3. Программирование на Фортране. Немного теории

> Всегда пишите код так, будто сопровождать его будет склонный к насилию психопат, который знает, где вы живете.
>
> **Мартин Колдинг**

> Теория — это когда всё известно, но ничего не работает. Практика — это когда всё работает, но никто не знает почему. Мы же объединяем теорию и практику: ничего не работает... и никто не знает почему!
>
> **Альберт Эйнштейн**

Если кто-то изучал Фортран ещё в советские времена, то он найдёт очень много и, честно признаемся, весьма приятных и удобных изменений в современном варианте Фортрана. Основные принципы программирования не изменились, но вот реализация этих принципов зачастую изменилась и, в некоторых случаях, весьма кардинально. Это связано с тем, что появившиеся позднее Фортрана многие языки программирования реализовывались на принципе «программист — балбес, за ним нужен глаз да глаз». Так что многие из балбесоспасательных принципов появилось и в Фортране. Поэтому прежде чем подступиться к примерам программ, просто необходимо кратенько описать, а каким образом собственно эти программы будут писаться. И вообще, как их нынче положено писать на Фортране.

Хочу сразу предупредить, что Фортран за свою долгую и весьма бурную историю накопил довольно много форм проделывания одного и того же дела. Я не буду описывать их все, т. к. для этого придётся писать целое собрание сочинений, а приведу только те формы, которые считаются современными на данном историческом этапе.

3.1 С чего начинается и чем заканчивается программа

> Всесоюзная Коммунистическая партия (большевиков) прошла долгий и славный путь от первых маленьких марксистских кружков и групп, появившихся в России в 80-х годах прошлого столетия, до великой партии большевиков, руководящей ныне первым в мире социалистическим государством рабочих и крестьян.
>
> **Краткий курс истории ВКП(б), 1939 год**

Вот как выглядит программа до того, как программист внесёт туда что-нибудь полезное:

```
! Комментарии, в которых описывается
! для чего нужна программа, каковы
! её особенности и принципиальные отличия
! от других, весьма многочисленных собратьев
program MyProgram

end program MyProgram
```

Программа начинается со служебного слова «program», потом идёт название программы, а заканчивается служебными словами «end program» и опять название программы. Те строки кода, которые начинаются с символа «!» являются комментариями и транслятором игнорируются.

Справедливости ради нужно сказать, что по фортрановским стандартам единственным необходимым словом здесь является слово «end». Всё остальное желательно, но не необходимо, поскольку нужно не транслятору, а самому программисту, чтобы не запутаться в собственном коде. Да-да, вы не ослышались, именно программисту. Дело в том, что сегодня программы из пяти-шести строчек — большая редкость. Обычно тексты программ начинаются от обеда и уходят в такую бесконечность, откуда без комментариев (иногда матерных) и прочих облегчающих понимание

дополнений, вернуться будет весьма проблематично. Поэтому всё это присутствует в исходных кодах программ для того, чтобы сам программист (или кто-то после него) спокойно ориентировался в текущем моменте кода. Именно поэтому использовать только слово «end» в конце программы или подпрограммы не стоит, чтобы в случае возникновения ошибки вы точно знали, к чему конкретно «end» относится.

Одна из особенностей Фортрана — регистронезависимость кода программ. Можно написать «program», а можно и «Program», это будет одно и тоже слово. В некоторых случаях позволяет избежать ошибок, связанных с написанием переменных или функций.

О комментариях. В отличие от таких языков как Си, Паскаль или Джава, в Фортране нет символов-ограничителей многострочных комментариев. Это не упущение разработчиков. Дело в том, что в Фортране строка с командой заканчивается там, где заканчивается сама строка, а не там, где стоит символ конца строки, как например в Си. Правда, если в одну строку всё не влезает, то ставится специальный символ продолжения — «&» и на следующей строке идёт продолжение того, что не влезло. Но вообще, позиция разработчиков — желательно чтобы всё, что относится к чему-то одному, влезло в одну строку, а не устраивать из программы бесконечность.

Кстати говоря, есть и предельная длина строки кода программы в Фортране — 132 символа. В старом Фортране было 80 символов. Если в одной строке кода вы не уложитесь в эти самые магические 132 символа, транслятор вас мягко пожурит и, слегка стесняясь, напишет: мол извините, панове, лишнее я вам отрежу. Так что имейте в виду. Чем определена именно такая длина строки я не знаю, но ничего неразумного тут не вижу, т. к. слишком длинную строку читать неудобно и больше 60 … 70 символов в строку помещать, по моему мнению, нецелесообразно. В длинной строке видишь голову и не видишь хвост, а если видишь хвост — не видишь голову. Если одна строка команд получается излишне длинной (бывает такое иногда) то код можно писать в несколько строк, в конце незаконченной строки ставя символ «продолжение следует...» - «&», тогда при компиляции программы компилятор строку с таким символов объединит со следующей строкой.

Ходят слухи, что именно такая длина (132 символа) определялась длиной строки, которую могла напечатать старая АЦПУ (алфавитно-цифровое печатающее устройство) на бумажном листе формата А3. Посмотрев в интернете я понял, что возможно это и не слухи. Часть АЦПУ печатает 128 символов в строке (ЕС-7030 и ЕС-7032), одно аж целых 156 (ЕС-7031) и два как раз 132 символа (ЕС-7033, СМ-6315). ☺

3.2 Типы данных и их преобразование

Базовых типов данных в Фотране всего пять штук:

- INTEGER — целочисленные, например 1, 2, 145, -18 ;
- REAL — числа с плавающей точкой (или, для Европы — запятой, но поскольку американские языки программирования используются в подавляющем большинстве, то ничего, кроме точки), например, 3.141592;
- CHARACTER — символьные, например, «Привет, чуваки и чувихи!». Символьные типы ограничиваются с двух сторон либо одинарными кавычками, либо двойными, разницы между ними нет. Главное, чтобы кавычка в конце строки была такая же, что и в начале. Это довольно удобно. Если в тексте вам нужно приводить кавычки, то они просто не должны быть такие же, как в ограничителе строки;
- LOGICAL — тут всего два значения — .true. и .false. ;
- COMPLEX — комплексный тип, состоит из двух частей — вещественной и мнимой, например, (3, 5). Данные этого типа представляются в скобках.

Модификации типов «знаковый» и «беззнаковый», как это есть в Си или Паскале, здесь совершенно отсутствует. Возможно это кому-то покажется недостатком, однако хочу напомнить, что в математике тоже нет отдельного набора чисел, которые без знака. Не ссылайтесь на слово «положительные» или «отрицательные», это к типу не относится и является всего лишь одной из характеристик чисел. Хотя с другой стороны, если вы работаете только с положительными

числами, то количества положительных будет в два раза меньше, чем могло бы влезть в этот тип данных, если бы он был беззнаковым.

Типы могут быть и составные, по аналогии с сишным «struct» и паскалевским «record». Например:

```
type student
        character(40)  :: name
        integer        :: group
        real           :: average
end type
```

Классы мы рассматривать не будем, потому что вряд ли это представляет интерес применительно к Фортрану, но объявляются они аналогично составному типу.

Переменные в Фортране можно объявить множеством разнообразнейших способов, вплоть до такой интересной ситуации, когда переменные можно вообще не объявлять. Сишники немедленно по этому поводу начнут ехидно хихикать и возбуждённо потирать ручки восклицая: «Ага! Мы же вам говорили, говорили...» и будут как всегда правы, и как всегда только в определённом, узком смысле. Дело в том, что Фортран создавался как язык высокого уровня (в отличие от Си ^), с которым будут работать непрограммисты. Математики же не забивают себе мозги размышлениями о типах данных. Вот и пусть транслятор там внутри себя с ними разберётся сам. Это же ему надо, не нам…

Если серьёзно, то для махонькой программки в четыре-пять строчек заморачиваться над типами и правда не стоит. Я даже знаю один язык программирования, чрезвычайно похожий на Си, ну просто одно лицо, где с типами переменных тоже не заморачиваются. Зовут этот язык PHP. Однако, бывают такие ситуации, когда верность результата вычисления сильно зависит от типа данных. И тут, конечно, над типами надо серьёзно поразмышлять. Мало того, транслятору надо позаботится, чтобы у программиста не вышла путаница с типами. Как раз на этот случай, в первой строке программы (после названия) стоит прописать специальную команду:

```
IMPLICIT NONE
```

тогда компилятор будет строго следить за программистом, чтобы у него там случайно ничего не выскочило левого, помимо объявленного в переменных.

В общем случае переменная объявляется так:

```
Тип_данных[(Размер_типа)] [, модификаторы] :: Имя_переменной [ = начальное_значение_если_нужно ]
```

Если не задан размер типа в байтах (4, 8, 16, а в некоторых крутых трансляторах, типа CRAY, даже и 32), то размер будет выставлен по умолчанию, который обычно равен 4 байта, если вы вручную не задали компилятору другой умолчальный размер.

В модификаторах наиболее часто используется слово «**PARAMETER**», который означает, что вы объявили неизменяемую переменную, т. е. константу.

Иногда возникает необходимость сохранения значения переменной объявленной внутри функции между вызовами этой функции. В этом случае применяется модификатор «**SAVE**».

Длина «Имя_переменной» в Фортран-90 (а мы в подавляющем большинстве случаев будем придерживаться этого стандарта) может быть до 31 символа.

Если не присвоить начальное значение, то в переменной будет какой-нибудь мусор. Для константы (которая «parameter») значение в виде мусора будет выглядеть весьма креативно, но совсем не эффективно, так что не забывайте константам присваивать значение.

Примеры:

```
integer :: i          ! Целочисленная переменная,
                      ! с умолчальным размером в 4 байта
real :: r             ! С плавающей точкой, размер — см. выше
real(16) :: rr        ! А это уже с размером в 16 байт,
                      ! соответствует типу extended в Си
```

```
logical :: k = .false.     ! Переменная с начальным значением

! Ниже константа размером в 8 байт
real(8), parameter :: my_pi = 3.14159265358979323846_8

! Символьные
character(6) :: hello1 = "Привет,"
character(9) :: hello2 = 'девчонки!'

! Переменная составного типа
type (student) :: I_Am
```

Особо следует сказать про размер символьных переменных. Их размер, как ни странно, задаётся не количеством символов, а количеством байт. У тех, кто позабыл, что многие современные ОС и языки программирования работают на кодировках типа UTF, это может вызвать появление в программах обрезанных строк. Поэтому перед тем, как отдавать программу в работу, надо обязательно выяснить, а сколько байт в используемой ОС содержит 1 символ. Для русского языка в ОС Linux и Windows обычно используют 2 байта. ОС типа BSD пока ещё используют 1 байт на символ, однако если в них включена поддержка UTF, то байт на символ будет больше. Поэтому максимальное количество символов, которые может содержать строка, надо обязательно умножить на 2.

Если вообще не указывать размер, то по умолчанию будет всего лишь один байт. При неизвестной длине строки в скобках, в качестве размера, нужно указать звёздочку. Однако такое упрощение годится исключительно для констант. А вот «безразмерных» строк-переменных в Фортране пока нет. Пример, который приведён ниже, даёт указание компилятору, что длина строки будет посчитана при фактическом присваиваемом значения:

```
character(*), parameter :: name = "Иванов Иван Петрович"
```

Отдельное слово про константы. Дело в том, что если мы объявили константу определённого типа с неумолчальным размером и присвоили ей какое-то значение, то у значения размер будет умолчальный (т. е. 4 байта), что сводит на нет задание размера у типа. Нет, конечно же само число останется таким же, как и было, вот только и ошибки округления будут как в 4-ёхбайтном числе. Объяснения этому в стандартах я не нашёл. Сия особенность касается всех чисел, которые обитают в программе до того, как они присвоются какой-либо переменной или константе. Чтобы размер значения константы соответствовал размеру типа, к числу добавляется подчёркивание и тот размер, который требуется, как это сделано в примере выше.

Для числовых констант с плавающей запятой (или их представления в научном формате) есть другой способ задания точности:

```
! Скорость света, real(4) → 2,99 * 10^8
real, parameter :: c = 2,99792458E8

! Гравитационная постоянная, real(8) → 6,67408 * 10^-11
real(8), parameter :: G = 6,67408D-11

! Постоянная Планка, real(16) → 6,626070040 * 10^-34
real(16), parameter :: Plank =  6,626070040Q-34
```

В первом числе буква «E» перед показателем степени (8) обозначает, что это число обычное, одинарной точности, т. е. размер 4 байта. Во втором числе стоит буква «D», которая обозначает, что это число двойной точности (double precision), т. е. размер 8 байт. А в третьем числе стоит буква «Q», обозначающее, что это число аж четверной точности (quad precision), т. е. размер 16 байт.

Вот такие пироги…☺

А ещё бывает такая ситуация, когда вы не знаете, какой размер у типа должен быть в программе. Этот размер можно подсчитать с помощью функций SELECTED_INT_KIND(a) для

целочисленного типа и SELECTED_REAL_KIND(a, b) для плавающей точки. Для первой функции параметр «a» - количество значащих цифр, для второй b — значение степени числа. Вот пример, целочисленный тип с пятью значащими цифрами, а вещественный — с пятью значащими цифрами и степенью от -99 до +99:

```
integer, parameter    :: kind1 = SELECTED_INT_KIND(5)
integer, parameter    :: kind2 = SELECTED_REAL_KIND(5, 99)
integer(kind1)        :: ogo
real(kind2)           :: aga
```

Отдельно нужно сказать про возможность вообще предварительно не объявлять переменные. Хотя такой подход считается небезопасным, но в современных стандартах (Фортран-2008 и Фортран-2018) пока ещё не отменён. И в грядущих стандартах тоже не слышно, чтобы эту возможность собирались отменять. Ходят слухи, что этой проблемой займутся в стандарте 202x, но это только слухи…

Изначально в Фортране была возможность, когда вновь вводимые в ходе работы программы переменные (не объявленные заранее) получали тип «по-умолчанию». Умолчальные типы определялись так:

- Переменными целого типа (INTEGER) будут считаться те, имена которых начинаются с букв «I», «J», «K», «L», «M», «N»;
- Переменные начинающиеся с любых других букв будут считаться типом с плавающей точкой (REAL).

«А как же другие типы?» спросите вы. А никак! ᴍ Умолчания для других типов не предусмотрено, их уже надо объявлять явно. Умолчальные типы вы можете назначать и сами с помощью оператора «IMPLICIT», например:

```
! Все переменные начинающиеся с буквы x,y,z - комплексные
IMPLICIT COMPLEX (X,Y,Z)

! Все переменные начинающиеся с буквы L - логические
IMPLICIT LOGICAL L

! Все переменные начинающиеся с букв a, b, c – символьные
IMPLICIT CHARACTER (A-C)
```

Но всё же лучше, во избежание непонятных ошибок, такой умолчальной типизацией переменных не пользоваться.

Преобразования численных данных из одного типа в другой, тоже численные, дело простое и лёгкое:

```
integer :: i, i1, i2
real :: r, r1, r2
complex :: cl

! Преобразовать целое число в нецелое
i = 5
r = real(i)

! Целое в комплексное (должно быть два целых)
i1 = 7
i2 = 4
cl = complex(i1, i2)
! или, в старых стандартах
cl = cmplx(i1, i2)
```

```
! Нецелое в комплексное
cl = complex(r1, r2)
! или, в старых стандартах
cl = cmplx(r1, r2)
```

Для преобразования нецелых чисел в целое функций очень много, т. к. нужно учитывать округление числа (вверх, вниз) или то, что у комплексного числа составляющих не одно, а целых два:

```
! Если нужна только целая часть, без округления
i = int( r )
! Действенная часть комплексного числа
i = int(cl)
! Округление в большую сторону
i = ceiling(r)
! Округление в меньшую сторону
i = floor(r)
```

Если размер типа у нас нестандартный, то вторым аргументом в эти функции можно подставить необходимый размер:

```
integer(8) :: i

i = int( r, 8)
```

Для преобразования чисел в строку специальных функций нет, т. к. для этого, так же как и в Си, используются команды ввода\вывода с указанием формата изображаемого числа:

```
! Целое число в строку (формат вывода можно не указывать,
! если вас не волнуют могущие там появиться пробелы)
character(20) :: cc

write (cc, *) i
write (cc, "(F10.8)") r
```

3.3 Массивы

Массивы — это самая любимая переменная в Фортране. Большая часть вычислений проводится с большими объёмами однотипных данных, которые как раз и представляются в виде массива. Объявляется массив так (к примеру, целочисленный):

```
integer, dimension(25,25,25) :: barmaley
```

То, что это будет именно массив, говорит модификатор «dimension» после чего в скобках задаётся размерность массива — он трёхмерный, по 25 индексов в каждом измерении. Индексы по умолчанию начинаются как и в математике с 1, но если вам сильно хочется, то начальный индекс вы можете задать сами.

Вот интересно, массив в Фортране как-то очень уж подозрительно похож на функцию или процедуру, но мы по такому малосущественному вопросу кукситься не будем… ☺

Если необходимо инициализировать весь массив каким-то одним числом, делается это просто:

```
barmaley = 1
```

Всем ячейкам, по всем трём измерениям присвоится 1. Ещё один интересный способ инициализации массива — с помощью неявного цикла:

```
integer :: i
! Используя переменную i присваиваем массиву числа от 1 до 10
integer, dimension(10) :: mm1 = ( / (i, i = 1,10) / )
! А здесь присваиваем только нечётные числа
integer, dimension(10) :: mm1 = ( / (i, i = 1,20,2) / )
```

Аналогично можно инициализировать и многомерные массивы. При этом предполагается, что ячейки идут подряд, друг за другом. На самом деле так и есть, только у массива есть и куча скрытых данных для служебного пользования. Надо предупредить, что в отличие от сишного двумерного массива где будет сначала заполнена первая строка, потом вторая и так далее, в Фортране массивы идут сверху вниз, т. е. сначала заполняется первый столбец, потом второй и так далее. Сишным и паскалевским программистам об этом нужно помнить.

Вообще с массивами можно делать любые математические операции, как с простой переменной, если конечно массивы позволяют с собой это делать, например:

```
integer, dimension(25,25,25) :: barmaley
integer, dimension(25,25,25) :: aybolit

barmaley = 8
aybolit = 9

print *, barmaley + aybolit
print *, sin(real(barmaley))
print *, cos(real(aybolit))
```

В приведённом выше коде все операции поэлементные, т. е. при сложении значение каждой ячейки одного массива складывается с соответствующем значением другого массива, а операция синуса применяется к каждой ячейке массива. А вот если надо, к примеру, умножить именно массив на другой массив, то нужно использовать специальную функцию:

```
print *, matmul(barmaley, aybolit)
```

Ещё один интересный момент про массивы. Не всегда изначально известно, сколько индексов должно быть у массива. Ничего страшного, можно объявить безразмерный массив, а после того, как размер станет известным — образмерить его:

```
! Безразмерный массив на 2 измерения
real(16), allocatable, dimension(:,:) :: sary_agach
integer :: i,j

! Без изысков, псевдослучайные числа
call random_seed()
call random_number(i)
call random_number(j)

! Выделяем память для массива
allocate(sary_agach(i,j))
```

Здесь добавлен модификатор к объявлению массива «allocatable», т. е. подразумевается, что память перед использованием массива будет выделена с помощью оператора «allocate». Если использовать массив больше не нужно то память стоит освободить с помощью:

```
deallocate(sary_agach)
```

В отличие от Си, массив можно передать как параметр в функцию или процедуру не указывая ни его размерность, ни количества индексов, т. к. всё это в скрытом виде содержится в самом массиве и определяется соответствующей функцией:

```
real, dimension(45,45) :: mm
integer :: s, i, u

! Общее количество элементов массива
s = size(mm)
! Нижняя граница (границы) массива
l = lbound(mm)
! Верхняя граница (границы) массива
u = ubound(mm)
```

Вторая и третья функции возвращают одно число, если массив одномерный и массив значений, если массив многомерный (размер выходного массива будет равен количеству измерений исследуемого массива).

Тут нужно заметить, что использование дополнительных функций делает подпрограмму более медленной. Поэтому используемые в ней свойства массива лучше передавать в качестве параметров, а не определять по ходу действий.

Можно пользоваться не всем массивом, а какой-то его частью. Это похоже на цикл с известным количеством повторений:

```
massiv([Начало]:[Конец][:Шаг])
```

Например:

```
integer, dimension(10) :: a

print *, a(5:7)
```

выведет на экран значения ячеек с 5 по 7 включительно. Шаг по умолчанию равен 1.

```
print *, a(:4)
```

выведет значения ячеек с 1 по 4 включительно.

Чрезвычайно интересное свойство Фортрана — присваивание значений элементам массива по маске (или по условию). Делается это с помощью операции WHERE:

```
real, dimension(10) :: a

CALL random_seed()
CALL rundom_number(a)

print '(10(X1F5.3)', a

where (a>0.5)
        a = 1.0
elsewhere
        a = 0.0
end where

print '(10(X1F5.3)', a
```

Несмотря на то, что кажется, будто значение здесь присваивается массиву целиком, на самом деле происходит поэлементное присваивание: если значение элемента массива больше чем 0.5, то ему присваивается 1.0, если же это не так, то 0.0.

3.4 Ввод\вывод данных

Согласитесь, без этого труд программиста теряет не только определённую изысканность, но и вообще какой либо смысл. Фортран изначально имел для вывода команду:

```
write (Номер_устройства, Формат_представления) Что-то_выводим
```

а для ввода:

```
read (Номер_устройства, Формат_представления) Куда_вводим
```

Здесь, **Номер_устройства** — это то устройства (экран, принтер, файл), в которое (или из которого) будет вводиться (или браться) информация. Есть несколько предопределённых номеров для разных стандартных устройств, для файлов тоже используются номера. Если вместо номера устройства поставить звёздочку, то транслятор воспримет это как устройство по умолчанию, т. е. для вывода будет использоваться экран, а в качестве устройства ввода — клавиатура. Это серьёзно упрощает программу.

```
Integer :: MyYears

write (*,*) "Будьте добры, сообщите свой возраст, сударь:"
read (*,*) MyYears
```

Для вывода только на экран есть и специальная команда:

```
print *, "Какой-нибудь высоконаучный текст"
```

Для ввода только с клавиатуры никакого специального оператора почему-то не предусмотрели. ^

«Формат_представления» данных может быть умолчальным (вторая звёздочка в командах write\read и первая в print) или задаваться вручную с помощью специального набора символов. Обычно сам транслятор правильно определяет, где у вас текст, а где числа, но может быть умолчальный вид вам не подойдёт. По умолчанию, числа с плавающей точкой представляются в научном формате, а целые числа имеют длину сколько цифр влезает в заданный размер типа. Иногда числа с плавающей точкой надо представить не в научном формате, а в том, как его пишут обычные люди, не обременённые высшим образованием. Иногда бывают всевозможные смешанные данные, которые в умолчальном формате выглядеть будут крайне предосудительно. Например, если сделать вывод массива по умолчанию, то любое количество ячеек массива будет выведено в одну строку, что, согласитесь, будет выглядеть довольно странно. Ведь, например, матрица должна при выводе иметь вполне определённый формат — заданный изначально количеством столбцов и строк и все ожидают, что на экране (или в файле) так и будет.

Собственный формат ввода\вывода обычно определяется отдельной командой **FORMAT**, которая предваряется номером метки и на этот номер ссылаются команды **READ\WRITE\PRIN**T. Надо сказать, что команда **FORMAT** может стоять в любом месте программного блока, хоть в начале, хоть в конце, естественно, в области видимости операторов ввода\вывода. При использовании своего формата, эти команды ссылаются на номер метки. Можно, конечно, задать строку формата и в самой команде ввода\вывода, но обычно такая строка сложная (а простую никто бы и не задавал, а зачем простое делать сложным? ☺), поэтому, чтобы не загромождать команду строку формата делают отдельно. Вот как это выглядит в общем случае (очень похоже на сишное представление):

```
10 FORMAT TW[.D]
```

Здесь T — тип данных (см. табл. 1);
W — общая длина данных;
D — количество десятичных знаков (или, для целого числа, минимальное количество цифр).

Сведения о наиболее употребительных форматах представления данных смотрите в таблице 2.

Таблица 2. Задание типа данных в формате ввода\вывода

Тип данных	Обозначение	Описание
Целые числа	I	В десятичном виде
	B	В двоичном виде
	O	В восьмеричном виде
	Z	В шестнадцатеричном виде
С плавающей точкой	F	Без экспоненты
	E	С экспонентой. Здесь можно дополнительно задать длину экспоненты в виде «Ечисло»
Текст	A	Здесь длину можно не задавать, она будет определяться фактическим числом символов. Но если нужно ограничение, то строка будет обрезаться на указанном количестве символов или дополнятся пробелами до нужного количества
Логический	L	

Перед символом типа данных можно поставить число количества повторений, таким образом можно строить, например, обрамление для текстовых таблиц. Хотя по нынешним временам это уже неактуально.

Есть и другие служебные символы, например «X» — вставить пробел и «T» — вставить табуляцию, «/» — вставить перевод строки. Их тоже можно использовать с числом повторов.

У команды write есть один интересный именованный параметр, который позволяет отменить перевод строки в конце:

```
Integer :: MyYears

write (*,*, advance="no") "Будьте добры, сообщите свой возраст, сударь:"
read (*,*) MyYears
```

Таким образом мигающий курсор ввода останется в конце фразы-просьбы.

Вывод массива на экран в Фортране — дело совсем пустяковое. В отличие от языков Си или Паскаль, здесь даже не надо использовать циклы. Вот пример для одномерного целочисленного массива:

```
integer, dimension(40) :: mm

….

10 FORMAT "4I10"

print 10, mm
```

Здесь в одной строке идёт повторение 4 целых чисел, у которых длина в 10 символов. Транслятор знает, что выводится массив, поэтому внутри себя применяет цикл до тех пор, пока массив не закончится. Таким образом у нас получается 10 строк по 4 числа в каждом. Правда порою нужно, чтобы между значениями массива был пробел, а то прочитать значения будет трудно, тогда можно сделать так:

```
10 FORMAT "X2(4I10)"
```

Здесь скобки — знак группировки символов, перед каждой группой символов (I10 — десять знаков в каждом числе массива) будет ставится 2 пробела (2X) и так по 4 группы символов в строке. Таким образом перед каждым числом из 10 знаков будет ставится два пробела, отделяя одно значение от другого. Конечно, само число может состоять и не из 10 символов (цифр), но принудительно будет обрезаться до этого числа.

С двумерными (многомерным) массивами слегка посложнее, т. к. нужно учитывать, а влезет ли строка массива на экран и сколько в ней колонок. Если не влезает, то приходится выводить массив по частям, благо в Фортране делать срезы массивов легко, опять же не применяя в явном виде никаких циклов. Но с учётом, что многомерный массив всё равно расположен в памяти последовательно (разделение на колонки и строки — это свойство самого массива, а вовсе не свойство хранилища), у вас есть все средства показать его правильно. Правда к большим матрицам вряд ли можно применить правильный показ их на экране, т. к. по умолчанию консоль всё равно показывает не больше 80 символов подряд, если только вы специально не настраивали свою консоль на вывод 150 символов или даже 200. Ведь каждая ячейка матрицы, в подавляющем большинстве случаев содержит числа не из одного символа. Но вывод в файл матрицы очень легко сделать правильно — по количеству столбцов и строк с необходимыми пробелами между ними, так что чтение глазами, для проверки, будет проходить без малейшего затруднения.

3.5 Подпрограммы, функции, модули

Да-да, именно «подпрограммы», а не «процедуры». ☺ Дело в том, что так исторически сложилось, что процедуры в Фортране всегда назывались «subroutine». Если переводить дословно, то «routine» можно перевести с математической точки зрения (не забывайте, что Джон Бэкус был математиком) как «последовательность команд» или, если взять уже более близкие нам времена и терминологию вычислительной техники — «программа для вычисления чего-то», следовательно «subroutine» будет переводиться как «подпрограмма». И ещё одна историческая особенность — эти процедуры-подпрограммы хранились в отдельных файлах, которые потом можно было использовать, не переписывая код, в других своих программах, надо только указать их в строке компиляции. Так что «подпрограмма» - это самое точное название. Впрочем, если говорить «процедура» никаких недопониманий тоже не будет, т. к. к этому термину все уже давно привыкли. Вдобавок, в новом Фортране все подпрограммы, которые использует ваша программа, можно объявить в самой программе, т. е. И в этом плане термин «процедура» будет вполне к месту.

Пользовательские подпрограммы и функции

Это, как и в других языках, отдельные части программы, которые выполняют какую-то внутреннюю задачу. И если с функциями всё понятно, они также возвращают одно единственное значение, то с процедурами несколько сложнее, т. к. в других языках процедура — это часть программы, которая не возвращает данных, но вот в Фортране изначально повелось, что процедура тоже возвращает данные и даже по нескольку штук сразу. Дело в том, что параметры в процедуру и функцию в Фортране всегда передавались по ссылке, а не по значению. Таким образом в процедуре эти параметры спокойно можно было менять и в основную программу они возвращались уже имея другое значение. В Фортране-90 такую ситуацию попытались разрулить, введя в переменные-параметры специальный модификатор INTENT, который определяет, это параметр входящий (IN), исходящий (OUT) или и то и другое сразу (INOUT). Теперь, если правильно указывать этот модификатор, можно не опасаться нечаянно испортить данные.

У Фортрана в заголовке процедуры\функции указываются только имена параметров, их типы определяются уже в теле самой процедуры\функции, рядом с внутренними переменными. Вот пример для процедуры:

```
subroutine karramba (par1, par2, par3)
integer :: par2
real, dimension(par2) :: par1
logical :: par3

….

end subroutine karramba
```

Здесь в процедуру передаётся три параметра:
par1 — целочисленный массив;
par2 — размер этого массива;
par3 — некий логический параметр.

По умолчанию, если не указан модификатор INTENT, значения параметров можно изменять и в программе, после выполнения процедуры, они будут изменённые. Если же у параметров специально указать INTENT, например:

```
subroutine karramba1 (par1, par2, par3)
integer, intent(IN) :: par2
real, intent(INOUT), dimension(par1) :: par1
logical, intent(OUT) :: par3

….

end subroutine karramba1
```

то параметр par1 вернётся в программу неизменным. Значение второго параметра можно менять, и предварительно в программе и в подпрограмме, в основную программу он вернётся изменённый. В третий параметр передавать перед вызовом процедуры что-либо бессмысленно, т. к. в процедуру он всё равно придёт с неким случайным значением (или, если его в процедуре проинициализировать, то со значением инициализации), но после отработки процедуры там будет именно то значение, какое параметру присвоили в процедуре.

Для функции всё тоже самое, только в заголовке, перед названием, надо указать тип возвращаемого значения, при этом возвращаемое значение присваивается переменной, которая имеет такое же название, как и сама функция:

```
logical function karramba2 (par1, par2)
integer, intent(IN) :: par2
real, intent(INOUT), dimension(par1) :: par1

….

! Возвращаемое значение
if (условие) then
  karramba2 = .true.
else
  karramba2 = .false.
end if

end function karramba2
```

Другой не менее популярный формат заголовка функции — с отдельной переменной результата, которая обозначается служебным словом **result**:

```
function karramba3 (par1, par2) result (res)
integer, intent(IN) :: par2
real, intent(INOUT), dimension(par1) :: par1
logical :: res

. . .

end function karramba3
```

При этом нужно не забыть объявит тип возвращаемой переменной в списке других переменных.

Если вызов функций в Фортране ничем не отличается от вызова в других языках программирования, то вызов процедуры (неважно, собственной или встроенной фортрановской) обязательно должен предваряться служебным словом «CALL»:

```
CALL karramba1(a, b, c)
```

Отношение процедуры\функции к основной программе бывает в двух видах:

1. Процедура\функция принадлежит программе.
2. Процедура\функция не принадлежит программе.

В первом случае процедура\функция «видит» все объекты основной программы, но для программы объекты процедуры\функции — тёмный лес. Такое тесное взаимоотношение оформляется использованием оператора «contains», при этом тело процедуры\функции располагается после кода основной программы, но до слов «end program»:

```
program shara-para

...

contains

 real function introfunc(a, b,c)

 ...

 end function  introfunc

 subroutine introsub(a, b, c)

 ...

 end subroutine introsub

end program shara-para
```

```
program lalala
use mathlib

...
```

end program lalala

Во втором случае, процедура\функция либо лежит в какой-то посторонней уже когда-то скомпилированной библиотеке, либо текст находится в отдельном от программы файле. Для использовании сторонней библиотеки нужно создать модуль с описанием интерфейса к библиотечной функции (как это делается в Паскале). Для использования просто из из другого файла, его нужно компилировать вместе с основной программой, указав файл исходника обязательно впереди неё.

Второй случай не рекомендуется, т. к. компилятор не может участвовать в проверке взаимодействия основной программы и посторонней функции.

Функции без побочных эффектов

У заголовка функций может присутствовать специальный модификатор «**PURE**». Он говорит о том, что функция не имеет побочных эффектов и компилятор может не тратить своё время на её дополнительную проверку.

«Побочный эффект» - это не ошибка, как думают многие, а всего лишь некое дополнительное действие, которое делает функция помимо своего прямого назначения. В качестве грубого примера такого эффекта можно взять функцию SIN() которая, помимо вычисления синуса числа, будет ещё вычислять и косинус. Казалось бы, почему нет? Но в этом случае функцию следовало бы назвать SINCOS() чтобы было понятно её предназначение.

Следить, чтобы не было побочных эффектов — дело программиста, поскольку компилятор выявить факт нужности того или иного действия в вашей функции в принципе не способен, да и не его это дело.

Внутренние (Intrinsic) подпрограммы и функции

Есть языки, у которых нет никаких собственных функций, а только описание, как и что должно происходить в программе, к примеру Си или Паскаль. Все процедуры\функции там вынесены во внешние модули, которые подключаются по мере надобности. Фортран же принадлежит к другой ветке языков — «всё своё ношу с собой», в котором определённый набор процедур\функций предусмотрен самим стандартом языка. Правда в современном Фортране стандартные функции всё равно вынесены в отдельный модуль, однако программисту предпринимать какие-либо действия по его подключению не надо — он (как и в Паскале модуль «system») подключается автоматически.

В GNU Фортране современных версий собственных процедур\функций ужасно много — примерно 300, точное значение не помню. В общем — практически на все случаи жизни. Так что вряд ли стоит их тут перечислять. Они хорошо описаны в документации [], в разделе «Intrinsic procedures». Практически ко всем прилагается небольшой работающий примерчик по использованию. Также, что немаловажно, написано, к какому стандарту Фортрана принадлежит процедура на тот случай, если вам придётся пользоваться какими-либо старыми версиями трансляторов.

В подавляющем большинстве случаев процедуры выполняют какую-нибудь элементарную операцию, например вычисление синуса или косинуса числа. Но есть и посложнее, например Гамма-функция или функция Бесселя, но такие появились только в современном Фортране, в старом их не было. Кроме того, есть довольно много процедур для общения с операционной системой, чтобы не было необходимости подключать какие-либо дополнительные библиотеки.

Модули

Модули — это аналог паскалевких «UNIT», в которых хранятся уже откомпилированные процедуры, функции, переменные и константы. При компиляции в программу просто вставляется двоичный код используемых из модулей объектов.

Обычно подпрограммы и функции выносят в дополнительный модуль тогда, когда их количество мешает воспринимать программу как единое целое. Либо вы сами эти дополнительные функции не писали и они вам нужны, а кто-то очень добрый их уже написал и выложил в интернет. В этом случае тоже бывает полезным разделение — где ваш код, а где не ваш. Это помогает в поиске ошибок.

Надо сказать, что выносить код функций\подпрограмм в отдельный файл можно было и в старом Фортране, однако при этом существовали определённые сложности и неудобства. Подпрограмма в отдельном файле не видела ничего, что есть в главной программе и если подпрограмме нужно было использовать какие-то общие переменные, то их нужно передавать туда в виде параметров. И если таких общих переменных очень много, то заголовок подпрограммы превращается в нечто непонятное.

В новом Фортране, с появлением компонентов-модулей, ситуация поменялась, т. к. теперь видимость констант, переменных и подпрограмм можно задавать специальным модификатором. Для тех объектов, которые должны быть видны везде задаётся модификатор «PUBLIC», а для тех, что будут видны только непосредственно в модуле - «PRIVATE». Модификатор «PUBLIC» задан по умолчанию. Такие модификаторы работают только в модуле, если указать их в главной программе, то компилятор выдаст ошибку.

Файл модуля начинается служебным словом «module» с названием модуля, а заканчивается словами «end module НазваниеМодуля».

```
module mathlib

real(8), save :: pi=3.14159265358979323846264338327950288419716939937_8
real(8), save :: e=4.803242d-10
...

CONTAINS

    real (8) function pythag ( a, b )

    ...

    end function pythag

    subroutine Hogwqrds ( a, b )

    ...

    end subroutine Hogwards

end module mathlib
```

К сожалению, введя паскалеподобный синтаксис, разработчики Фортрана не сделали и другой хороший, опять таки паскалеподобный, шаг для удобной компиляции модулей и программ их использующих. Компиляция модулей происходит с обязательным ключом «-c», чтобы отключить линковщик и не создавать «нормальную» программу. Если этот ключ позабыть, то компилятор будет выдавать ошибку. В Паскале компилятор и без нас знает, как компилировать модуль, а как нормальную программу.

```
gfortran -c mathlib
```

Ключ «-o», чтобы задать имя выходного файла, указывать не нужно, компилятор по умолчанию создаст два файла с тем же именем, что и у исходного кода:

- mathlib.o — объектный файл, в котором содержатся в двоичном виде всё, что входит в модуль. Его нужно добавлять при компиляции программы использующей этот модуль;
- mathlib.mod — это заголовочный файл, по аналогии с сишным «что-то_там.h» только в откомпилированном, двоичном виде.

Если ваша программа будет использовать какой-либо фортрановский модуль, то сразу после названия программы название этого модуля увзывается в строке подключаемых модулей после слова «use»:

```
program test_mathlib
use mathlib

...

end program test_mathlib
```

и тогда в главной программе можно использовать любой объект из модуля, если только этот объект не сделали невидимым с помощью модификатора «PRIVATE».

Компилируется программа так:

```
gfortran mathlib.o test_mathlib.f90 -o test_mathlib
```

А можно и так:

```
gfortran mathlib.f90 test_mathlib.f90 -o test_mathlib
```

если вам доступен исходный код модуля. ☺ Главная программа в строке компиляции обязательно должна указываться последней, т. к. всё что в неё входит из других файлов должно быть уже откомпилировано к моменту компиляции основной программы.

3.6 Перегрузка функций и операторов

В современный Фортран, как и в другие языки общего назначения, была добавлена такая интересная, с точки зрения упрощения кода основной программы, возможность, как перегрузка функций\подпрограмм и операторов.

Если кто-то видел подпрограммы старого Фортрана, то наверняка заметили, что любая функция обязательно присутствует, как минимум, в двух экземплярах — для вычисления с обычной точностью (REAL) и для вычисления с двойной точностью (REAL(8), который раньше называли DOUBLE PRECISION). Например, в библиотеке пакета научных программ фирмы IBM (которую мы, в своё время, без спросу позаимствовали. Но только — тс-с-с, чтобы ЦРУ не узнало... ☺) существовала подпрограмма HARM, которая делала комплексный трёхмерный анализ Фурье с числами обычной точности (REAL) и подпрограмма DHARM — то же самое, только для чисел с двойной точностью. А если алгоритм мог работать и с комплексными числами, то обязательно делался и третий экземпляр, где к имени подставлялась начальная буква «С». Программист, который писал свою программу с использованием таких дополнительных функций, запросто мог ошибиться с типами данных или с именем функции, т. к. хорошо, если компилятор подсказывал, что перепутаны типы данных, а ведь мог и не подсказать, если формально всё было правильно, только тип двойной точности, куда влезало значение одинарной точности, вычислялся дольше и вентиляторы процессора в ужасе и панике выли дурными голосами от такой нагрузки, предупреждая о катастрофическом перегреве. Но программисты отчего-то на подсказки вентиляторов внимание никогда не обращают… ☺

С появлением перегрузки от нескольких экземпляров функций, которые делают одно и то же, только с разными типами данных, избавится конечно же не удалось, однако все их можно свести к одному названию, а компилятор уже в соответствии с используемыми типами данных будет подставлять в двоичный код подходящую процедуру.

Удобнее всего перегрузку организовать именно в модулях, а не в главной программе, т. к. перегрузка по умолчанию предполагает существенное увеличение размеров кода, который лучше с глаз убрать, чтобы не путаться. А в главной программе можно не думать, какую функцию взять, чтобы использовать с тем или иным типом параметров, название будет всегда одно и то же. В стандартном Фортране, например, мы можем использовать функцию SIN() как с параметром REAL, так и с REAL(8), COMPLEX или вообще это будет массив. На самом деле это несколько

отдельных функций, которые компилятор подставляет при компиляции ориентируясь на тип данных, который вы передаёте в функцию SIN().

Как это оформляется? Допустим, у нас есть функция, которая переводит переданный ей параметр из градусов в радианы, чтобы потом можно было вычислить синус этой величины. Если мы её напишем для типа REAL, а передавать туда будем REAL(8), то при компиляции получим ошибку. Хорошо, давайте, на всякий случай она будет работать с типом REAL(8). Отлично, большую часть времени её использования видимых проблем не будет, пока мы ей не передадим аргумент в виде REAL(16). Пусть это бывает крайне редко, но бывает. И что, делать универсальную функцию именно с этим типом? Получим уменьшение производительности и при том малорадостном факте, что REAL(16) нам будет требоваться только в 1 случае из 100. Введя перезагрузку мы от такой проблемы избавимся.

```
!-------------------------------------------------
! Модуль для преобразования градусов в радианы
! с разными типами аргументов.
!-------------------------------------------------
module grad2rad

! Константы, которые потребуются для расчётов
real, parameter :: Pi = 3.1415
real(8), parameter :: Pi8 = 3.1415_8
real(16), parameter :: Pi16 = 3.1415_16

! Функцию с этим названием используем в своей програме
interface g2r
    module procedure g2r4, g2r8, g2r16
end interface g2r

contains

! Реализация функции g2r в зависимости от типа параметра
real(4) function g2r4(g)
real(4) :: g

g2r4 = g * 180.0/Pi

end function g2r4

real(8) function g2r8(g)
real(8) :: g

g2r8 = g * 180.0/Pi

end function g2r8

real(16) function g2r16(g)
real(16) :: g

g2r16 = g * 180.0/Pi

end function g2r16

end module grad2rad
```

Здесь хотелось бы обратить особое внимание на явное указание **real**(4) вместо умолчального **real**. В чём тут дело? В компиляторе. ☺ Дело в том, что ключом компилятора мы можем задать умолчальное количество байтов для элементарных типов и оно от 4 может

отличаться. Таким образом, если мы укажем для **real** ключом умолчальное значение 8, то у нас получится две одинаковые функции, а вот для **real**(4) никакой функции не будет. Специально для такого случая мы указали у всех типов их величину в байтах явно.

Итак, определяем модуль для перерасчёта градусов в радианы, в котором определяем константу Pi разных типов, которая будет участвовать в перерасчёте (само число ПИ я расписывать целиком не стал, просто обозначил его тип с помощью постфиксов - _8 и _16).

После констант определяем с помощью «interface» название функции, которая у нас будет работать со всеми типами данных - «g2r» и указываем, какие конкретно процедуры будут подставляться при том или ином типе данных — g2r4, g2r8 и g2r16.

Теперь, когда мы подключим этот модуль и применим в своей программе функцию g2r, то в откомпилированной программе получим ту функцию, которая определяется типом переданного параметра. Нам самим об этом беспокоиться больше не нужно.

Кстати, не обращайте внимание что в модуле присутствует «module procedure», а вовсе не «module function», как по логике должно было бы быть. Просто раньше самыми популярными были именно процедуры (подпрограммы), поэтому разработчики решили с отдельными названиями не заморачиваться. ☺

Теперь давайте займёмся перегрузкой операторов. В Фортране вы можете наблюдать, что арифметические операторы можно применять как к переменным «одиноких» типов данных, так и к переменным массивов без явного использования циклов или каких-то подпрограмм. Хотя какая-то подпрограмма при работе с массивами всё же используется, только очень глубоко в подполье и нам она не видна.

Обычно перегрузка операторов нужна в случае использования каких-то сложных составных типов, которые мы сами определяем и чтобы работа с ними была похожа на работу с обычными «одинокими» типами. Для примера сделаем, как и в прошлый раз, отдельный модуль. Допустим нам надо работать с объектами по их трёхмерным координатам.

```
module D3Object

! Составной тип отображающий точку
! в трёхмерном пространстве
Type D3Point
        integer :: x
        integer :: y
        integer :: z
end type D3Point

! Эта функция будет подставляться
! вместо оператора «+»
interface operator(+)
        module procedure add_3d
end interface operator(+)

! Эта функция будет подставляться
! вместо оператора «-»
interface operator(-)
        module procedure subtract_3d
end interface operator(-)

! Эта функция будет подставляться
! вместо оператора «*»
interface operator(*)
        module procedure mult_3d
end interface operator(*)

! Эта функция будет подставляться
```

```
! вместо оператора «=»
interface assignment(=)
        module procedure assign_3d, assign_3dint
end interface assignment(=)

contains

! Реализация функций

elemental type(3DPoint) function add_3d(a, b)
type(3DPoint), intent(in) :: a, b

add_3d%x = a%x + b%x
add_3d%y = a%y + b%y
add_3d%z = a%z + b%z

end function add_3d

elemental type(3DPoint) function subtract_3d(a, b)
type(3DPoint), intent(in) :: a, b

subtract_3d%x = a%x - b%x
subtract_3d%y = a%y - b%y
subtract_3d%z = a%z - b%z

end function subtract_3d

elemental type(3DPoint) function mult_3d(a, b)
type(3DPoint), intent(in) :: a, b

mult_3d%x = a%x * b%x
mult_3d%y = a%y * b%y
mult_3d%z = a%z * b%z

end function mult_3d

elemental subroutine assign_3d(a, b)
integer, intent(in) :: b
type(3DPoint), intent(inout) :: a

a%x = b
a%y = b
a%z = b

end function assign_3d

! Подпрограмма для показа содержимого
!  3DPoint на экране
subroutine show_3D(a)
type(3DPoint), intent(in) :: a

print *, a%x
print *, a%y
print *, a%z

end subroutine show_3D
```

```
end module D3Object
```

Здесь всё сделано по аналогии с предыдущим примером, только в интерфейсной части дополнительно возникло слово «operator», что вне всякого сомнения говорит о том, что мы теперь собираемся перегружать не функции\процедуры, а именно операторы, которые указаны в скобках.

В части реализации появился дополнительный модификатор функций «elemental». Это означает, что функция будет производить операции с элементами передаваемых её типов данных, вне зависимости от того, одинокий это тип или массив. Этот модификатор автоматически подразумевает, что функция у нас без побочных эффектов и делает только то, что от неё требуется, что ускоряет работу, поскольку компилятор не добавляет туда дополнительные процедуры проверки.

Интересно выглядит замена присваивания. Здесь вместо слова «operator» нужно использовать «assignment». Самим присваиванием должна заниматься почему-то не функция, а подпрограмма, но как и предыдущие функции обязательно с двумя параметрами — первый мы должны увидеть на выходе, поэтому у него модификатор «inout», а второй только входной, поэтому с модификатором «in». Именно его значения мы будем присваивать выходному параметру. Вторая подпрограмма присваивания нужна на тот случай, если мы захотим, по аналогии с массивом, присвоить всем внутрилежащим переменным объекта одно и то же целочисленное значение.

Теперь, если подключить этот модуль к программе, с переменными типа 3DPoint можно будет производить арифметические операции, как с одинокими типами (естественно в рамках того, что определено в модуле).

```
program test_D3Object
use D3Object

type (D3Point) :: a, b, c

! Начальные значения
a = 2
b = 3
c = 0

! Арифметические операции
c = a * b

call show_3D( c )

c = a - b

call show_3D( c )

end program test_3DObject
```

3.7 Циклы и проверка условий

Эти два понятия связаны друг с другом. Правда условия могут существовать и сами по себе, но вот циклы без проверки условий существовать никак не могут, ведь выход из цикла осуществляется именно по какому-то условию.

Сначала поговорим об условиях. Они бывают простые и сложные. Простые — это когда на условие (даже если оно длинное, как обещание политиков про счастливую жизнь трудящихся) можно дать ответ «ДА» или «НЕТ». Сложное условие — это когда ответ зависит от кучи нюансов и количество ответов может быть довольно велико, как количество блюд в ресторане «Распутин».

Простые условия обрабатываются операторами «IF … END IF», когда нужно какое-то безальтернативное действие или «IF … ELSE ... END IF», когда если условие выполняется, то действует один блок кода, а если нет — то другой:

```
if (sin(alfa)>=0.7) then
        print *, "Угол тупой"
else
        print *, "Угол острый"
end if
```

Проверяемое условие обязательно должно быть в скобках.

Сложное, многовариантные условия обрабатывается оператором «SELECT CASE … END CASE»:

```
integer :: year

print *, "Сколько Вам лет?"
read (*,*) year

select case (year)
        case (0:10)
                print *, "Вы ещё очень молоды"
        case (11:20)
                print *, "Вы пока ещё молоды"
        case (21:40)
                print *, "Вы до сих пор молоды"
        case (41:60)
                print *, "Вы молоды душой"
        case (61:80)
                print *, "Вы молоды духом"
        case (81:100)
                print *, "Вы молоды в своих мечтах"
        case default
                print *, "Либо Вы ещё, либо Вы уже..."
end select
```

Здесь представлена ситуация выбора значения в пределах нескольких диапазонов. Если никакой диапазон не подходит, то выбирается последнее действие по умолчанию (default).

Циклы в других языках программирования обычно представлены в двух разных форматах:

- цикл, когда заранее известно количество повторений
- цикл, когда количество повторений заранее неизвестно.

В Фортране конструкция цикла всего одна и базируется она мало того что на неизвестном количестве повторений, но ещё и на отсутствии в операторе цикла каких-либо условий по выходу из него:

```
do

        ! Что-то делаем

end do
```

На самом деле такая запись цикла универсальная. Мы можем условие выхода поставить как в начале цикла, так и в его конце.

Вариант для цикла с неизвестным количеством повторений выглядит так:

```
do
        ! Что-то делаем
```

```
        ...
        ! Проверяем условие выхода из цикла
        if (условие_выхода) then
                exit
        end if
end do
```

Здесь оператор «EXIT» прекращает работу цикла и передаёт продолжение программы за пределы «end do». Если вместо «EXIT» будет стоять оператор «CYCLE», до весь программный блок, который находится до «end do» будет проигнорирован, а продолжение действия будет передано в начало цикла.

Модификация DO для цикла с известным количеством повторений выглядит так:

```
integer :: i

do i=Начальное_значение,Конечное_значение[,Шаг]

        ! Что-то делаем

end do
```

Внутри цикла значение «i» во всю можно использовать, но ни в коем случае самостоятельно не изменять.

Если не указывать значение «Шаг», то переменная «i» будет изменяться от начального значения до конечного (включая и то и другое) с шагом в 1. Иначе переменная будет изменяться на значение «Шаг». Если «Шаг» положительное число, то оно будет плюсоваться к «i», если же отрицательное — то минусоваться. Переменная «i» в обязательном порядке должна быть целочисленного типа.

3.8 Работа с файлами

Поговорим только про текстовые файлы, т. к. это наиболее универсальное средство обмена данными. Прежде чем прочитать что-то из файла или записать что-то в файл, его нужно открыть командой «open» с несколькими обязательными именованными параметрами:

```
open ( unit=Номер_Файла, file="Название_Файла", status="Статус", &
action="Что_делать_с_файлом")
```

Здесь:

- unit=Номер_файла — это номер, целое число, используя которое можно с файлом производить какое-либо действие — читать, писать информацию, а потом закрыть. В современных версиях Фортрана «unit=» можно не писать, а ограничится только номером;
- file – здесь мы пишем в кавычках полное название файла, если надо, то и с путём к нему;
- status – если файл до этого уже существовал, то статус ему присваиваем «OLD». Пытаясь открыть файл с таким статусом, которого на самом деле нет, мы получим ошибку выполнения программы. Если нужно создать файл которого нет, то статус будет «NEW», однако если такой файл всё же есть, то открытие с этим статусом приведёт к ошибке выполнения. В случае, если нам надо полностью заменить файл, причём неважно, существовал он или нет, то статус будет «REPLACE»;
- action – что будем делать с файлом — писать туда «WRITE», читать оттуда «READ» или добавлять информацию к существующей «APPEND». Опция на самом деле необязательная, т. к. по умолчанию файл открывается и на чтение и на запись, однако очень полезная с точки зрения безопасности данных. Если вы хотите только читать из файла, то «READ» не даст вам ничего туда записать. Если хотите добавлять данные не затирая уже существующие, то «APPEND»

сразу после открытия переведёт указатель файла в самый его конец, тем самым убережёт существующие данные от затирания. Так что эту опцию я вам настоятельно рекомендую использовать.

Есть и другие опции, но они уже добавляются в случае крайней необходимости. Если вам интересно, почитаете про них самостоятельно в других книжках.

Открытый файл обязательно нужно закрыть, причём сразу же после его использования, не дожидаясь конца работы программы. Это критично в случае, если вы туда что-то пишете. Вдруг свет отключат? Файлы обычно буферизируются для скорости работы, так что незаписанный буфер может и пропасть без вести.

Закрывают файл просто:

```
close ( Номер_файла )
```

Чтение данных из файла и запись в файл делаются теми же операторами, что и при работе с экраном, только обязательно нужно добавить номер файла:

```
 ! Вот сейчас мы из файла что-то прочитаем
read ( Номер_файла, Формат_чтения ) Переменная [, Переменная, ... ]

write ( Номер_файла, Формат_чтения ) "А вот мы в файл что-то записали"
```

Когда вы работаете с большим количеством файлов (больше двух ☺), то в номерах файлов легко запутаться и лучше всего будет вместо номеров использовать переменную с каким-нибудь говорящим именем. Чтобы облегчить работу с файлами, в Фортран-2008, в команду «open», введён новый именованный параметр «newunit», которому присваивается переменная номера файла. Сам номер при этом можно не определять — «newunit» возьмёт ближайший свободный. На практике это выглядит так:

```
integer :: MyVeryBeautyFile
character(100) :: message

open (newunit=MyVeryBeautyFile, file="MyVeryBeautyFile.txt", &
        status="OLD", action="READ")
read (MyVeryBeautyFile,'(a)') message
print *, message
close (MyVeryBeautyFile)
```

Как видим, работа с файлом теперь совсем простая и, самое главное, понятная.

3.9 Фортран и графика

Правильнее было бы назвать этот подраздел «Фортран и графики», но уж как назвал, так и назвал… ☺ В Фортране, в отличие от его родного сыночка Бейсика, никаких встроенных средств по работе с графикой нет. То есть абсолютно никаких. Но это ещё не значит, что графикой в Фортране пользоваться совсем уж никак нельзя. К примеру в Си или Паскале тоже нет никаких графических функций\процедур, просто создан интерфейс к какой-нибудь графической библиотеке. Для Windows это к примеру WinAPI, поскольку в Windows графика прямо встроена в систему. Для взаимодействия Фортрана с WinAPI написана специальная книжка [27]. Есть так же довольно много модулей, которые организовывают взаимодействие Фортрана с другими графическими библиотеками, например с OpenGL [28][29] или с GTK [30]. Однако в этом подразделе я расскажу как без всяких модулей строить именно графики, которые в науке применяются очень широко.

Графики можно строить на основе текстового файла с данными, который скармливается какой-нибудь специальной программе по рисованию графиков. Мне, например, очень нравится

программа Gnuplot, которая может строить как двумерные, так и трёхмерные графики на основе цифр из текстового файла. Кроме цифр она понимает и формулы и по формуле тоже может построить график. Готовые графики можно выводить как непосредственно на экран, так и в какой-нибудь файл рисунка (gif, jpeg, svg и ещё довольно много графических форматов).

3.10 Параллельное программирование

Работа параллельной программы всегда начинается с одного потока, который проводит какие-то начальные действия, а далее создаются несколько потоков, по которым распределяются части данных, над которыми проводятся одинаковые вычисления (см. рис. 16). Либо это будут не части одного массива данных, а отдельные функции которые обрабатывают свои массивы данных. По ходу работы программы параллельные потоки или процессы (параллельные области вычислений) могут создаваться несколько раз, в зависимости от потребностей вычислительного алгоритма.

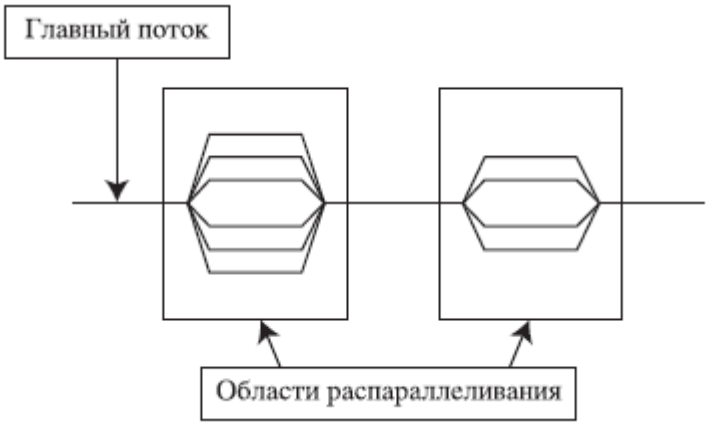

Рисунок 16. По ходу работы программы создаются параллельные потоки или процессы

3.10.1 OpenMP

Применение OpenMP для параллельного программирования по сравнению с MPI очень простое. И если учесть, что сегодня подавляющая часть процессоров делается многоядерными, то параллельное программирование с помощью OpenMP может запросто входить составной частью в любую программу, при этом программисту не придётся излишне напрягать своё серое вещество.

Начало области параллельных вычислений в программе обозначается после применения специальных директив компилятору, которые всегда начинаются с «!$OMP» с дополнительными опциями. А конец параллельной области обозначается с помощью «!$OMP END». Такой вид директив сделан для того, чтобы когда компилятор не умеет работать с OpenMP, он бы воспринимал подобные директивы как обычный комментарий. Создание потоков при этом автоматизировано и участия программиста не требует.

Если количество потоков не заданно явно, то создаётся столько потоков, сколько вычислительных ядер найдено в системе. В свою очередь, каждый поток может создавать свою область параллельных вычислений. Количество потоков вручную может быть задано специальной опцией в директиве, служебной функцией или специальной системной переменной. Когда создаётся область параллельной работы, то тот поток, где он создаётся, получает номер 0, а все остальные от 1 до Количество_ядер — 1.

Кроме директив есть ещё некоторый набор служебных функций, которые помогают при работе параллельных секций программы. Для их использования в вашей системе должна быть установлена специальная библиотека libgomp. Для операционных систем типа UNIX такая библиотека есть в репозитории ОС. Если речь идёт о Windows, то в комплекте cygwin и mingw эта библиотека есть.

Чтобы использовать дополнительные функции из этой библиотеки, необходимо подключить модуль с описаниями этих функций;

```
use omp_lib
```

В общем случае директива выглядит так:

```
!$OMP PARALLEL [ОПЦИИ][,ОПЦИИ][,ОПЦИИ ...]

    ! Здесь мы что-то высчитываем

!$OMP END PARALLEL
```

Сразу же нужно предупредить, что успешное распараллеливание возможно только тогда, когда переменные в цикле не зависят друг от друга. Например, при обработке массива, если каждая ячейка массива может вычислятся совершенно отдельно от других ячеек, то будет всё прекрасно работать. А вот если вычисление ячеки, зависит от вычисления предыдущей ячейки, то ошибку компилятор вам не выдаст, но и пралаллельную область создавать тоже не будет. Программисту нужно очень внимательно следить за этим моментом, чтобы распараллеливание было успешным.

В подразделе «[ОПЦИИ]» указываются разные уточнения по распараллеливанию, например, как использовать переменные и прочие полезные штуки. Опций может быть несколько, тогда они перечисляются через запятую.

Наиболее часто используемыми опциями являются:

- по работе с циклами;
- задание области видимости переменных в параллельной области;
- условие входа в параллельную область работы;
- задание участка кода, который выполняется только в главном потоке.

Распараллеливание для цикла выглядит так:

```
!$OMP DO [ОПЦИИ]

    ! Здесь мы что-то высчитываем

!$OMP END DO
```

Некоторые опции для параллельного цикла. Они задают область видимости и начальную инициализацию переменных в параллельной области):

- private(список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
- firstprivate(список) – задаёт список переменных, для которых порождается локальная копия в каждой нити. Локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
- lastprivate(список) – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;
- reduction(оператор:список) – задаёт оператор и список общих переменных. Для каждой переменной создаются локальные копии в каждой нити. Локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги). Над локальными копиями переменных после завершения всех итераций цикла выполняется заданный оператор. Оператор может быть одним из: «+», «*», «-», «.and.», «.or.», «.eqv.» , «.neqv.», «max», «min», «iand», «ior», «ieor». Что делает каждый из операторов понятно из его названия, логические операторы с буквой «i» - это побитовые операции. Порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

Условие входа в параллельную область задаётся по аналогии с обычным IF:

```
!$OMP DO IF(Какое-то_условие)

    ! Что-то делаем

!$OMP END DO
```

Задавать условие входа нужно тогда, когда в ходе проектирования\отладки программы выясняется, что на распараллеливание уходит больше ресурсов, чем если бы программа работала в однопоточном режиме.

Те части программы, которые должны выполняться только в главном потоке (например, вывод на экран, чтобы там не создавался винигрет) оформляются так:

```
!$OMP MASTER

    ! Что-то делаем

!$OMP END MASTER
```

Кроме директив компилятору, большую помощь программе оказывают всякие служебные функции OpenMP. Вот несколько самых популярных:

```
integer :: n
real(8) :: r

! Узнать текущее количество потоков
n = omp_get_num_threads()

! Узнать, сколько потоков вообще можно создать в системе
n = omp_get_max_threads()

! Узнать, в каком именно потоке мы сейчас работем
n = omp_get_thread_num()

! Узнать, какое количество вычислительных ядер есть в системе
n = omp_get_num_procs()

! Узнать, работаем мы сейчас в параллельной области или нет
! Если нет, возвращает 0, если да — не 0
n = omp_in_parallel()

! Метка времени. Полезна при замерах времени выплнения функций
r = omp_get_wtime()
```

Про последнюю функцию следует сказать особо. Дело в том, что если получать метку времени стандартной подпрограммой (например cpu_time()), то можем столкнуться с такой ситуацией, когда время работы будет определено неправильно. Даже может быть, что функция отработала отрицательное количество секунд. ☺ Связано это с тем, что таймеры в разных потоках не синхронизированы, поэтому при измерении времени в программах с параллельными вычислениями, нужно использовать именно эту функцию.

Есть и другие функции…

Из интересного… В последних версиях OpenMP появилась поодержка не только главного процессора, но и разных дополнительных, например GPU, что резко увеличивает количество потоков, которые можно создавать в системе. А это значит, что скорость параллельных вычислений может увеличится на порядок по сравнению с системами, где для вычислений использовался только главный процессор.

3.10.2 MPI

MPI, по сравнению с OpenMP, это дело куда более сложное. Процессы работают чаще всего на разных компьютерах. Работой процессов заведует специальный менеджер, который запускается на главном компьютере и инициирует запуск приложений числом, определённым в параметрах запуска, на тех компьютерах, которые записаны в конфигурационном файле менеджера. Сама программа должна лежать на каком-то сетевом ресурсе, который доступен всем без исключения компьютерам-участникам параллельного вычисления. Во время работы программы, менеджер организует передачу данных между всеми участвующими в вычислениях процессами.

Чтобы можно было использовать MPI, необходимо предварительно установить какую-нибудь MPI-библиотеку на все компьютеры, где будет запускаться MPI-программа. Чаще всего используются две наиболее популярные свободные библиотеки – MPICH и OpenMPI. Для UNIX-систем нет никаких проблем установить их из репозитория своей ОС. Правда надеятся на то, что библиотека будет самой свежайшей версии не стоит ☺. С Windows дела похуже, если вы не используете CYGWIN. В CYGWIN есть и OpenMPI. В случае использования других дистрибутивов придётся искать в интернете откомпилированные версии для Windows и тут уж они могут быть совсем старые и, вдобавок, работать только с Visual Си.

Я на свой компьютер MPI из репозитория никогда не ставлю, предпочитаю скачать с сайта разработчика свежую версию в исходниках, собрать её у себя и установить. Обычно пользуюсь библиотекой OpenMPI не потому что питаю к ней нежные чувства, а потому что она у меня всегда компилировалась без малейших проблем.

В свою программу нужно обязательно добавить модуль MPI. Если у вас gfortran версии 5.4 и выше, то модуль будет таким:

```
use mpi_f08
```

Если версия ниже или компилятор будет давать ошибку подключения этого модуля, тогда используем более старое название:

```
use mpi
```

В случае, если вы каким-то непонятно печальным образом используете до сих пор Фортран-77, то подключать будем заголовочный файл:

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

из которого будут браться все функции MPI. Такой заголовочный файл будет работать и современных фортранах, но выглядеть это будет совсем не по современному.

С компиляцией программы тоже не всё просто так. Необходимо использовать специальный скрипт MPI:

```
mpifort ./Ваша_программа.f90
```

Этот добавляет к компилятору gfortran необходимые ключи (их много). Хотя все ключи можно посмотреть с помощью опций, но у меня не получилось откомпилировать программу используя gfortran с этими ключами. Так что будем пользоваться mpifort.

Если библиотека MPI сильно уж старая (как у меня на рабочих серверах — версия openmpi – 1.10), то вместо mpifort там есть mpif90, которая используется точно так же.

С запуском откомпилированной программы ситуация аналогичная – если запускать её просто так, то будет всего один процесс. Запускается программа как опция к специальной программе, которая заведует и менеджером MPI:

```
mpirun -np Количество_процессов Ваша_программа [другие опции]
```

Если планируется использовать программу на нескольких компьютерах, то необходимо заполнить файл конфигурации /etc/openmpi-default-hostfile, куда записываются все IP или имена

хостов, где будут одновременно запущены процессы вашей программы. И, конечно же, для всех хостов должен быть доступен тот файловый ресурс, с которого запускается программа. Естественно на всех этих хостах должен быть один и тот же пользователь, пароль которого вы знаете. Как конфигурировать этот файл я рассказывать не буду, это сильно далеко выходит за рамки начального уровня, об этом можно почитать в документации к библиотеке MPI (вот, например, для OpenMPI [ ]).

Вот ещё несколько ключей запуска, которые иногда могут представлять интерес:

mpirun -cpu-per-procs N

Здесь N – количество ядер выделяемых для работы одного процесса.

mpirun -max-restarts N

Здесь N – количество попыток перезапустить процесс, если что-то с ним случилось во время работы.

mpirun -output-filename ИмяФайла

Если экран для вывода сообщений недоступен, а для вычислительных кластеров это реальная ситуация, то вывод программы будет происходить в указанный файл.

Если вам нужно узнать, чем именно предстоит пользоваться, то можно использовать ключ:

mpirun -V

Если нужно вы хотите запускать столько процессов на одной машине, сколько у неё есть вычислительных ядер, то запуск будет с таким ключом:

mpirun -use-hwthread-cpus ./ИмяПрограммы

Работа в самой программе организуется с помощью специальных функций. До начала каких либо вычислений, в программе необходимо объявить, что она готова работать под руководством менеджера. Делается это с помощью функции:

mpi_init(error)

где error – это переменная, куда помещается код ошибки в случае каких-либо проблем.

Эта функция инициализирует клиентскую часть библиотеки MPI с помощью нескольких внутренних функций, которые программисту не видны.

Перед завершением работы программы в обязательном порядке нужно сообщить менеджеру, что вы устали и работать больше не хотите:

mpi_finalize(error)

Вообще, вся работа по параллелизму осуществляется, в отличие от OpenMP, исключительно функциями из внешней библиотеки MPI и, увы, полностью в ручном режиме. В принципе использование только функций довольно удобно, т.к. если вы хотите организовать MPI в каком-нибудь другом языке, кроме Фортран и Си, то достаточно будет правильно описать заголовки этих функций в том языке. Однако ручной режим распараллеливаня требует дополнительных мозговых усилий, иногда совсем нетривиальных.

В начале планируются режимы распараллеливания – по данным, по функциям или всё вместе, в зависимости от взаимосвязи данных и их распределению для подзадач.

Далее, далее необходимо спланировать обмен данными между разными процессами и при завершении вычислений – синхронизацию этих данных. Данные передаются с помощью специальных сообщений. В сообщение вкладывается некий блок, который имеет тип данных специфический для MPI, но в принципе все эти типы соответствуют типам Фортрана, если

программа написана на Фортране, или Си, если программа написана на Си. К недостаткам подобных передач сообщений следует отнести то, что просто так передать другому процессу к примеру многомерный массив не получится. Перед передачей его нужно обязательно сделать одномерным или, в терминах MPI, упаковать.

Приём\передача данных могут быть как синхронными, так и асинхронными. Синхронная передача – это когда процесс запускает функцию приёма сообщения и та приостанавливает работу программы, пока ей чего-нибудь не передадут. Аналогично и функция передачи данных приостановит действие программы, пока у него данные не примет процесс, кому адресована передача. Асинхронные функции приёма\передачи никого никогда не ждут, они просто отправляют сообщение с данными менеджеру, а тот хранит их у себя в памяти, пока процесс-адресат не подключится и не заберёт свои данные.

Синхронная передача данных осуществляется с помощью функции:

```
mpi_send(buf, count, datatype, dest, msgtag, comm, error)
```

где buf – блок передаваемых данных;
count – их количество;
datatype – тип передаваемых данных (MPI тип);
dest – номер процесса-получателя;
msgtag – идентификатор блока передаваемых данных;
comm – номер коммуникатора (канала приёма\передачи);
error – номер ошибки.

Пожалуй из не совсем понятного в этих параметрах – msgtag и comm. Процесс может передавать другому процессу не один, а несколько блоков данных. Это особенно актуально для асинхронного приёма\передачи, чтобы не запутаться в данных. Поэтому всем блокам перед передачей нужно присвоить уникальный идентификатор. С comm немного сложнее. Процессы обычно организуются в группы. Между процессами одной группы есть канал связи, по которому идут данные. Но групп может быть несколько и один и тот же процесс может участвовать в нескольких группах, поэтому номер канала (или, по другому говоря, номер группы) обязательно должен присутствовать. Если у нас всего одна группа\канал, то обычно её обозначают предопределённой константой «MPI_COMM_WORLD», которая обозначает «все, кто меня слышит». Однако на вычислительном кластере у группы со своими каналами может быть много, поэтому канал должен иметь свой уникальный номер.

MPI типы данных – это отдельные типы, которые имеют своё название. Их размер фортрановским не соответствует, но в них передаётся как раз размер типа Фортрановских переменных. MPI-ные типы и какие фортрановские типы они представляют привелены в таблице 3.

Таблица 3. MPI типы данных и какие типы Фортрана они представляют

Тип MPI	**Тип Фортран**
MPI_INTEGER	INTEGER
MPI_REAL	REAL(4)
MPI_DOUBLE_PRECISION	REAL(8)
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	Что-то вроде универсального типа в один байт

Синхронный приём данных делает функция:

mpi_recv(buf, count, datatype, source, msgtag, comm, status, error)

Здесь в buf помещается принятая информация, количество которой count и с типом данных datatype. Source – номер процесса пославшего сообщение. Status – аттрибуты сообщения, которые представляют из себя массив. Посмотреть его значения можно так:

- status(MPI_SOURCE) – номер процесса пославшего сообщение;
- status(MPI_TAG) – идентификатор (номер) сообщения;
- status(MPI_ERROR) – код ошибки. Если ошибки нет, то 0.

Функции приёма\передачси имеют несколько разновидностей. Есть разновидность, которая записывает данные в специальный локальный буфер программы, после чего работу свою закончит, а буфер будет дожидаться подключения процесса-адресата. Есть функция, которая ждёт непосредственного подключения адресата, а есть функция, которая срабатывает только в том случае, если обнаружится уже подключённый процесс-адресат.

Асинхронные приём\передача осуществляются с помощью функций *mpi_irecv()*\ *mpi_isend()* с теми же параметрами. При этом функции своего адресата не дожидаются, но использовать buf до того, как адресат принял сообщение уже не получится.

Если нужно переслать данные всем без исключения существующим процессам, то используют функцию широковещательной передачи:

mpi_bcast(buffer, count, datatype, root, comm)

Эта функция используется как для отсылки, так и для приёма данных, что очень удобно — не надо для разных процессов (отсылающий — принимающие) использовать разные функции, как это было в предыдущей ситуации.

Здесь все параметры функции уже знакомы, кроме параметра root. Это номер процесса, который делает рассылку буфера по всем процессам, которые подключены к каналу comm. Побочный эффект – себе он тоже присылает свой же буфер. ☺

Синхронизация данных между процессами делают с помощью функции:

mpi_reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

Это операция групповая, которая в конце расчёта собирает по процессам данные, производит с ними какую-то операцию (например, суммирование) и складывает в буфер главного процесса.

Здесь, sendbuf – это буфер отправляемых данных к главному процессу;

recvbuf – буфер главного процесса, куда складываются данные после сбора и устаканивания;

count – как всегда количество данных в буфере;

datatype – опять же, как всегда, тип данных в буфере;

op – та операция, которую нужно провести с данными;

root – номер главного процесса;

comm – номер канала.

Операции над данными в буфере очень похожи на операции в OpenMP, только обозначение их немного другое:

- MPI_MAX и MPI_MIN – соответственно поиск самого большого и самого маленького значения в буферах всех процессов;
- MPI_MAXLOC и MPI_MINLOC – определяет в каком именно процессе содержится максимум или минимум;
- MPI_SUM и MPI_PROD – суммирование или перемножение;
- MPI_LAND, MPI_LOR, MPI_LXOR – логические действия;
- MPI_BAND, MPI_BOR, MPI_BXOR – побитовые действия.

Кроме неё существуют и другие функции синхронизации, но про них можно почитать отдельно.

Все функции синхронизации данных, а так же функция широковещательной рассылки должны быть видны всем процессам без исключения, иначе смысл их использования теряется. А вот функции обмена данными, где указываются и процесс-источник и процесс-получатель могут быть дополнены условиями проверки номера процесса, чтобы такие функции использовались только в предназначенных им процессах.

Выше я писал, что переслать или принять многомерный массив не получится. В принципе, матрицу можно было бы передать другому процессу последовательно-построчно, предварительно передав количество строк. Однако суммарное время передачи при таком способе будет выше, чем если бы мы передавали матрицу единым блоком. Один из способов - матрицу можно предварительно выстроить в одну линию, а после получения обратно переделать в исходный вид.

Лирическое отступление на тему «Я, матрицы и MPI»

Когда я учился в магистратуре, у меня там была контрольная работа по MPI на тему решения СЛАУ. Я сначала как-то не сильно вдавался в подробности, накатал по быстрому программку с одномерным массивом и она у меня сработала совершенно правильно. Однако та же программа, но с матрицей компилироваться отказалась напрочь. Проблема определилась быстро – отправка матрицы происходила без проблем, но вот принимал я вовсе не матрицу, а неизвестно что, какой-то кошкин хвост. Вдоволь нафилосовствовавшись по поводу умственных способностей создателей MPI, я побыстрому накалякал процедуру, которая из матрицы делала одномерный массив, отсылала его другим процессам, а после получения приводила этот массив обратно к первозданному виду. Преподаватель проверив мою контрольную, ничего по этому поводу не сказал, видимо решил, что у меня свободного времени навалом, раз я занимаюсь подобными чудачествами. Он только как-то странно покачал головой и поставил мне очередную пятёрку.☺ И только позже, уже углубившись в тему, я понял, что изобрёл велосипед, т.к. MPIщики подобным вопросом тоже в своё время озабачивались...☺

Кроме того, для удобства программиста в MPI есть специальные функции для создания типов определяемых пользователем и специальная функция упаковщик (MPI_PACK), которая как раз и занимается выстраиванием всего и вся в одну линию, которую уже можно передавать другим процессам, а те, в свою очередь, распаковывают данные функцией MPI_UNPACK. Надо признаться дело это не такое уж и простое, как кажется, поэтому писать об это я не буду, лучше почитать какую-нибудь специальную книжку по MPI.

В программе с MPI обычно заранее никогда не известно, сколько всего процессов взаимодействующих через один канал связи будет запущено, поэтому наряду с номером текущего процесса:

```
mpi_comm_rank(comm, rank, error)
```

Здесь, rank – это номер текущего процесса;
error – если возникла ошибка.

крайне необходимо знать общее количество процессов подключённых к одному каналу:

```
mpi_comm_size(comm, size, error)
```

Здесь size – это количество процессов сидящих на канале comm.

Для замеров времени в тестах производительности используется функция:

```
mpi_wtime()
```

возвращающее время в секундах, прошедшее «с сотворения мира». Тип возвращаемого значения real(8), а разрешение таймера можно узнать с помощью:

```
mpi_wtick()
```

В принципе этих сведений достаточно, чтобы начать работу с MPI. Однако функций там великое множество, практически на все случаи жизни, так что работу стоит начинать с каким-нибудь учебником перед носом.

3.10.3 COARRAY

Для того, чтобы не отказываться от распределённых вычислений и немного упростить программу, в начале 90-ых годов прошлого века придумали COARRAY. Сделали это два человека — Роберт Нумич, преподаватель университета Минесоты и Джон Рид из Окфорда. В то время они работали по контракту с фирмой Cray, где делают суперкомпьютеры. Вернувшись к концепции распараллеливания данных, они разработали модель данных, когда одна и та же переменная с одним и тем же названием, которая фигурирует в разных процессах, является ячейкой единого массива. Таким образом работа с межпроцессными данными будет иметь вид работы с обычным массивом. Однако на нижнем уровне работа с такими многопроцессными данными строится на базе какой-нибудь транспортной библиотеки, например MPI, функции которой просто скрыты от глаз программиста. Для такого «скрытого» использования MPI приходится подключать дополнительную библиотеку, например OpenCoarrays [32].

Сначала COARRAY было просто расширением стандарта Фортран-95 и звалось Co-Array, но после длительных и бурных обсуждений в комитете по Фортрану, примерно в 2005 году это расширение вошло в разрабатываемый стандарт Фортран-2008. Что интересно, разработчики Co-Array были членами этого самого комитета, поэтому обсуждение новой фичи Фортрана на удивление напоминало, если кто читал книгу братьев Стругацких «Обитаемый остров», спор главного героя книги Максима с одной из работниц научного учреждения Рыбой (Рыба — это не имя, это её так Максим прозвал ☺):

«Он принялся одеваться, а она застелила его постель, хотя Максим всегда говорил, что будет делать это сам, выдвинула на середину комнаты стол, который Максим всегда отодвигал к стене, решительно отвернула кран отопления, который Максим всегда заворачивал до упора, и все однообразные «не надо» Максима разбивались о её не менее однообразные «надо».

Застегнув балахон у шеи на единственную сломанную пуговицу, Максим подошел к столу и поковырял завтрак двузубой вилкой. Произошел обычный диалог:

– Не хочу. Не надо.

– Надо. Еда. Завтрак.

– Не хочу завтрак. Невкусно.

– Надо завтрак. Вкусно.

– Рыба, – сказал ей Максим проникновенно. – Жестокий вы человек. Попади вы ко мне на Землю, я бы вдребезги разбился, но нашел бы вам еду по вкусу.

– Не понимаю, – с сожалением сказала она. – Что такое «рыба»?

С отвращением жуя жирный кусок, Максим взял бумагу и изобразил леща анфас. Она внимательно изучила рисунок и положила в карман халата. Все рисунки, которые делал Максим, она забирала и куда-то уносила.» ☺

Все доводы разработчиков Co-Array, что это удобно и программистам так надо, разбивались о доводы других членов комитета — «неудобно» и «не надо», ведь есть уже MPI. Впрочем, по истечение то ли 10, то ли 12 лет таких высоконаучных споров, Co-Array всё же победил, правда стал называться CoArray. ☺

Надо сказать, что CoArray — это дело пока что новое, поэтому библиотек для реализации раз-два и обчёлся. На сайте GNU Fortran рекомендуют использовать именно OpenCoarrays совместно с транспортной библиотекой либо MPI, либо GASNet.

Кроме библиотек можно было бы использовать специализированный компилятор, который разрабатывался в Rice University, что было бы весьма интересно, но там с разработкой возникли какие-то проблемы, потому что после появления информации о пре-альфа версии компилятора 2.0 в 2012 году на ихнем сайте больше ничего не выкладывали. А жаль…

Ну, что ж, будем использовать OpenCoarrays.

Установка библиотеки OpenCoarrays

Библиотека OpenCoarrays на сегодняшний день есть в репозиториях большинства общераспространённых дистрибутивов Linux. Однако нужно проверить версию библиотеки, т. к. бывает и довольно старая и вместо установки пакета, лучше скачать у разработчиков исходник и

собрать библиотеку самостоятельно. Из *BSD систем библиотека есть в FreeBSD, для других — информация отсутствует… ☺ .

Итак, чтобы работать с CoArray нужно (кроме gfortran):

- Одна из библиотек CoArray верхнего уровня, например, OpenCoarrays;
- Библиотека нижнего (транспортного) уровня MPI, например, OpenMPI или MPICH.

Устанавливается OpenCoarrays в Ubuntu\Debian очень легко, с помощью штатного менеджера пакетов, поэтому описывать этот процесс я не буду. У других дистрибутивов дела могут быть несколько посложнее...

Вообще, OpenCoarrays поставляется разработчиками в виде исходников и снабжёна специальным скриптом-инсталлятором «install.sh». Этот инсталлятор последовательно запускает набор скриптов, которые скачивают с сайтов разработчиков все необходимые компоненты:

- компиляторы (gfortran, gcc, g++);
- транспортную библиотеку нижнего уровня MPI (по умолчанию это MPICH);
- ещё кое-какие дополнительные компоненты, которые непосредственно в CoArray не участвуют, но нужны при сборке самой библиотеки.

Надо сказать, это довольно удобно и в том плане, что вся установка не требует прав администратора (в Windows) или root (в Linux), т. к. всё скачивается и устанавливается в тот каталог, где находится и OpenCoarrays. Другое дело, что в Linux системах компиляторы уже могут быть установлены изначально и другие не требуются. Для OpenCoarrays версия GNU компиляторов Си\Фортран должна быть не ниже 5.1. Это согласуется с readme gfortran'a, в котором тоже написано, что полноценно CoArray поддерживается начиная именно с этой версии. Так что если у вас версия ниже, то компиляторы скачивать тоже нужно.

Для Windows мы условимся, что у нас уже установлен дистрибутив CYGWIN. Далее скачиваем и распаковываем OpenCoarrays с сайта github.com [30] в домашний каталог (естественно, CYGWIN мы запустили и работаем в нём ☺) и если у нас нет никаких компиляторов, то в каталоге OpenCoarrays запускаем инсталлятор с такими ключом:

```
./install.sh --yes-to-all
```

При этом инсталятор не надоедает нам вопросами и обходится умолчальными параметрами. Надо сказать, что по умолчанию компиляторы GCC могут скачиваться и не самые свежие, а всего лишь версии 6.1. Другие скачиваемые пакеты:

- bison 3.0.4;
- cmake 3.4.0;
- flex 2.6.0;
- mpich 3.2.0.

Надо сказать, что если эти компоненты в вашей системе уже установлены и инсталлятор их нашёл, то скачиваться и компилироваться они не будут, что сильно сократит время сборки. Особенно долго компилируется MPICH, а в Windows её компиляция вообще может закончится ошибкой. Так что я вам настоятельно рекомендую предварительно всё это установить из репозитория вашей операционной системы. Для Windows MPICH есть на сайте Microsoft [], только называется оно «Microsoft MPI». ☺ В качестве библиотеки MPI не обязательно устанавливать именно MPICH. Я, например, всегда пользовался OpenMPI, так что на ваше усмотрение можно ставить ту, что вам больше нравится.

Если у нас Linux, где нет своего готового пакета OpenCoarrays и в котором уже установлены компиляторы gfortran, gcc и g++ (или в CYGWIN эти компиляторы уже ранее были установлены), то запускаем инсталлятор с такими ключами, которые говорят, что компиляторы скачивать не надо:

```
./install.sh --with-fortran /usr/bin/gfortran \
             --with-c /usr/bin/gfortran \
             --with-cxx /usr/bin/gfortran \
             --yes-to-all
```

(здесь символ «\» означает, что строка с командой не заканчивается и следует продолжение).

Ключи:

--with-fortran – указывает, какой именно компилятор Фортрана мы используем;

--with-c – тоже самое про компилятор Си;

--with-cxx – тоже самое про компилятор Си++.

Компилятор Си++ в ключах я указал на всякий случай. Мало ли, вдруг мне захочется применить CoArray и в Си++ тоже, когда-нибудь в очень отдалённом будущем.☺ Однако без gfortran и без gcc при компиляции файлов совершенно точно не обойтись, так как системные библиотеки компилируются с помощью gcc, а модуль с функциями для Фортрана — с помощью gfortran.

На моём домашнем компьютере установлена ОС ROSA Linux Desktop Fresh R9, поэтому никакого пакета с CoArray в его репозиториях нет. Попробуем установить с помощью install.sh с ключами, уведомляющими, что компиляторы у меня уже есть и другие не нужны.

После работы инсталлятора в каталоге OpenCoarrays, и далее в prerequisites/installations образовались каталоги, из которых самые нужные четыре:

- bin — в нём содержатся два файла: caf и cafrun;
- include — в нём содержатся заголовочные файлы для Си и Фортрана;
- lib64 — здесь живут сами библиотеки OpenCoarrays. Lib64 – потому что у меня 64-ёх разрядная ОС. Для 32-ух разрядной, каталог будет называться lib;
- share — здесь справочные man-файлы по использованию caf и cafrun.

Файлы caf и cafrun выполняют ту же функцию, что и mpifort с mpirun из пакетов MPI. Нетрудно предположить, что количество ключей компиляции с применением CoArray значительно увеличилось, так что лучше не тратить по часу на составление строки этих ключей (которая, как в математике — функция стремящаяся к бесконечности ☺), а использовать специальные скрипты, которые подставляют то, что надо:

- caf — скрипт, компилирующий вашу программу с подстановкой необходимых ключей MPI и OpenCoarray;
- cafrun — запускает многопроцессную программу с указанным количеством процессов. Правда здесь нужно сказать, что использование вместо неё mpirun будет иметь точно такой же эффект. Так что смысл этого скрипта мне непонятен, разве что в плане названия, говорящего о том, что мы используем всё же CoArray, а не просто MPI.

Таким образом, компилироваться программа будет такой командой:

```
caf ./ИмяПрограммы.f90 -o ИмяПрограммы
```

а запускаться такой:

```
cafrun -np КоличествоПроцессов ./ИмяПрограммы
```

Примечание: caf — это ничто иное, как «CoArray Fortran». ☺

Лирическое отступление по поводу сборки CoArray своими силами.

Честно говоря, собирать большие и сложные программы (которые сам не писал ☺) я маленько опасаюсь, потому что были случаи, когда я, как ни старался, собрать программу не смог, несмотря на то, что все требуемые компоненты для сборки имелись. Поэтому и здесь у меня имелись изрядные опасения. Однако сборка прошла удачно, всё что я ожидал в двоичном виде появилось и настал самый животрепещущий момент — проверить, а как это сооружение будет работать, потому что сборка сборкой, но цель то у меня была другая — получить систему программирования с помощью CoArray.

С замиранием сердца запускаю компилирование тестовой программы «HelloWorld». Откомпилировалось. И, самое страшное — запуск программы на исполнение. Мать честная, всё работает!!! ☺ Хорошо, теперь сделаем что-нибудь посложнее — организуем обмен данными между процессами. Опять сердце замирает в ожидании… Но, видимо, на этот раз я родился в комассивной рубашке — всё отработало без сучка и задоринки. Уф, пронесло! Ничего плохого не случилось, одно только хорошее… ☺

Если в Linux всё прошло просто блестяще, то в CYGWIN с первого раза ничего не получилось — проблем навыдавала MPICH. Вывалилось с полсотни ошибок компиляции, причём ошибки такие, что с ходу и не сообразишь, в чём там проблема. Отказались компилироваться файлы, где вызывались функции winsock2, на которых строится программирование сети в Windows. Основная ошибка — неверный тип данных для функций из winsock.h для функций типа gethostbyaddr, gethostbyname и прочих вполне стандартных. Я изрядно удивился, но копаться в ошибках не стал, т. к. они были полностью на совести разработчиков MPICH. Решил, как Владимир Ильич Ленин, пойти другим путём — поставил OpenMPI, который стандартно входит в комплект пакетов CYGWIN и запустил инсталлятор с теми же ключами ещё раз. В отличие от первого раза, никаких ошибок не последовало и сборка пролетела очень быстро без всяких ошибок. В каталогах бинарных файлов появились необходимые caf и cafrun и готовые библиотеки.

Теперь осталось только скопировать (в Linux — с правами root, естественно) содержимое bin в /usr/bin, содержимое lib64 в /usr/lib64 (или, для 32-ух разрядной ОС lib в /usr/lib), содержимое include в /usr/include и содержимое share в /usr/share и можно тут же, ни секунды не медля, приступать к работе.

Применение CoArray

Работа с CoArray очень сильно напоминает работу с обычными массивами, только вместо круглых скобок в таких комассивах применяются квадратные, вот например, объявление переменной, которая должна присутствовать одновременно во всех запущенных процессах нашей программы:

```
! Для простой переменной
integer, codimension[*] :: x

! Или можно ещё проще
integer :: x[*]
```

Звёздочка в квадратных скобках обозначает, что мы не знаем, сколько у нас будет процессов. Если знаем, то в квадратных скобках можно поставить вполне определённое число.

Так же как и в MPI вводом\выводом в ресурсы, которые нежелательно использовать совместно, здесь будет заниматься только один процесс, который мы будем именовать главным. В отличие от MPI, который изначально ориентировался на Си, поэтому первый процесс, он же главный, имеет номер 0, в CoArray, как в Фортране и принято, первый процесс имеет номер 1. Так же как и в MPI в первую очередь нужно определиться, сколько всего процессов мы имеем и какой номер текущего процесса.

Кстати, по поводу названия функции. В CoArray процессы называются «образы» («images»), отсюда следует, что большинство функций, которые относятся к этому делу, будут иметь в своём название слово «image».

После того, как в главном процессе переменной «x» присвоено значение, его нужно разослать по остальным процессам:

```
integer :: img_count, img_current, i

! Количество доступных процессов
img_count = num_images()
! Номер текущего процесса
img_current = this_image()

! Если это первый процесс, главный по вводу\выводу,
! то здесь вводим с клавиатуры значение переменной
if ( img_current == 1 ) then
  read (*,*) x

  ! Заносим значение x текущего (первого) процесса
  ! во все остальные
  do i = 2, img_count
```

```
    x[i] = x
  end do
end if

! Ждём, пока все процессы получат свой значения для x
SYNC ALL
```

Здесь мы сначала определяем, в каком процессе мы находимся (функция this_image()) и если он равен номеру 1 (это и будет главный процесс). В комассивах, как и в обычных массивах, начальный индекс по умолчанию равен 1, и именно в нём выводим на экран запрос ввода значения в переменную. Далее, мы рассылаем полученое значение по всем остальным процессам . Остальные процессы в это время терпеливо ждут (SYNC ALL), пока не отработает текущий код по рассылке данных. Т.е. те процессы, что уже получили свои данные дожидаются, пока их получат все остальные и ничего не делают. Это убережёт от ситуации, когда алгоритм начнёт вычисления, не дождавшись правильных данных, т. к. процессы работают с разными скоростями.

По поводу SYNC ALL есть один нюанс. Если посмотреть внимательно на код, то становится понятно, что ожидать свои данные должны все процессы, кроме первого, т. к. первый как раз занят работой по получению и распределению данных, поэтому его синхронизировать не нужно. На этот случай есть похожий оператор:

```
SYNC IMAGES(*)
```

который приостанавливает все другие, кроме текущего, процессы, пока те не получат свои данные.

Если нам нужно обменяться данными между текущим и каким-то определённым процессом, это будет так же просто:

```
x[5] = x
```

Здесь переменная «x» процесса номер 5 получила данные из переменной «x» текущего процесса.

Есть несколько функций, начинающиеся с префикса «CO_», которые по аналогии с MPI заведуют пересылкой данных или отвечают за их сбор после вычисления в остальных процессах.

Процедуры для рассылки данных по всем процессам:

```
CO_BROADCAST(A, SOURCE_IMAGE)
```

здесь A — это данные, которые нужно разослать,
SOURCE_IMAGE — номер процесса, откуда беруться данные A для рассылки.

Функции сборки данных после вычисления по процессам:

```
CO_MAX(A [, RESULT_IMAGE] )
```

ищет максимальное значение по всем процессам работавшим с A и возвращает максимальное значение в A процесса с номером RESULT_IMAGE. Если RESULT_IMAGE не указывать, то максимальное значение будет разослано по всем процессам.

```
CO_MIN(A [, RESULT_IMAGE] )
```

то же самое, только идёт поиск минимального значения.

```
CO_SUM(A [, RESULT_IMAGE] )
```

здесь все значения A суммируются.

Если требуется более сложный алгоритм по сборке данных, то можно для этого написать собственную функцию и её название указывать в процедуре:

CO_REDUCE(A, OPERATOR, [, RESULT_IMAGE])

где OPERATOR — это название той функции, которая проводит сборку данных по процессам применяя какой-нибудь сложный алгоритм. Вот например:

```
program test
real :: val

call random_seed()
call random_number(val)

SYNC ALL

call co_reduce (val, result_image=1, operator=myfunc)

if (this_image() == 1) then
  if (val == 1.0) then
    print *, "Мы получили псевдослучайное число весьма близкое к 1.0"
  else
    print *, "Максимальное псевдослучайное число по процессам = ", val
  end if
end if

contains

pure function myfunc(a,b)
real, value :: a, b
real :: myfunc

if (val > 0.998) then
  myfunc = 1.0
else
  myfunc = val
end if

end function myprod

end program test
```

Здесь мы в каждом из процессов запускаем генератор псевдослучайных чисел (random_seed()\random_number()), затем на всякий случай синхронизируем все процессы (SYNC ALL), чтобы быть уверенными, что все процессы получат сгенерируемое псевдослучайное число к моменту сбора данных из них. А далее, с помощью co_reduce, собираем значения из процессов по сложному условию: если псевдослучайное число больше 0.998, то оно будет равно 1.0.

По функции сбора данных есть жёсткие правила, без соблюдения которых компилятор работать откажется:

- Функция должна быть без побочных эффектов, т. е. делать только то, что от неё требуют. Это тщательнейшим образом проверяет программист перед тем, как присвоить функции этот статус — **pure**;
- У функции обязательно должно быть два аргумента того же типа данных, что и собираемое значение. В эти аргументы заносятся собираемые по процессам (два соседних процесса) значения и сравниваются между собой, либо с ними проводят ещё какие-либо манипуляции. В данном примере они не используются, но это надо иметь в виду.

И широковещательная и сборочные процедуры в обязвтельном порядке должны быть доступны всем процессам.

Теперь поговорим о вещах несколько более сложных, если не знать которые, то дальше поработать уже никак не удастся. Я имею в виду работу с массивами неопределённых заранее размеров.

Несмотря на большой соблазн сделать как в MPI, т. е. выделить память под массив и заполнить его данными в первом процессе, а потом разослать уже готовый массив остальным, увы, ничего из этого не выйдет. Самое печальное — ни компилятор, ни работающая программа не выдадут вам ни одной ошибки, просто на процедуре рассылки данных программа перестанет что-либо делать. Поэтому, как только вы получили размер массива, его нужно разослать по всем процессам и уже в каждом процессе выделять память под этот массив. После чего в первом процессе можно заполнять массив данными и рассылать по остальным процессам.

Теоретически, это всё, что нужно, чтобы немедленно начать работать. Однако по аналогии с MPI без справочника под носом к серьёзным программам лучше не приступать. ☺

3.11 Пара слов о скорости выполнения программ

Хотя Фортран и считается языком быстрых вычислений, но иногда можно заметить, что какие-то программы хоть и написаны на Фортране, скоростью никак не блещут. В чём проблема, вроде бы автор обещал бешеную скорострельность? ☺ При рассмотрении исходных кодов «медленных» программ регулярно выясняется, что проблема у программы не техническая, а, так сказать, организационная и к языку это не имеет ни малейшего отношения, т. к. подобный «медленный» результат будет и при использовании этого же алгоритма с любым другим языком. А может будет ещё и хуже. Ошибка в этом случае допущена ещё на этапе проектирования программы, т. е. при составлении алгоритма. Давайте посмотрим что может мешать «скорострельности» программы.

Ни для кого не секрет, что в самом начале алгоритм составляется для некоей идеальной вычислительной машины, обладающей бесконечными ресурсами. Для небольших программ, с небольшими объёмами данных в этом нет ничего предосудительного. Сегодняшний компьютер таким программам и будет казаться идеальным вычислителем. Проблемы начинаются, когда объём данных начинает превышать некую планку, когда занятая всеми программами и всеми данными для них оперативная память начинает приближаться к границе физически установленных на компьютер планок с ОЗУ. Ещё чуть-чуть — и операционная система начинает сбрасывать код и данные в виртуальную память, т. е. на жёсткий диск. А скорость при работе с жёстким диском, как минимум, в 1000 раз меньше скорости работы с оперативной памятью. Что мы видим при этом в окошке диспетчера задач на вкладке «Быстродействие»? Процессор загружен на все 100%, физическая память загружена на 100% и процессор, вместо того, чтобы заниматься полезными делами, занят перегонкой данных из памяти на диск и с диска обратно в память. Короче, мраки и вешалки…

Что делать? А давайте посмотрим из чего состоит наша программа. Начнём с данных. Не секрет, что в подавляющем большинстве случаев для вычислений используются так называемые численные методы, которые хоть и не дают точных результатов, зато работают быстро. Для них используются какие-то типы данных, которые имеют пределы — минимальное значение и максимальное значение. Здесь можно выявить первый скоростной фактор — а какие именно типы используются.

К примеру, целочисленные вычисления всегда проходят быстрее, чем вычисления с плавающей точкой. Надо посмотреть, где можно заменить числа нецелые на числа целые.

Размер типа данных. В пределах каких значений будут ваши данные? Real(16) обрабатываются дольше, чем real(8). Опять нужно посмотреть, можно ли заменить длинные типы на более короткие, может быть ваши исходные данные и результаты вполне влезут в более короткий тип.

Использование подпрограмм и функций. Подпрограммы и функции вызываемые из внешней библиотеки обычно работают дольше, чем встроенные в Фортран. Опять нужно внимательно посмотреть — может быть в Фортране есть аналоги внешних функций.

В качестве повода для размышления.

Вообще применение подпрограмм и функций увеличивает расходы ресурсов, как процессора, так и памяти. Программный код с отдельными функциями смотрится превосходно. Вот только слегка проигрывает в скорости работы. В некоторых случаях вполне можно заменить вызов функции вставкой кода этой функции непосредственно в тело программы. Очень большого прироста скорости не будет, тем не менее таким способом можно получить некоторый выигрыш. Здесь определённого совета я вам дать не могу, потому что нужно соблюдать баланс между скоростью работы и понятностью кода. А тут уже надо смотреть на многое — например, требует ли программа постоянной модификации и сопровождения. Если да, то лучше делать программу понятнее. Если нет, по типу сделали и забыли, то можно всё сделать в главной программе.

Работа с внешними устройствами, например с файлами. Если такую работу включить в процесс вычислений, мы опять сели в лужу работы с диском, который в 1000 раз замедлит скорость вычисления. Лучше сделать мух отдельно, а котлеты отдельно, т. е. работу с файлами вынести за рамки основных вычислений и вообще свести к минимуму.

Про работу с данными есть ещё интересный нюанс. Самые быстрые вычисления будут в том случае, если все без исключения данные влезли в регистры процессора. Правда тут уже стоит обратится не к языкам высокого уровня, а к ассемблеру. ☺ Тем не менее, имейте в виду. Следующий порог быстродействия — когда все данные влезают в кэш процессора. Самый быстродействующий кэш — кэш первого уровня, он работает на частоте процессора. Кэши второго и третьего уровня работают помедленнее, но всё равно намного быстрее, чем обычная оперативная память.

Параллельные вычисления работу программы обычно ускоряют, но далеко не всегда. Иногда получается так, что большие объёмы пересылаемых между потоками\процессами данных наоборот замедляют работу программы. Тут нужны сравнительные тесты с теми объёмами данных, которые планируется использовать при повседневной работе программы.

Как показывает практика, выбор количества потоков\процессов в параллельной программе влияет на быстродействие. В многопоточной программе лучше за пределы предоставляемых процессором вычислительных ядер не выходить, чтобы процессор не начал терять дополнительное время на переключение потоков в вашей задаче (он и так это уже делает, чтобы обслуживать другие программы — зачем его затруднять ещё больше?). В многопроцессной программе максимальное быстродействие (при запуске на одном компьютере) будет при количестве процессов не больше, чем физических ядер процессора (не тех, что показывает процессор при включённом гипертрейдинге, а тех, что управляют процессором и в спецификации интеловских процессоров обозначаются как «Cores»).

3.12 Как переделать тексты программ из старого формата в новый

Хотя этот подраздел уже сильно напоминает практику, но всё-таки здесь будут показаны те принципы, которыми стоит руководствоваться при переделки кодов старых программ в новый формат. Так что я его решил поместить именно в раздел теории.

В Интернете можно найти громадное количество текстов программ на Фортране практически на все случае жизни. К сожалению, наверное ¾ из них будут написаны на старом Фортране, в старом стиле. Оно и понятно, если алгоритм кардинально не поменялся, зачем переписывать готовую и хорошо работающую программу. И современные компиляторы пока ещё могут большинство таких старых программ компилировать без проблем. Однако как только все компиляторы перейдут на стандарт Фортран-2018, вся эта громадная масса кода перестанет компилироваться, потому что в Фортране-2018 уберут как раз те фундаментальные основы, которые делали код программ непонятным, но на которых всё фортрановское программирование и строилось. Поэтому понимание, как именно нужно переделывать программы в современный формат необходимо прямо сейчас. Особенно это касается студентов. ☺

Вне зависимости от специфики программы, начинать нужно с символа обозначения комментариев — «C». В старом коде он всегда располагается в первой колонке строки, поэтому прекрасно виден. Его необходимо заменить на символ комментария нового формата — «!». Дело в

том, что как только вы перейдёте на новый формат, символ «C» в первой колонке перестанет восприниматься компилятором как комментарий, поэтому на каждую такую строчку компилятор будет воспринимать как ошибку.

Давайте рассмотрим маленький, но очень показательный пример, где не будет компилироваться практически каждая строчка кода. В книге [ ] я взял программу решения системы линейных уравнений:

```
      real A(20,21), X(20)
1     type *, " ?N"
      accept *, N
      call matr(A, X)
      call gauss(N, A, X, S)
      if (S .ne. 0.) GO TO 2
      type *, " DET=0"
      GO TO 1
2     DO 3 I=1, N
            TYPE 4, I, X(I)
      GO TO 1
4     format (1X, "X", I2, "=", 1PE13,6)

      end
```

На каких строках будут ошибки:

Ввод\вывод данных — accept\type. Оператор ACCEPT, взять данные с клавиатуры, существовал до Фортран-77 включительно, но потом его упразднили. Упоминания про TYPE, аналог PRINT, я не нашёл, может быть плохо искал, но это могло быть какое-то расширение для ЭВМ ЕС, учитывая год издания книги. Эти проблемные операторы легко заменяются на READ и PRINT. Кусок кода в новом формате будет выглядеть так:

```
integer :: N            ! Потому что объявить N позабыли
print *, " ?N"
read (*,*) N
```

Строка с условием. Она пока ещё будет работать, но там есть непонятные символы «.ne.». Это старое обозначение операции сравнения данных «НЕ РАВНО». Кстати, из Фортрана такие названия перекочевали в ряд других систем. Чем можно заменить такие не очень читаемые операции можно посмотреть в табл. 4.

Таблица 4. Старые и современные обозначения операций сравнения данных

Старое обозначение	Новое обозначение	Название
.eq.	==	Равно
.ne.	/=	Не равно
.gt.	>	Меньше
.ge.	>=	Меньше или равно
.lt.	<	Больше
.le.	<=	Больше или равно

Из IF желательно убрать так же и GO TO. Здесь мы сталкиваемся с тем, что в подобных конструкциях действие по условию не такое, как в новом Фортране или других языках. IF без GO TO работает так:

```
ЕСЛИ (условие_верное) ТО
      Программный_Блок_1
ИНАЧЕ
```

```
        Программный_Блок_2
КОНЕЦ ЕСЛИ
```

В старых программах IF с GO TO обозначали (~ 99% случаев) что при .TRUE. в условии IF необходимо пропустить Программный_Блок_1, поэтому для его выполнения нужно писать так:

```
ЕСЛИ (условие_НЕ_верное) ТО
        Программный_Блок_1
КОНЕЦ ЕСЛИ

Программный_Блок_2
```

т. е. в исправленной программе необходимо будет изобразить IF в полном формате и поменять операцию сравнения на противоположную. Новый код будет выглядеть так:

```
if (S == 0.) then
        write (*,*) " DET=0"
end if
```

Но это ещё не всё…☺ Поскольку из текста явно видно, что S == 0 это явная ошибка, а при такой ошибке в исходной программе предлагается вновь начать вводить уравнение, то стоит немного задуматься — а мы вообще это уравнение собираемся вводить вручную или предпочитаем брать кем-то полученные заранее данные из файла? Скорее всего второе. Поэтому продолжать программу, которая прочитает тот же файл с теми же «нерешаемыми» исходными данными не стоит. Программа получилась не универсальная.

И здесь у нас возникает развилка: либо уравнение нерешаемое и мы выходим из программы, либо оно решаемое и мы выводим на экран или в файл данные. Здесь стоит дополнить наше IF оператором ELSE, но сначала надо разобраться с циклом, который тоже совсем не современен.

Раньше цикл DO использовали совместно с номером метки программного кода, которая обозначала последнюю строку цикла. Нынче же у цикла есть явное начало и явный конец, поэтому цикл исправляем на современный вид:

```
do I=1, N
        write (*,4) I, X(I)
end do

4       format (1X, "X", I2, "=", 1PE13,6)
```

и тут мы видим ещё одну не объявленную переменную, которую надо будет объявить в начале программы. Вот теперь можно сделать IF с ELSE:

```
if (S == 0.) then
        write (*,*) " DET=0"
else
        do I=1, N
                write (*,4) I, X(I)
        end do
end if
```

Вот теперь программа будет выглядеть вполне по современному — все переменные заранее объявлены, отменённые операторы заменены на существующие, от GO TO мы избавились и условия с циклами приобрели логически законченный вид.

По поводу циклов нужно добавить, что в 70-ых годах, и особенно после появления Фортран-77, циклы стали закрывать оператором «CONTINUE», правда от метки это всё равно не избавляло. Цикл в этом случае выглядит так:

```
C  Set input values of the input/output parameters.
      T = 5.0
      DO 20, I = 1, N
            VM(I) = 1.0
20    CONTINUE
```

Здесь начало и конец цикла определить очень легко, поэтому переделка в новый формат займёт немного времени.

А вот кстати, про одну вещь мы явно забыли — массив в исходной программе снабжён постоянными индексами, хотя по ходу программы явно видно, что количество ячеек массива заранее неизвестно. Это тоже надо исправить, объявив безразмерный массив, а потом выделить для него память.

Вот что у нас в результате получилось:

```
program linear_eq
! Заставим компшлятор проверять соответствие используемых
! и объявленных переменных
implicit none

! Объявим все необходимые переменные
integer :: N    ! Количество индексов в одном измерении
integer :: I    ! Для перебора ячеек массива
! Массив индексов и св. членов
real, allocatable, dimension(:,:) :: A
real, allocayable, dimension(:) :: X    ! Массив ответов

! Запрашиваем количество ячеек массива в одном измерении
write (*,*) "Введите размер одного измерения массива:"
read (*,*) N

! Выделяем память для массивов
allocate (A(N,N))
allocate(X(N))

! Вызываем необходимые подпрограммы
call matr(A, X)
call gauss(N, A, X, S)

! И проверяем результат вычисления
if (S == 0.) then
      write (*,*) " DET=0"
else
      do I=1, N
            write (*,4) I, X(I)
      end do
end if

! Формат вывода результата
4     format (1X, "X", I2, "=", 1PE13,6)

end program linear_eq
```

Несмотря на то, что количество строк переделанной программы явно увеличилось по сравнению с исходной, новая программа читается, а главное понимается, без малейших затруднений. Этому способствует ещё и добавление комментариев, которых раньше не было.

При передаче массивов с заранее неизвестным количеством ячеек в подпрограмму или функцию, использовался хитрый трюк, который современные компиляторы ни в коем случае не пропустят:

```
      SUBROUTINE SUB1(N, A)

      INTEGER N          ! Здесь передаётся количество ячеек массива
      REAL A(1)          ! А это сам массив. Тут кроется подвох —
                         ! количество ячеек на самом деле не равно 1

      DO 10 I = 1, N
            A(I) = какое-то_вычисление
10    CONTINUE

      END
```

такую ситуацию можно исправить двумя способами — либо применить традиционную методику работы с безразмерными массивами, с выделением памяти под массив:

```
subroutine sub1(N, A)

integer :: N                              ! Здесь передаётся количество ячеек массива
real, allocatable, dimrnsion(:) :: A      ! Это безразмерный массив
integer :: i

! Выделяем память под массив
allocate(A(N))

! А теперь его используем
do I = 1, N
      A(i) = какое-то_вычисление
end do

end subroutine sub1
```

либо указывая передаваемое количество ячеек, как размер массива:

```
subroutine sub1(N, A)

integer :: N                    ! Здесь передаётся количество ячеек массива
real, dimrnsion(N) :: A         ! Это безразмерный массив
integer :: i

! А теперь его используем
do I = 1, N
      A(i) = какое-то_вычисление
end do

end subroutine sub1
```

По собственному опыту могу сказать, что применение второго варианта очень хорошо ускоряет работу подпрограммы.

В качестве инициализации переменных, раньше регулярно применялся оператор DATA. Выглядело это так:

```
real ONE, TWO
```

```
data ONE/1.01/, TWO/2.02/
```

В новом Фортране это будет выглядеть так:

```
real :: ONE = 1.01, TWO = 2.02
```

хотя оператор DATA пока ещё не планируют отменять и в принципе его можно не трогать.

Другая проблема с объявлением данных возникнет после принятия стандарта Фортран-2018 для оператора COMMON, который обозначает блок глобальных данных, которые видны не только в главной программе, но и в подпрограммах. Этот оператор будет признан устаревшим и в стандарте 2020 его, можно в этом не сомневаться, отменят вообще, т. к. по утверждению разработчиков эта конструкция часто приводит к ошибкам. В качестве собственного комментария хочу сказать, что его собирались объявить устаревшим ещё в Фортран-90, но видимо упустили из виду.

Проблема с общей видимостью переменных в новом Фортране решается либо использованием подпрограмм, которые являются составной частью основной программы т. е. расположены в тексте главной программы после оператора CONTAINS, либо использованием модулей, в которых переменные из главной программы видны по умолчанию.

В старом коде объявление COMMON выглядело так:

```
program Rocket
  real Rocketmass,Fuelmass,Impulse,Burnrate,Endthrust
  common /c1/Rocketmass,Fuelmass,Burnrate,Initthrust,Endthrust

  ...

end


  function Thrust(t)
  implicit none

  common /c1/Rocketmass,Fuelmass,Burnrate,Initthrust,Endthrust
  real Rocketmass,Fuelmass,Burnrate,Initthrust,Endthrust
  real Thrust,t

  Thrust=Initthrust
  if (t .ge. Endthrust) then
    Thrust=0
  endif

  return
  end
```

В новом Фортране это выглядит так:

```
program Rocket
real :: Rocketmass,Fuelmass,Impulse,Burnrate,Endthrust

...

contains

real function Thrust(t)
implicit none
```

```
      real t

      Thrust=Initthrust
      if (t >= Endthrust) then
        Thrust=0
      endif

      end function Thrust
end program Rocket
```

Таким образом мы избавились от дублирования объявления общих переменных и в главной программе и во входящей в неё функции. В функции переменные главной программы будут видны.

Это самые лёгкие «неправильности» старых программ и они обычно решаются влёт. Наибольшие трудности в «распутывании» смысла старых программ вызывает многочисленные «GO TO». Если вам попалась программа без перекрёстных ссылок, когда «GO TO» отсылает вас сначала вперёд, потом назад, потом опять вперёд, а потом опять назад, то считайте, что вам крупно повезло и восстановить логику работы программы можно сравнительно быстро. А вот с перекрёстными ссылками зачастую приходится рисовать алгоритм работы, иначе запутаешься очень быстро. Вот именно в таких случаях корифеи программирования очень многословно и эмоционально ругают «GO TO». ☺

Разберём случай средней сложности. На сайте [] найдена подпрограмма обработки ошибок одной из вычислительных подпрограмм. Это случай применения вычисляемого GO TO. Я её довольно сильно упростил, чтобы только продемонстрировать идею, как её привести в удобочитаемый вид:

```
   SUBROUTINE UTZF10(IERR)

 140 PRINT 141
     GO TO (133,143,144), IERR
 141 FORMAT(' БИБЛИОТЕКА НИВЦ МГУ,ПОДПРОГРАММА ZF14R:',
    *' ФАТАЛЬНАЯ ОШИБКА')

 133 PRINT 132
 132 FORMAT(' N 1 - СХОДИМОСТЬ НЕ ДОСТИГНУТА В ПРЕДЕЛАХ ЗАДАННОГО',
    *' ЧИСЛА ИТЕРАЦИЙ'/7X,'ПО КРАЙНЕЙ МЕРЕ',
    *' ДЛЯ ОДНОГО КОРНЯ. ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ'/7X,'КОРНЕЙ',
    *' ПОЛОЖЕНЫ РАВНЫМИ 3.4E38')
     RETURN

 143 PRINT 145
 145 FORMAT(' N 2 - ЗНАЧЕНИЯ ОДНОГО ИЛИ НЕСКОЛЬКИХ КОРНЕЙ НЕ БЫЛИ',
    *' НАЙДЕНЫ,'/7X,'ПОСКОЛЬКУ ПРОИЗВОДНАЯ ЗАДАННОЙ',
    *' ФУНКЦИИ СТАЛА СЛИШКОМ МАЛОЙ.'/7X,'ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ',
    *' КОРНЕЙ ПОЛОЖЕНЫ РАВНЫМИ 3.4E38')
     RETURN

 144 PRINT 146
 146 FORMAT(' N 3 - ЗНАЧЕНИЯ НЕСКОЛЬКИХ КОРНЕЙ НЕ БЫЛИ НАЙДЕНЫ,',
    *' ПОСКОЛЬКУ ЛИБО'/7X,'СХОДИМОСТЬ НЕ БЫЛА',
    *' ДОСТИГНУТА В ПРЕДЕЛАХ ЗАДАННОГО ЧИСЛА'/7X,'ИТЕРАЦИЙ,ЛИБО',
    *' ПРОИЗВОДНАЯ ЗАДАННОЙ ФУНКЦИИ',
    *' СТАЛА СЛИШКОМ МАЛОЙ.'/7X,'ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ КОРНЕЙ',
    *' ПОЛОЖЕНЫ РАВНЫМИ 3.4E38',/7X,
    *'В ПЕРВОМ СЛУЧАЕ И - 3.4E38 - ВО ВТОРОМ.')
     RETURN
```

```
      END
```

Здесь сразу бросается в глаза большое количество символов «*» в начале строк. К умножению это не имеет никакого отношения, а всего лишь обозначает продолжение предыдущей строки. В новом формате такое «продолжение работать не будет, поэтому все символы «*» в началах строк мы удаляем, а вот в конце предыдущих строк ставим символ «&», который в новом формате обозначает «продолжение следует...».

Странный вид «GO TO» наводит на мысль, что это множественный выбор в зависимости от значения чего-то. И действительно, в зависимости от значения переменной «IERR» в которую помещается номер ошибки либо 1, либо 2, либо 3, осуществляется переход на метку, которая у «GO TO» в скобках по порядку:

1 — переход на метку 133;
2 — переход на метку 143;
3 — переход на метку 144.

Для замены тут само собой напрашивается конструкция «SELECT CASE». И в переделанном виде подпрограмма будет выглядеть так:

```
      subroutine utzf10(ierr)
      implicit none

      integer :: ierr

      print 141
      select case (ierr)
            case (1)
                  print 132
            case (2)
                  print 145
            case (3)
                  print 146
      end select

      141 FORMAT('БИБЛИОТЕКА НИВЦ МГУ, ПОДПРОГРАММА ZF14R: ',&
            ' ФАТАЛЬНАЯ ОШИБКА')

      132 FORMAT(' N 1 - СХОДИМОСТЬ НЕ ДОСТИГНУТА В ПРЕДЕЛАХ ЗАДАННОГО',&
     ' ЧИСЛА ИТЕРАЦИЙ'/7X,'ПО КРАЙНЕЙ МЕРЕ',&
     ' ДЛЯ ОДНОГО КОРНЯ. ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ'/7X,'КОРНЕЙ',&
     ' ПОЛОЖЕНЫ РАВНЫМИ 3.4E38')

      145 FORMAT(' N 2 - ЗНАЧЕНИЯ ОДНОГО ИЛИ НЕСКОЛЬКИХ КОРНЕЙ НЕ БЫЛИ',&
     ' НАЙДЕНЫ,'/7X,'ПОСКОЛЬКУ ПРОИЗВОДНАЯ ЗАДАННОЙ',&
     ' ФУНКЦИИ СТАЛА СЛИШКОМ МАЛОЙ.'/7X,'ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ',&
     ' КОРНЕЙ ПОЛОЖЕНЫ РАВНЫМИ 3.4E38')

      146 FORMAT(' N 3 - ЗНАЧЕНИЯ НЕСКОЛЬКИХ КОРНЕЙ НЕ БЫЛИ НАЙДЕНЫ,',&
     ' ПОСКОЛЬКУ ЛИБО'/7X,'СХОДИМОСТЬ НЕ БЫЛА',&
     ' ДОСТИГНУТА В ПРЕДЕЛАХ ЗАДАННОГО ЧИСЛА'/7X,'ИТЕРАЦИЙ, ЛИБО',&
     ' ПРОИЗВОДНАЯ ЗАДАННОЙ ФУНКЦИИ',&
     ' СТАЛА СЛИШКОМ МАЛОЙ.'/7X,'ЗНАЧЕНИЯ СООТВЕТСТВУЮЩИХ КОРНЕЙ',&
     ' ПОЛОЖЕНЫ РАВНЫМИ 3.4E38',/7X,&
     'В ПЕРВОМ СЛУЧАЕ И - 3.4E38 - ВО ВТОРОМ.')

      end subroutine utzf10
```

Как видите, без «GO TO» подпрограмма стала немного компактнее и главное — совершенно понятной.

Другая разновидность вычисляемого GO TO встречается достаточно редко, но тем не менее об этом стоит упомянуть:

```
      IF (LB.EQ.0) ASSIGN 9 TO K
      ...
      GO TO K
      ...
9     AKX = 0.
```

Здесь номер метки предварительно вычисляется и присваивается с помощью «ASSIGN» какой-либо переменной. В дальнейшем эта переменная используется в качестве метки в операторе GO TO. Никаким другим способом присвоить номер метки какой-либо переменной было нельзя, кроме как оператором «ASSIGN».

Это самый необходимый минимум сведений по переделке программ из старого формата в новый, который позволит в большинстве случаев без проблем провести переделку.