второй замер времени
    fullarr = sqrt(exp(sin(fullarr))) ! в данном случае наша функция
будет применена ко всем элементам массива без использования цикла
    t2 = omp_get_wtime()
    time1 = t2 - t1 ! временные затраты на вычисление функции с
массивом в виде аргумента
    write(*, *) "Массив в виде аргумента :", time1
```

Проверим результаты наших замеров:

```
if(time > time1) then       ! проверка
        write(*, *) "Второй способ дал выигрыш в проивзодительности"
end if
```

Вывод программы:

```
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % gfortran -fopenmp lab
01_3.f90
(base) sergejloginov@MacBook-Air-Sergej 2_openmp % ./a.out
 Поэлементно :   2.7000904083251953E-005
 Массив в виде аргумента :   4.0000304579734802E-006
 Второй способ дал выигрыш в проивзодительности
```

Второй способ действительно дал выигрыш в производительности.

Задание № 4

- Реализовать функцию *map*, которая в качестве аргументов принимает функцию заданного вида и массив произвольной длины. Затем она поэлементно применяет переданную функцию к каждому элементу массива и возвращает результат этого применения.
- Реализовать этот же функционал с помощью элементарной функции.

Стоит оговориться, что в коде реализована функция вида x^2, тк была не совсем понятна формулировка "полином от действительного числа". В любом случае функция меняется в несколько строчек кода, поэтому при необходимости ее можно заменить.

Для начала напишем модуль *main_fun*, в котором будет описана сама функция map. Также необходимо описать абстрактный интерфейс функции, которую будем передавать в качестве аргумента:

```
module main_fun     ! в этом модуле опишем саму функцию map
    implicit none
    abstract interface  ! задаем абстрактный интерфейс для функции-
аргумента
        pure function f(x)
            real, intent(in) :: x
            real :: f
        end function f
    end interface
    contains
    function map(func, x)   ! описываем функцию map
        implicit none
        procedure(f) :: func    ! функция аргумент
        real, intent(in), dimension(1:) :: x    ! входной массив
```

```
        real, dimension(1: size(x)) :: map  ! возвращаем массив как
    результат функции
        integer :: n, i

        n = size(x)

        do i = 1, n
            map(i) = func(x(i))
        end do

    end function
end module main_fun
```

В модуле *side_function* реализуем функцию, которую будем передавать в качестве аргумента (реализация в соответствии с объявленным ранее интерфейсом), для передачи в качестве аргумента она должна быть чистой:

```
module side_function    ! описываем вспомогательную функцию в этом
модуле
    contains
    pure function funcc(x)  !вспомогательная функция
        implicit none
        real, intent(in) :: x
        real :: funcc
        funcc = x**2
    end function funcc
end module side_function
```

В модуле *elemental_function* реализуем элементарную функцию *elem*.

```
module elemental_function   ! элементарная функция для второго задания
    implicit none
    contains
    elemental function elem(x)
        real :: elem
        real, intent(in) :: x
        elem = x**2
    end function
end module
```

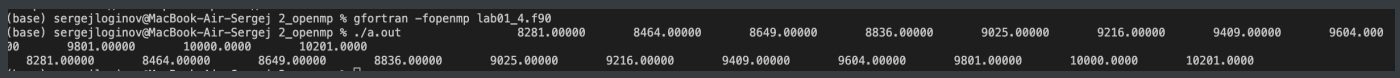
В самой программе объявляем необходимые переменные и массивы: x - вспомогательный с элементами типа real, от которого будем производить дальнейшие вычисления, y - для хранения значений, полученных через функцию *map*, z - для хранения значений, полученных из элементарной функции *elem*.

Для проверки выведем элементы массива с индексами 90-100:

```
program main
    use side_function
    use main_fun
    use elemental_function
    implicit none
    real, allocatable :: x(:), y(:), z(:)
    integer :: i, n
    n = 100
    if( .not. allocated(x) ) allocate(x(1:n)) ! выделяем память(с
проверкой выделения) под массив
    if( .not. allocated(y) ) allocate(y(1:n)) ! выделяем память(с
проверкой выделения) под массив
    if( .not. allocated(z) ) allocate(z(1:n)) ! выделяем память(с
проверкой выделения) под массив
    do i = 1, n
        x(i) = i + 1.0
    end do
    y = map(funcc, x)
```

```
        write(*, *) y(90:100)
        z = elem(x)
        write(*, *) z(90:100)
    end program main
```

Вывод программы:



Получили одинаковые результаты.