РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ
Факультет физико-математических и естественных наук

М. Н. Геворкян, А. В. Королькова, Д. С. Кулябов

Параллельное программирование

Лабораторные работы

Учебное пособие

Москва
Российский университет дружбы народов
2014

УДК 004.272 (076.5)
ББК 018.2*32.973
Г 27

Утверждено
РИС Учёного совета
Российского университета
дружбы народов

Рецензент —
старший научный сотрудник ЛИТ ОИЯИ
кандидат физико-математических наук *О. И. Стрельцова*;
начальник сектора телекоммуникаций УИТО РУДН
кандидат физико-математических наук, доцент *К. П. Ловецкий*

Геворкян М. Н.

Г 27 Параллельное программирование : лабораторные работы : учебное пособие / М. Н. Геворкян, А. В. Королькова, Д. С. Кулябов. — Москва : РУДН, 2014. — 87 с. : ил.

ISBN 978-5-209-06152-6

Пособие представляет собой лабораторный практикум по дисциплине «Параллельное программирование» и предназначено для студентов направлений «Математика и компьютерные науки», «Фундаментальная информатика и информационные технологии», «Прикладная математика и информатика».

УДК 004.272 (076.5)
ББК 018.2*32.973

ISBN 978-5-209-06152-6

Оглавление

1. Fortran

1.1. История языка Fortran

В 1953 г. группа учёных-программистов из IBM во главе с Джоном Бэкусом начала разработку альтернативы языку ассемблер для мейнфрейма IBM 704. Основной задачей нового языка программирования стало упрощение процесса написания программ.

Первый компилятор FORTRAN появился в апреле 1975 г. FORTRAN стал первым языком высокого уровня, а его компилятор — первым оптимизирующим компилятором. По производительности программы, написанные на FORTRAN, лишь немногим уступали программам, написанным вручную на ассемблере.

Язык быстро завоевал популярность и стал использоваться для написания вычислительных программ в самых различных областях науки и инженерного дела. Собственно вычислительные задачи и были основным назначением нового языка, название которого произошло от FORMula TRANslator.

Успех Фортрана повлёк его активное развитие и с 1958 по 1961 г. — вышло три редакции языка Fortran II, III, IV. В 1966 г. Американская Ассоциация стандартов (ASA, а ныне ANSI) выпустила стандарт Fortran 66, основанный на редакции Fortran IV со множеством нововведений. Работа над стандартом окончательно завершилась лишь в 1972 г. Следующим стандартом стал Fortran 77 (работа шла с 1977 г. по 1980 г.).

Современная версия языка берёт своё начало с начала 90-х гг. XX века, когда был выпущен стандарт Fortran 90. Начиная с этой версии, Фортран приобрёл все атрибуты современного процедурного языка высокого уровня. Вышедшие позже Fortran 95 (1997 г.), Fortran 2003 и Fortran 2008 (2008–2010 гг.) добавили некоторые новшества, но серьёзно синтаксис не поменяли.

Хотя Фортран и является самым старым языком высокого уровня, он активно развивается до сих пор (о чем свидетельствует появление новых стандартов) и используется на практике в разнообразных научных и инженерных задачах.

Важно отметить, что за время существования Фортрана на нём было написано множество хорошо отлаженных и оптимизированных библиотек (например, LAPACK, IMSL). Кроме этого, Фортран сравнительно прост и лаконичен. Он обладает удобными средствами работы с массивами (в том числе и динамическими), встроенным типом комплексных чисел и удобными средствами ввода-вывода.

Наиболее распространённым бесплатным компилятором Фортрана является `gfortran`, который поддерживает практически все конструкции стандарта Fortaran 95 и многие конструкции стандартов Fortaran 2003 и 2008.

1.2. Элементы и объекты программы

Сразу же рассмотрим простейший пример программы `Hello World!` на Фортране:

```
program HelloWorld
  print *, "Здравствуй Мир!"
end program HelloWorld
```

Для компиляции программы следует сохранить исходный код в файле с расширением `f90`. При этом рекомендуется именовать файл с исходным кодом так же, как названа программа (в нашем случае файл будет называться `HelloWorld.f90`). Сохранять файл следует в кодировке utf, в этом случае не будет проблем с сообщениями, напечатанными кириллическими символами. Для компиляции программы следует воспользоваться одной из следующей команд:

```
gfortran HelloWorld.f90 -o HelloWorld
gfortran HelloWorld.f90
```

В первом случае указана опция `-o`, поэтому созданный исполняемый файл будет именоваться `HelloWorld`. Во втором случае опция `-o` не указана, поэтому исходный файл будет назван стандартным именем `a.out`. Напомним, что запуск исполняемого файла из командной строки осуществляется следующим образом:

```
./HelloWorld
```

После запуска программы `HelloWorld` в консоль будет напечатано сообщение:

```
Здравствуй Мир!
```

1.2.1. Структура программы

В общем случае структура программы на языке Фортран укладывается в следующую схему:

```
program <имя_программы>
<Раздел_описания_переменных>
<Раздел_операторов>
[Внутренняя подпрограмма]
end program <имя_программы>
[Внешняя подпрограмма]
```

Прежде чем переходить к пояснению, приведём ещё один пример программы:

```
! Комментарии начинаются с восклицательного знака
program Example
! Раздел описания переменных
! Объявление трёх действительных переменных
  real :: x, y, z
```

```
! Раздел операторов
! Присвоение значения переменной
  x = 6.2
  y = 2.0
  z = x*y
! Вывод значения переменной на экрана
  print *, "z = ", z
end program Example
```

В данном примере содержится только одна программа, называемая *главной*. Текст главной программы открывается ключевым словом `program` со следующим за ним именем программы и должен быть закрыт посредством `end` с тем же именем программы. Любая программа начинается с раздела объявления переменных. Только когда все переменные объявлены, можно переходить к разделу операторов, в котором совершаются действия над объявленными ранее переменными. Присвоение значений переменным можно осуществлять и в области описания, но рекомендуется поступать так лишь в случае констант.

Алфавит Фортрана (Fortran 90) состоит из 26 букв английского алфавита (регистр символов не играет роли и различается только в строковых переменных), а также из перечисленных ниже символов:

=	равно	:	двоеточие
+	плюс	_	нижнее подчёркивание
−	минус	!	восклицательный знак
*	умножить	"	кавычки
/	слэш	%	процент
(	левая скобка	&	амперсанд
)	правая скобка	;	точка с запятой
,	запятая	<	меньше
.	точка	>	больше
$	знак доллара	?	вопросительный знак
'	апостроф		

Все остальные символы можно использовать только в комментариях и в строках. Заметим также, что длина одной строки текста программы не должна превышать 132 символа и 39 строк продолжения. Строки продолжения должны начинаться с символа &.

1.2.2. Типы данных

Следующие типы данных являются встроенными и доступны всегда:

`integer`	Целый
`real`	Вещественный
`complex`	Комплексный
`character`	Текстовой
`logical`	Логический

Целый, вещественных и комплексный типы данных могут быть как одинарной, так и двойной точности:

Точность	integer	real	complex
Одинарная	4 байта	4 байта	8 байт
Двойная	8 байт	8 байт	16 байт

Рассмотрим на примере объявление переменных различных типов и присваивание им конкретных значений:

```
program Example
  implicit none
  integer :: i
  real :: x1, x2, x3
  complex :: z
  ! Ниже объявляются переменные двойной точности
  integer(kind = 8) :: j
  real(kind = 8) :: y
  ! Обратите внимание, что для комплексных чисел
  ! двойной точности следует указывать kind=8, а не 16:
  ! на действительную и мнимую часть выделяется
  ! по 8 байт
  complex(kind = 8) :: u
  ! Логическая переменная
  logical :: l, t
  ! Строка длиной не более 50 символов
  character(len=50) :: s
  ! Константа вещественного типа
  real, parameter :: pi = 3.1416
  ! Присвоим объявленным переменным значения
  i = 2
  x1 = 3.6
  x2 = 1e3 ! 1000
  x3 = 2.3e-4 ! 0.00023
  z = (2.0,-3.0)
  l = .true.
  t = .false.
  s = "Строка не более 50 символов длиной"
end program Example
```

Стоит пояснить директиву `implicit none`, указанную до области описания переменных. `Fortran 90` не принадлежит к языкам со строгой типизацией и, если тип некоторой переменной не задан, то он определяется автоматически по имени. Переменные с именами, начинающимися на `i, j, k, l, m, n`, относятся к целому типу одинарной точности, остальные переменные — к вещественному типу одинарной точности. Упомянутая директива `implicit none` устанавливает строгую типизацию, и, во избежание лишних ошибок и путаницы, настоятельно рекомендуется во всех

программах указывать её и объявлять все использованные переменные явным образом.

Объявление константы `pi`

```
! Константа вещественного типа
  real, parameter :: pi = 3.1416
```

иллюстрирует использование атрибутов переменной. Атрибуты позволяют указать дополнительные свойства переменной. В данном примере атрибут `parameter` указывает, что объявляемая переменная является константой, и присвоенное ей в области описания значение изменяться не может. Массивы также задаются с помощью соответствующего атрибута, но их мы рассмотрим позже в соответствующем разделе. Другие атрибуты также будут рассматриваться по мере необходимости.

1.2.3. Встроенные операции и функции

Как уже говорилось выше, Фортран изначально создавался для математических расчётов, поэтому в нём реализованы практически все элементарные функции и алгебраические операции. Кроме операторов сложения, вычитания, умножения и деления (`+ - * /`) в Фортран встроен оператор возведения в степень `**` и следующие математические функции: `sqrt()` — квадратный корень, `sin(), cos(), tan()` — тригонометрические функции, `asin(), acos(), atan()` — обратные тригонометрические функции, `exp(), log(), log10()` — экспонента, натуральный и десятичный логарифмы, `sinh(), cosh(), tanh()` — гиперболические функции. Заметим, что перечисленные функции принимают в качестве аргумента как действительные, так и комплексные переменные одинарной и двойной точности.

Функция	Пояснение
`nint(x)`	ближайшее целое $\lfloor x \rceil$
`anint(x)`	вещественная форма ближайшего целого
`ceiling(x)`	округление вверх $\lceil x \rceil$
`floor(x)`	округление вниз $\lfloor x \rfloor$
`abs(x)`	абсолютное значение числа $\lvert x \rvert$
`aimag(z)`	мнимая часть числа $\mathrm{Im} z$
`real(z)`	действительная часть числа $\mathrm{Re} z$
`conjg(z)`	сопряжение комплексного числа $\bar{z}$
`cmplx(x,y)`	комплексное число $x + \mathrm{i}y$
`mod(a,b)`	остаток от деления a на b $a \mod b$
`sign(x,y)`	$\mathrm{sign}(y)\lvert x \rvert$
`sign(1,x)`	$\mathrm{sign}(x)$

Опишем теперь операторы, применяемые для работы с логическими переменными. Логических констант две

```
.true.
.false.
```

Логические *отношения* позволяют сравнить две переменные между собой. Они часто используются в операторах условия и цикла. Допускаются как специфические для фортрана обозначения (первая колонка в таблице ниже), так и стандартные для большинства языков программирования (вторая колонка):

`.lt.`	`<`	$<$
`.le.`	`<=`	$\leq$
`.eq.`	`==`	$=$
`.ne.`	`/=`	$\neq$
`.gt.`	`>`	$>$
`.ge.`	`>=`	$\geq$

Важно, что проверку *равенства* вещественных чисел, учитывая неизбежную погрешность их представления, следует производить с некоторой точностью, для чего можно использовать функцию `abs()`, например:

```
abs(x-2.6) < 1e-10
```

Логические *операции* позволяют вычислять логические выражения, что обычно используется для составления более сложных условий в операторах цикла или ветвления. В таблице ниже логические операции расположены в порядке приоритетов вычисления:

`.not.`	отрицание
`.and.`	логическое умножение (и)
`.or.`	логическое сложение (или)
`.xor.`	исключающее или
`.eqv.`	эквивалентность
`.neqv.`	неэквивалентность

Как и в большинстве языков программирования, вычисление отношений предшествует выполнению логических операций. В выражении

```
(x .gt. 0) .and. (x .le. 2)
```

вначале будут вычислены выражения в скобках, а результаты затем использованы для вычисления логического И.

1.3. Операторы управления

Современный Фортран поддерживает все стандартные операторы управления: циклы, разветвления, разрывы цикла и т.д.

1.3.1. Разветвления

Простейший условный оператор имеет следующий синтаксис:

```
if(<логическое выражение>) <действие>
```

Пример использования:

```
if(x < y) x = 0
```

Более полный вариант логического оператора

```
if(<логическое выражение>) then
   <блок операторов>
end if
```

можно использовать в случае большого блока операторов. Пример использования:

```
if(x < y) then
  x =0
  y =3
end if
```

Если необходимо ветвление, то используется следующая конструкция:

```
if(<логическое выражение>) then
  <блок операторов №1>
else
  <блок операторов №2>
end if
```

Пример использования:

```
if(x < y) then
  x = 0
  y = 3
else
  x = 3
  y = 0
end if
```

В случае множественного ветвления конструкция усложняется:

```
if(<логическое выражение №1>) then
  <блок операторов №1>
else if(<логическое выражение №2>) then
  <блок операторов №2>
else if(<логическое выражение №3>) then
  <блок операторов №3>
  . . . . . .
else if(<логическое выражение № N>) then
  <блок операторов № N>
else
  <блок операторов № N+1>
end if
```

Пример использования:

```
if(x < y) then
  x = 0
  y = 3
else if(x == y) then
  x = 3
  y = 0
else
  x = 1
```

```
  y = 1
end if
```

В некоторых случаях удобнее применять оператор `select`:

```
select case (<логическое выражение>)
  case(<список №1>)
  <блок операторов №1>
  case(<список №2>)
  <блок операторов №2>
  . . . . . .
  [case default
  <блок операторов № N>]
end select
```

Результат вычисления логического выражения может совпадать со значениями из списка 1, 2 и т.д. В случае, если ни в одном из списков вычисленного выражения не оказалось, то выполняется блок операторов, указанный после конструкции `case default`.

1.3.2. Циклы с шагом и условием

Цикл с шагом задаётся следующей конструкцией:

```
do i =n1,n2[,n3]
  <блок операторов>
end do
```

Здесь `n1` — начальное значение счётчика итераций `i`, `n2` — конечное значение `i`, а `n3` — шаг, на который увеличивается счётчик после очередного витка цикла. По умолчанию `n3` равен 1 и его можно не указывать. Счётчик итерации может быть как целого типа, так и вещественного. Изменение значения переменной `i` в теле цикла не допускается и будет вызывать ошибку компиляции.

Кроме цикла с шагом в Фортране определён цикл с условием:

```
do while(<логическое выражение>)
  <блок операторов>
end do
```

Тело цикла выполняется, пока истинно логическое выражение.

Для прерывания витков цикла раньше времени (например, при появлении ошибки) предусмотрены следующие операторы:

- `exit` — выход из программы (или из подпрограммы/функции в главную программу);
- `continue` — полный выход из цикла;
- `cycle` — выход из текущей итерации цикла.

Обратите внимание, что в отличие от языка `C` команда `continue` приводит к полному выходу из цикла, а не только к прекращению текущей итерации и переходу к следующей (для чего используется `cycle`).

1.4. Процедуры: подпрограммы и функции

Все примеры, которые мы рассматривали до сих пор, содержали лишь главную программу. Однако, очевидно, что достаточно сложная программа должна быть разделена на относительно самостоятельные части, каждая из которых решает отдельную подзадачу. Для этой цели в Фортране используются *подпрограммы* и *функции* (общее название *процедуры*).

Различают процедуры трёх типов:

– внутренние процедуры располагаются внутри кода главной программы (т.е. между директивами `program` и `end program`);
– внешние процедуры располагаются вне кода главной программы (до, после или во внешнем файле);
– модульные процедуры находятся в предкомпилированном файле — модуле.

1.4.1. Подпрограммы

Код любой подпрограммы должен удовлетворять следующему шаблону:

```
subroutine <имя_подпрограммы> (<список_аргументов>)
  <Блок операторов>
end subroutine <имя_подпрограммы>
```

Вызов подпрограммы осуществляется с помощью ключевого слова `call`

```
call <имя_подпрограммы> (<список_аргументов>)
```

Рассмотрим простой пример, где определена подпрограмма `sum`, суммирующая два своих аргумента:

```
subroutine sum(x,y,res)
! Входной параметр
  real, intent(in) :: x,y
! Выходной параметр
  real, intent(out) :: res
  res = x + y
end subroutine sum
program main
  implicit none
  real :: x, y, res
  x = 2.0
  y = 3.0
! Вызов подпрограммы
  call sum(x,y,res)
  print *, res
end program main
```

Значения переменных, переданных в подпрограмму в качестве аргументов, могут быть изменены — никаких запретов на это не накладывается.

Однако для неоднократного использования подпрограммы желательно чётко установить, какие из аргументов будут перезаписаны, а какие нет. Этой цели служит атрибут `intent`:

- Атрибут `intent(in)` означает, что переменной перед передачей в подпрограмму следует присвоить некоторое значение, так как оно используется для работы в подпрограмме.
- Атрибут `intent(out)` означает, что переменной, передаваемой в качестве аргумента в подпрограмму, будет в любом случае присвоено некоторое новое значение, а старое значение (если оно было) будет утеряно. Такие аргументы используются для возвращения результатов работы подпрограммы.
- Атрибут `intent(inout)` означает, что переменной должно быть присвоено некоторое значение, которое необходимо для вычислений, но в ходе работы подпрограммы данной переменной будет присвоено новое значение. Аргументы такого типа позволяют более экономно использовать память, так как одна и та же переменная используется как для передачи данных в подпрограмму, так и для передачи данных из подпрограммы.

Применение атрибута `intent` необязательно, однако желательно для облегчения использования подпрограммы.

1.4.2. Функции. Рекурсивные функции

В отличие от подпрограмм, функции по окончании своей работы возвращают вычисленное значение. Возвращаемое значение присваивается переменной с тем же именем, что и вызываемая функция.

Рассмотрим пример использования функции:

```
real function f(x)
  real :: x
  f = x
end function f
real function g(x) result(res)
  real :: x
  res = x**2
end function g
program main
  implicit none
  real, external :: f, g
  ! Напечатаем сумму двух функций
    print *, f(1) + g(2)
end program main
```

В данном примере функция `f(x)` записывает возвращаемое значение в переменную `f`, а функция `g(x)` — в переменную `res`. Для использования определённых функций в основной программе (или подпрограммах) следует объявить имя функции как переменную соответствующего типа с

атрибутом `external`. После этого функции можно использовать непосредственно в выражениях.

Ключевым отличием функций от подпрограмм является запрет на переопределение аргументов в теле функции. Все переданные в функцию аргументы не должны быть изменены в процессе работы функции.

Рассмотрим иллюстрацию определения рекурсивной функции на примере вычисления факториала числа.

```
! Программа, вычисляющая факториал
program factorial
  implicit none
! Интерфейсный блок (для рекурсивной функции обязателен)
  interface
    recursive function fact(n) result(fn)
      integer :: n
      integer :: fn
    end function
  end interface
  integer :: n, m, i
  write(*,*) "Введите целое число"
  read(*,*) n
  do i=1,n,1
    m = fact(i)
  end do
  print *, 'Факториал равен'
  print *, m
end program factorial
! Обязательно указание имени возвращаемой переменной,
! отличного от имени функции
recursive function fact(n) result(fn)
  integer :: fn
  integer :: n
  if(n .gt. 0) then
    fn = n*fact(n-1)
  else
    fn = 1
  end if
end function fact
```

Отличительной особенностью рекурсивных функций является обязательное наличие интерфейсного блока в теле программы, вызывающей функцию, и переопределение переменной, в которую будет записано возвращаемое значение.

Заметим, что использование модулей позволяет избежать использования интерфейсного блока.

1.4.3. Модули

При разработке сложной программы так или иначе приходится использовать множество подпрограмм и функций. Можно выделить все процедуры в отдельные файлы. Однако при компиляции главной программы все используемые в ней процедуры из внешних файлов также будут компилироваться, что может быть весьма затратно по времени. Для решения этой проблемы в Фортране следует использовать *модули*.

Для создания модуля в файле `<название_модуля>.f90` необходимо определить следующую конструкцию:

```
module <название_модуля>
  implicit none
  contains
  <определение процедур>
end module <название_модуля>
```

Процедуры из предыдущих примеров можно организовать в виде модуля следующим образом:

```
! Модуль под названием mylib
module mylib
  implicit none
  contains
  subroutine sum(x,y,res)
    ! Входной параметр
    real, intent(in) :: x,y
    ! Выходной параметр
    real, intent(out) :: res
    res = x + y
  end subroutine sum
  real function f(x)
    real :: x
    f = x
  end function f
  real function g(x) result(res)
    real :: x
    res = x**2
  end function g
end module mylib
```

В главной программе следует использовать директиву `use <имя_модуля>`:

```
program main
  use mylib ! Используем модуль
  implicit none ! Обязательно после директивы use
  ! Далее как обычно
end program main
```

Для компиляции модуля и программы с модулем следует в консоли выполнить следующие команды:

```
gfortran -c mylib.f90
gfortran -c main.f90
gfortran mylib.o main.o
```

В результате по окончании компиляции будет создан файл модуля с расширением `.mod`, объектные файлы с расширением `.o` и исполняемый файл под стандартным именем `a.out`. В дальнейшем, если в сам модуль изменений не вносилось, то достаточно пересобрать только главную программу.

1.5. Массивы и работа с ними

1.5.1. Описание массива

Одним из достоинств Фортрана является простая, но в то же время эффективная работа с массивами.

Массив — это упорядоченное множество однотипных элементов. Массив характеризуется следующими свойствами:

- типом значений элементов массива;
- рангом или числом измерений (в стандарте F90 не более 7);
- граничными парами — диапазоном индексов по каждому измерению.

Два массива считаются равными, если равны все их элементы, стоящие в одинаковых позициях.

Приведём примеры описания массивов:

```
real, dimension(-10:10) :: a, b
real, dimension :: x(1:100)
integer, dimension(1:3,1:5) :: M
```

Как видно из приведённого примера, для описания массива необходимо использовать атрибут `dimension`. В первом случае были заданы два вещественных одномерных массива `a,b` с граничными парами `-10` и `10`. Указание граничных пар при атрибуте `dimension` позволяет компактно задавать сразу несколько массивов одинаковой размерности и с одинаковым диапазоном индексов. Во втором случае был описан также одномерный вещественный массив, но граничные пары были указанны непосредственно в скобках после имени массива. Такую запись удобно применять, когда описываются массивы разных типов, рангов и размерностей. Наконец последний массив описан как целочисленный двумерный массив с коэффициентами по первой размерности от `1` до `3`, а по второй — от `1` до `5`.

Протяжённость массива по некоторому измерению называется *экстентом*. Произведение экстентов даёт *размер* массива, т.е. число элементов (размер массива не следует путать с размерностью массива). Массивы одинаковой *формы*, т.е. с одинаковыми экстентами соответствующих размерностей, считаются конформными. Следует учитывать, что для конформности двух массивов совпадения граничных пар не требуется.

При задании матриц с помощью массивов следует учитывать порядок расположения элементов массива в памяти. Например для следующей матрицы/массива

$$a(1:3,1:3) = \begin{pmatrix} a(1,1) & a(1,2) & a(1,3) \\ a(2,1) & a(2,2) & a(2,3) \\ a(3,1) & a(3,2) & a(3,3) \end{pmatrix}$$

элементы в памяти располагаются по столбцам, т.е. $a(1,1)$, $a(2,1)$, $a(3,1)$, $a(1,2)$, $a(2,2)$, $a(3,2)$, $a(1,3)$, $a(2,3)$, $a(3,3)$.

Фортран предоставляет широкие возможности для доступа как к частям массива, так и к каждому из его элементов. *Вырезка* из массива представляет собой подмассив. Если по какому-то измерению опущены обе границы (двоеточие следует сохранить), то говорят о *сечении* массива. Например, `a(i,:)` означает `i`-ю строку приведённой выше матрицы `a`:

```
real, dimension(1:5,1:5) :: A
A(1:2,1:3) ! вырезка из массива A
A(1,:) ! сечение (первая строка)
A(:,2) ! сечение (второй столбец)
```

1.5.2. Задание массивов

Для задания значений массивов в Фортране предусмотрены различные средства. Опишем несколько одинаковых массивов и зададим их значения несколькими различными способами:

```
real, dimension(1:4) :: a, b, c, d
a = 2.0 ! Все элементы массива равны двум
! Использование конструктора массивов
b = (/1.0, 2.0, 3.0, 4.0/)
! Использование оператор data
data c/4.0, 3.0, 2.0, 1.0/
! Использование квадратных скобок
d = [ 1.0, 3.0, 4.0, 2.0 ]
```

Оператор `data` в общем случае может применяться для присваивания значений не только массивам, но и обычным переменным. Он имеет следующий синтаксис:

```
data <список объектов>/<список значений>/
```

Конструктор массивов может применяться для задания значений элементов массива, зависящих от индекса. Например:

```
a(1:12) = (/ (2*i, i=1,10), 12.0, 15.0/)
```

Первые 10 элементов заданы с помощью выражения от индекса, а последние два заданы явно. В результате мы получим массив с элементами $2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 12, 15$. Хотя конструктор позволяет задавать только одномерные массивы, но с помощью оператора `reshape` одномерный массив легко можно преобразовать в многомерный:

```
! Преобразуем одномерный массив из 12 элементов в
! двумерный массив 3 на 4
d = reshape( (/ (2*i, i=1,10), 12.0, 15.0/), (/3,4/) )
```

1.5.3. Действия над массивами

К любому элементу массива можно обратиться по соответствующим индексам массива, например, `v = a(2,1) + b(3,3)`. Рассмотрим ещё несколько примеров, иллюстрирующих поэлементные действия с массивами:

```
real, dimension(5) :: p, q
real, dimension(7) :: r
p = 2.0 ! Все элементы равны 2
r = 3.2 ! Все элементы равны 3,2
q = p + 0.89*r(3:7)
! Возможно применение стандартной функции.
! Значение функции будет вычислено для
! каждого элемента массива
p = exp(p)
```

Конструкция `where` позволяет запрограммировать действия с некоторыми элементами массива, отбираемые в соответствии с логической маской. Общий вид данной конструкции:

```
where (<логическое выражение массив>)
  <действия над элементами массива>
[elsewhere
  <действия над элементами массива>]
end where
```

```
! Обнулим все отрицательные элементы массива a
where(a < 0) a = 0
```

Более гибким в применении является оператор `forall`:

```
forall(<индексное выражение>,[<сколярная логическая маска>])
   <действия над элементами массива>
 end forall
```

```
! Сделаем замену в некоторой части массива всех
! положительных элементов на их квадратные корни
forall(i = 3:6, j =-3:3, M(i,j)>0) M(i,j)=sqrt(M(i,j))
```

В Фортран встроено много функций, позволяющих узнать характеристики массива, а также совершать матричные операции над одномерными (вектор) и двухмерными (матрица) массивами. Наиболее полезные функции приведём в виде таблицы:

Функция	Пояснение
`lbound(a,[dim])`	начальный индекс по измерению `dim`
`hbound(a,[dim])`	конечный индекс по измерению `dim`
`shape(a)`	вектор экстентов
`size(a,[dim])`	экстент по заданному измерению
`dot_product(a,b)`	скалярное произведение двух векторов
`matmul(A,B)`	перемножение матриц
`maxval(a)`	максимальный элемент массива
`minval(a)`	минимальный элемент массива
`sum(a)`	сумма элементов массива
`transpose(a)`	транспонирование (вектора, матрицы)

В функциях `lbound` и `hbound` второй аргумент не обязателен в случае одномерного массива. В функции `size` второй аргумент также можно не указывать, тогда функция вернёт суммарное число элементов по всем размерностям.

Заметим, что в Фортране нет встроенных функций для нахождения обратной матрицы и определителя матрицы. Этот недостаток легко устраняется сторонними библиотеками, об одной из которых (LAPACK) будет рассказано далее.

Для иллюстрации применения некоторых из приведённых выше функций, рассмотрим простую программу.

```
! Простая программа определения
! максимального элемента массива
program min_max
  implicit none ! не определять тип переменной по имени
  real, dimension(1:100) :: a
  real :: maximum
  integer :: i
  maximum = 0
  i = 0
  ! заполняем массив случайными числами в пределах [0,1]
  call random_number(a)
  do i = lbound(a,1),ubound(a,1),1
    ! print *, a(i)
    if (maximum <= a(i)) then
      maximum = a(i)
    end if
  end do
  print *, "Информация о массиве:"
  print *, "    левая граница =", lbound(a,1)
  print *, "    правая граница =", ubound(a,1)
  print *, "Наш максимум = ", maximum
  print *, "Встроенный максимум = ", maxval(a)
end program min_max
```

1.5.4. Динамические массивы

Для создания гибких программ часто необходимо создавать массивы, размерности которых вычисляются в самой программе, вводятся пользователем или передаются как параметр при запуске программы. Для таких случаев предусмотрено описание динамических массивов, граничные пары которых при объявлении не указываются. Такие массивы называются динамическими, и для их описания следует использовать атрибут `allocatable`:

```
! Опишем двумерный массив a
! и одномерный массив b
real, allocatable :: a(:,:), b(:)
```

После описания такого массива уже в области операторов следует выделить под него память оператором `allocate(<список массивов>)`:

```
allocate(a(1:n,1:m), b(1:k))
```

Переменным `n, m, k` должны быть присвоены конкретные значения. После завершения работы с массивами память можно освободить, используя оператор `deallocate(<список массивов>)`:

```
deallocate(a(1:n,1:m), b(1:k))
```

Хотя освобождение памяти, выделенной для динамических массивов, при завершении программы должно осуществляться автоматически, полагаться на это не рекомендуется, и следует всякий раз использовать оператор `deallocate`.

1.5.5. Передача массива в качестве аргумента процедуры

В некоторых случаях необходимо передать процедуре массив в качестве аргумента. При этом размерность массива должна быть известна заранее, но число элементов может вычисляться непосредственно в теле процедуры. Следующая процедура вычисляет произведение всех элементов одномерного массива:

```
real function ArrayProd(A)
  implicit none
! Следует указать только начальный индекс
  real, intent(in), dimension(1:) :: A
  integer :: m,i
  m = size(A)
  ArrayProd = 1
  do i = 1,m,1
  ArrayProd = ArrayProd*A(i)
  end do
end function ArrayProd
```

Для использования данной функции в основной программе следует включить её в состав модуля (см. раздел 1.4.3) или же снабдить интерфейсным блоком:

```
program prod
! Функция ArrayProd входит в состав модуля modprod
  use modprod
  implicit none
  real, dimension(1:4) :: a
  data a/1,2,3,4/
  write(*,*), ArrayProd(a)
end program prod
```

1.5.6. Функция, возвращающая массив

Имеется возможность создавать функции, возвращаемое значение которых является массивом:

```
! Функция, возвращающая массив
function fvector(x)
   implicit none
   ! Массив переменных
   real, intent(in) :: x
   ! Тип возвращаемых значений
   real, dimension(1:3) :: fvector
   fvector(1) = sin(x)
   fvector(2) = cos(x)
   fvector(3) = x**2
end function
```

Следует включить данную функцию в состав модуля, иначе потребуется написать блок интерфейса. При вызове функции будет возвращён массив значений:

```
real, parameter :: x = 2.0
print *, "fvector(x) = ", fvector(x)
```

или

```
real, dimension(1:3) :: fv
fv = fvector(x)
print *, "fvector(x) = ", fv(1), fv(2), fv(3)
```

1.6. Ввод и вывод. Форматирование

1.6.1. Ввод-вывод

Ввод и вывод по умолчанию происходит в консоль, в которой запущена программа. Этой цели служат два оператора:

```
! Ввод
read(*,*), "Список ввода"
read *, "Список ввода"
! Вывод
write(*,*), "Список вывода"
print *, "Список вывода"
```

Звёздочки означают стандартный формат и стандартное устройство ввода (клавиатура) и вывода (консоль). Общий синтаксис данных операторов таков:

```
read(<устройство ввода>,<формат ввода>),
   <список переменных>
write(<устройство вывода>,<формат вывода>),
   <список переменных>
```

Следующий пример пояснит применение данных операторов для ввода и вывода данных через консоль:

```
integer :: n, i
real, dimension(1:100) :: a
logical :: l
write(*,*), "Введите n"
read(*,*), n
write(*,*), "Введите ",n,"первых элементов:"
! Вводим значения через пробел
read(*,*), (a(i), i = 1,n)
! Распечатываем значения
write(*,*), (a(i), i = 1,n)
write(*,*), "Введите правду или ложь"
! Логические .true. и .false. Вводятся как T и F.
read(*,*), l
```

Для организации вывода в файл следует вначале открыть файл на запись с помощью оператора `open`, который имеет два обязательных аргумента:

```
open(<номер устройства>,file=<путь к файлу>)
```

Для закрытия файла применяется оператор `close`:

```
close(<номер устройства>,status=<keep или delete>)
```

Допустим, что в файле `in.txt` записаны три действительных числа, расположенные в одну строку через пробел. Необходимо открыть файл на чтение, считать данные и записать их в другой файл `out.txt`:

```
! Номер устройства может быть любым (больше 3)
! для разных файлов — разные номера
open(4,file="in.txt")
open(5,file="out.txt")
! Указываем номер устройства
read(4,*), x, y, z
write(5,*), x, y, z
! Закрываем файлы, сохраняя информацию
close(4, status = "keep")
close(5, status = "keep")
```

1.6.2. Форматирование ввода-вывода

При необходимости вывода в отличном от умолчания формате применяются спецификации формата. Они задаются в виде текстовой строки в аргументе операторов вывода или с помощью специального оператора `format`. Спецификации формата лучше рассматривать на примерах:

- `i10` — целое число максимум из 10 позиций;
- `f10.5` — вещественное число максимум из 10 знаков всего и 5 знаков после запятой;
- `e12.4` — вещественное с плавающей точкой из 12 знаков (например, `0.50200E+10`);
- `a10` — текст из 10 символов.

Продемонстрируем на примере использование оператора `format`:

```
integer :: k
real :: x
character :: s
! Перед оператором format указывается метка
! Меткой может быть произвольное целое число
! Метка позволяет обратиться к помеченному
! месту в программе
10 format(i10,f10.5,a10)
write(10,*), k, x, s
```

Вместо использования `format` можно непосредственно указать строку спецификации формата в операторе `write`:

```
write(*,"(i10,f10.5,a10)"), k, x, s
```

Это удобно, когда нужно указать отдельную спецификацию формата вывода для каждого случая.

Кроме настройки вывода стандартных типов данных, с помощью спецификации формата можно управлять позицией в строке, вставкой пробелов, табуляций, переводом на новую строку: `t<n>` — отступ на `n` позиций от начала строки, `<n>x` — вставка `n` пробелов, `t<n>` — вставка `n` символов табуляции.

Следующий пример показывает, как распечатать в консоль несколько элементов массива:

```
real, dimension(1:20) :: a
! Заполним массив случайными числами
call random_number(a)
write(*,"(10(2x,f5.2))"), a
```

Хотя массив `a` содержит 20 элементов, но при выводе в одной строке будет лишь 10 из них, следующие 10 будут перенесены на следующую строку. Кроме того, между элементами массива будут вставлены 2 пробела (`2x`).

Часто необходимо организовать распечатку элементов массива в виде матрицы, но размерность массива заранее неизвестна, так как вычисляется уже во время работы программы или вводится пользователем. В этом случае можно использовать следующий приём:

```
    character(len=20) :: fm
    write(fm,*), m
    write (*,"("//adjustl(fm)//"(f8.3), t1)") (a(i), i=1,m)
```

Вначале задаётся строковая переменная `fm`, затем в эту переменную записывается некоторое целое число `m`. Дальше строковая переменная включается в спецификацию формата с помощью конкатенации строк (оператор `//`) и оператора `adjustl`, который убирает лишние пробелы. Таким образом можно организовать форматированный вывод любых матриц и векторов, что значительно упрощает восприятие результатов программы.

Ниже приведён код программы, иллюстрирующий как действия над матрицами, так и разнообразное форматирование при записи результатов в файл. Для лучшего понимания работы программы следует скомпилировать её, запустить и посмотреть результат работы. Обратите внимание на порядок задания элементов матрицы и на то, в какой последовательности перебираются коэффициенты массива при выводе в файл.

```
!-------------------------------------
! Действия с матрицами/массивами
! и форматированный вывод в файл
!-------------------------------------
program matrix
  integer :: i, j, k
  real :: s
  real, dimension (1:5,1:4) :: a
  real, dimension (1:4,1:7) :: b
  real, dimension (1:5,1:7) :: c
  real, dimension (1:4,1:5) :: aT
! открываем файл для записи
  open(10,file = "matrix.out") ! Открыли файл
! заполняем массив случайными числами в пределах [0,1]
  !call random_number(a)
  ! Заполняем матрицу. Заполнение идет по столбцам
  ! 1.000   0.000   0.000   0.000
  ! 0.000   0.000   1.000   0.000
  ! 0.000   1.000   0.000   0.000
  ! 0.000   0.000   0.000   1.000
  ! 0.000   0.000   0.000   0.000
  data a/1,0,0,0,0, 0,0,1,0,0, 0,1,0,0,0, 0,0,0,1,0/
  ! 11.000  12.000  13.000  14.000
  ! 21.000  22.000  23.000  24.000
  ! 31.000  32.000  33.000  34.000
  ! 41.000  42.000  43.000  44.000
  ! 51.000  52.000  53.000  54.000
  !data a/11,21,31,41,51,12,22,32,42, &
  !    52,13,23,33,43,53,14,24,34,44,54/
  call random_number(b)
  call random_number(c)
```

```
!---
  write(10,*), "# Матрица A"
! Записываем в файл
  write(10,'(4(f8.3), t1)') ((a(i,j), j = 1,4), i = 1,5)
  write(10,*), "# Матрица B"
  write(10,'(7(f8.3), t1)') ((b(i,j), j = 1,7), i = 1,4)
! Использование встроенной функции для перемножения
  c = matmul(a,b)
!----
  write(10,*), "# C — результат переумножения матриц A и B"
  write(10,'(7(f8.3), t1)') ((c(i,j), j = 1,7), i = 1,5)
!-----
! Сложение вырезок
  c(1:4,1:4) = a(1:4,1:4) + b(1:4,1:4)
  write(10,*), "# C — результат сложения вырезок из A и B"
  write(10,'(4(f8.3), t1)') ((c(i,j), j = 1,4), i = 1,4)
! Транспонирование матрицы A
  aT = transpose(a)
  write(10,*), "# AT — результат транспонирования матрицы A"
  write(10,'(5(f8.3), t1)') ((aT(i,j), j = 1,5), i = 1,4)
  !write (*,'(5(f8.3), t1)') ((c(i,j), j = 1,5), i = 1,4)
! Закрываем файл после того, как завершили с ним работу
  close(10, status = "keep")
end program matrix
```

1.7. Алгоритмы умножения матриц

1.7.1. Стандартный алгоритм

Рассмотрим две матрицы $A[m \times n]$ и $B[n \times k]$, которые при умножении дают матрицу $C[m \times k]$:

$$\begin{pmatrix} a_1^1 & a_1^2 & a_1^3 & \dots & a_1^n \\ a_2^1 & a_2^2 & a_2^3 & \dots & a_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_m^1 & a_m^2 & a_m^3 & \dots & a_m^n \end{pmatrix} \times \begin{pmatrix} b_1^1 & b_1^2 & b_1^3 & \dots & b_1^n \\ b_2^1 & b_2^2 & b_2^3 & \dots & b_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_m^1 & b_m^2 & b_m^3 & \dots & b_m^n \end{pmatrix} =$$

$$= \begin{pmatrix} c_1^1 & c_1^2 & c_1^3 & \dots & c_1^n \\ c_2^1 & c_2^2 & c_2^3 & \dots & c_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_m^1 & c_m^2 & c_m^3 & \dots & c_m^n \end{pmatrix},$$

где

$$c_i^l = \sum_{j=1}^{m} a_i^j b_j^l, i = 1, \ldots, n, \;\; j = 1, \ldots, m, \;\; l = 1, \ldots, k.$$

Каждый элемент матрицы C является скалярным произведением строки матрицы A на столбец матрицы B. Стандартный алгоритм умножения требует nmk операций умножения и $n(m-1)k$ операций сложения и алгоритмически реализуется следующим образом:

```
do l = 1,m,1
  do i = 1,k,1
  c(l,i) = a(l,1)*b(1,i)
  do j=2,n,1
  c(l,i) = c(l,i) + a(l,j)*b(j,i)
  end do
  end do
end do
```

На первый взгляд, никак нельзя уменьшить объём работы, необходимый для перемножения двух матриц. Однако исследователям не удалось доказать минимальность и в результате были найдены другие алгоритмы, умножающие матрицы более эффективно. Рассмотрим один из таких алгоритмов.

1.7.2. Умножение матриц по алгоритму Винограда

Одним из алгоритмов умножения матриц является алгоритм Винограда (Shmuel Winograd). Он даёт небольшой выигрыш в скорости вычисления произведения, но может быть эффективно распараллелен. Основная идея алгоритма весьма проста и заключается в преобразовании суммы $\sum\limits_{j=1}^{m} a_i^j b_j^l$ следующим образом:

$$c_i^l = \underbrace{\sum_{j=1}^{m/2} (a_i^{2j-1} + b_{2j}^l)(a_i^{2j} + b_{2j-1}^l)}_{(1)} - \underbrace{\sum_{j=1}^{m/2} a_i^{2j-1} a_i^{2j}}_{(2)} - \underbrace{\sum_{j=1}^{m/2} b_{2j-1}^l b_{2j}^l}_{(3)}.$$

Количество операций сложения и умножения приведено в следующей таблице:

Слагаемое	Сложений	Умножений
(1)	$knm/2$	$kn(m+m/2+1)$
(2)	$n(m/2-1)$	$nm/2$
(3)	$k(m/2-1)$	$km/2$

В слагаемом (2) участвуют только элементы матрицы A, а в слагаемом (3) — только элементы матрицы B. Для каждой матрицы A и B, участвующих в умножении, эти слагаемые можно вычислить заранее и сохранить в память (особенно это актуально для больших матриц). При необходимости нахождения произведения $A \times B$ следует вычислить (1) и к полученному результату прибавить (2) и (3), вычисленные ранее.

Ниже приведён код, реализующий алгоритм Винограда [4]:

```
d = n/2
! Вычисление rowFactor для матрицы A
do i = 1,m,1
  rowFactor(i) = A(i,1)*A(i,2)
  do j = 2,d,1
  rowFactor(i) = rowFactor(i) + A(i,2*j-1)*A(i,2*j)
  end do
end do
! Вычисление columnFactor для матрицы B
do i = 1,k,1
  columnFactor(i) = B(1,i)*B(2,i)
  do j = 2,d,1
     columnFactor(i) = columnFactor(i) +
        B(2*j-1,i)*B(2*j,i)
  end do
end do
! Вычисление матрицы C
do i =1,m,1
  do j =1,k,1
    C(i,j) = -rowFactor(i) - columnFactor(j)
    do l = 1,d,1
      C(i,j) = C(i,j) + (A(i,2*l-1)+B(2*l,j))*
          (A(i,2*l)+B(2*l-1,j))
    end do
  end do
end do
! Прибавление членов в случае нечётной
! размерности матрицы
if (mod(n,2) == 1) then
  do i = 1,m,1
    do j =1,k,1
      C(i,j) = C(i,j) + A(i,n)*B(n,j)
    end do
  end do
end if
```

1.8. Задания для лабораторной работы

1.8.1. Задание №1

Написать программу, решающую квадратное уравнение с корнями любого вида.

1.8.2. Задание № 2

Написать программу, вычисляющую числа Фибоначчи с помощью рекурсивной функции.

1.8.3. Задание № 3

Написать подпрограмму, распечатывающую переданную ей в качестве аргумента матрицу (двухмерный массив) с любым числом строк и столбцов. Для этого воспользуйтесь приёмом, указанным в конце раздела 1.6. Подпрограмму требуется оформить как модуль.

1.8.4. Задание № 4

Написать программу перемножения матриц стандартным способом и с помощью алгоритма Винограда. Сравнить производительность этих алгоритмов и встроенной функции `matmul` для больших матриц. Текущее время можно получить с помощью процедуры `ctime(t)`.

1.9. Содержание отчёта

Отчёт должен включать:
1) титульный лист;
2) формулировку цели работы;
3) описание результатов выполнения задания:
   - листинги программ;
   - результаты выполнения программ (текст, снимок экрана, таблицы и графики в зависимости от задания);
4) выводы по каждому заданию лабораторной работы.

1.10. Контрольные вопросы

1. Какие встроенные типы данных поддерживает Фортран? Как описать вещественную переменную двойной точности?
2. Почему комплексная переменная двойной точности занимает в памяти 16 байт?

3. Для чего применяются дополнительные атрибуты при описании переменной?
4. Как с помощью встроенных функций вычислить дробную часть вещественного числа?
5. Как правильно проверить, равны ли две вещественные переменные?
6. Какой оператор прерывает виток цикла и переходит к следующей итерации?
7. Чем подпрограмма отличается от функции?
8. Какова структура программы на фортране?
9. В чём преимущество модулей?
10. Необходимо описать 5 массивов размерности (1:100,-100:1). Как сделать это наиболее кратко?
11. Что такое экстент, размер и ранг массива?
12. Что такое сечение и вырезка массива?
13. Какие способы задания значений элементов массива вы знаете?
14. Как называется встроенная функция, вычисляющая определитель матрицы (двухмерного массива)?
15. Что такое динамический массив и как он задаётся?
16. Опишите основные операторы для ввода-вывода данных. Как записать данные в файл?

2. Библиотека LAPACK

2.1. Назначение библиотеки LAPACK

Библиотека LAPACK (Linear Algebra PACKage) служит для решения задач линейной алгебры (решение систем линейных алгебраических уравнений (СЛАУ), различные матричные разложения). Подпрограммы LAPACK написаны на языке `Fortran`, хорошо оптимизированы и для повышения быстродействия используют BLAS (Basic Linear Algebra Subprograms) — подпрограммы, оптимизированные под конкретные архитектуры.

Изначально процедуры LAPACK были написаны под `Fortran 77`, однако к версии 3.2 (2008) были переписаны под `Fortran 90`.

Процедуры LAPACK делятся на следующие виды:

- *процедуры-драйверы* (driver routines) — для решения конкретных задач линейной алгебры (решение СЛАУ, нахождение собственных векторов и т.д.);
- *процедуры-вычислители* (computational routines) — вычислительные задачи (различные разложения матриц);
- *вспомогательные процедуры* (auxiliary routines).

2.2. Процедуры LAPACK

При компиляции программы, использующей процедуры LAPACK, следует использовать одну из следующих команд:

```
gfortran -L /usr/local/lib -llapack -lblas src.f90 -o prog
gfortran $(pkg-config --libs lapack) src.f90 -o prog
```

Названия всех процедур закодированы по одинаковой схеме: `pmmaaa`, где буква `p` задаёт тип чисел (вещественный или комплексный) матрицы и может принимать следующие значения:

p	Расшифровка
s	действительный тип одинарной точности `real(kind=4)`
d	действительный тип двойной точности `real(kind=8)`
c	комплексный тип одинарной точности `complex(kind=8)`
z	комплексный тип двойной точности `complex(kind=16)`

Буквы `mm` задают тип матрицы, что позволяет использовать более оптимизированные процедуры для работе с ними. Наиболее часто используемыми типами являются:

mm	Расшифровка
GE	матрица общего вида
DI	диагональная матрица
OR	ортогональная матрица (вещественная)
UN	унитарная матрица (комплексная)
GT	трёхдиагональная матрица
SY	симметричная матрица

Наконец aa задают конкретный алгоритм, например sv — решение СЛАУ, или trf — вычисление LUP-разложения матрицы.

Познакомимся теперь с некоторыми конкретными процедурами. Будем рассматривать процедуры для действительных чисел одинарной точности. Однако все рассматриваемые процедуры имеют реализации и для действительных чисел двойной точности, и для комплексных чисел одинарной и двойной точности.

2.3. Процедура sgesv решения СЛАУ

Рассмотрим применение процедуры для решения систем линейных алгебраических уравнений общего вида:

$$\begin{cases} a_{11}x_1+ & a_{12}x_2+ & a_{13}x_3+ & \ldots+ & a_{1n}x_n = & b_1, \\ a_{21}x_1+ & a_{22}x_2+ & a_{23}x_3+ & \ldots+ & a_{2n}x_n = & b_2, \\ a_{31}x_1+ & a_{32}x_2+ & a_{33}x_3+ & \ldots+ & a_{3n}x_n = & b_3, \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1}x_1+ & a_{m2}x_2+ & a_{m3}x_3+ & \ldots+ & a_{mn}x_n = & b_m. \end{cases}$$

Исходными данными, передаваемыми подпрограмме, являются матрица A и вектор **b**:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1,n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2,n-1} & a_{1n} \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3,n-1} & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & a_{m-1,3} & \ldots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{m,n-1} & a_{mn} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{m-1} \\ b_m \end{bmatrix}.$$

Компактно система запишется следующим образом:

$$\mathrm{A}\mathbf{x} = \mathbf{b}.$$

Как уже говорилось выше, процедура, решающая СЛАУ с *квадратной*

матрицей A$[n \times n]$ общего вида и действительными коэффициентами одинарной точности называется `sgesv` и имеет вид:

```
subroutine sgesv(n, nb, A, lda, pivot, B, ldb, info)
   integer, intent(in) :: n
   integer, intent(in) :: nb
   real, dimension(lda,*), intent(inout) :: A
   integer, intent(in) :: lda
   integer, dimension(*), intent(out) :: pivot,
   real, dimension(ldb,*), intent(inout) :: B
   integer, intent(in) :: ldb
   integer, intent(out) :: info
```

Расшифруем математический смысл параметров.

Параметр	Расшифровка
`n`	размерность системы уравнений (матрицы A)
`nb`	число векторов **b**
`A`	матрица A
`lda`	главная размерность массива `A` (`lda >= max(1,n)`)
`pivot`	вектор, задающий матрицу перестановок
`B`	массив, содержащий вектор(ы) b
`ldb`	число правых частей (векторов b)
`info`	индикатор успешности выполнения процедуры

Следует отметить, что обычно `lda = n` и `ldb = 1`.

Более подробного разъяснения требует `pivot`. Пусть задана матрица

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

Известно, что элементарное преобразование, заключающееся в перестановке первой и второй строки матрицы, можно задать с помощью элементарной матрицы вида

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

так как

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{31} & a_{12} & a_{13} \end{pmatrix}$$

Матрицы, задающие перестановку строк или столбцов, называются *матрицами перестановки* (permutation matrix). Они представляют собой матрицу, каждая строка которой содержит лишь один ненулевой элемент, равный

единице. Поэтому для экономии памяти компьютера данная матрица задаётся в виде вектора (одномерного массива). При этом i-й элемент массива равен номеру столбца, где в i-й строке стоит единица. В нашем случае

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad p = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}.$$

Ниже приведён пример использования процедуры `sgesv`.

```
! Решение системы уравнений
program lapack_ex1
  implicit none
!----------------------------------------
  real, allocatable :: A(:,:), b(:)
  integer, allocatable :: pivot(:)
  integer :: n, i, j, ok
  ! переменная для задания форматирования
  character(len=20) :: fmt
!----------------------------------------
  write(*,*), "Введите размерность СЛАУ"
  read(*,*) n
  allocate(A(1:n,1:n))
  allocate(b(1:n))
  allocate(pivot(1:n))
! Заполняем числами от 0 до 1
  call random_number(A)
  call random_number(b)
!!
  write(fmt,*), n
  write(*,*), "-----  Матрица A  -----"
  write (*,"("//adjustl(fmt)//"(f8.3), t1)") &
    ((A(i,j), j = 1,n), i = 1,n)
!!
  call SGESV(n, 1, A, n, pivot, b, n, ok)
!!
  write(*,*), "-----  Матрица A  -----"
  write (*,"("//adjustl(fmt)//"(f8.3), t1)") &
     ((A(i,j), j = 1,n), i = 1,n)
  write(*,*), "--- Вектор перестановок ---"
  write (*,"("//adjustl(fmt)//"(i3), t1)") &
                                (pivot(i), i = 1,n)
  write(*,*), "--- Решение системы ---"
  write (*,"("//adjustl(fmt)//"(f8.3), t1)") &
                                     (b(i), i = 1,n)
end program lapack_ex1
```

2.4. Процедура `sgetrf` вычисления LUP-разложения

При работе с матрицами большую роль играют различные матричные разложения. В LAPACK не реализованы, например, такие операции, как нахождение определителя матрицы и нахождение обратной матрицы. Однако вычислить определитель можно, воспользовавшись LUP-разложением, а найти обратную матрицу — с помощью SVD-разложения.

Рассмотрим вначале LUP-разложение и соответствующую процедуру LAPACK.

Любую невырожденную матрицу A можно представить в виде

$$\mathrm{PA} = \mathrm{LU},$$

где L — нижнедиагональная матрица, U — верхнедиагональная матрица, P — матрица перестановок (элементарная матрица).

Алгоритм LUP-разложения, реализованный процедурой `sgetrf`, может обрабатывать любые невырожденные матрицы и при этом обладает высокой устойчивостью:

```
subroutine sgetrf(m, n, A, lda, pivot, info)
  integer, intent(in) :: m
  integer, intent(in) :: n
  real, dimension(lda,*), intent(inout) :: A,
  integer, intent(in) :: lda
  integer, dimension(*), intent(out) :: pivot
  integer, intent(out) :: info
```

Многие параметры совпадают с параметрами процедуры `sgesv`:

Параметр	Расшифровка
`m`	число строк матрицы A
`n`	число столбцов матрицы A
`A`	матрица A
`lda`	главная размерность массива `A` (`lda >= max(1,m)`)
`pivot`	вектор, задающий матрицу перестановок
`info`	индикатор успешности выполнения процедуры (0 — успех)

Результат вычислений (коэффициенты матриц L и U) записывается в массив `A`, причём у матрицы L диагональ всегда единичная, поэтому при такой записи информация о ней не теряется:

$$\mathrm{A} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1,n-1} & u_{1n} \\ l_{21} & u_{22} & u_{23} & \dots & u_{2,n-1} & u_{1n} \\ l_{31} & l_{32} & u_{33} & \dots & u_{3,n-1} & u_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{m-1,1} & l_{m-1,2} & l_{m-1,3} & \dots & u_{m-1,n-1} & u_{m-1,n} \\ l_{m1} & l_{m2} & l_{m3} & \dots & l_{m,n-1} & u_{mn} \end{bmatrix}.$$

Матрица P восстанавливается по вектору перестановок **p**, коэффициенты которого записываются в массив `pivot`.

Рассмотрим теперь пример использования процедуры `sgesv`.

```
! LUP разложение
program lapack_ex2
  implicit none
!----------------------------------------
  real, allocatable :: A(:,:)
  integer, allocatable :: pivot(:)
  integer :: m, n, i, j, lda, info
!----------------------------------------
  write(*,*), "Введите размерность матрицы m и n"
  read(*,*) m, n
  lda = n
  allocate(A(1:m,1:n))
  allocate(pivot(1:m))
! Заполняем числами от 0 до 1
  call random_number(A)
!!
  call sgetrf(m, n, A, m, pivot, info)
!!
end program lapack_ex2
```

2.5. Процедура `sgeev` вычисления собственных векторов и собственных значений

Собственным вектором линейного преобразования (матрицы) A называется такой ненулевой вектор **x**, что для некоторого $\lambda \in \mathbb{R}, \mathbb{C}$ справедливо

$$A\mathbf{x} = \lambda\mathbf{x}.$$

Собственным значением линейного преобразования A называется такое число $\lambda \in \mathbb{R}, \mathbb{C}$, для которого существует собственный вектор, т.е. уравнение $A\mathbf{x} = \lambda\mathbf{x}$ имеет ненулевое решение **x**. Различают также правый A**x** и левый **y**A собственные векторы. Для нахождения собственных векторов и собственных значений служит процедура `sggev`:

```
    subroutine sggev(jobvl, jobvr, n, A, lda, wr, wi, Vl,
       ldvl, Vr, ldvr, work, lwork, info)
     character(len=1), intent(in) :: jobvl,
     character(len=1), intent(in) :: jobvr,
     integer, intent(in) :: n,
     real, dimension(lda,*), intent(inout) :: A,
     integer, intent(in) :: lda,
     real, dimension(*), intent(out) :: wr,
     real, dimension(*), intent(out) :: wi,
```

```
      real, dimension(ldvl,*), intent(out) :: Vl,
      integer, intent(in) :: ldvl,
      real, dimension(ldvr,*), intent(out) :: Vr,
      integer, intent(in) :: ldvr,
      real, dimension(*), intent(out) :: work,
      integer, intent(in) :: lwork,
      integer, intent(out) :: info
```

Поясним смысл передаваемых параметров:

- `jobvl` может принимать два значения: `'N'` — левый собственный вектор матрицы A **не** вычисляется и `'V'` — левый собственный вектор матрицы A вычисляется;
- `jobvr` аналогичен `jobvl`, но задаёт настройки вычисления для правого собственного вектора;
- `n` — размерность матрицы A;
- `lda` — главная размерность массива `A`; в общем случае `lda >= max(1,n)`, но обычно `lda = n`;
- `wr` и `wr` — массивы, содержащие действительную и мнимую части вычисленных собственных значений;
- `Vl` и `Vr` — левый и правый собственные векторы; в случае если j-е собственное значение действительно, то `u(j) = Vl(:,j)`, j-й столбец массива `Vl`; в случае же если j-е и $j+1$-е собственные значения являются комплексно сопряжённой парой, то `u(j) = Vl(:,j) + i*Vl(:,j+1)` и `u(j+1) = Vl(:,j) - i*Vl(:,j+1)`; то же справедливо и для `Vr`;
- `ldvl` и `ldvr` — главные размерности `Vl` и `Vr`; требуется, чтобы `ldvl,ldvr >= n`;
- `work` и `lwork` — служебный массив, необходимый для вычислений, и его размерность (для хорошей производительности рекомендуют брать `lwork > = 4*n`);
- `info` — индикатор успешности выполнения (может принимать следующие значения: 0 — выполнение успешно, > 0 — ошибка и $= i < 0$, где i означает, что неверно задан i-й аргумент).

Пример использования процедуры `sgeev` приведён ниже.

```
! Нахождение собственных значений
program lapack_ex3
  implicit none
!-----------------------------------------------------
  real, allocatable :: A(:,:), Vl(:,:), Vr(:,:)
  real, allocatable :: wr(:), wi(:), work(:)
  character(len=1) :: jobvl, jobvr
  integer :: n, i, j, lda, ldvl, ldvr, lwork, info
!-----------------------------------------------------
  write(*,*), "Введите размерность квадратной матрицы n"
  read(*,*) n
  lda = n
  ldvl = n
  ldvr = n
```

```
  jobvl = 'V'
  jobvr = 'V'
  lwork = 5*n
  allocate(A(1:n,1:n))
  allocate(Vl(1:n,1:n))
  allocate(Vr(1:n,1:n))
  allocate(work(1:lwork))
  allocate(wr(1:n))
  allocate(wi(1:n))
! Заполняем числами от 0 до 1
  call random_number(A)
!!
  call sgeev(jobvl, jobvr, n, A, lda, wr, wi, &
     Vl, ldvl, Vr, ldvr, work, lwork, info)
!!
  if (info .eq. 0) then
    write(*,*) "Успешное выполнение"
  else if(info .gt. 0) then
    write(*,*) "Ошибка!"
  else if(info .lt. 0) then
    write(*,*) "Неправильно задан параметр №", abs(info)
  end if
end program lapack_ex3
```

2.6. Процедура `sgesvd` вычисления SVD-разложения

Ещё одним полезным разложением матриц, часто используемым на практике, является *сингулярное разложение* или, иначе, SVD-разложение (Singular Value Decomposition). Любая матрица M порядка $m \times n$, элементы которой являются комплексными числами, может быть представлена в виде

$$\mathrm{M} = \mathrm{U}\Sigma\mathrm{V}^{\dagger},$$

где U — унитарная матрица порядка $m \times m$, Σ — диагональная матрица порядка $m \times n$ с неотрицательными вещественными коэффициентами на диагонали, V — унитарная матрица порядка $n \times n$, $\mathrm{V}^{\dagger}$ — эрмитово-сопряжённая матрица к V.

Элементы σ_{ii} диагонали матрицы Σ называются *сингулярными числами* матрицы M и определены с точностью до их перестановки. Обычно их выстраивают по убыванию $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ldots \geq \sigma_n$. Столбцы матриц U и V называют соответственно *левым и правым сингулярными векторами*. В случае действительной матрицы M матрицы U и V ортогональны.

SVD-разложение можно использовать для вычисления ранга матрицы, нахождения обратной и псевдообратной матриц, нахождения приближённого решения переопределённой системы уравнений и т.д.

Значение ранга матрицы равно количеству ненулевых сингулярных чисел. В случае квадратной матрицы, если хотя бы одно значение $\sigma_{ii} = 0$, то матрица вырожденная ($\det(M) = 0$).

Для вычисления обратной матрицы вначале следует удостовериться, что матрица невырожденная. Если это так, то обратная матрица вычисляется по формуле

$$\mathrm{A}^{-1} = \mathrm{V}\Sigma^{-1}\mathrm{U}^T, \quad \Sigma^{-1} = \mathrm{diag}\left(\frac{1}{\sigma_{11}}, \ldots, \frac{1}{\sigma_{nn}}\right).$$

Рассмотрим теперь процедуру библиотеки `LAPACK`, осуществляющую это разложение. Так как процедура работает с действительными матрицами, то эрмитово сопряжение заменяется на простую транспозицию:

```
sgesvd (jobu, jobvt, m, n, A, lda, s, U, ldu, Vt,
    ldvt, work, lwork, info)
character*1, intent(in) :: jobu
character*1, intent(in) :: jobvt
integer, intent(in) :: m,
integer, intent(in) :: n,
real, dimension(lda, *), intent(inout) :: A,
integer, intent(in) :: lda,
real, dimension(*), intent(out) :: s,
real, dimension(ldu, *), intent(out) :: U,
integer, intent(in) :: ldu,
real, dimension(ldvt, *), intent(out) :: VT,
integer, intent(in) :: ldvt,
real, dimension(*), intent(out) :: work,
integer, intent(in) :: lwork,
integer, intent(out) :: info
```

Поясним смысл передаваемых процедуре аргументов:

- Параметр `jobu` может принимать четыре значения: `'A'` — все m колонок матрицы U возвращаются в массиве `U`, `'S'` — первые $\min(m, n)$ столбцов матрицы U (левые сингулярные векторы) возвращаются в массиве `U`, `'O'` — первые $\min(m, n)$ столбцов матрицы U (левые сингулярные векторы) возвращаются в перезаписываемом массиве `A`, `'N'` — столбцы U не вычисляются.
- Параметр `jobvt` задаёт точно такие же параметры, но для строк матрицы V^T (правые сингулярные векторы).
- Параметры `m, n` задают число строк и столбцов матрицы A, $m, n \geqslant 0$.
- Параметр `A` — изначальная матрица, которую следует подвергнуть разложению.
- Параметр `lda` — главная размерность `A`, `lda >= max(1,m)`.
- Параметр `S` — двухмерный массив, соответствующий матрице Σ.
- Параметр `U` и `Vt` — это матрицы U и V^T.
- Параметры `ldu` и `ldvt` — главные размерности U и V^T.

– Параметры `work, lwork, info` — служебные.

```
!!  Использование процедуры для SVD разложения
!!  действительной матрицы одинарной точности
program lapack_ex4
  implicit none
  character(len=1) :: jobu, jobvt
  real, allocatable :: A(:,:), U(:,:), Vt(:,:), s(:), work(:)
  integer :: m, n, lda, ldu, ldvt, lwork, info, i, j
!-------------------------------------------------------
  write(*,*), "Введите размерности матрицы m и n"
  read(*,*) m, n
  lda = m
  ldu = m
  ldvt = m
  jobu = 'A'
  jobvt = 'A'
  ! Служебная переменная, чем больше, тем лучше
  lwork = 20*max(m,n)
  allocate(A(1:m,1:n))
  allocate(U(1:m,1:m))
  allocate(Vt(1:n,1:n))
  ! Диагональные элементы сингулярной матрицы
  allocate(s(1:min(n,m)))
  ! Служебный массив
  allocate(work(1:lwork))
  ! Заполняем числами от 0 до 1
  call random_number(A)
!!
  call sgesvd(jobu, jobvt, m, n, A, lda, s, &
    U, ldu, Vt, ldvt, work, lwork, info)
!!
end program lapack_ex4
```

2.7. Задания для лабораторной работы

2.7.1. Задание № 1

Проверьте работоспособность четырёх вышеприведённых примеров. Добавьте в программы код, организующий распечатку результатов. Проверьте правильность получаемых результатов на простых примерах матриц 3×3 (т.е. требуется задать массив вручную и вычислить решение СЛАУ, собственные векторы, матрицы разложения и т.д.).

2.7.2. Задание № 2

Вычислить определитель матрицы, используя LAPACK тремя разными способами: с помощью собственных векторов матрицы, с помощью SVD и LUP разложений. Какой способ наиболее прост? Какой способ наиболее быстр?

2.7.3. Задание № 3

Вычислить обратную или псевдообратную матрицу с помощью SVD разложения.

2.8. Содержание отчёта

Отчёт должен включать:

1) титульный лист;
2) формулировку цели работы;
3) описание результатов выполнения задания:
   - листинги программ;
   - результаты выполнения программ (текст, снимок экрана, таблицы и графики в в зависимости от задания);
4) выводы по каждому заданию лабораторной работы.

2.9. Контрольные вопросы

Для ответов на некоторые вопросы следует воспользоваться документацией с официального сайта LAPACK http://www.netlib.org/.

1. Как расшифровывается аббревиатура LAPACK?
2. Что такое BLAS и зачем он использован в LAPACK?
3. Процедуры LAPACK являются подпрограммами или функциями?
4. Какова схема названий процедур библиотеки LAPACK?
5. Какие языки программирования поддерживает LAPACK? На каком языке написан сам LAPACK?
6. Как будет именоваться процедура по вычислению собственных векторов для комплексной трёхдиагональной матрицы?
7. Есть ли в LAPACK процедура для вычисления определителя? Если есть, то приведите её название и аргументы.
8. Есть ли в LAPACK процедура для решения переопределённой или неопределённой системы алгебраических линейных уравнений? Если есть, то приведите её название и аргументы.
9. В описании аргументов процедуры sgesvd об аргументе jobvt было сказано кратко, так как он аналогичен jobu. Дайте его подробное описание по аналогии с jobu.
10. Что делает процедура sgeqrf? Дайте краткое описание.

3. OpenMP

3.1. Введение

OpenMP (Open Multi-Processing) — открытый стандарт для создания многопоточных программ. Поддерживаются языки `Fortaran`, `C` и `C++`.

OpenMP является реализацией *многопоточности (multithreading)*. Многопоточность означает, что выполняемый процесс может состоять из нескольких *потоков* (thread), выполняемых одновременно. Термин *thread* также переводят буквально как *«нить»*.

Все потоки выполняются в адресном пространстве одного процесса, из чего следует, что многопоточная технология программирования в общем и OpenMP в частности может применяться на многопроцессорных системах с разделяемой памятью (например, на компьютерах с многоядерным процессором). Это обстоятельство определяет как достоинства многопоточности, так и недостатки.

К достоинствам можно отнести:

- время, требуемое для создания и уничтожения нити в рамках процесса, мало по сравнению с временем, требуемым на создание и уничтожение полноценного нового процесса;
- относительная простота программирования многопоточных программ;
- эффективное использование многоядерных процессоров.

К недостаткам можно отнести:

- многопоточная программа может работать лишь на системах с разделяемой памятью (грубо говоря, в рамках одного компьютера, пусть и оснащённого несколькими процессорами);
- сложность адаптации последовательных алгоритмов для многопоточного выполнения.

Ввиду малых ресурсов, необходимых для создания и уничтожения потоков, в течение работы программы потоки могут создаваться и уничтожаться по многу раз. Однако выполняемый процесс имеет по крайней мере один *главный поток (master thread)*, который создаётся в самом начале и не уничтожается в течение всего времени его работы. Главный поток создаёт набор *подчинённых (slave)* потоков и задача распределяется между ними.

3.2. OpenMP и Fortran

Для компиляции программы, использующей OpenMP директивы, необходимо указать опцию `-fopenmp`

```
gfortran -fopenmp src-file.f90 -o prog-name
```

где `src-file.f90` — имя исходного файла с кодом программы, `prog-name` — имя исполняемого файла. В случае если не указать опцию `-o`, исполняемый файл получит стандартное имя `a.out`. Для проверки поддержки OpenMP можно использовать простейшую программу следующего вида:

```
program ex1
!$ print *, "OpenMP поддерживается!"
end program ex1
```

Области кода, которые следует выполнять параллельно, выделяются с помощью специальных директив препроцессора соответствующего языка — прагм. В случае языка Fortran все директивы в общем случае выглядят следующим образом:

```
!$omp <директива> [опции]
  <код>
!$omp end <директива>
```

Следует также отметить, что программирование с помощью OpenMP реализовано в рамках идеологии одна программа — множество данных (Single Program Multiple Data, SPMD). В рамках этой идеологии для двух нитей используется одинаковый код. В идеальном случае код параллельной и последовательной версий программ должен отличаться лишь директивами OpenMP, однако это случается нечасто.

Кроме директив предпроцессору в библиотеке OpenMP реализованы некоторые вспомогательные функции, позволяющие получать отладочную информацию. Для использования этих функций следует подключить заголовочный файл `omp_lib.h`.

Рассмотрим применение функций для замера времени выполнения участков кода:

```
program ex2
  implicit none
  include "omp_lib.h"
  real :: t1, t2, tick
  t1 = omp_get_wtime()
  call sleep(1)
  t2 = omp_get_wtime()
  write(*,*) "Спали ", t2-t1, " сек."
  write(*,*) "Точность таймера: ", omp_get_wtick()
end program ex2
```

Функции

```
integer :: omp_get_wtime()
integer :: omp_get_wtick()
```

возвращают соответственно системное время в секундах и разрешение таймера в секундах, которое можно рассматривать как меру точности таймера.

3.3. Параллельные и последовательные области

При запуске программы порождается нить мастер, которая начинает выполнение программы. Как только выполнение доходит до директивы `parallel`, порождаются дополнительные нити (число которых зависит от настроек системы и заданных опций).

Директива, задающая начало и конец параллельной области, выглядит следующим образом:

```
!$omp parallel [опции]
  <код параллельной области>
!$omp end parallel
```

Рассмотрим следующий пример:

```
program ex3
  implicit none
  include "omp_lib.h"
  integer :: i, n, tn
  real :: t1, t2
  write(*,*) "Последовательная область"
!$omp parallel num_threads(4)
  write(*,*) "Параллельная область"
!$omp end parallel
  write(*,*) "Последовательная область"
end program  ex3
```

В данном примере создаётся одна параллельная область. Опция `num_threads` позволяет явно указать количество создаваемых нитей. Количество нитей, создаваемых по умолчанию, задаётся переменной окружения `OMP_NUM_THREADS`. Оптимальное количество нитей зависит как от конфигурации системы, так и от алгоритма, и в каждом случае подбирается индивидуально.

Все нити, порождённые при входе в параллельную область, начнут выполнять указанные в ней команды. Например, при выполнении программы `ex3`:

```
gfortran -fopenmp ex3.f90 && ./a.out
```

в консоли будут распечатаны следующие сообщения:

```
Последовательная область
  Параллельная область
  Параллельная область
  Параллельная область
  Параллельная область
Последовательная область
```

Кроме опции `num_threads` с директивой `parallel` можно использовать следующие опции:

– `if(условие)` — выполнение параллельной области по условию;

- `default(private | firstprivate | shared | none)` — всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс `private`, `firstprivate` или `shared` соответственно; none означает, что всем переменным в параллельной области класс должен быть назначен явно;
- `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;
- `shared(список)` – задаёт список переменных, общих для всех нитей;
- `copyin(список)` – задаёт список переменных, объявленных как `threadprivate`, которые при входе в параллельную область инициализируются значениями соответствующих переменных в главной нити;
- `reduction(оператор:список)` – задаёт оператор и список общих переменных, для каждой переменной создаются локальные копии в каждой нити; над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор — это: `-, +, *, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

Иногда бывает необходимо, чтобы какой-то участок кода в параллельной области был выполнен только одной нитью. Закрывать параллельную область и открывать вновь было бы слишком ресурсоёмко. Для этой цели следует использовать директиву `single`:

```
!$omp single
  <код>
!$omp end single
```

Рассмотрим теперь ещё один пример программы, в котором иллюстрируется применение директивы `single` и функции `omp_get_thread_num()`:

```
program ex4
  implicit none
  include "omp_lib.h"
  integer :: i, n, tn
  real :: t1, t2
! Сравнить результаты для переменной объявленной
! как private и public (по умолчанию)

  write(*,*) "В переменную tn записывается номер нити"
  write(*,*) "Если эта переменная в параллельной &
              области не объявлена как private"
  write(*,*) "То для всех нитей она общая:"

!$omp parallel num_threads(4)
```

```
  tn = omp_get_thread_num()
  write(*,*) "Нить № ", tn
!$omp end parallel

  write(*,*) "Если же tn объявлена как private"
  write(*,*) "То для всех нитей создаётся отдельный экземпляр:"

!$omp parallel num_threads(4) private(tn)
  tn = omp_get_thread_num()
  write(*,*) "Нить № ", tn
!$omp end parallel

! Следующая часть иллюстрирует применение директивы single,
! которая используется для того, чтобы в параллельной области
! указанные в её рамках команды исполнялись только мастер-нитью
  t1 = omp_get_wtime()
!$omp parallel num_threads(100) private(tn)
  !$omp single
    tn = omp_get_thread_num()
    write(*,*) "Одна нить найдёт для нас число созданных нитей"
    n = omp_get_num_threads()
    write(*,*) "Создано нитей:", n
    write(*,*) "Остальные нити опять распечатают свои номера"
  !$omp end single
    tn = omp_get_thread_num()
    write(*,*) "Нить № ", tn
!$omp end parallel
  t2 = omp_get_wtime()
  write(*,*) "На создание и закрытие параллельной области &
             и нитей ушло сек", t2-t1
end program  ex4
```

Следует пояснить использование опции `private(tn)` во второй параллельной области. В OpenMP переменные в параллельных областях разделяются на два основных класса:

– `shared` — общие, все нити видят одну и ту же переменную;
– `privat` — локальные, каждая нить видит свой экземпляр переменной.

По умолчанию все переменные в параллельной области общие, кроме счётчиков итераций. Поэтому в первой параллельной области значение переменной `tn` будет одинаковым для всех нитей и её значение после параллельной области будет зависеть от того, какая нить последней записала в эту переменную свой номер. В консоль в данном случае будет выдано 4 сообщения типа `Нить № n`. Во второй же параллельной области значение переменной `tn` установлено как `private` и поэтому оно разное для каждой нити. В консоль в данном случае будут выданы следующие сообщения:

```
 Нить №  0
 Нить №  1
 Нить №  2
 Нить №  3
```

Отдельно стоит отметить, что при каждом запуске программы последовательность выдаваемых сообщений будет разной, ввиду других фоновых процессов, влияющих на время выполнения нашей программы.

3.4. Параллельные циклы

Если в параллельной области встречается оператор цикла `do`, то он будет выполнен всеми нитями одновременно. Для того чтобы указать на необходимость параллельного выполнения итераций цикла, следует использовать директиву `!$omp do`

```
!$omp do
  <код>
!$omp end do
```

Теперь итерации цикла в рамках этой директивы будут при возможности разделены между нитями. Отнюдь не каждый цикл может быть распараллелен. Например, следующий код:

```
do i = 2,n,1
  a(i) = a(i-1) + a(i-2)
end do
```

параллельному выполнению не поддаётся, так как каждый шаг вычисления непосредственно зависит от двух предыдущих.

Для возможности распараллеливания каждая нить должна иметь возможность выполнять свою часть цикла независимо от остальных нитей. В некоторых случаях последовательный алгоритм хорошо поддаётся распараллеливанию без дополнительных модификаций. К таким алгоритмам относятся многие действия над массивами (матрицами), что иллюстрирует следующий пример:

```
program ex4
  implicit none
  include "omp_lib.h"
  integer n,i,j,tn
  real :: t1, t2
  real, allocatable :: A(:,:), B(:,:), C(:,:)
  n = 4000
! Выделяем память
  allocate(A(1:n,1:n))
  allocate(B(1:n,1:n))
  allocate(C(1:n,1:n))
! Заполняем числами от 0 до 1
  call random_number(A)
  call random_number(B)
  call random_number(C)
  t1 = omp_get_wtime()
! Итерационные счётчики приватные по умолчанию
!$omp parallel shared(A,B,C) private(i,j,tn)  num_threads(4)
```

```
  tn = omp_get_thread_num()
  !$omp do
    do i = 1,n,1
      do j = 1,n,1
        C(i,j) = A(i,j) + B(i,j)
        !write(*,*) "Нить ", tn, "сложила элементы", i,j
      end do
    end do
  !$omp end do
!$omp end parallel
  t2 = omp_get_wtime()
  write(*,*) "На параллельное вычисление ушло сек", t2-t1

! Заново заполняем числами от 0 до 1
  call random_number(A)
  call random_number(B)
  call random_number(C)

  t1 = omp_get_wtime()
  !$omp do
    do i = 1,n,1
      do j = 1,n,1
        C(i,j) = A(i,j) + B(i,j)
        !write(*,*) "Нить ", tn, "сложила элементы", i,j
      end do
    end do
  t2 = omp_get_wtime()
  write(*,*) "На последовательное вычисление ушло сек", t2-t1
end program  ex4
```

В данном примере проиллюстрировано лишь сложение матриц. Более ресурсоёмкий процесс перемножения матриц также хорошо поддаётся распараллеливанию, что будет предложено показать самостоятельно в первом задании лабораторной работы.

В подавляющем большинстве случаев последовательный вариант алгоритма не поддаётся параллелизации и его приходится модифицировать. В простейших случаях параллельная версия лишь немного отличается от исходной. Например, вычисление суммы или произведения множества чисел (элементов массива) можно разбить на части согласно формулам:

$$\sum_{i=1}^{n} a(i) = \sum_{i=1}^{m} a(i) + \sum_{i=m+1}^{n} a(i),$$

$$\prod_{i=1}^{n} a(i) = \prod_{i=1}^{m} a(i) + \prod_{i=m+1}^{n} a(i)$$

Однако чаще параллельный вариант алгоритма существенно отличается от последовательного, и при его реализации нарушается идеология Single Program Multiple Data. Это означает, что уже невозможно превратить параллельный вариант программы в последовательный, лишь убрав директивы OpenMP.

3.5. Параллельные секции

Несмотря на то, что OpenMP реализует идеологию Single Program Multiple Data, все же имеются средства для ручного распараллеливания программы и контроля над тем, какую часть кода будет выполнять та или иная нить. Этой цели служит директива `sections`:

```
!$omp sections
  <блок_секций>
!$omp end sections
```

Данная директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью. Для иллюстрации рассмотрим следующий пример:

```
program ex5
  implicit none
  include "omp_lib.h"
  integer :: tn
!$omp parallel private(tn)
  tn=omp_get_thread_num()
!$omp sections
  !$omp section
  write(*,*) "Первая секция, процесс ", tn
  !$omp section
  write(*,*) "Вторая секция, процесс ", tn
  !$omp section
  write(*,*) "Третья секция, процесс ", tn
!$omp end sections
  write(*,*) "Параллельная область, процесс ", tn
!$omp end parallel
end program  ex5
```

3.6. Задания для лабораторной работы

3.6.1. Задание № 1

В данном задании предлагается написать параллельную версию *решета Эратосфена* — известного алгоритма по нахождению простых чисел. Ниже дан последовательный вариант этого алгоритма, реализованный на

фортране. Заметим, что переменная n задаёт натуральное число, до которого необходимо производить поиск простых чисел.

```
program Eratosthenes
  implicit none
  include "omp_lib.h"
  integer :: i,j
  integer :: n
  integer :: p_num, th_num
  ! Замер времени
  real :: t1, t2
  logical, allocatable :: A(:)
  ! Открыли файл
  open(10,file = "primes.txt")
  write(*,*), "Введите n ="
  read(*,*) n
  write(*,*), "Введите th_num ="
  read(*,*) th_num
! Выделяем память под массив
  allocate(A(2:n))
!-------------------------------------
  A = .true.
  t1 = omp_get_wtime()
  do i = 2,n,2
    A(i) = .false.
  end do
! Непосредственно алгоритм
  do i = 2,nint(sqrt(real(n))),1
    if(A(i) .eqv. .true.) then
      do j = i**2,n,i
        A(j) = .false.
      end do
    end if
  end do
  t2 = omp_get_wtime()
  print *, "Время работы алгоритма: ", t2-t1
!-------------------------------------
! Записываем найденные простые числа в файл
  p_num = 0
  do i = 2,n,1
    if(A(i)) then
      p_num = p_num + 1
      write(10,*) i
    end if
  end do
  write(*,*), "Найдено ", p_num, " простых чисел"
! Закрываем файл после того как завершили с ним работу
```

```
  close(10, status = "keep")
end program Eratosthenes
```

1. С помощью директив OpenMP требуется распараллелить части программы (где это возможно).
2. Необходимо исследовать влияние опции `dynamic` директивы `!$omp do` на время выполнения программы. Даёт ли динамическое распределение блоков выигрыш в данном случае?
3. Необходимо исследовать, как зависит время выполнения программы от числа нитей при достаточно большом фиксированном n. Определить оптимальное число нитей.

3.6.2. Задание № 2

Реализовать параллельное перемножение квадратных матриц стандартным алгоритмом и алгоритмом Винограда. Замерить выигрыш по времени параллельного варианта программ по сравнению с последовательным. Сравнить быстродействие параллельного варианта со встроенной функцией `matmul`.

3.6.3. Задание № 3

Данное задание демонстрирует невозможность непосредственного распараллеливания простейшего алгоритма сортировки (сортировка пузырьком). Однако можно перестроить алгоритм так, что становится возможным параллельное выполнение его итераций. В приведённом ниже исходном коде реализована классическая сортировка пузырьком, модифицированный её вариант (последовательный) и параллельный вариант. Необходимо распараллелить параллельный вариант с помощью директив OpenMP и сравнить быстродействие с двумя последовательными вариантами.

```
program bubble
  implicit none
  include "omp_lib.h"
  integer :: i,j,n, first
  logical :: sorted
  real :: t1, t2
  real, allocatable :: a(:), b(:)
  n = 200000
  allocate(a(1:n))
  allocate(b(1:n))
  call random_number(a)
  b = a
!============================
! Оригинальный алгоритм сортировки пузырьком
!============================
```

```
  t1 = omp_get_wtime()
  do i = 1,n,1
    do j = 1,n-i,1
      if(a(j) .gt. a(j+1)) then
        call swap(a(j),a(j+1))
      end if
    end do
  end do
  t2 = omp_get_wtime()
!============================
  write(*,*) "Затратили время (последовательно)", t2-t1
  ! write(*,'(f8.3)') a
!============================
! Переработанный алгоритм пузырька
! Odd-Even transposition sort
!============================
  ! t1 = omp_get_wtime()
  ! sorted = .false.
  ! do while(.not. sorted) ! Делать пока sorted ложь
    ! sorted = .true.
    ! do i = 2,n-1,2
      ! if(b(i) .gt. b(i+1)) then
        ! call swap(b(i),b(i+1))
        ! sorted = .false.
      ! end if
    ! end do
    ! do i = 1,n-1,2
      ! if(b(i) .gt. b(i+1)) then
        ! call swap(b(i),b(i+1))
        ! sorted = .false.
      ! end if
    ! end do
  ! end do
  ! t2 = omp_get_wtime()
!============================
!============================
! Переработанный алгоритм пузырька для распараллеливания
!============================
  t1 = omp_get_wtime()
  do i = 1,n,1
    first = mod(i,2)
    do j = first,n-1,2
      if(b(j) > b(j+1)) then
        call swap(b(j),b(j+1))
      end if
    end do
```

```
  end do
  t2 = omp_get_wtime()
  write(*,*) "Затратили время (параллельно)", t2-t1
end program  bubble
! - Подпрограмма для перестановки элементов массива
subroutine swap(a, b)
  real :: a, b
  real :: tmp
  tmp = a
  a = b
  b = tmp
end subroutine swap
```

Последовательный и параллельный алгоритмы иллюстрируются на рис. 3.1 и 3.2:

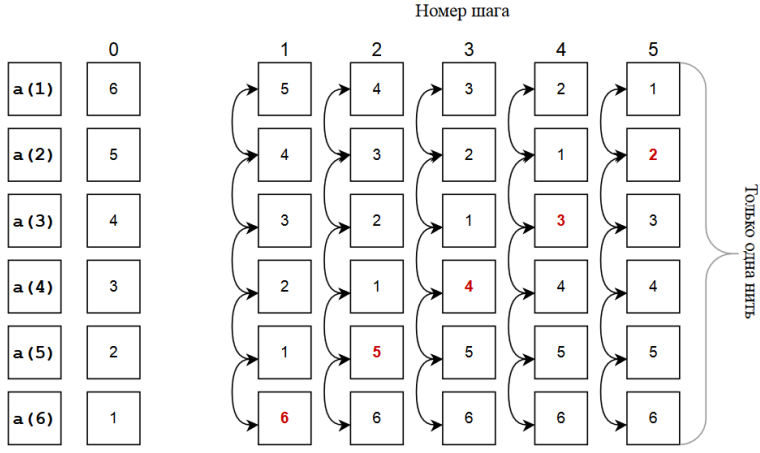

Рис. 3.1. Последовательный алгоритм сортировки пузырьком

3.7. Содержание отчёта

Отчёт должен включать:

1) титульный лист;
2) формулировку цели работы;
3) описание результатов выполнения задания:
   – листинги программ;

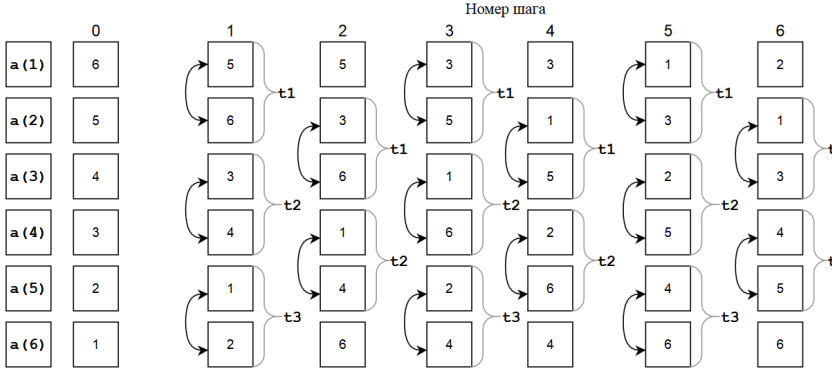

Рис. 3.2. Параллельный алгоритм сортировки пузырьком

– результаты выполнения программ (текст, снимок экрана, таблицы и графики в в зависимости от задания);

4) выводы по каждому заданию лабораторной работы.

3.8. Контрольные вопросы

1. В чём заключается основное отличие процесса от потока/нити?
2. Какие преимущества даёт технология OpenMP?
3. Оправдано ли использование технологии OpenMP (и вообще многопоточного программирования) при написании пользовательских программ для персонального компьютера?
4. Как реализуется применение OpenMP при написании программ на языке Fortran? На языках C/C++?
5. Перечислите наиболее используемые директивы OpenMP.
6. Перечислите наиболее используемые функции из библиотеки OpenMP.
7. Какие переменные окружения влияют на число создаваемых потоков?
8. Поясните смысл идеологии Single Program Multiple Data на конкретном примере (можно воспользоваться одной из вышеизложенных программ).

4. MPI

4.1. Введение

Для компьютеров с раздельной оперативной памятью технология OpenMP уже не применима. В таких системах основным способом параллелизации программ является организация обмена данными между процессами. Одной из технологий, позволяющих организовать взаимодействие параллельно выполняемых процессов, является технология обмена сообщениями MPI.

Аббревиатура MPI расшифровывается как Message Passing Interface — интерфейс передачи сообщений. MPI изначально поддерживается языками `C` и `Fortran`, однако существуют библиотеки и для других языков.

В отличие от OpenMP основной парадигмой программирования MPI является MIMD (Multiple Instructions Multiple Data), что дословно можно перевести как множество инструкций – множество данных, а на практике означает, что параллельный вариант кода программы существенно отличается от последовательного. Напомним, что при программировании с использованием OpenMP, напротив, код параллельной программы в идеальном случае отличается от кода параллельной программы только наличием директив OpenMP.

4.2. MPI и Fortran

MPI является стандартом и не включает в себя конкретные реализации процедур, поэтому для Unix систем существует несколько реализаций MPI, в том числе и коммерческих. Мы будем рассматривать свободно распространяемую реализацию под названием mpich. Именно эта версия доступна для использования в дисплейных классах.

Для компиляции и запуска Фортран-программы с использованием MPI следует выполнить следующие команды:

```
mpif90  src-file.f90 -o prog-name
mpirun -np N prog-name
```

С помощью первой программы исходный файл `src-file.f90` программы компилируется в исполняемый файл `prog-name`, а с помощью команды `mpirun` полученный файл запускается с поддержкой MPI. Опция `-np N` указывает число `N` параллельных процессов, которые необходимо породить. Заметим также, что в зависимости от версии `mpich` для запуска исполняемого файла на локальном компьютере может быть необходимо указание дополнительной опции `-host localhost`, т.е.

```
mpirun -np N -host localhost prog-name
```

Перейдём теперь к рассмотрению основ MPI применительно к языку Фортран. Для использования подпрограмм и функций MPI необходимо подключить заголовочный файл: `include 'mpif.h'`. Для предотвращения конфликтов с другими библиотеками все дополнительные объекты, процедуры, константы, типы данных, используемые в MPI, имеют префикс `mpi_`.

При запуске программы с MPI порождается множество параллельных процессов. Заметим, что, в отличие от OpenMP, это не нити, а полноценные процессы, их порождение и завершение сравнительно ресурсоёмко. Поэтому фиксированное число процессов порождается единожды, и в ходе работы нельзя создать дополнительные процессы или же уничтожить работающие (хотя в современных версиях MPI эта возможность есть).

У каждого процесса своё адресное пространство и свой набор переменных. Все взаимодействие между ними осуществляется с помощью передачи и приёма сообщений.

При передаче сообщений от процесса к процессу необходимо иметь возможность однозначно идентифицировать каждый процесс, а также группы процессов. Для этих целей используется два атрибута:

– *коммуникатор* процессов (`comm_id`) — идентификатор группы процессов;
– *идентификатор* процесса (`proc_id`).

Любой процесс однозначно адресуется `comm_id` и `proc_id`.

По умолчанию создаются три коммуникатора:

– `mpi_comm_world` — все созданные процессы;
– `mpi_comm_self` — только текущий процесс;
– `mpi_comm_null` — ни один процесс.

Указанные названия можно использовать непосредственно в программе, так как они определяются при подключении библиотеки.

Каждое сообщение имеет следующие атрибуты:

– номер процесса отправителя;
– номер процесса получателя;
– коммуникатор процесса получателя;
– идентификатор сообщения (tag, тег).

Идентификатор сообщения является неотрицательным числом типа `integer` и позволяет программисту при необходимости разбить все передаваемые сообщения на классы.

4.3. Основные процедуры MPI

Перейдём к рассмотрению основных и некоторых вспомогательных процедур MPI.

В каждой программе обязательно должны присутствовать вызовы следующих двух подпрограмм:

```
integer :: ierr
call mpi_init(ierr)
call mpi_finalise(ierr)
```

Первая подпрограмма инициализирует параллельную часть программы, а вторая завершает параллельную часть. После `mpi_finalise` обращение к любой процедуре MPI запрещено, в том числе и к процедуре `mpi_init`. Из этого следует, что в рамках программы допустима только одна параллельная часть.

Рассмотрим первый пример:

```
program ex1
  implicit none
  include 'mpif.h'
  integer :: ierr
  logical :: flag
! В зависимости от реализации MPI следующее сообщение
! будет напечатано или только одним процессом, или же
! всеми созданными процессами
  write (*,*),"До параллельной части"
  call mpi_init(ierr)
!-----------------------------------
  call mpi_initialized(flag, ierr)
  if(flag) then
! Следующее сообщение обязательно будет распечатано
! всеми процессами
    write (*,*), "Параллельная часть"
  end if
!-----------------------------------
  call mpi_finalize(ierr)
  write (*,*), "После параллельной части"
end program ex1
```

В этом примере используется процедура

```
mpi_initialized(flag, ierr)
```

которая записывает в переменную `flag` значение `.true.` в случае, если процедура вызвана в параллельной области, и `.false.`, если процедура вызвана вне параллельной области.

Следующие две процедуры используются для получения идентификаторов процесса (`comm_id`) и коммуникатора (`proc_id`):

```
integer :: comm_id, comm_size, proc_id, ierr
call mpi_comm_size(comm_id, comm_size, ierr)
call mpi_comm_rank(comm_id, proc_id, ierr)
```

В переменную `comm_size` записывается число процессов в коммуникаторе `comm_id`. Следующий пример иллюстрирует применение данных процедур:

```
program ex2
  implicit none
  include 'mpif.h'
  integer :: ierr
! Число процессов в коммуникаторе
  integer :: comm_size
```

```
! Номер процесса
  integer :: proc_id
  call mpi_init(ierr)
!------------------------------------
  call mpi_comm_size(mpi_comm_world, comm_size, ierr)
  call mpi_comm_rank(mpi_comm_world, proc_id, ierr)
  write(*,*), "Процесс №", proc_id, ", всего ", comm_size
!------------------------------------
  call mpi_finalize(ierr)
end program ex2
```

Следующий пример (ex3) показывает использование процедуры mpi_get_processor_name, которая позволяет получить имя используемого процессора (это актуально, например, в случае кластера):

```
    integer :: length
    character(len = mpi_max_processor_name) ::
       processor_name
       mpi_get_processor_name(processor_name, length, ierr)
```

Обратите внимание на использование переменной length, благодаря которой имя процессора выводится в консоль без дополнительных знаков пробела:

```
program ex3
  implicit none
  include 'mpif.h'
  integer :: ierr
! Количество символов в имени процессора
  integer :: length
! Имя процессора
  character(len = mpi_max_processor_name) :: processor_name
  real(kind = 8) :: time
!!!
  call mpi_init(ierr)
!------------------------------------
  time = mpi_wtime(ierr)
  call mpi_get_processor_name(processor_name, length, ierr)
  write(*,*), "Процессор ", processor_name(1:length), &
              " Время ", time
!------------------------------------
  call mpi_finalize(ierr)
end program ex3
```

4.4. Приём и передача сообщений

Подновляющее число процедур MPI служит для приёма и отправки сообщений между процессами. Все процедуры по приёму и передаче сообщений можно разделить на два класса:

– приём и передача с блокировкой (с синхронизацией);
– приём и передача без блокировки (асинхронные).

Здесь мы изучим лишь процедуры с блокировкой. Для дальнейшего изучения следует обратиться к методическому пособию [1].

Две основные процедуры MPI: `mpi_send` — отправка сообщения и `mpi_recv` — приём сообщения. Рассмотрим их по порядку.

```
<type> :: msg(*)
integer :: count, datatype, proc_id, &
   msgtag, comm_id, ierr
mpi_ssend(msg, count, datatype, proc_id, &
   msgtag, comm_id, ierr)
```

– `count` — число элементов в массиве-сообщении `msg`;
– `datatype` — тип элементов в массиве `msg`;
– `proc_id` — идентификатор процесса-получателя;
– `msgtag` — тег сообщения (идентификатор сообщения);
– `comm_id` — идентификатор коммуникатора;
– `ierr` — служебная переменная для записи кода ошибки.

Следует подробнее остановиться на параметре `datatype`. В MPI для передачи сообщений определены свои типы, эквивалентные стандартным типам Фортрана:

Тип данных MPI	Расшифровка
`mpi_integer`	целые числа
`mpi_real`	вещественные числа
`mpi_double_precision`	вещественные числа двойной точности
`mpi_complex`	комплексные числа
`mpi_logical`	логический тип
`mpi_character`	текстовой тип

Все эти типы данных уже предопределены, и их следует использовать при вызове процедур.

Параметр `proc_id` может принимать специальное значение `mpi_proc_null`, адресующее несуществующий процесс. Операции по пересылке сообщения этому процессу завершаются немедленно с успешным кодом завершения, и выполнение программы продолжается.

Следует также иметь в виду модификацию процедуры `mpi_send`, процедуру `mpi_ssend`, которая осуществляет пересылку сообщения с синхронизацией. Это означает, что передающий процесс не продолжит работу, пока посланное им сообщение не будет принято. Использовать пересылку с синхронизацией следует внимательно, во избежании блокировки.

Рассмотрим теперь процедуру по приёму сообщения:

```
<type> :: msg(*)
integer :: count, datatype, proc_id, msgtag, comm_id, ierr
integer :: status(mpi_status_size)
mpi_recv(msg, count, datatype, proc_id, &
         msgtag, comm_id, status, ierr)
```

Здесь `status` — целочисленный массив размера `mpi_status_size`, в который записывается информация о переданном сообщении. Перечислим важнейшие его элементы:
- `mpi_source` — номер процесса отправителя;
- `mpi_tag` — идентификатор сообщения;
- `mpi_error` — код ошибки.

Параметр `proc_id` может принимать предопределённое значение `mpi_any_source`, а параметр `mstag` также может принимать значение `mpi_any_tag`.

Приведём теперь пример использования процедур передачи и приёма:

```
program ex4
  implicit none
  include 'mpif.h'
  integer, parameter :: n = 100
  integer :: proc_id, ierr, tag
  integer, dimension(1:mpi_status_size) :: status
  character(len = n) :: msg1
  character(len = n) :: msg2
  msg1 = "Привет параллельному процессу!"
  tag = 10
  call mpi_init(ierr)
  !---------------------------------
  call mpi_comm_rank(mpi_comm_world, proc_id, ierr)
  if (proc_id .eq. 0) then
    call mpi_ssend(msg1, n, mpi_character, 1,
      tag, mpi_comm_world, ierr)
    write(*,*),"Процесс №", proc_id,
      "послал сообщение: ", msg1
  else if (proc_id .eq. 1) then
    call mpi_recv(msg2, n, mpi_character, 0,
      tag, mpi_comm_world, status, ierr)
    write(*,*),"Процесс №", proc_id, &
        "принял сообщение: ", msg2
    write(*,*), "status(mpi_source)=", status(mpi_source)
    write(*,*), "status(mpi_tag)=", status(mpi_tag)
    write(*,*), "status(mpi_error)=", status(mpi_error)
  end if
  !---------------------------------
  call mpi_finalize(ierr)
end program ex4
```

Схема, показанная в данном примере, используется в любой программе, написанной с использованием MPI. Вначале параллельной области вызывается процедура `mpi_comm_rank` и в переменную `proc_id` записывается идентификатор процесса. Так как в MPI нет разделяемых переменных, то значения `proc_id` будут различаться для каждого процесса. Это можно использовать для выделения участков кода, предназначенных для выполне-

ния конкретным процессом, что и было сделано в вышеизложенном примере с помощью операторов условия. Следует всегда помнить, что никакого автоматического распараллеливания программы в MPI не предусмотрено и разделить участки кода между процессами *необходимо вручную*.

```
integer :: proc_id, msgtag, comm_id, ierr
integer :: status(mpi_status_size)
mpi_probe(proc_id, msgtag, comm_id, status, ierr)
```

Процедура `mpi_probe` получает информацию о структуре ожидаемого сообщения с идентификатором `mstag` от процесса с идентификатором `proc_id` в коммуникаторе `comm_id`. При этом ожидание сопровождается блокировкой. Важно, что `mpi_probe` сообщение не принимает и после него надо вызывать `mpi_recv`. В этом случае будет получено то сообщение, информацию о котором получила процедура `mpi_probe`. В нижеследующем примере показано применение этой процедуры.

```
program ex5
  implicit none
  include 'mpif.h'
  integer:: proc_id, ierr
  integer, dimension(1:mpi_status_size) :: status
  integer :: im
  real :: rm
  call mpi_init(ierr)
  !--------------------------------
  call mpi_comm_rank(mpi_comm_world, proc_id, ierr)
! Две переменные разного типа для отправки
  im = proc_id
  rm = 1.0*proc_id
! Отправка сообщений двумя процессами
  if(proc_id .eq. 1) then
    call mpi_ssend(im, 1, mpi_integer, 0, 5,
         mpi_comm_world, ierr)
  end if
  if(proc_id .eq. 2) then
    call mpi_ssend(rm, 1, mpi_real, 0, 5,
         mpi_comm_world, ierr)
  end if
! Прием отправленных сообщений
  if(proc_id .eq. 0) then
    call mpi_probe(mpi_any_source, 5,
      mpi_comm_world, status, ierr)
! Если первым пришло сообщение от первого процесса
    if(status(mpi_source) .eq. 1) then
      call mpi_recv(im, 1, mpi_integer, 1, 5,
        mpi_comm_world, status, ierr)
      call mpi_recv(rm, 1, mpi_real, 2, 5,
        mpi_comm_world, status, ierr)
```

```
! Если первым пришло сообщение от второго процесса
    else if(status(mpi_source) .eq. 2) then
      call mpi_recv(rm, 1, mpi_real, 2, 5,
        mpi_comm_world, status, ierr)
      call mpi_recv(im, 1, mpi_integer, 1, 5,
        mpi_comm_world, status, ierr)
    end if
  write(*,*) "Процесс 0 получил", im, " от процесса 1, ", &
             rm, "от процесса 2"
  end if
  !---------------------------------
  call mpi_finalize(ierr)
end program ex5
```

В завершение приведём пример по параллельному вычислению суммы массива:

```
program ex6
  implicit none
  include 'mpif.h'
  integer :: proc_id, ierr, tag
  integer, dimension(1:mpi_status_size) :: status
  real :: a, b, aa, bb
  real :: array_sum
  real(kind = 8) :: t1, t2
  real, dimension(1:100000000) :: array
  tag = 10
  call random_number(array)
  call mpi_init(ierr)
  !---------------------------------
  call mpi_comm_rank(mpi_comm_world, proc_id, ierr)
! Процесс 0 суммирует половину массива
! и отправляет результат процессу 2
  if (proc_id .eq. 0) then
    a = sum(array(1:50000000))
    call mpi_send(a, 1, mpi_real, 2, tag,
       mpi_comm_world, ierr)
    write(*,*), "Процесс 0 закончил суммирование"
! Процесс 1 суммирует вторую половину массива
! и также отправляет результат процессу 2
  else if (proc_id .eq. 1) then
    b = sum(array(50000001:100000000))
    call mpi_send(b, 1, mpi_real, 2, tag,
       mpi_comm_world, ierr)
    write(*,*), "Процесс 1 закончил суммирование"
! Процесс 2 складывает полученные от проц 0 и 1 результаты
  else if (proc_id .eq. 2) then
    t1 = mpi_wtime(ierr)
```

```
    call mpi_recv(aa, 1, mpi_real, 0, tag,
      mpi_comm_world, status, ierr)
    call mpi_recv(bb, 1, mpi_real, 1, tag,
      mpi_comm_world, status, ierr)
    array_sum = aa + bb
    t2 = mpi_wtime(ierr)
! Распечатываем затраченное время
    write(*,*), "t2-t1", t2-t1
! Для сравнения быстродействия суммируем последовательно
    t1 = mpi_wtime(ierr)
    array_sum = sum(array)
    t2 = mpi_wtime(ierr)
    write(*,*), "t2-t1", t2-t1
  end if
  !---------------------------------
  call mpi_finalize(ierr)
end program ex6
```

4.5. Задания для лабораторной работы

4.5.1. Задание № 1

Написать программу по перемножению матриц по стандартному алгоритму и по алгоритму Винограда. Распараллелить её, используя MPI. Сравнить быстродействие по сравнению с последовательным вариантом и в зависимости от числа используемых процессов.

4.5.2. Задание № 2

Исследование эффективности реализации двухсторонних передач данных.

Постановка задачи. Разработать алгоритм передачи данных между процессами по кругу:

- второй процесс передаёт данные первому процессу;
- третий процесс, получив данные от нулевого, передаёт их второму,
- i-й процесс принимает данные от (i-1) процесса и пересылает их (i+1) процессу, последний процесс передаёт данные нулевому и т.д.

Требуется:

- реализовать алгоритм с использованием блокирующих и неблокирующих передач программы;
- исследовать время решения задачи для длин сообщений: 128, 512, 1024, 2048 и 4096 байт;
- число циклов круговой передачи задавать параметром командной строки;

– построить графики времени выполнения одного цикла передачи, усреднённого по числу кругов, для сообщений разной длины и разного числа процессов.

4.5.3. Задание № 3

Распараллелить программу, вычисляющую определённый интеграл, с помощью MPI. В качестве примера взять любой вычисляемый в конечном виде (для проверки результата работы программы) определённый интеграл.

4.6. Содержание отчёта

Отчёт должен включать:
1) титульный лист;
2) формулировку цели работы;
3) описание результатов выполнения задания:
   – листинги программ;
   – результаты выполнения программ (текст, снимок экрана, таблицы и графики в в зависимости от задания);
4) выводы по каждому заданию лабораторной работы.

4.7. Контрольные вопросы

1. Назовите главные отличия OpenMP и MPI. В каких случаях какую из этих технологий предпочтительнее использовать?
2. Как на практике реализуется парадигма Multiple Instructions Multiple Data (MIMD)?
3. Какие типы данных определены в MPI? Как они соотносятся с типами данных языка Фортран?
4. Кратко опишите суть технологии MPI.
5. Какие разновидности процедуры `mpi_send()` существуют? В чем их отличие друг от друга?
6. Укажите основные различия между синхронным (блокирующим) и асинхронным способами передачи сообщений.
7. На основании чего в MPI можно однозначно идентифицировать тот или иной процесс?
8. Последовательная часть программы (до директивы `mpi_init(ierr)`) будет выполняться всеми процессами одновременно или же только одним процессом?

Литература

1. Антонов А. С. Параллельное программирование с использованием технологии MPI. — Москва : Издательство МГУ, 2004. — 77 с. — ISBN: 5-211-04907-1.
2. Антонов А. С. Параллельное программирование с использованием технологии OpenMP. — Москва : Издательство МГУ, 2009. — 77 с. — ISBN: 978-5-211-05702-9.
3. Богачев К. Ю. Основы параллельного программирования. — Москва : Бином, 2003. — 342 с. — ISBN: 5-94744-037-0.
4. Макконнелл Д. Анализ алгоритмов. — 3 изд. — Москва : Техносфера, 2009. — 416 с. — ISBN: 978-5-94836-216-8.
5. Старченко А. В., Данилкин Е. А., Лаева В. И., Проханов С. А. Практикум по методам параллельных вычислений. — Москва : Издательство Московского университета, 2010. — 200 с. — ISBN: 978-5-211-05976-4.
6. Рыжиков Ю. И. Современный Фортран. — Санкт-Петербург : Корона Принт, 2009. — 288 с. — ISBN: 978-5-7931-0514-9.
7. Якобовский М. В. Введение в параллельные методы решения задач. — Москва : Издательство Московского университета, 2013. — 328 с. — ISBN: 918-5-211-06382-2.

Учебно-методический комплекс дисциплины

«Параллельное программирование»

Рекомендуется для направления подготовки
«Математика и компьютерные науки»
и «Прикладная математика и информатика»

Квалификация (степень) выпускника: бакалавр

Программа дисциплины

Цели и задачи дисциплины

Целью дисциплины является введение учащихся в предметную область современных параллельных вычислений.

В процессе преподавания дисциплины решаются следующие задачи:

- освоение архитектурных принципов реализации параллельной обработки в вычислительных машинах;
- изучение методов и языковых механизмов конструирования параллельных программ;
- овладение параллельными вычислительными методами.

Место дисциплины в структуре основной образовательной программы

Дисциплина относится к базовой части профессионального цикла.

Требования к входным знаниям, умениям и компетенциям студента: требуется пройти обучение по дисциплинам «Операционные системы», «Языки и методы программирования», «Численные методы».

Требования к результатам освоения дисциплины

Процесс изучения дисциплины направлен на формирование следующих компетенций (указываются в соответствии с ФГОС ВПО):

Для направления «Математика и компьютерные науки»:

ОК: 6, 7, 10–13; ПК: 1–3, 7–11, 14, 19, 20, 22–24:

ОК-6 — способность применять в научно-исследовательской и профессиональной деятельности базовые знания в области фундаментальной и прикладной математики и естественных наук;

ОК-7 — обладать значительными навыками самостоятельной научно-исследовательской работы;

ОК-10 — умение быстро находить, анализировать и грамотно контекстно обрабатывать научно-техническую, естественнонаучную и общенаучную информацию, приводя ее к проблемно-задачной форме;

ОК-11 — обладать фундаментальной подготовкой в области фундаментальной математики и компьютерных наук, готовность к использованию полученных знаний в профессиональной деятельности;

ОК-12 — владеть значительными навыками самостоятельной работы с компьютером, программирования, использования методов обработки информации и численных методов решения базовых задач;

ОК-13 — владеть базовыми знаниями в области информатики и современных информационных технологий, навыками использования программных средств и навыками работы в компьютерных сетях, умение создавать базы данных и использовать ресурсы Интернета;

ПК-1 — умение определять общие формы, закономерности, инструментальные средства отдельной предметной области;

ПК-2 — умение понять поставленную задачу;

ПК-3 — умение формулировать результат;

ПК-7 — умение грамотно пользоваться языком предметной области;

ПК-8 — умение ориентироваться в постановках задач;

ПК-9 — знание корректных постановок классических задач;

ПК-10 — понимание корректности постановок задач;

ПК-11 — владение навыками самостоятельного построения алгоритма и его анализа;

ПК-14 — владение навыками контекстной обработки информации;

ПК-19 — владение методом алгоритмического моделирования при анализе постановок математических задач;

ПК-20 — владение методами математического и алгоритмического моделирования при анализе и решении прикладных и инженерно-технических проблем;

ПК-22 — умение увидеть прикладной аспект в решении научной задачи, грамотно представить и интерпретировать результат;

ПК-23 — умение проанализировать результат и скорректировать математическую модель, лежащую в основе задачи;

ПК-24 — владение методами алгоритмического моделирования при анализе управленческих задач в научно-технической сфере, а также в экономике, бизнесе и гуманитарных областях знаний.

Для направления «Прикладная математика и информатика»:

ОК: 11, 12, 14, 15; ПК: 1–3, 6, 7, 9, 10:

ОК-11 — способность владения навыками работы с компьютером как средством управления информацией;

ОК-12 — способность работать с информацией в глобальных компьютерных сетях;

ОК-14 — способность использовать в научной и познавательной деятельности, а также в социальной сфере профессиональные навыки работы с информационными и компьютерными технологиями;

ОК-15 — способность работы с информацией из различных источников, включая сетевые ресурсы сети Интернет, для решения профессиональных и социальных задач;

ПК-1 — способность демонстрации общенаучных базовых знаний естественных наук, математики и информатики, понимание основных фактов, концепций, принципов теорий, связанных с прикладной математикой и информатикой;

ПК-2 — способность приобретать новые научные и профессиональные знания, используя современные образовательные и информационные технологии;

ПК-3 — способность понимать и применять в исследовательской и прикладной деятельности современный математический аппарат;

ПК-6 — способность осуществлять целенаправленный поиск информации о новейших научных и технологических достижениях в сети Интернет и из других источников;

ПК-7 — способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным, профессиональным, социальным и этическим проблемам;

ПК-9 — способность решать задачи производственной и технологической деятельности на профессиональном уровне, включая: разработку алгоритмических и программных решений в области системного и прикладного программирования;

ПК-10 — способность применять в профессиональной деятельности современные языки программирования и языки баз данных, операционные системы, электронные библиотеки и пакеты программ, сетевые технологии.

В результате изучения дисциплины студент должен:

Знать:

- общие реализации параллельной обработки в вычислительных машинах;
- методы и языковые механизмы конструирования параллельных программ;
- параллельные вычислительные методы;
- корректные постановки классических задач;
- корректность постановок задач.

Уметь:

- быстро находить, анализировать и грамотно контекстно обрабатывать научно-техническую информацию, приводя ее к проблемно-задачной форме;
- использовать ресурсы Интернета;
- определять общие формы, закономерности, инструментальные средства отдельной предметной области — операционных систем;
- понять поставленную задачу;
- формулировать результат;
- грамотно пользоваться языком предметной области;
- ориентироваться в постановках задач;
- увидеть прикладной аспект в решении научной задачи, грамотно представить и интерпретировать результат;
- проанализировать результат и скорректировать математическую модель, лежащую в основе задачи.

Владеть:

- способностью применять в научно-исследовательской и профессиональной деятельности базовые знания в области фундаментальной и прикладной математики и естественных наук;

- значительными навыками самостоятельной научно-исследовательской работы;
- фундаментальной подготовкой в области компьютерных наук;
- значительными навыками самостоятельной работы с компьютером, программирования, использования методов обработки информации;
- базовыми знаниями в области информатики и современных информационных технологий, навыками использования программных средств и навыками работы в компьютерных сетях;
- навыками самостоятельного построения алгоритма и его анализа;
- навыками контекстной обработки информации;
- методом алгоритмического моделирования при анализе постановок математических задач;
- методами математического и алгоритмического моделирования при анализе и решении прикладных и инженерно-технических проблем;
- методами алгоритмического моделирования при анализе управленческих задач в научно-технической сфере;
- параллельными вычислительными методами.

Объём дисциплины и виды учебной работы

Общая трудоемкость дисциплины составляет 3 зачетные единицы.

№	Вид учебной работы	Всего часов	Семестры
			7
1.	**Аудиторные занятия (всего)**	**51**	**51**
	В том числе:		
1.1.	Лекции	-	-
1.2.	Прочие занятия	-	-
	В том числе:		
1.2.1.	*Практические занятия (ПЗ)*	-	-
1.2.2.	*Семинары (С)*	-	-
1.2.3.	*Лабораторные работы (ЛР)*	51	51
1.2.4.	*Из них в интерактивной форме (ИФ)*	51	51

2.	**Самостоятельная работа студентов**	**57**	**57**
	В том числе:		
2.1.	Курсовой проект (работа)	-	-
2.2.	Расчетно-графические работы	-	-
2.3.	Реферат	-	-
2.4.	Подготовка и прохождение промежуточной аттестации	27	27
2.5.	*Другие виды самостоятельной работы:*		
2.5.1.	Самостоятельная проработка дополнительных материалов по дисциплине	30	30
2.5.2.	Выполнение домашних заданий	-	-
3.	**Общая трудоёмкость (ак.часов)**	**108**	**108**
4.	**Общая трудоёмкость (зач. ед.)**	**3**	**3**

Содержание дисциплины

Содержание разделов дисциплины

№ п/п	Наименование раздела дисциплины	Содержание раздела
1.	Язык Фортран	Основные сведения о языке Фортран. История развития. Его преимущества в области научных вычислений по сравнению с другими языками высокого уровня. Структура программы. Типы данных. Встроенные операции и функции. Операторы управления и ветвления. Массивы и работа с ними: описание массивов, задание массивов, динамические массивы, основные функции работы с массивами как матрицами. Ввод и вывод

2.	Библиотека LAPACK	Основные сведения. Матричные разложения и их использование для численных расчётов. Система наименований подпрограмм LAPACK. Матричные разложения. Процедуры SGESV, SGETRF, SGEEV, SGESVD
3.	Технология OpenMP	Основные сведения. OpenMP и Fortran. Нити и процессы. Параллельные и последовательные области. Параллельные циклы и параллельные области. Автоматическое распараллеливания циклов
4.	Технология MPI	Основные сведения. Способы распараллеливания численных методов. Основные процедуры MPI. Типы данных MPI. Способы передачи сообщений. Прием и передача сообщений процессами

Разделы дисциплин и виды занятий

№ п/п	Наименование раздела дисциплины	Лекц.	Практические занятия и лабораторные работы			СРС	Всего час.
			ПЗ/С	ЛР	Из них в ИФ		
1.	Язык Фортран	-	-	15	15	15	**30**
2.	Библиотека LAPACK	-	-	12	12	14	**26**
3.	Технология OpenMP	-	-	12	12	14	**26**
4.	Технология MPI	-	-	12	12	14	**26**
Итого:		**-**	**-**	**51**	***51***	**57**	**108**

Описание интерактивных занятий

№ п/п	№ р/д	Тема интерактивного занятия	Вид занятия	Труд. (час.)
1	1	Написание программ на языке Fortran	Лаб. раб., выполняемая малой группой (2–3 чел.)	15

2	2	Решение задач линейной алгебры с помощью библиотеки LAPACK	Лаб. раб., выполняемая малой группой (2–3 чел.)	12
3	3	Написание параллельных программ с помощью технологии OpenMP	Лаб. раб., выполняемая малой группой (2–3 чел.)	12
4	4	Написание параллельных программ с помощью технологии MPI	Лаб. раб., выполняемая малой группой (2-3 чел.)	12

Лабораторный практикум

№ п/п	**№ р/д**	**Наименование лабораторных работ**	**Труд. (час.)**
1.	2	Написание программ на языке Fortran	15
2.	2	Решение задач линейной алгебры с помощью библиотеки LAPACK	12
3.	3	Написание параллельных программ с помощью технологии OpenMP	12
4.	4	Написание параллельных программ с помощью технологии MPI	12
Итого:			**51**

Практические занятия (семинары)

Практические занятия (семинары) не предусмотрены.

Примерная тематика курсовых проектов (работ)

Курсовые работы не предусмотрены.

Учебно-методическое и информационное обеспечение дисциплины

а) **Основная литература**

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. — Н. Новгород: ННГУ, 2001.
2. Богачев К.Ю. Основы параллельного программирования. — М.: БИНОМ. Лаборатория знаний, 2003.
3. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002.
4. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. — СПб.: БХВ-Петербург, 2002.

б) **Дополнительная литература**

1. Березин И.С., Жидков И.П. Методы вычислений. — М.: Наука, 1966.
2. Дейтел Г. Введение в операционные системы. Т. 1. — М.: Мир, 1987.
3. Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. — М.: Мир, 1981.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНТО, 1999.
5. Корнеев В.В. Параллельные вычислительные системы. — М.: Нолидж, 1999.
6. Корнеев В.В. Параллельное программирование в MPI. — Москва-Ижевск: Институт компьютерных исследований, 2003.
7. Тихонов А.Н., Самарский А.А. Уравнения математической физики. — М.: Наука, 1977.

в) **Программное обеспечение:**

1. ОС Linux,
2. Компилятор gcc (gfortran),
3. Библиотека LAPACK,
4. Библиотеки MPI, OpenMP.

г) **Базы данных, информационно-справочные и поисковые системы:** не требуются.

Материально-техническое обеспечение дисциплины

Москва, ул. Орджоникидзе, д. 3, корп. 1, 5, лаборатория «Управление инфокоммуникациями»: ауд. 110: комплект жидкокристаллический дисплей Sharp PNL702B, Монитор 24”Acer V243HAOBD, системный блок (процессор Intel Core i7-2600 OEM <3.40GHz, 8Mb, 95W, LGA1155(Sandy Bridge)>, 16GB ОП, HDD 2 TB), ноутбук Toshiba Satellite 17/300GB Intel Core2 2.4 GHz (9 шт.); ауд. 116: проектор DMS800 с интерактивной доской Board

1077, HP xw7800, Intel Core2 2.4 GHz (8 шт.); дисплейные классы ДК3, ДК4, ДК5, ДК6, ДК7, Intel Core i3-550 3.2 GHz – 60 шт.

Методические рекомендации по организации изучения дисциплины

Учебным планом на изучение дисциплины отводится один семестр. Промежуточный контроль знаний предусматривает: проведение проверочной работы в середине семестра, подготовку и сдачу лабораторных работ в течение семестра.

Первый модуль составляют:

- Основные сведения о языке Фортран. История развития. Его преимущества в области научных вычислений по сравнению с другими языками высокого уровня. Структура программы. Типы данных. Встроенные операции и функции. Операторы управления и ветвления. Массивы и работа с ними: описание массивов, задание массивов, динамические массивы, основные функции работы с массивами как матрицами. Ввод и вывод.
- Основные сведения. Матричные разложения и их использование для численных расчётов. Система наименований подпрограмм LAPACK. Матричные разложения. Процедуры SGESV, SGETRF, SGEEV, SGESVD.

Второй модуль составляют:

- Основные сведения. OpenMP и Fortran. Нити и процессы. Параллельные и последовательные области. Параллельные циклы и параллельные области. Автоматическое распараллеливание циклов.
- Основные сведения. Способы распараллеливания численных методов. Основные процедуры MPI. Типы данных MPI. Способы передачи сообщений. Прием и передача сообщений процессами.

Фонды оценочных средств

Примерные тестовые задания

1. Какая из этих архитектур процессора относится к CISC?
(a) x86
(b) ARM
(c) PowerPC
(d) SPARC

2. Какие компьютеры можно отнести к классу MIMD классификации Флина?
(a) Однопроцессорные последовательные системы.
(b) Векторно-конвейерные системы.
(c) Многопроцессорные системы.
(d) Многоядерные системы.

3. Какие компьютеры можно отнести к классу NISD классификации Флина?
(a) Однопроцессорные последовательные системы.
(b) Векторно-конвейерные системы.
(c) Многопроцессорные системы.
(d) Ни один — класс пуст.

4. Какие компьютеры можно отнести к классу SIMD классификации Флина?
(a) Однопроцессорные последовательные системы.
(b) Векторно-конвейерные системы.
(c) Многопроцессорные системы.
(d) Многоядерные системы.

5. Какие компьютеры можно отнести к классу SISD классификации Флина?
(a) Однопроцессорные последовательные системы.
(b) Векторно-конвейерные системы.
(c) Многопроцессорные системы.
(d) Многоядерные системы.

6. Какие параметры оптимизирует процессорная архитектура RISC?
(a) Число циклов на одну инструкцию
(b) Время на один цикл
(c) Число инструкций на задачу

7. Какой класс классификации Флина не был дополнительно разбит на подклассы Вангом и Бриггсом?
(a) SISD
(b) SIMD
(c) MIMD
(d) MISD

8. Какой параметр оптимизирует процессорная архитектура CISC?
(a) Число циклов на одну инструкцию
(b) Время на один цикл
(c) Число инструкций на задачу

9. К какому классу из классификации Флина применяется классификация Джонсона?
(a) SISD
(b) SIMD
(c) MIMD
(d) MISD

10. К какому классу из классификации Флина применяется классификация Хокни?
(a) SISD
(b) SIMD
(c) MIMD
(d) MISD

11. К какому классу по классификации Фенга относится большинство современных компьютеров?
(a) Разрядно-последовательные/пословно-последовательные
(b) Разрядно-параллельные/пословно-последовательные
(c) Разрядно-последовательные/пословно-параллельные
(d) Разрядно-параллельные/пословно-параллельные

12. Конвейерная обработка характеризуется
(a) Загрузкой операндов в векторные регистры
(b) Операциями с векторами
(c) Операциями с матрицами
(d) Выделением отдельных этапов выполнения общей операции

13. Сколько типов вычислительных машин выделяется в классификации Шора?
(a) 3
(b) 4
(c) 5
(d) 6

14. Возможна ли передача параметров программе, написанной на Фортране через командную строку (например, ./prog -e 10 -p 60)? Укажите все подходящие варианты.
(a) Да, с использованием сторонней библиотеки
(b) Да, в стандарте F2003
(c) Нет

15. Выделение памяти для динамического массива предполагает:
(a) Использование специального оператора
(b) Использование указателя на начало массива
(c) Память всегда выделяется динамически
(d) Динамические массивы в F90 не реализованы

16. Какие из вариантов определения массива сработают в gfortran?
(a) real, dim(1:10) :: A[]
(b) real,dimension[1:10] :: A(:)
(c) real, dimension(1:10) :: A
(d) real, dimension(10) :: A

17. Какие из приведённых ниже типов данных не поддерживает Фортран?
(a) Целый (integer)
(b) Вещественный (real)
(c) Текст (text)
(d) Строка (string)

18. Какие из приведённых ниже типов данных не поддерживает Фортран?
(a) Целый (integer)
(b) Вещественный (real)
(c) Текст (text)
(d) Строка (string)

19. Какие из приведённых ниже типов данных не поддерживает Фортран?
(a) Комплексные числа
(b) Вещественный двойной точности
(c) Символьный (character)
(d) Кватернионы

20. Какие из приведённых ниже типов данных поддерживает Фортран?
(a) boolean
(b) bool
(c) logical
(d) int

21. Какие из приведённых ниже типов данных поддерживает Фортран?
(a) list
(b) tuple
(c) logical
(d) int

22. Какой вызов процедуры под названием proc верен?
(a) call proc(1,x)
(b) callproc proc(1,x)
(c) subprogram proc(1,x)
(d) exec proc(1,x)

23. Какой из вариантов определения массива не сработает в gfortran?
(a) real, dimension(1:n) :: x
(b) real :: x(1:n)
(c) real, dimension(n) :: x
(d) real, dimension :: x[1:n]

24. Как расшифровывается название языка «Fortran»?
(a) Formula Translator
(b) Formula Transcoder
(c) Расшифровки не имеет
(d) Назван в честь прозвища Джона Бэкуса

25. Может ли процедура менять значения переменных, переданных ей в качестве аргументов?
(a) Да, если это подпрограмма
(b) Да, если это функция
(c) Нет

26. Может ли функция менять значения переменных, переданных ей в качестве аргументов?
(a) Нет
(b) Да
(c) Устанавливается при определении функции
(d) Зависит от реализации компилятора

27. Типизация, используемая в современном Фортране (при указании директивы ‘implicit none‘)
(a) строгая статическая типизация
(b) строгая динамическая типизация
(c) нестрогая статическая типизация
(d) нестрогая динамическая типизация

28. Фортран это язык...
(a) Основанный на объектно-ориентированном подходе
(b) Не основанный на объектно-ориентированном подходе
(c) Компилируемый в байт-код
(d) Машинного уровня

29. Фортран — это язык...
(a) Компилируемый
(b) Интерпретируемый
(c) Функциональный
(d) Императивный

Перечень тем для контроля знаний

1. Цели и задачи введения параллельной обработки данных.
2. Различие многозадачных, параллельных и распределённых вычислений.
3. Способы построения многопроцессорных вычислительных систем.
4. Виды параллельных вычислительных систем.
5. Классификация параллельных вычислительных систем.
6. Модели параллельных вычислительных систем.
7. Оценка эффективности параллельных вычислений.
8. Уровни распараллеливания вычислений.
9. Расширение существующих языков программирования.
10. Общие способы распараллеливания алгоритмов.
11. Параллельные численные алгоритмы линейной алгебры.
12. Параллельные численные алгоритмы решения дифференциальных уравнений в частных производных.

Календарный план

Неделя	Самостоятельная работа студента	Число часов	Лабораторные занятия	Число часов
1	Цели и задачи введения параллельной обработки данных	**3**	Основные сведения о языке Фортран. Его преимущества в области научных вычислений по сравнению с другими языками высокого уровня	**3**
2	История введения параллелизма	**3**	Работа с массивами. Написание подпрограмм и функций. Работа с вводом/выводом	**3**
3	Различия многозадачных, параллельных и распределённых вычислений	**3**	Примеры программ. Лабораторная работа № 1	**3**
4	Проблема использования параллелизма	**3**	Лабораторная работа № 1. Консультации	**3**
5	Принципы построения параллельных вычислительных систем	**3**	Библиотека LAPACK. Основные сведения. Матричные разложения и их использование для численных расчётов	**3**
6	Классификация МВС	**3**	Система наименований подпрограмм LAPACK. Пример использования. Лабораторная работа № 2	**3**
7	Моделирование и анализ параллельных вычислений	**3**	Лабораторная работа № 2. Консультации	**3**
8	Принципы разработки параллельных алгоритмов и программ	**3**	Технология OpenMP. Основные сведения. Нити и процессы. Способы замера времени исполнения. Примеры	**3**

9	Оценка эффективности параллельных вычислений. Оценка коммуникационной трудности параллельных алгоритмов	**3**	Параллельные секции. Автоматическое распараллеливание циклов	**3**
10	Характеристики топологий сети передачи данных. Алгоритмы маршрутизации. Методы передачи данных.	**3**	Лабораторная работа №3. Консультации	**3**
11	Уровни распараллеливания вычислений. Этапы построения параллельных алгоритмов и программ	**3**	Лабораторная работа № 3. Консультации	**3**
12	Технологические аспекты распараллеливания. Системы разработки параллельных программ	**3**	Технология MPI. Основные сведения. Способы распараллеливания численных методов	**3**
13	Создание специализированных языков программирования. Расширение существующих языков программирования	**3**	Дальнейшие сведения о MPI. Типы данных. Способы передачи сообщений	**3**
14	Стандарт OpenMP	**3**	Примеры программ с использованием MPI	**3**
15	Стандарт MPI	**3**	Лабораторная работа № 4. Консультации	**3**
16	Консультации	**3**	Консультации	**3**
17	Контрольное тестирование	**3**	Консультации	**3**

Балльно-рейтинговая система

Рейтинговая система оценки знаний студентов

Раздел	**Тема**	**Формы контроля уровня освоения ООП**				**Баллы темы**	**Баллы раздела**
		Тест	**ЛР**	**ИЗ**	**Экзамен**		
Язык программирования Fortran и Библиотека LAPACK	Fortran	7,5	15	2,5	-	25	**25**
Язык программирования Fortran и Библиотека LAPACK	LAPACK	7,5	15	2,5	-	25	**25**
Технологии параллельного программирования	OpenMP	-	15	2,5	7,5	25	**25**
Технологии параллельного программирования	MPI	-	15	2,5	7,5	25	**25**
Итого	-	15	60	10	15	-	**100**

Таблица соответствия баллов и оценок

Баллы БРС	Традиционные оценки РФ	Оценки ECTS
95–100	5	A
86–94		B
69–85	4	C
61–68	3	D
51–60		E
31–50	2	FX
0–30		F
31–51	Зачет	Passed

Правила применения БРС

1. Раздел (тема) учебной дисциплины считаются освоенными, если студент набрал более 50 % от возможного числа баллов по этому разделу (теме).
2. Студент не может быть аттестован по дисциплине, если он не освоил все темы и разделы дисциплины, указанные в сводной оценочной таблице дисциплины.
3. По решению преподавателя и с согласия студентов, не освоивших отдельные разделы (темы) изучаемой дисциплины, в течение учебного семестра могут быть повторно проведены мероприятия текущего контроля успеваемости или выданы дополнительные учебные задания по этим темам или разделам. При этом студентам за данную работу засчитывается минимально возможный положительный балл (51 % от максимального балла).
4. При выполнении студентом дополнительных учебных заданий или повторного прохождения мероприятий текущего контроля полученные им баллы засчитываются за конкретные темы. Итоговая сумма баллов не может превышать максимальное количество баллов, установленное по данным темам (в соответствии с приказом Ректора № 564 от 20.06.2013). По решению преподавателя предыдущие баллы, полученные студентом по учебным заданиям, могут быть аннулированы.
5. График проведения мероприятий текущего контроля успеваемости формируется в соответствии с календарным планом курса. Студенты обязаны сдавать все задания в сроки, установленные преподавателем.
6. Время, которое отводится студенту на выполнение мероприятий текущего контроля успеваемости, устанавливается преподавателем. По завершении отведённого времени студент должен сдать работу преподавателю, вне зависимости от того, завершена она или нет.
7. Использование источников (в том числе конспектов лекций и лабораторных работ) во время выполнения контрольных мероприятий возможно только с разрешения преподавателя.
8. Отсрочка в прохождении мероприятий текущего контроля успеваемости считается уважительной только в случае болезни студента, что подтверждается наличием у него медицинской справки, заверенной круглой печатью в поликлинике № 25, предоставляемой преподавателю не позднее двух недель после выздоровления. В этом случае выполнение контрольных мероприятий осуществляется после выздоровления студента в срок, назначенный преподавателем. В противном случае отсутствие студента на контрольном мероприятии признается не уважительным.
9. Если в итоге за семестр студент получил менее 31 балла, то ему выставляется оценка F и студент должен повторить эту дисциплину в установленном порядке. Если же в итоге студент получил 31–50 баллов, т. е. FX, то студенту разрешается добор необходимого (до 51) количества баллов путем повторного одноразового выполнения предусмотренных контрольных мероприятий, при этом по усмотрению преподавателя аннулируются соответствующие предыдущие результаты. Ликвидация задолженностей проводится в период с 07.02 по 28.02 (с 07.09 по 28.09) по согласованию с деканатом.

Сведения об авторах

Геворкян Мигран Нельсонович — кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Королькова Анна Владиславовна — кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Кулябов Дмитрий Сергеевич — кандидат физико-математических наук, доцент, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Учебное издание

Мигран Нельсонович Геворкян
Анна Владиславовна Королькова
Дмитрий Сергеевич Кулябов

Параллельное программирование

Тематический план изданий учебной и научной литературы
2014 г., №18

Редактор *И.Л. Панкратова*
Технический редактор *Н. А. Ясько*
Компьютерная вёрстка *М. Н. Геворкян*, *А. В. Королькова*, *Д. С. Кулябов*
Дизайн обложки *Н. А. Ясько*

Подписано в печать 20.10.2014 г. Формат 60×84/16. Печать офсетная.
Усл. печ. л. 5,5. Тираж 500 экз. Заказ № 1428.

Российский университет дружбы народов
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3

Типография РУДН
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3, тел. 952-04-41