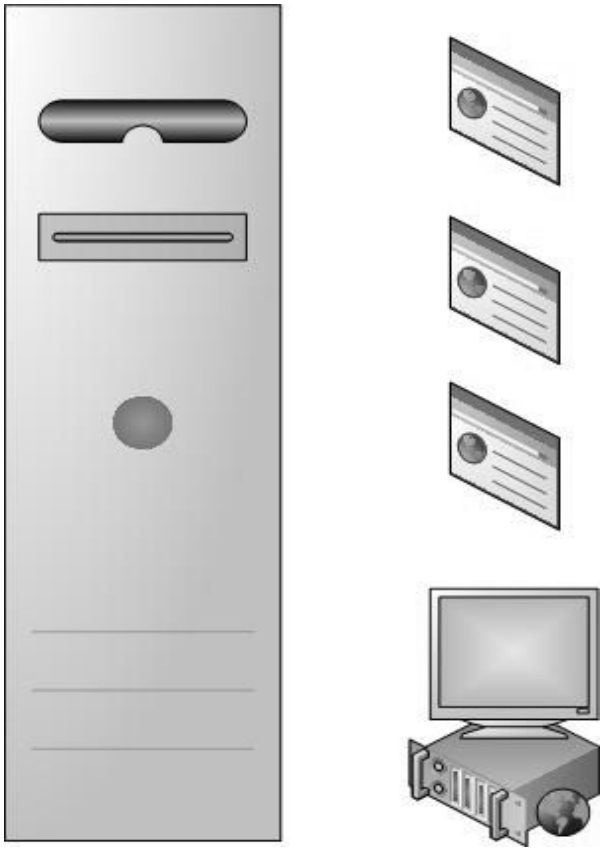
В.Ф. Звягин, С.В. Федоров

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ В ОПТИКЕ И ОПТОИНФОРМАТИКЕ



Санкт-Петербург

2009

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ПРИОРИТЕТНЫЕ НАЦИОНАЛЬНЫЕ ПРОЕКТЫ «Образование»

ПОБЕДИТЕЛЬ КОНКУРСА ИННОВАЦИОННЫХ ОБРАЗОВАТЕЛЬНЫХ ПРОГРАММ ВУЗОВ

В.Ф. Звягин, С.В. Федоров

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ В ОПТИКЕ И ОПТОИНФОРМАТИКЕ

Учебное пособие

ИТМО
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Санкт-Петербург

2009

В.Ф. Звягин, С.В. Фёдоров, Параллельные вычисления в оптике и оптоинформатике: Учебное пособие. – СПб: СПбГУ ИТМО, 2009. – 109с.

Аннотация Параллельные высокопроизводительные вычисления можно организовать по-разному: либо на основе автоматической параллелизации программ на Фортран-95, либо распараллеливая программы вручную. Изложены новейшие концепции Фортрана-95, обеспечивающие автоматическую параллелизацию. Рассмотрены два подхода к распараллеливанию: либо директивы OMP, либо вызовы функций библиотеки MPI. Типовые численные методы предметной области доведены до программной реализации. Задания исследовательского характера, требуют измерения времени последовательных и параллельных вычислений и построения графиков. Предусмотрены задания для самостоятельного решения в виртуальных лабораториях.

Пособие предназначено для студентов, специализирующихся в области фотоники и оптоинформатики, а также в оптике и других областях, требующих проведения высокопроизводительных вычислений.
200600.68.02 - Оптические технологии передачи, записи и обработки информации (Магистр техники и технологии)
Рекомендовано к печати ученым советом факультета фотоники и оптоинформатики СПбГУ ИТМО 18.02.09, протокол № 5

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007-2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

Введение

До недавнего времени и техника и среды проектирования Программного обеспечения поддерживали чисто последовательные вычисления. Ситуация меняется. Появилась потребность в параллельных вычислениях, и, действительно, во многих задачах не обязательно расчеты выполнять строго последовательно. Логика задачи часто позволяет выполнять расчеты параллельно-последовательно, что ускоряет решение. Когда для краткости говорят о "*параллельном* программировании", то, конечно, на самом деле суть - в организации параллельно-последовательных вычислений. Чисто последовательные или чисто параллельные вычисления – это упрощенный подход к вычислениям.

Имеется несколько подходов к организации параллельных вычислений

- одна программа - одни данные, случай обычных последовательных вычислений (на этот подход ориентирован Фортран до стандарта – Фортран-95), по-английски SISD – single instructions –single data
- одна программа - много данных, случай, наиболее часто встречающийся на практике (на этот подход сориентирован Фортран-95 и выше), по-английски SIMD – Single Instructions – Multiple Data
- много программ – одни данные, по-английски MISD – multiple instructions – single data
- много данных – много программ, по-английски MIMD – multiple instructions – multiple data

Далее по тексту будем различать 3 термина

- параллельное программирование *на* Фортране-95 с автоматической *параллелизацией*
- *распараллеливание* – технология, подразумевающая написание последовательной программы с последующим добавлением строк кода на специальном языке (либо OMP, либо MPI), превращающем последовательную программу в *параллельную*
- *параллельное приложение* – программа, которая использует несколько процессоров впараллель, при этом неважно каким путем это достигнуто (параллелизация или распараллеливание)

В Фортран-95 мы задачу не распараллеливаем, а просто решаем не так, как в других языках. Языковые средства позволяют обобщить проблему, описав задачу на более высоком уровне – векторы, матрицы и действия над ними. Действия над компонентами *независимы* и в языке порядок их выполнения не декларируется. На основе сведений о независимости компиляторы поступают по-разному:

- Intel IFC-9,10,11 проведёт автоматическую *параллелизацию;*

- не пройдет никакой *параллелизации* в IFC-5.0, MS FPS-4.0, Compaq-6.0.

Параллельное приложение будет выполняться:

- с параллельной обработкой компонент, если процессоров 2 и более;
- чисто последовательно, если процессор единственный.

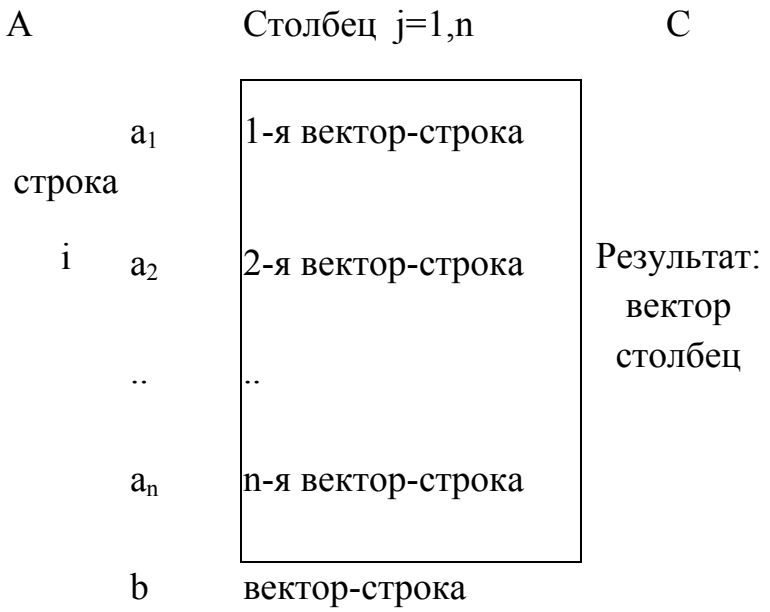
Для выполнения параллельных расчетов нужна поддержка:

- *в языках программирования*, как-то:
  - *Фортран-95* - единственный язык программирования, непосредственно обеспечивающий поддержку параллельно-последовательных расчетов;
  - для других языков остается возможность внедрить директивы препроцессора (OMP), или вызовы библиотеки (MPI) в исходный код, чтобы распараллелить вычисления;
- *в средах проектирования*, как-то:
  - *Intel Fortran Compiler* – IFC9, IFC10 – уникальный компилятор, с автоматическим распараллеливанием программ на *Фортран-95;*
  - MPI, OMP – библиотека и технология распараллеливания последовательной программы на C, Java, Фортран.
- *аппаратная* - в современных компьютерах частично уже есть аппаратная поддержка, её перечень от простого к сложному:
  - ПК с 2 ядрами в одном процессоре Intel Core Duo;
  - ПК с 4 ядрами в одной микросхеме Quad Intel Core 2;
  - ПК с 8 ядрами в двух процессорах на материнской плате ;
  - *кластер* из нескольких одинаковых компьютеров, связанных локальной сетью, под управлением центрального компьютера;
  - *суперсервер* - кластер из нескольких одинаковых процессоров, связанных внутренней локальной скоростной сетью, под управлением процессора, назначенного центральным (это вариант кафедры ФиОИ);
  - *суперкомпьютер*, в котором, не вдаваясь в детали, еще больше реализовано аппаратно (см. Top 500 и Top 50 России и СНГ).

Глава 1. Что такое параллельное программирование

Принципы параллельного программирования на Фортране

Принципы параллельного программирования на Фортране разъясняются на характерном примере умножения квадратной матрицы на вектор-строку $c_i = \sum_j A_{ij} b_j$ где A-квадратная матрица *n* на *n*, *b* - вектор-строка из *n* элементов, *c* - вектор-столбец из *n* элементов



Составим схему вычислений с объёмом вычислений, пропорциональным n^2 - квадрату размерности матрицы A

j=1,n	**A*b**	**Процесс – скалярное произведение векторов**	**C -вектор-столбец результат**
вектор-строка	a_1 b	$c_1 = \sum_j A_{1j} b_j$	$c_1 = a_1$ b
вектор-строка	a_2 b	$c_2 = \sum_j A_{2j} b_j$	$c_2 = a_2$ b
вектор-строка	a_n b	$c_n = \sum_j A_{nj} b_j$	$c_n = a_n$ b

Для распараллеливания по данным необходимо,

- чтоб данные параллельных процессов были независимы ba_1 ba_2 .. ba_n;
- чтоб некоторые общие данные были продублированы;
  (см. вектор-строка b в примере) - идет обычный размен памяти на быстродействие;
- чтоб результаты вычислялись независимо для c_1 c_2 .. c_n;
- чтоб процессоров имелось достаточное количество - в теории *n*, при нехватке CPU используется реальное количество CPU, но при этом приходится частично жертвовать параллельностью;
- чтоб программа для всех CPU была одна и та же (в примере - скалярное произведение векторов), а данные разные;
- чтоб параллельные процессы были бы синхронизированы, то есть результат вектор-столбец *c* нельзя использовать в дальнейших

вычислениях, пока не завершились все запущенные параллельные процессы;
- чтоб в расчетах на распараллеленных участках не было ввода-вывода.

Пример программы на Фортране для сравнения параллельных и последовательных вычислений

Составим программу на Фортране, которая решает эту задачу. Сама по себе задачка несложная. Текст программы приводится полностью по нескольким причинам, а именно:
- чтоб иметь готовую программу для экспериментов;
- чтоб сравниваться с программой на "С", решающей ту же задачу;
- для первоначального ознакомления с организацией программы на Фортране, её компиляцией и выполнением;
- чтоб почувствовать суть параллелизации, для чего размерности матриц варьируются от n=10 вплоть до экстремального значения n~14000 - нас будет интересовать не значения, полученные в результате умножения, а время его выполнения в зависимости от размерности задачи.

Задача подразделена на следующие 3 программные единицы (ПЕ):
- *главная программа* `program matamula`
  - для матриц задает последовательно 17 размерностей *n*=`Dima(k)` из `10,100,500, 1000,2000.. 14000;`
  - для каждой из размерностей *n* в цикле вызывает процедуру `seqvcalc(n),` получая от неё время умножения `Timer(k);`
  - выводит таблицу с двумя колонками "время в зависимости от размерности *n* " ;
- *процедура* `seqvcalc`
  - по заданному *n* динамически наделяет памятью массивы *Vec*`(1:n)`, *Matr*`(1:n,1:n)`, *Prod*`(1:n);`
  - заполняет числами *Vec, Matr ;*
  - пользуясь стандартной программой *matamul* *Prod = matamul(Matr,Vec)*, умножает матрицу *Matr* на вектор *Vec,* делая отсечку интервала времени функцией *tima()* и сохраняя его в `Timer`;
  - освобождает память для повторного использования;
- *функция* `tima()` опрашивает текущее время суток до соток секунды
  - интерфейс к измерителю времени *gettim( )* прописан в модуле, специфичным для каждого компилятора;
    - либо IFPORT *у фирмы INTEL* для IFC10;

- либо MSFLIB *у фирмы* MS для FPS4.0; поскольку на модуль MSFLIB программы настроены по умолчанию, то его можно и не использовать;

- Вы можете опрашивать процессорное время, вызвав другой таймер, см. помощь по F1;
- если параллельно Вашей программе кто-то работает, то он съедает ресурсы, - Ваши измерения будут искажены – поэтому даже мышку нельзя трогать!

Эта программа из 35 строчек выглядит таким образом.

```
program matamula   ! главная программа
 implicit none
  integer,parameter::NofDima=17! всего исследуется 17 размерностей
   integer :: k
    integer,dimension(1:NofDima):: &
! размерности матриц от n=10 до n=14000
Dima=(/10,100,500,(k,k=1000,(NofDima-3)*1000,1000)/)
     real ,dimension(1:NofDima):: Timer
 open(6,file="timelooses.txt")
 do k=1,NofDima ! цикл по размерностям
   call seqvcalc(Dima(k))
 enddo
 write(6,11)( Dima(k),Timer(k),k=1,NofDima)
11 format ("время в зависимости от размерности n "/(i7,f8.2))
                  contains
subroutine seqvcalc(n) ! вычислительно-измерительная процедура
implicit none
integer,intent(in)::n
real*8,dimension(1:n,1:n)::Matr !автоматич. динамические массивы
    real*8,dimension(1:n):: Vec,Prod ! массивы
     real startTimer
      real*8 :: x=0.73
 Matr=-sin(2*x); Vec=cos(-x/2)  ! заполнение Vec,Matr
 startTimer=tima()              ! стартовое время
  Prod=matmul(Matr,Vec)         ! умножение Matr на Vec
 Timer(k)=tima()-startTimer     ! интервал времени
end subroutine seqvcalc

end program matamula

 real function tima(  )   ! таймер
  USE IFPORT !для FPS4.0 – эту строчку надо закомментировать, а для IFC10-открыть
   implicit none
    INTEGER*2 our,minut,sec,100
```

```
  call GETTIM(our,minut,sec,h100)
  tima=our*3600+minut*60+sec+h100*0.01
end function tima
```

Эксперименты с программой на Фортране

Написанная программа экспериментальна и экстремальна – она ставит ПК на грань работоспособности: когда будет исчерпана физическая память, Windows доберётся до виртуальной памяти и Ваш ПК превратится в паровоз с КПД 13-15% , у которого весь пар уйдёт в гудок, а именно:

- винчестер будет гудеть;
- а расчет будет стоять.

Программу скомпилировали дважды практически безо всяких изменений:

1. на новом IFC 10.1 для *параллельно-последовательных* вычислений с автоматической параллелизацией;
2. на старом MS FPS4.0 для *последовательных* вычислений.

Вызвав *диспетчер задач* по Alt^Ctrl^Del, мы выполнили программы в указанном порядке, на 2-х ядерном ПК с Core Duo. Снимок экрана по *PrintScreen* сохранен как рисунок

Рис. 1. Скриншот выполнения параллельной и последовательной программ на 2-х ядерном ПК.

Сравним по рисунку поведение двух программ:

- сверху - хронология загрузки 2 процессоров, видно, что она разная в 2 программах:
  1. узкие пики – *параллельные вычисления*: равномерно до 91% загружены оба CPU;
  2. широкие участки - *последовательные вычисления*: оба CPU загружены, но с перекосом и всего лишь на 50%;
- внизу справа - хронология использования файла подкачки, видно, что память нарастает с разной скоростью и до разного уровня:
  1. *параллельно*: память нарастает быстрее и до уровня, чуть более высокого;
  2. *последовательно*: память нарастает медленнее;
- по нижнему рисунку измерим времена исполнения программ по временной развертке - просто по клеточкам, как бы по длительности импульса:
  - 2 единицы для *параллельно-последовательных* вычислений ;
  - 19 единиц для *последовательных* вычислений;
  - отметим не 2, а 10-кратное увеличение быстродействия?

Знакомство с компиляторами в экстремальных вычислениях

Знакомясь с компиляторами в экстремальных вычислениях, следует иметь в виду следующее:

- `n` - размерность матрицы в программе умножения матрицы на вектор, максимум 14000 и зависит от NofDima=17, а что подойдет для Вашего ПК ;
- следует подобрать *NofDima*, взяв его больше-меньше, оценив предельную размерность матрицы в расчетах на конкретном ПК (проведите эксперименты *дома, в классе,* на Core Duo, *на кластере*);
- каждый ПК “умирает” по-своему с ростом `n` - понаблюдайте как, а причина банальна – мучения с виртуальной памятью, зависящей от наличной физической RAM;
- для сравнения *параллельного и последовательного* расчета по *PrintScreen* сделайте снимок с экрана *диспетчера задач* и сохраните как рисунок*;* затем по таблице постройте графики зависимости времени расчета от размерности задачи `n`.

Сравнение программ умножения матрицы на вектор на Фортране и Си

Написаны 3 принципиально разные программы. Сравним их по числу строк кода:

1. 35 строк – рассмотренная выше универсальная фортрановская программа;
2. 116 строк - *последовательная* программа на Си, заимствованная в Нижегородском университете;
3. 279 строк - *параллельная* программа на Си, заимствованная в Нижегородском университете.

Конечно, различия по объёму заметны, но если присмотреться к составу 3 программ, то становится понятна и причина столь заметных отличий:

- Программа на Фортране универсальна для обоих способов вычислений, и ее можно просто по-разному скомпилировать;
- *на С нельзя создать* универсальную программу по отношению к 2 способам вычислений;
- причина - на С вообще *нельзя* создать программу для параллельных вычислений;
- вскрывается “обман”: из 279 строк состоит не параллельная, а *последовательная* программа на С, в которую добавлены команды *на другом языке* - на MPI, который как раз и умеет распараллеливать .

Если программировать на Фортране-95, то компилятор IFC извлечет сведения для параллелизации из программы, и в большинстве случаев этого будет достаточно, чтобы автоматизировать распараллеливание.

- Для любой ли программы на Фортране можно эффективно провести параллелизацию?

- Нет, конечно, - только если Вы ее написали соответственно.

- А как это?

- Вот это и есть предмет изучения в нашем курсе!

Конечно, можно распараллеливать вручную с MPI или OMP, но это уже в особых случаях.

Если программировать на С, то в нём *нет никаких сведений для* параллелизации, и нет другого выхода как изучать и применять MPI или OMP. Отметим, что программу на С придётся не только снабдить

директивами распараллеливания, но и изменить для целей распараллеливания. MPI-директивы построены как вызовы подпрограмм. Команды языка управляют распараллеливанием. MPI достаточно хорошо продуман и позволяет препарировать обычную программу, выделяя в ней параллельные и последовательные *участки. Команды MPI россыпью внедряют* в программу на обычным языке программирования. Таким обычным языком может быть Фортран, C или Java. Создание параллельного приложения - дело тонкое, поэтому, конечно, даже новый Intel Фортран со своим передовым компилятором и высоким уровнем автоматизации не могут похвастаться исчерпывающим набором средств параллелизации. Пакеты MPI и конкурирующий пакет OMP дают низкоуровневые средства распараллеливания. Технология работы такова:

- программист изучает MPI;
- сам вручную реализует логику *параллельно-последовательных* вычислений;
- отлаживает свою программу + логику *параллельно-последовательных* вычислений;
- если долго мучиться что-нибудь получится.

Глава 2. Fortran - язык вычислений в науке и технике

Здесь не ставится задача полного описания Фортрана; даётся его описание с точки зрения организации *параллельно-последовательных* вычислений. Далее изложим Фортран в целом кратко, а в том, что касается *параллельно-последовательных* вычислений – подробнее.

Если Вы внимательно познакомитесь с Фортраном:

- то будете озадачены – ни слова о параллельном программировании;
- тем не менее, Вы не будете разочарованы – есть всё для параллельного программирования;
- тут нет парадокса, в Фортране говорится о независимости вычислений:
  - независимость можно трактовать и как *параллельность* вычислений;
  - независимость можно трактовать и как *последовательность* вычислений;
  - всё будет зависеть от техники:
    - техники компиляции;
    - техники - в смысле многих CPU.

Историческая справка

Программы на ***ForTran*** = ***For***mula***Tran***slator стали писать так давно, что тогда их вводили с перфокарт, откуда и пошли правила построчного оформления операторов, тогда как в C и Паскале - это лишь традиция, а не

правила языка. В этом смысле поначалу Фортран не привычен, но к хорошему быстро привыкают. Этот язык могли изучать еще Ваши родители и их учителя. Я впервые его изучал, как и Вы, на 5-ом курсе. Что интересно, я и сейчас продолжаю его изучать, потому что это единственный язык, который всё живет и живёт, потому что всё изменяется и изменяется. Расхожее мнение среди пижонов: “Фортран - старый язык, Фортран – примитивный язык”– всё это в корне верно. Действительно, его история - самая внушительная. Признано, что его простота, а можно сказать и примитивность – это залог жизнеспособности. Развитие языка органично сочетается с преемственностью и стандартизацией языка:

- *1954* – впервые стали писать формулы для ЭВМ, это было на языке Фортран;
- *1966* – 1-ый *стандарт* языка;
- *1977* – этим *стандартом* узаконен стиль структурного программирования, выпущен лучший компилятор *MS Fortran 5.1* для DOS;
- *1990 – стандарт* f90 внес самые радикальные изменения:
  - свободная форма текста программы;
  - конформные параллельные вычисления, секции массивов
  - функции для параллельных вычислений;
  - модули, 3 способа передачи данных между программами проекта;
  - выпущен лучший для этого стандарта компилятор *MS Fortran Power Station 4.0* в составе интегрированной (Studio+Compiler+Help) среды проектирования Developer Studio для Windows ;
  - следующая версия этого компилятора - *Digital Visual Fortran 5.0* фирмы DEC уже разрознена MS Visual Studio / Compiler / Help;
- *1995* – секционированные вычисления **forall** и ветвления **where** наряду со скалярными циклами **do** и **if,** отметим, что компилятор *Compaq Visual Fortran 6.0* содержит в розницу MS *Visual* Studio / Compiler / Help;
- *2003* – обработка исключений, классы, объектно-ориентированное программирование, упрощение смешанного программирования (с языком “C”);
- *2008* – ***Intel Fortran Compiler 10*** (ifc10) по стандарту F03 с автоматизацией распараллеливания в составе Compiler / Help из Internet, ifc10 запускают из командной строки или интегрируют в *MS Visual* Studio 2005.

Оформление программы

Оформление программы - тут ряд аспектов для знакомства:

- построчное оформление простых и составных операторов, для программных единиц;
- построение проекта из программных единиц (ПЕ);
- обмен данными между ПЕ;
- модули – новинка Fortran-90.

Построчное оформление программы **.f90* в *свободной* форме

Построчное оформление программы **.f90* в *свободной* форме, характеризуется следующим:

- пустые строки и пробелы улучшают читаемость текста;
- "**!**" открывает полнострочный комментарий или комментарий после оператора;
- *простой* оператор занимает одну строку;
- несколько коротеньких простых операторов размещают в одной *видимой* строке, используя *невидимый* конец строки "**;**" вместо <Enter>;
- *невидимый* конец строки "**;**"не используют при оформлении *составного* оператора, который состоит, как минимум, из 3 *видимых* строк,

  заголовок
  .. *начинка..*
  end ***заголовок***
- блок – часть составного оператора, между строками с *ключевыми* словами, например,

  if() ***then***
  .. блок да..
  else
  .. блок нет..
  endif
- длинные строки (но не комментарии), можно переносить пословно, как в обычном языке, используя знак переноса "**&**" ;
- малые и большие буквы *не различаются* в именах и *ключевых* словах (не как в "С");
- в Фортране имеются *ключевые* слова, но они *не зарезервированы* в отличие от "С";

- не допускаются двусмысленности между ключевыми словами и объектами программы
  integer :: max,i1,i2,i3 ! max-переменная
  i1=***max***(i2,i3) ! ***max-*** стандартная функция;
- не стоит писать, но не вызывает двусмысленности у компилятора
  integer :: integer=73 ! ***integer***-целый тип, integer-целая переменная.

Структура проекта, подпрограммы, функции, модули, интерфейсы

Управляя сложностью решения задачи, проект делят на части, организуя обмен данными между ними.
Терминология и принятые сокращения:

1. Программу составляют из *Программных Единиц*, ПЕ – наименьшая самостоятельно компилируемая часть текста решения задачи ;
2. Проект может включать: единственную *главную программу* - ***program*** и в любом количестве ***module***-*модули* и подпрограммы (ПП) вида ***subroutine***-*процедуры,* ***function***-*функции;*
3. *Вызывающими* ПЕ могут быть главная программа или ПП
   - *вызываемую* функцию, наподобие математического F(x,y..), упоминают в выражении, куда и возвращается её единственный ответ
   - процедуру *вызывают* оператором ***call*** *имя*_процедуры(x,y..)
   - после имени *вызываемой* ПП перечисляют *фактические* параметры x,y.., которые являются объектами вызывающей ПЕ;
4. *Формальные* параметры в заголовке вызываемой ПП – это её объекты
   видПП имярек(*формальные* параметры) ! видПП- это ***subroutine*** или ***function***
   .. *описание* функции (если *видПП* - *функция*)**..**
   .. *описания* формальных параметров**..**
   .. *описания* **..** ! не включают в интерфейс
   .. *действия* **..** ! не включают в интерфейс
   end видПП имярек
5. *Фактические* параметры *вызывающей* ПЕ должны быть попарно сопоставлены *формальным* параметрам вызываемой ПП и согласованы по типу, по форме и по предназначению ***intent***(*in*)-входной, ***intent***(*out*)-выходной, ***intent***(*inout*)-входной/выходной. *Формальные* параметры могут быть заявлены как необязательные (***optional***);
6. Наряду с *позиционными* параметрами при явно описанном интерфейсе можно пользоваться *ключевыми* параметрами. *Векторные* и *элементные* функции, как и механизм перегрузки

также требуют описания явного интерфейса: *Interface .. .. end Interface* – составной оператор *вызывающей* ПЕ, описывающий шаблон вызова ПП. Если из ПП (см. п.4) удалить мелкое *описания* и *действия* как раз получится начинка **.. ..** интерфейса к ПП.;

7. *Модули* отличаются тем, что их нельзя вызывать (в них нет действий), но можно использовать их описания через ***use*** напрямую или по цепочке модулей, но без тавтологии (рекурсии);
8. Вызовы программ могут осуществляться по цепочке, допустима *рекурсия,* т.е. вызов самое себя.

Проект состоит из самостоятельных *программных единиц* (ПЕ), как-то:

- ***program*** имярек ! главную программу загружают средствами ОС
  .. описания **..**
  .. действия **..**
  end program имярек
- ***subroutine*** имярек(формальные параметры) ! *процедуру* вызывают ***call***
  .. *описания* параметров**..** ! *по назначению* ***in***-входы ***out***-выходы ***inout***: входы/выходы
  .. *описания* **..**
  .. *действия* **..**
  end subroutine имярек
- ***function*** имярек(формальные параметры) ! упоминают в выражении под *имярек*
  .. *описание* функции .. ! *по назначению* ***in***-только входы
  .. *описания* параметров**..**
  .. *описания* **..**
  .. *действия***,** включая *имярек*= **..** ! ответ связан с *имярек*
  end function имярек
- ***module*** имярек ! *модуль используют,* ***use*** *имярек* - первое из описаний
  .. *описания* **..**
  [***contains***
  ..модульные ПП..]
  end module имярек

К числу вспомогательных операторов относят перечисленные в таблице.

оператор	**назначение**	**Комментарии**
End*	конец конструкции ПЕ Единственный в ПЕ	нормальное завершение в ***Program***-задачи в ***Subroutine***, ***Function*** – подзадачи
Stop	досрочное, часто аварийное, завершение задачи	в любых ПЕ в любом количестве

оператор	назначение	Комментарии
Return	досрочное завершение ПП	в любом количестве в ПП
Pause	пауза до нажатия *Enter*	Аналог ***read***(*,*)
Interface ***.. ..*** ***end Interface***	составной оператор описания шаблона ПП	явное описание правил вызова ПП на Фортране или на другом языке
F(x,y..)=*выражение*	Определение *операторной* функции	Вызов только в ПЕ, где дано определение
Include**	включить описания и определения из файла в текст ПЕ	директива среди описаний
Use имяРек	использовать модуль имяРек	до других описаний
Call имяРек(..A)	оператор вызова ПП	вызвать процедуру имяРек(A)
имяРек(..A)	упоминание в формуле	вызов функции в формуле
***intent*(**in**)** ***intent*(**out**)** ***intent*(**inout**)**	***intent*** - атрибут *предназначения* параметра ПП	*(in) Входной* *(out) выходной* *(inout) входной-выходной*
external ***intrinsic***	имена ПП, используемых как аргументы при вызове ПП	***external*** **–** внешние ПП ***intrinsic*** - внутренние функции Ф90
Optional	атрибут или оператор ПП	атрибут необязательного параметра
Public* / *private	глобальные/локальные объекты модуля	атрибут или оператор модуля
common**	общие блоки памяти	Оператор описания

Примечание (*) вид программы и имя программы можно повторить в операторе ***end***; для модуля, модульных и внутренних программ – это делать обязательно.

Примечание (**) оператор, объявленный устаревшим в Ф90

Ф*ункция* и *процедура* различаются следующими свойствами:

- по способу вызова: функцию - упоминают в выражении, процедуру вызывают ***call;***
- по способу возврата ответа и количеству ответов:
  - у функции единственный ответ, возвращаемый в формулу, где её упоминают;
  - ответы процедуры вернутся через фактические параметры, сопоставленные формальным *выходным* (*out*, *inout*) параметрам;
- по атрибутам:
  - функция имеет тип, форму и имя возвращаемого значения (по нему и вызов);
  - процедура имеет только имя для вызова;
- назначением параметров;

- все аргументы функции – только входные (*in*);
- аргументы процедуры разнообразны *in*-вход, *out*-выход, *inout*-вход/выход.

Помимо ПП-функций в ходу *встроенные* и *операторные* функции.

В Фортран-90 программные единицы по взаиморасположению стали подразделять на *внешние* и *внутренние, при этом:*

- ***program*** и ***module*** могут быть только *внешними;*
- *подпрограммы* ***subroutine*** и ***function*** могут быть и *внешними* и *внутренними.*

Внешняя ПЕ выглядит так:

Заголовок внешней ПЕ

... .. !текст внешней ПЕ, а после

contains

внутренние ПЕ *! вкладывают пачкой* внутрь внешней ПЕ перед

end *внешней* ПЕ

Допускается *3 уровня вложения* для модуля: 1) модуль, 2) модульная ПП, 3) *внутренняя* ПП.

Допускается *2 уровня вложения* для других видов ПЕ:

1) внешняя ПЕ, 2)*внутренняя* ПП

Область видимости объекта – это ПЕ, где над объектом можно выполнять действия, перечислим свойства *области видимости*:

- в область видимости включаются объекты, описанные в рассматриваемой ПЕ ;
- если ПП является внутренней, то из внешней ПЕ-*носителя* данных к области видимости присоединяются поимённо объекты, которых нет во внутренней ПП;
- если ПЕ использует модуль, то к области видимости поимённо присоединяются объекты модуля-носителя данных, коллизии имён не допускается;
- используемый модуль в свою очередь мог присоединить объекты используемых в нём модулей и т.д., но рекурсии не допускается;
- локальные объекты ПЕ вида ***subroutine*** и ***function***, не попавшие в область видимости других ПЕ, считаются приватными, не видимыми извне;
- объекты модуля по умолчанию считаются публичными ***public***-объектами, видимыми извне по использованию, приватные ***private***-объекты модуля надо описывать.

Модуль – это контейнер описаний глобальных объектов и *модульных* ПП для их обработки. Модуль в отличие от других ПЕ не вызывают (в нём нет команд), а используют. Принцип *инкапсуляции* - сокрытие глобальных объектов в модульных ПП – это основное предназначение модуля в

суперпроектах. Начинающему программисту не просто сделать обоснованный выбор, вида ПЕ. На примере факториала, см. разные виды ПЕ - примеры 4.1, 4.2, 4.3, 4.4.

Пример 4. Вычислить факториал n! = 1*2*3*.. *n . Из математики известен особый случай: 0!=1. Иллюстрируется реализации одной и той же задачи вычисления факториала, как программ разного вида.

Пример 4.1. Подпрограмма-функция *Factorial*.

```
Function Factorial ( n ) ! заголовок подпрограммы-функции
  real*8 Factorial ! тип результата
  integer,intent(in) :: n ! единственный входной параметр
  integer k ! при умножении автоматически переводится а двойную точность
 Factorial = 1
 if( n == 0 ) return  ! из математики известен особый случай: 0!=1
    do k=1,n
      Factorial = Factorial *k
    enddo
end Function Factorial
```

Пример 4.2. Функцию *Factorial*, можно определить по-другому, используя стандартную функцию product, зависящую от вектора

```
Function Factorial ( n ) ! заголовок подпрограммы-функции
 Integer k
 real*8 Factorial  ! тип результата
if ( n > 0 ) then
   Factorial =product( (/ ( dble(k), k=1,n) /) )
        ! dble(k) преобразует тип аргумента в вещественный двойной точности
        ! ( dble(k), k=1,n ) - неявный цикл
        ! (/ ( dble(k), k=1,n) /) - конструктор массива
        ! product( (/ ( dble(k), k=1,n) /) ) - векторная функция, вычисляющая произведение
Else
        Factorial = 1
 Endif
End Function Factorial
```

Пример 4.3. Функцию *Factorial* можно оформить не только как самостоятельную ПЕ, но и как *операторную* функцию, объявленную в составе ПП ***Loto***. Изменилось определение функции *Factorial*, но не изменился вызов функции *Factorial.*

```
Subroutine Loto ( m,n, S,P ) ! заголовок подпрограммы Loto
 Integer,intent(in) :: m, n ! (in) - входные параметры
  real,intent(out) ::  S,P  ! (out) - выходные параметры
   Integer k
! это описание используется в  определении операторной функции Factorial
     real*8 Factorial  !  тип (dble(k)) => тип(product) => тип(Factorial)
```

```
      ! операторная функция в составе ПП Loto
      Factorial(n)=product((/(dble(k),k=1,max(1,n)) /))
          !     при n=0 получим max(1,0)=1,
          ! но при n=1 и при n>1 получим max(1,n)=n
   S=Factorial(n)/(Factorial(m)*Factorial(n-m))
   P = 1 / S
 End Subroutine Loto
```

Пример 4.4. Реализует вычисление факториала *из учебного проекта* вообще не в виде функции, а в виде процедуры

- не изменился способ вычисления факториала и имя входного параметра *n*
- изменился вид ПП с function на Subroutine и ее имя с *Factorial* на pro*Fact,* изменился интерфейс
- подпрограмму вида Subroutine по имени pro*Fact* только вызывают, но с ее именем не связывают результатов
- тот же результат *Factorial* стал выходным параметром вместо имени функции
- в соответствии с интерфейсом изменится вызов подпрограммы

```
interface
 SUBROUTINE proFact ( n, Factorial) ! заголовок процедуры
  integer,intent(in) :: n !  единственный входной параметр
  real*8,intent(out)::Factorial
   ! добавился выходной параметр вместо имени ПП
 end SUBROUTINE proFact
end interface
```

Оболочка ПП показана прописными буквами она сменилась, начинка– нет.

```
SUBROUTINE proFact ( n, Factorial) ! заголовок процедуры
  integer,intent(in) :: n ! единственный входной параметр
  real*8,intent(out) :: Factorial ! тип результата
   integer k
    Factorial = 1
    if ( n = = 0  ) return
    do k=1,n
      Factorial = Factorial *k
    enddo
END SUBROUTINE proFact
```

В соответствии с интерфейсом изменится вызов подпрограммы из ПП Loto - добавятся новые промежуточные переменные factorN, factorN, factorNM

```
integer n,m
real*8 factorN, factorN, factorNM, S
 Call proFact(n,factorN)
    Call proFact(m,factorM)
      Call proFact(n-m,factorNM)
          S = factorN  /  ( factorM)*factorNM )
```

Оценивая 4 варианта реализации вычисления факториала, можно отметить следующее:

- вычисление факториала предпочтительнее оформить не в виде процедуры, см. 4.4, а в виде функции, которую легче вызывать;
- это можно сделать, потому что результат – одно число;
- вызов функции не меняется, если ее по-разному определять, см. 4.1, 4.2, 4.3;
- самое краткое определение – у операторной функции, см. 4.3, видимо, из-за использования стандартных функций *product, dble,max.*

Чтобы легче понять разницу между возможными видами программ, классифицируем их по двум направлениям:

- *по горизонтали* - сколько исполняемых *операторов* в программе по принципу "0" "1" "много";
- *по вертикали* - сколько *результатов* (выходов), возвращаемых в вызывающую ПЕ, считая подпрограмму черным ящиком, у которого при вызове:
  - “много входов - много выходов";
  - "много входов - 1 выход";
  - "ни входов - ни выходов".

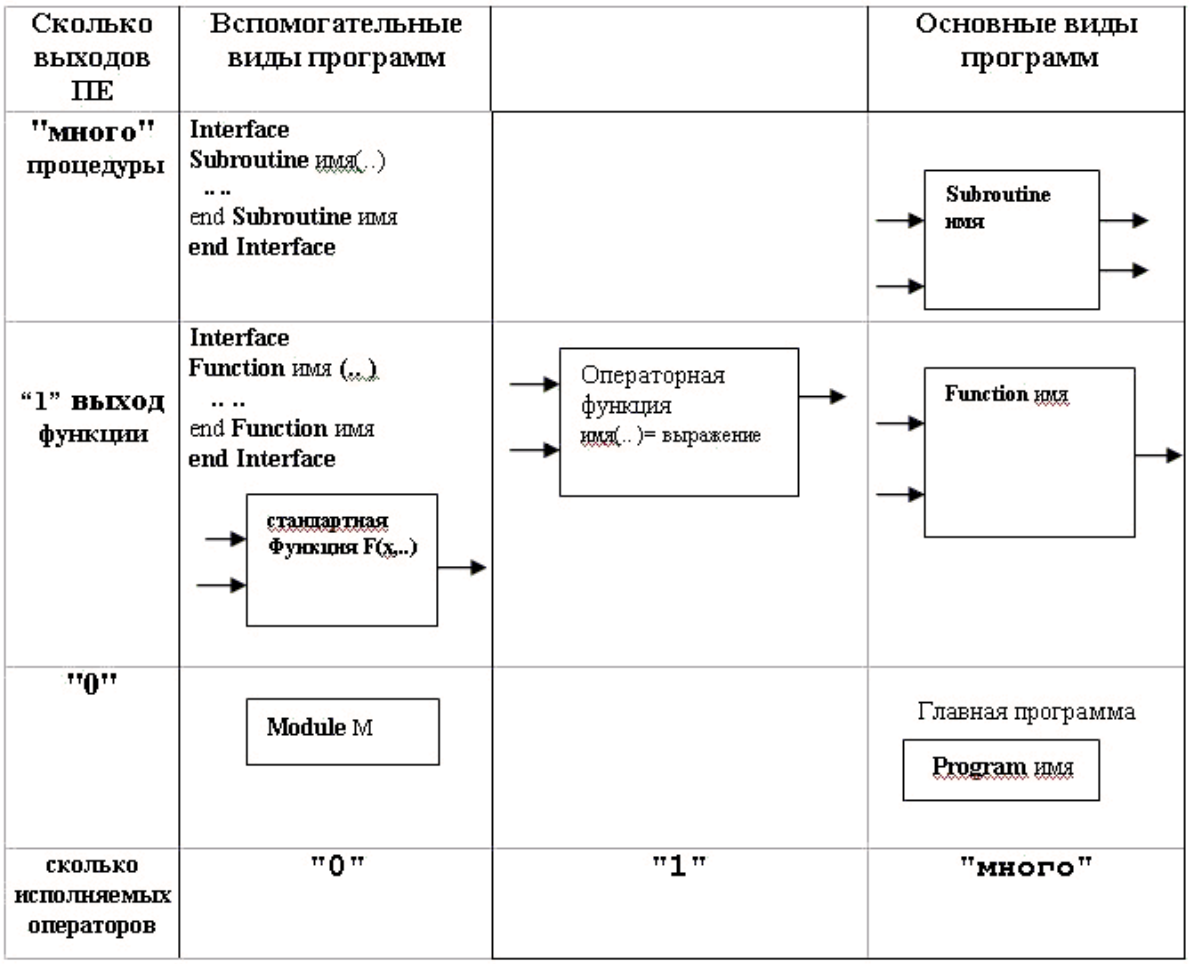


Рис. 2. Классификация программных единиц

Используя таблицу, проведем сравнительную классификацию программы по видам. В правую колонку "много" попали три основных вида ПЕ. Только многими исполняемыми операторами можно запрограммировать сложную задачу, разделенную на подзадачи в виде ПП. По разным причинам в колонке “0”, без исполняемых операторов, сосредоточено 4 представителя :

- интерфейсы функций и процедур, которые являются описаниями вызывающей ПЕ;
- стандартные функции, реализованные, скорее всего не на Фортране;
- модуль - самостоятельный вид ПЕ, не содержащий выполняемых операторов и состоящий:
  - из описаний общедоступных ***public***-объектов и локализованных в модуле ***private***-объектов;
  - из интерфейсов вызываемых ПП и из модульных ПП для обработки public-объектов.

Операторная функция - единственной строчка *имя***(..v)=e**, которая содержит формулу, напоминающую оператор присваивания. Её можно считать и исполняемым оператором.

Часто по своему усмотрению программист может оформить ПП и как ***Function*** и как ***Subroutine***. Умение удачно разделить задачу на подзадачи и выбрать подходящий вид ПЕ приходит с опытом. При большом числе входов/выходов удобнее передавать данные не через параметры, а через посредство заимствования объектов.

В центральной строке "1" сосредоточены все 4 вида функций, каждую из которых независимо от способа определения, можно вызывать единообразно, просто упоминая в формуле вызывающей программы. Способ определения зависит от числа исполняемых операторов, написанных на Фортране. Для стандартной и интерфейсной функции (ИФ) исполняемых операторов просто нет. Точнее говоря, они написаны в другом месте:

- стандартная функция, скорее всего, на Ассемблере, а не на Фортране;
- интерфейсная функция, возможно, например, на “С”, а не на Фортране;
- интерфейсная функция, возможно, например, и на Фортране, но в другом месте, например, кем-то, написана, скомпилирована и помещена в библиотеку, к примеру, IMSL.

Операторная функция (ОФ) имеет единственный исполняемый/описательный оператор. Это не самостоятельная ПЕ, а конструктивное определение, действующее только в той ПЕ, чьими обозначениями она пользуется. ПП вида ***Function*** *имя* **(..P)** является самостоятельной ПЕ, содержащей любое число операторов. *Имя* – одновременно и название функции, и переменная, отнесенная к одному из 5 базовых типов данных или производному типу данных (только в Ф90). По размерности и типу в Ф90 для функции возможны разные случаи:

1. функция может быть скаляром, требуется описания типа вызываемой функции;
2. функция может быть *массивоподобной*, то есть быть вектором, таблицей;
3. функция может быть *элементной, перенимая* форму у аргумента (скаляр или массивоподобная):

Примечание: для случаев 2,3 нужен явный интерфейс, чтобы использовать механизм *перегрузки*

Интерфейсы функций и процедур важны при смешанном программировании на “С” и Фортране. Интерфейсы функций и процедур поставляются в составе фирменных пакетов, в том числе ***use*** *msflib* для низкоуровневой графики**.**

Богатая, ориентированная на Windows библиотека Си, может быть использована в Фортране, если использовать готовые или написать свои интерфейсы, что требует понимания Фортрана и в какой-то мере Си. Можно применить и функции из dll, реализующих саму среду Windows. Применение функций из динамически связываемых библиотек dll требует их предварительной регистрации. Процедура на “С” истолковывается как ***Function*** либо ***Subroutine*** с соответствующим вызовом. Вызов стандартных функций так естественен, что кое-кто даже не догадывается, что за этим порой стоит программа, для повышения эффективности написанная на машинно-ориентированном Ассемблере.
В нижней строке "0" показаны два черных ящика без передачи данных через параметры:

- ***Program*** - начинает работу *приложения****,***
  end нормально завершает *приложение* и завершает текст;
- ***Module***
  - это упаковка для данных;
  - нет выполняемых операторов;
  содержит, если надо, ПП для обработки данных***;***
  end завершает текст модуля.

Модули – новинка Fortran-90

Модули многолики, вот лишь некоторые из них:

- контейнер данных;
- систематизация сложных проектов;
- интерфейсы;
- интерфейсы к программам на С;
- элементность функций - механизм *перегрузки формы;*
- механизм *перегрузки типов;*
- механизм *перегрузки операций;*
- *инкапсуляция - контейнер* данных и программ для их обработки;
- *определение классов.*

Не претендуя на полное описание, попытаемся немного прояснить предназначение модулей на примерах.

Как функция перенимает форму у параметра

Модуль ***gorner*** поясняет механизм *перегрузки* имен в результате которого функция ***polinom*** будет перенимать форму у параметра:

- *x - скаляр – тогда* ***polinom*** *тоже скаляр;*
- *X - вектор – тогда* ***polinom*** *тоже вектор.*

Далее приводится текст названного модуля.

```
module gorner
 implicit none
 integer,private :: i

 interface polinom !сводит воедино скалярную и векторную функцию
   MODULE PROCEDURE gscalar, gvector
 End interface

contains

function  gscalar(A,na,X)
 implicit none
  integer,intent(in):: na ! степень полинома
   real,dimension(0:na) :: A
    real,intent(in):: x
     real gscalar
gscalar = A(0) ! коэффициент при старшей степени полинома
do i=1,na
 gscalar=gscalar*x + A(i)
enddo
end function  gscalar

function  gvector(A,na,X)
 implicit none
 integer,intent(in):: na
 real,intent(in),dimension(:) ::  X
 real,dimension(1:size(X))::gvector ! перенимает форму X
 real,dimension(0:na) :: A
gvector = A(0)
do i=1,na
  gvector=gvector*x + A(i)
enddo
end function  gvector

end module gorner
```

Интерфейсы вызываемых программ

В данном примере подписан интерфейс к программе на C из FPS4.0, которая меняет кодировку си-подобной строки из ASCII в ANSI

```
module proInterface
   interface
```

```
    subroutine OemToChar(int1_name,AnsiName)
    !MS$ATTRIBUTES ALIAS:'_OemToCharA@8' :: OemToChar
      integer*1,intent(in),dimension(*) :: int1_name
      integer*1,intent(out),dimension(*) ::  AnsiName
    end subroutine OemToChar
  end interface
end module proInterface
```

Глава 3. Конформные массивы, встроенные функции, выражения и присваивания

Конформные массивы и секции - это база для построения конформных выражений и присваиваний. Параллельные вычисления - это эффективный по быстродействию код. Массивы, точнее говоря конформные массивы - это база для построения конформных выражений и присваиваний, на которых в свою очередь основаны параллельные вычисления.

Характеристики массивов

Массив – это множество однотипных элементов, объединенных общим именем. Доступ к элементам производится по индексам, их может быть от 1 до 7. При хранении в памяти справедливо следующее:

- в Фортране принята простая индексная модель массива:
  - массив – это прямоугольная таблица с неизменными по длине сторонами, называемыми размерностями;
  - модель нечувствительна к числу измерений массива;
  - модель нечувствительна к способу наделения массива памятью;
- в ***Fortran*** матрица хранится по столбцам - 1-ый индекс-номер строки, 2-й индекс-номер столбца;
- в “C” не так, модель путаная, матрица хранится по строкам, индексы столбца и строки наоборот, но чаще пользуются указателями или указателями на указатели, которые для многомерных пересчитывают вручную.

К характеристикам массива относят следующие характеристики:

- *Имя* массива;
- *Тип* массива – все элементы однотипны:
  - *базовые* типы ***integer real complex logical character*** ;
  - типы, определяемые программистом ***type (****имярек****)*** ;

  - в отличие от “С” Фортран придерживается строгой типизации: каждому типу - только свои операции;
- *ранг,* или число измерений массива, или число индексов от 0 до 7:
  - *1- вектор, одномерный массив*;
  - *2 – матрица, двумерный массив*;
  - ***3 –*** система массивов, *трехмерный массив*;
  - ***4 - 7***.. многомерные массивы;
  - *0* – нет индексов - *простая переменная*;
- по каждому измерению в описании следует задать на выбор:
  - либо, что предпочтительнее, *диапазон* возможных значений индекса *начало:конец,* тогда
    размерность= конец - начало+1*; начало* и *конец* – это целые числа, в том числе отрицательные и 0; притом *конец > начало*
    почему предпочтительнее: *двусмысленности не возникает*
    `real*8,dimension``(1:8)::` T- массив из 8 элементов
    `read(*,*)` T`(1:8)` как и **`read(*,*)`** T - читать все 8 элементов;
  - либо *размерность* – неотрицательное число, указывающее на количество возможных значений индекса; тогда диапазон
    1: *размерность*
    почему по-старому хуже - *возникает двусмысленность*:
    `real` T(8) - массив из 8 элементов;
    `read(*,*)` T(8) - читать только 8-й элемент, а не всё;
- форма массива *array* – это целочисленный вектор, составленный из размерностей; форму измеряют ***shape***(*array*);
- массивы, одинаковые по форме, называют *конформными* – это ключевое понятие в параллельном программировании;
- размер массива (количество элементов) можно измерить ***size***(*array*) или подсчитать через *форму* по формуле ***product***(***shape***(*array*)); ***product –*** это произведение;
- память под массив = <*память на элемент*> * ***size***(*array*) ;
- способ наделения массива памятью:
  - *статический*;
  - *динамический*;
  - *динамический автоматический* - только в подпрограмме;
  - *с переменной размерностью* - только в подпрограмме;
  - *перенимающий размерность* - только в подпрограмме;

- прочие атрибуты ***intent, save, parameter, private, public/***

Примеры:

Массивы `Mr,M2,Mp` *конформны*

```
real*8,dimension(1:2,1:3):: Mr,M2
real*8,dimension(0:1,-1:1):: Mp
```

shape(`Mr`)=***shape***(`M2`)= ***shape***(`Mp`)= [2,3].

Массивы `T, P` *неконформны*

```
real*8,dimension(0,3):: T
real*8,dimension(3,0):: P
```

Неконформны из-за того, что ***shape***(T)=[0,3], ***shape***(P)=[0,3], ***shape***(T)≠***shape***(P), хотя ***size***(T)=***size***(P)=0

Параллелизация начинается с описания *конформных* массивов, но это лишь база.

Примеры:

```
real*8,dimension(1:100,1:100):: Matr,  Matr2
 real*8,dimension(1:100):: Vec,Prod
  real*8 :: x=0.73
```

Параллелизация продолжается конформными формулами и конформными присваиваниями над векторами и матрицами, но этим не исчерпывается.

Правильно: **`Matr = -(2*x);  Matr2 = abs(Matr)-4.`**

Правильно: **`Prod = 2*x**(1./3)+4.`**

Сомнительно: **`Prod = 2*Vec**(1./3)+4`** так как одна из компонент вектора **Vec** может оказаться отрицательной.

Сомнительно: **`Vec=cos(-1/(2*Prod))`** так как одна из компонент вектора **Prod** может оказаться нулевой.

Неправильно: **`Matr = Vec + Prod`** - это сложение конформных векторов, но неконформное присваивание.

Обратим внимание на следующее:

- в примерах имеются только формулы и присваивания, нет циклов, хотя вычисляется много элементов;
- в правильные выражения и присваивания входят конформные объекты и *элементные* функции, *за исключением скаляров*, которые как бы *перенимают форму* у окружающих массивов;
- написание формул по виду не различается для скаляров, векторов и матриц, а способ обработки – передается в ведение компилятора;
- вообще, со стороны программы нет никаких указаний о порядке выбора компонент вектора или матрицы - именно такие “векторные”,

а лучше говорить *конформные* выражения и присваивания – именно они - основа распараллеливания;

- нет циклов, порождающих последовательность действий;
- проверка ОДЗ требует покомпонентного контроля, например,

```
where ( Vec >=0 )
Prod = 2*Vec**(1./3)+4
elsewhere !  только для отрицательных компонент - по-другому
  Prod = -2*(-Vec)**(1./3)+4
endwhere
```

Секция и конструктор массива

Секцией называют рабочий виртуальный массив, который определяют на базе обычного массива, один или несколько индексов которого задают множественным индексом одним из способов:

- либо в виде триплета, задающего арифметическую прогрессию;
- либо в виде целочисленного вектора;
- индексы должны быть из числа возможных значений по выбранному измерению.

Понятие секции массива важно для понимания новых конструкций, управляющих параллельными вычислениями. Применение секций во многих случаях позволяет *избежать циклов*, ориентирующих на последовательную обработку массивов. Важно, что в этом случае для компилятора остается свобода выбора способа обработки. При наличии многих процессоров создаются предпосылки для параллелизации.

Примеры по секциям и конструкторам

Пусть заданы описания конформных массивов X,Y

```
real,dimension(1:4) :: X,Y   ! массивы X,Y  по 4 элемента
read(1,*) X  ! Чтение массива целиком .
Y=X   !  задаёт пересылку массива целиком.
```

При пересылке только элементов с нечетными номерами индексов

X	**1**	2	**3**	4
Y	**1**		**3**	

```
Do i=1,3,2  ! по-старому пришлось бы задавать цикл
  Y(i)=X(i)
Enddo
```

Это не хорошо по 2 причинам:

1. выглядит архаично и громоздко;
2. главное - фиксирует *последовательность* действий.

К счастью, Fortran-90 ввел в обращение *секции массива*, которые формируются из базового массива. Ими можно пользоваться практически везде, где можно было пользоваться массивами.

Используя секции, пересылку, написанную выше по-старому, можно записать короче Y(1:3:2)=X(1:3:2).

Если надо поменять местами элементы массива с нечетными и четными номерами

X	**1**	2	**3**	4
Y	2	4	**1**	**3**

пишут Y(1:3:2)= X(2:4:2) ; Y(2:4:2)= X(1:3:2) .

Если надо составить массив Y из четных, потом нечетных элементов

X	**1**	2	**3**	4
Y	2	4	**1**	**3**

то, используя *конструктор* массива, вместо Y(1:2)=X(2:4:2); `Y(3:4)=X(1:3:2)` пишут еще короче `Y = (/ X(2:4:2), X(1:3:2) /)` . Фортран-95 разрешил более простые скобки конструктора Y = [X(2:4:2), X(1:3:2)].

Заметим, что, если надо поменять местами “на месте”

Y	**1**	2	**3**	4

то решение `Y(1:3:2)= Y(2:4:2); Y(2:4:2)= Y(1:3:2)`

– будет неправильным

Y	**1**	2	**3**	4
Y	2	4	**2**	**4**

а решение с *конструктором* `Y = [ Y(2:4:2), Y(1:3:2)]` – будет правильным

Y	**1**	2	**3**	4
Y	2	4	**1**	**3**

Имея тот же смысл, что и в цикле **Do** *индекс=начало,конец,шаг* тройка целых чисел, составляющих триплет пишется с другим знаком препинания *начало:конец:шаг* Объясняется это местоположением триплета – его пишут на месте индекса.

В написании цикла **Do** *индекс=начало,конец* [*,шаг*] *необязательным* является только *шаг,* подразумевается 1.

В написании триплета [*начало*] **:** [*конец*] [**:** *шаг*] *необязательным* является всё, кроме первого двоеточия. Все недостающие сведения берутся из описания массива.

Оформляя *секцию* массива, мы фактически создаем новый виртуальный массив со своими размерностными характеристиками, которые могут отличаться от базового массива, что интересно, не только в меньшую, но и в большую сторону.

Пусть описан массив **real,dimension**(1:4,1:6) :: M
Применение секций поясним на примерах:

M(1,:) – 1-я строка матрицы, одномерный массив из 6 элементов ;

M(i,:) – i-я строка матрицы, одномерный массив из 6 элементов ;

M(:,2) – 2-й столбец матрицы, одномерный массив из 4 элементов ;

M(:,j) – j-й столбец матрицы, одномерный массив из 4 элементов;

M(1:3:2,2:6:2) – матрица, двумерный массив 3 строки, 3 столбца;

M(1:3:2,6:2:-2) – матрица, двумерный массив 3 строки, 3 столбца, столбцы в обратном порядке;

M([1,3,4,2], 2:6:2) – секция с векторным индексом - матрица, двумерный массив 4 строки, 3 столбца, порядок следования строк изменен на [1,3,4,2] ;

M([1,3,4,2,3,1], 2:6:2) – секция с векторным индексом - матрица, двумерный массив 6 строк, 3 столбца, порядок следования строк изменен на [1,3,4,2,3,1], мало того строки 1 и 3 дублированы, столбцы только с четными номерами.

Распараллеливание и ввод-вывод в Фортране

Распараллеливая, главное понимать: выводить в параллельных ветвях нельзя, ведь это несовместимо со стремлением к быстродействию. Если вывод необходим, результат запоминают в промежуточном массиве.

Вкратце ввод-вывод в Фортране

Фортран из немногих языков, в котором ввод-вывод реализован операторами языка. В С и Паскале ввод-вывод реализован на уровне вызова процедур.
Концепции ввода-вывода в Фортране состоят в следующем:

- подсистема ввода-вывода, основана на принципе *интерпретации* – стоит вам написать один ***write*** – и в Ваш исполняемый код добавится несколько десятков килобайт – это и есть сразу весь интерпретатор;
- основной способ ввода-вывода – последовательный, как в кассетном магнитофоне – эта аналогия помогает правильному восприятию;
- каждый **write***(куда,как) что* - дает ответ на 3 вопроса – *куда? как? и что?* выводить;
- ответ на вопрос *куда?* – это условный номер фортрановского устройства, независимого от конкретных физических устройств (консоль, принтер, файл ..) ;

- самый популярный ответ на вопрос как? – это ссылка по номеру ***m*** из оператора **`write`** ***(*,m) что*** на оператор ***m*** **`format`** ***(*** *форматная* строка***)*** ;
- *формат* – уникальное описание, заимствованное многими языками из Фортрана.

Аналогии из обыденной жизни, которые помогают понять *формат* :

- **`format`** - это заранее заготовленный текстовой бланк:
  - текстовка и наполнение бланка*:*
    - *'тексты в апострофах'* или *"тексты в кавычках"*;
    - пробельные поля вроде *4x* - четыре пробела;
    - через "," идёт перечисление полей, переход к новой строке - "/";
  - *дескрипторы* ***i f e l a*** – это разнообразные пустые *окошечки* для заполнения ручкой;
  - список *что?* - это ручка, заполняющая окошечки бланка;
- *формат* обязан на каждую выводимую величину из списка *что?* дать способ преобразования ее в текст, помещаемый в *окошечко* ширины *α* - строго в соответствии *с типом:*
  - ***I****α для* **`integer`**;
  - ***F****α.d или* ***E****α.d для* **`real`** с d цифрами после точки;
  - ***L****α для* **`logical`**;
  - ***A****α или* ***A*** *для* **`character`**;
  - *2 штуки* ***F****α.d или* ***E****α.d* для **`complex`**;
- *формат* – это *почти формула*, например:
  - f5.2,f5.2,f5.2 короче выглядит как 3f5.2;
  - f5.2, f5.2,4x, f5.2, f5.2,4x, f5.2, f5.2,4x короче выглядит как 3(2f5.2,4x) ;
- одним **`write`** можно оформить вывод целиком таблицы, матрицы, массива:
  - **`write`** задает сколько выводить;
  - **`format`** задает сколько будет колонок в таблице и ее заголовок;
  - повторное сканирование последней пары скобок внутри ***format*** порождает нужное ;
  - *число_строк = сколько_выводить / колонок.*

Пример – одномерный массив M вывести в 2 колонки с заголовком

```
real,dimension(1:12) :: M
Write(2,11) M ! вывод массива для контроля
```

```
11 format(2x,'исходный массив в 2 колонки'/ &  ! заголовок
          'нечетный  |  четный номер'/&    ! 2-я строка
          (2x,F7.1,'  |',F7.1)  )  ! сканируемый участок выделен жирными ( )
```

Привлекательным интерактивным устройством отображения информации является дисплей, однако ни один язык не содержит команд для работы с этим мощным устройством; используются пакеты подпрограмм. Пакет низкоуровневой графики для раскраски миллиона точек экрана одинаково доступен и в Фортране и в Си.

Вывод-ввод с использованием секций

Особенно популярны и удобны секции для ввода-вывода двумерных массивов, которые в Фортране располагается в памяти по столбцам, а удобны для ввода – по строкам.

11	12	13
21	22	23

real,dimension (1:2,1:3) :: M

Команда **read** (1,*) M предполагает подготовку данных как в памяти - по столбцам, что эквивалентно

read (1,*) M(1,1), M(2,1), M(1,2), M(2,2), M(1,3), M(2,3)

данные: 11,21 12,22 13,23

При бесформатном вводе разделителями данных могут быть ***пробел запятая Enter*** *в любых сочетаниях.*

Построчный ввод с использование секции-строки *M(i,:)* и встроенного цикла по *i* пишут так **read** (1,*) (M(i,:), i=1,2)

данные можно записать в 1 строку так 11,12,13 21,22,23

или в 2 строки так

11, 12, 13

21, 22, 2

Или даже в 4 строки так, что малоосмысленно, но тоже правильно

11, 12

13

21

22, 23

Обратим внимание, что написание явного цикла

```
Do i=1,2
  read (1,*)  M(i,:)  ! каждый новый read начинает чтение с новой строки
enddo
```

исключит возможность правильного ввода таких данных 11,12,13 21,22,23

Свои особенности, связанные с написанием форматов, есть при выводе массивов

```
write(1,7)   ( M(i,:), i=1,2 )
7 format('Матрица'/ (3f8.2) )
```

! Заголовок 'Матрица' выведется однократно с переходом к новой строке,
! а участок, выделенный **жирными** скобками, повторится дважды, каждый повтор скобок сопровождается переходом к новой строке.

При необходимости повторители формата могут быть переменными или выражениями, заключенными в угловые скобки, например,

7 **format**('Матрица'/ <N>(f8.2))

Такой переменный повторитель пишется перед скобками, заключающими участок сканирования, и не может быть именованной константой.

Запрет ввода-вывода на параллелях

Сразу главное: ввод-вывод *запрещен* на параллелях. Понятно почему:

- распараллеливание – это максимально быстродействующий код;
- наличие ввода-вывода погубило бы это стремление, тому 2 причины:
  - работа с медленными внешними устройствами;
  - использование подсистемы ввода-вывода, организованной по принципу интерпретации, что крайне медленно – достаточно для внешних устройств, *но несовместимо* с принципом распараллеливания;
- ввод-вывод следует размещать на последовательных участках, там, где быстродействие не критично;
- действует все тот же универсальный принцип – размениваем память на быстродействие, а если с параллелей нужны результаты – их надо припасать в дополнительных массивах - см. массив Nofan на параллелях в Вычисление ряда для синуса

```
Integer*1,save,dimension(0:NofX) :: Nofan
 .. ..
where( abs(Sn-f)>eps )
 An = -An*x2 / ((2*n+2)*(2*n+3)); sn=sn+an
 Nofan=n+1
```

! интересно, что это эквивалентно Nofan(kx)=n+1=Nofan(kx)+1,
! где kx-номер в векторе X

```
Endwhere
```

В массиве Nofan нет необходимости при последовательной реализации

Встроенные функции Фортрана

Встроенные функции Фортрана – главное понять: многое уже распараллелено на уровне функций и фирменных библиотек.
Встроенные функции - неотъемлемая часть языка Фортран. Каждая из них имеет следующее:

- уникальное имя;
- тип аргумента (ов) ;
- тип возвращаемого значения;
- форма ответа, чаще всего повторяющая варьируемую форму аргумента, работает механизм так называемой *перегрузки формы*;
- если аргументов у функции несколько, они перечисляются в определенном порядке;
- имеются необязательные и ключевые параметры.

Полный список функций имеется в помощи по Фортрану.

В зависимости от выполняемых действий функции принято разбивать на группы: справочные, элементные, итоговые, преобразующие тип, преобразующие форму, функции векторной и матричной алгебры. Помимо встроенных функций, заявленных в описании языка, имеются традиционно портируемые функции, функции низкоуровневой графики, функции реалистичной графики OpenGL, функции и процедуры IMSL - библиотеки, реализующие 1500 математических методов. Через написание интерфейсов доступны функции языка Си. При параллельном программировании очень важно удостовериться, что библиотечные программы распараллелены.

Если аргументов у функции несколько, они перечисляются в определенном порядке. Имеются необязательные и ключевые параметры. В зависимости от выполняемых действий функции принято разбивать на группы.

Числовые справочные функции

Результат зависит от типа и разновидности аргумента, но не от его значения.

Функция	**Возвращаемое значение для одинарной точности**
`huge`(x)	Наибольшее число диапазона значений аргумента. *huge*(1)=2 147 483 674. HUGE(1.0)=3.402823E+38.
`tiny`(x)	Наименьшее отличное от нуля число диапазона значений вещественного аргумента. *tiny*(1.0)=1.1755494E-38
`epsilon`(x)	Ближайшее к единице значение вещественного аргумента. *epsilon*(1.0)=1.192093E-07, иначе говоря *epsilon*(1.0) +1.0 >1

В Фортране-90 большинство числовых и математических функций стали *элементными*: если аргумент-массив, то и результат-массив той же формы и того же типа, полученный применением функции поэлементно. В Фортране-90 большинство числовых и математических функций перенимают тип у своих аргументов, работает механизм так называемой *перегрузки*.

Элементные функции и итоговые функции – верный механизм поддержки параллельных вычислений:

- во многих случаях они исключают циклы ***do*** как *последовательные* структуры;
- сами по себе функции имеют скрытую *параллелизацию.*

Числовые функции

Функция	Возвращаемое значение
max(a1,a2,a3,..)	Максимальное из значений аргументов, *max*(-8.0,6.0,2.0)=>6.0 все аргументы должны быть либо все целые, либо все вещественные
min(a1,a2,a3..)	Минимальное из значений аргументов *min*(-8.0,6.0,2.0) => –8.0, аргументы должны быть либо все целые, либо все вещественные
abs(a)	Значением является |*a*|
mod(a,p)	Остаток от деления *a* на *p*. *mod*(17,2)=1. *mod*(18,2)=>0
int(a)	Целая часть аргумента. *int*(-5.7) имеет значение –5, *int*(0.9)=>0
nint(a)	Ближайшее к аргументу целое число. *nint*(-5.7) равно –6. *nint*(0.9)=>1
sign(a,b)	Абсолютное значение *a* со знаком *в*. sign(1.0,-1.0E-25) = –1.0

Математические элементные функции

В математике при работе с тригонометрическими функциями величину угла можно задавать в радианах, см. табл., или в градусах (к имени функции добавится окончание **d**). Следует обратить внимание на то, что функции этой группы являются элементными, то есть одинаково

применимыми и к скаляру, и к вектору, и к матрице, вообще к массиву любого ранга, заимствуя у аргумента его форму.

Функция	Возвращаемое значение
sqrt(x)	Квадратный корень из *x* ***sqrt(~~integer~~ x);*** ***real*** *x*>0: ***real sqrt(real*** *x); **complex sqrt(complex*** *x)*
exp(x)	Экспонента e^x
log(x)	log(*x*>0) в Фортране - это ln*x* *по основанию е*
log10(x)	Lg*x* (*x*>0) *по основанию* 10
sin(x)sind(x)	sin*x* (*x*-в радианах), тогда как *sind(x- в* градусах*)*
cos(x)cosd(x)	cos*x* (*x*-в радианах), тогда как *cosd* (x - *в* градусах)
tan(x) tand(x)	Tg*x* (*x*-в радианах) тогда как tand(x- *в* градусах)
cotan(x) **cotand(x)**	ctg*x* (*x*-в радианах, $x\neq0$) *x в* градусах
asin(x) **asind(x)**	arcsin*x* (|*x*|<1.0). Результат - в радианах. $-\pi/2<asin(x)<\pi/2$ Результат *в* градусах
acos(x) **acosd(x)**	arccos*x* (|*x*|<1.0). Результат - в радианах $-\pi\leq acos(x)\leq\pi$ Результат *в* градусах
atan(x) **atand(x)**	arctg*x*. Результат - в радианах $-\pi/2<atan(x)<\pi/2$ Результат *в* градусах
sinh(x)	Sh*x*- синус гиперболический
cosh(x)	Ch*x*- косинус гиперболический
tanh(x)	Th*x*- тангенс гиперболический

Справочные функции массивов

Функция	Возвращаемое значение не зависит от значений элементов массива
shape(A)	Форма массива - целочисленный вектор: • у которого число элементов равно рангу массива A; • значения - размерности в порядке следования измерений.
size(A)	Количество элементов массива, по-другому ***product***(***shape***(A))

Действия векторной и матричной алгебры как функции

Важные фунции матричной алгебры составляют данную группу:

- транспонирование матрицы transPmatrA=**transpose**(matrA)
- умножение матриц ResultMatr = **matmul**(matrA, matrB)
- скалярное произведение векторов ScalProizv=**dot_product**(VecC,VecD)

Встроенные функции для изменения формы массива

Любой многомерный массив представляет собой прямоугольную таблицу с неизменными длинами сторон и располагается в памяти компьютера в виде одномерного массива. С помощью вызова функции `Matrix=reshape` (*source*, *shape* [, *pad*] [, *order*]) исходный одномерный массив *source* любого типа меняет свою форму на форму, заданную параметром *shape*. В результате *source* переоформляется в массив заданной новой формы *shape*. В помощи Фортрана описана дополнительная пара необязательных параметров [, *pad*] [, *order*] .

В примере одномерный массив *source* = [1, 4, 5, 2, 3, 6], заданный конструктором, переоформляется в новую матрицу с 2 строками и 3 столбцами. Форма матрицы также задана конструктором *shape* = [2, 3]

```
Integer,dimension(1:2,1:3):: Matrix
```

result = **reshape** (*source*, *shape*)

Matrix=**reshape** (*source* = `[1,4, 5,2, 3,6],` *shape* = `[2,3] )`

Матрица имеет вид

```
1     5     3
4     2     6
```

Напомним, что в Фортране матрица располагается в памяти компьютера по столбцам.

С помощью функции **spread** в новый массив объединяется заданное количество копий исходного массива, образуя массив с новым указанным дополнительным измерением, синтаксис**:**

result = **spread** (*source*, *dim*, *ncopies*),

здесь *Source – входной* массив *любого типа;* массив, который надо реплицировать (повторить) вдоль заданного измерения;

Dim - входной целый параметр – номер измерения, вдоль которого надо реплицировать *source, причем* $1 \le dim \le n+1$, где n – это ранг массива *source;*

Ncopies – входной целый параметр – указывает *сколько раз* реплицировать *source.*

*Возвращаемое значение - м*ассив того же типа, что и исходный, но на одно измерение больше.

```
integer AR2(1:2,1:3), AR4(1:3,1:4)
AR2 = spread((/1,2,3/),dim= 1,ncopies= 2)
AR4 = spread((/1,2,3/), 2, 4)
!   1 1 1 1
!   2 2 2 2
!   3 3 3 3
```

Итоговые функции – редукция массивов и положение min, max

Функции этой группы заменяют циклы: **sum -** суммирует, **product-** перемножает, и т.д. Массив, обязательный 1-ый аргумент - никогда не опускают. Направление суммирования в массиве с числом измерений (рангом) более 1 регулируется аргументом *dim*. В тех случаях, когда задан параметр *Mask,* а ниже в примерах это *Mask*=A<0, - сперва вычисляется рабочий логический массив, показанный под описанием массива A, а затем отбираются только те элементы массива A, которым в маске соответствует "T".

Итоговые функции – применительно к одномерным массивам

Серия примеров этого раздела иллюстрируется на примере массива

номер элемента 1 2 3 4 5 6 7 8 9 10

real,dimension(1:10)::A= [4.5,7.5,3.,-3.,0.,-4.,7.5,1.,0.,1.1]

Mask=A<0→ [F F F T F T F F F F]

Итогом по *одномерному* массиву (ранг=1) всегда будет *скаляр, поэтому* ***dim*** в последующих примерах опускаем:

- *число* - для ***sum, product,*** *min****Val,*** *max****Val*** здесь ***Val****ue-значение*;
  - *число* = **sum**(*A*) →**27.5** - *сумма* элементов *A* - скаляр того же типа ***real***;
  - *число*=**product**(A,*Mask*=A<0)→(-3.)*(-4.)=**12.** – *произведение отрицательных*;
  - *число*= **minVal(**A**)** → -4. - минимальное *значение*;
  - *число*= **maxVal(**A**)** → 7.5 - максимальное *значение*;
- *число*=**sum**(A, *Mask*= A<0)→(-3.)+(-4.)=-7. – *сумма* отрицательных;
- *число*= **minVal *(abs(A)) →0***. – минимум по абсолютной величине;
- *число*= **maxVal (***A*,*Mask*=A<0**)** →-3. - максимум среди отрицательных [-3.,-4.] ;
- *число* =**sum**(*A* (1:9:2))→4.5+3+0+7.5+0=**15.** - сумма элементов с *нечётными* номерами 1,3,5,7,9;
- *свойства* числового массива ***all, any, count:***
  - *логическое*= **all** (*Mask= A<0*) →.**false**.=нет, – *Все ли* числа в *A* отрицательны?

- *логическое*= **any** (*Mask= A<0*) → .**true**.=да – *Есть ли* числа в *A* отрицательны?
- *целое* = **count** (*Mask=A<0*) →2 - *Количество* отрицательных [-3.,-4.].

Положение min или max элемента указывается при помощи **minLoc** - *№ минимального* элемента, при помощи **maxLoc** - *№ максимального* элемента. Положение элемента в общем случае задается *целочисленным вектором*. Почему вектор? – На положение минимума в 1-мерном укажет 1 индекс, в 2-мерном - 2 индекса, и т.д. – индексы как раз собраны в целочисленный вектор. Далее ряд примеров:

- *целое* =**sum**(**minLoc**(A)) →**sum**([6])=6 - найден №для *минимума,* равного -4; чтоб превратить вектор **minLoc**(A) = [6] в скаляр **6** использовали ***sum***;
- *целое* =**sum**(**maxLoc** *(A,Mask=A<0))* →4 – номер наибольшего -3 из [-3.,-4.] ;
- *N1max* **=** **sum**(**maxloc**(A)) →**sum**([2])=2 – № 1-ого *максимума*, равного 7.5 особенность: ищется № первого слева *максимума* 7.5*, которых* в массиве A имеется 2 штуки;
- отступив *N1max элементов*, можно найти № максимума, 2-го слева *N2max*=*N1max*+**sum**(**maxloc**(*A*(*N1max*+1:))) → 2+**sum**(**maxloc**([3,-3,0,-4,**7.5**,1,0,11]))=2+5=7;
- № последнего max ищется разворотом массива с возвратом к исходной нумерации 10-№+1 11-**sum**(**maxloc**(A(10:1:-1))) → 11-**sum**(**maxloc**([1.1,0,1,7.5,-4,0,-3, 3,7.5,4.5]))=11-**sum**([4])=**7 .**

Итоговые функции – применительно к двумерным массивам (матрицам)

Варианты задания аргумента *dim* будем иллюстрировать на примере матрицы **integer,dimension**(1:2,1:3) **::** M2. По *двумерному* массиву *функции* возвращают итог в форме *скаляра* или в форме *вектора* в зависимости от наличия и значения аргумента *dim :*

- *dim* – это номер измерения, если он не указан, то суммируется вся матрица M2 и ответ является скаляром

11	12	13
21	22	23

Ответ: **sum**(M2)→102- скаляр

- *dim* – это номер измерения, если *dim*=1, то исчезают строки, то есть **sum**(M2, *dim*=1) вызывает суммирование строками, что дает *вектор-строку* с суммами

dim=1	11	12	13	
	21	22	23	
	↓	↓	↓	
sum*(M2, dim=1)*	*32*	*34*	*36*	*Ответ: вектор-строка*

- *dim* – это номер измерения, если *dim*=2, то исчезают столбцы, то есть **sum**(M2, *dim*=2) вызывает суммирование столбцами, что дает *вектор- столбец* с суммами

11	12	13	→	Ответ:	36
21	22	23	→	**sum**(M2,*dim=2*)	66
				вектор- столбец	

Номер *minLoc*-минимального, *maxloc*-максимального - всегда вектор из целых чисел:

- *intVector*=**minLoc**(M2)→[1,1] - адрес *минимального* элемента матрицы, равного 11;
- *intVector*=**maxLoc**(M2) → [2,3] - адрес максимального элемента массива, равного 23;
- *intScalar*=**sum**(**maxLoc**(M2(:,2))) →**sum**(***maxLoc*** ([12,22]))= ***sum***([2])=2 - № max элемента из 2-го столбца.

Дополнительные разъяснения о ключевых и позиционных аргументах

Как можно задавать аргументы:

- *по порядку,* без *ключевых слов,* пример: *s*=***sum***(*A,* 1, *A<0*), можно опускать только последние аргументы;
- *в любом порядке* с указанием *ключевых слов* перед знаком "=", можно опускать любой аргумент,
  пример: *s*=**sum**(*Mask= A<0 , Array=A);*
- комбинируя первые 2 способа:
  верно: *s*=**sum**(*A, Mask=A<0*) - *A*-на своём, на 1-ом месте,
  здесь *Mask- ключевой* аргумент*;*
  неверно: s=**sum**(*A, A<0*) , так как *Mask =A<0* - 3-й аргумент, а попадает как позиционный аргумент не на своё 2-е место.

Аргументы функций по порядку и их ключевые слова в библиотеке:

Для числовых функций ***sum, product, minVal, maxVal***

1. *Array* = массив числового типа;
2. [*dim*] опущен или 1≤*dim*≤ *ранг* (*Array*):

- *ранг*=1, ответ только в форме скаляра, что делает *dim*-незначимым для одномерного массива;
- *ранг*=2 - двумерный массив, *dim* определяет *форму* ответа – скаляр, *dim=1* вектор-строка, *dim=2* вектор-столбец;
- *ранг*>2 - многомерный массив, ответ - многомерный массив, на одно указанное измерение *dim* меньше;

3. [*Mask*]= *логический* массив, конформный числовому массиву, определяемый как свойство элементов числового массива.

Для ***all, any, count,*** как функций от логического массива *Mask:*

1. *Mask*=логический массив, как свойство элементов числового массива;
2. [*dim*] определяет *форму* ответа аналогично числовому массиву.

Для ***minLoc*** или ***maxLoc,*** здесь ответ в форме целочисленного вектора ***Loc****ation* - номер элемента массива, функция с тремя параметрами:

1. *Array* = одномерный массив числового типа;
2. [*dim*] определяет *форму* ответа – появился в Ф95;
3. [*Mask*]=логический массив, как свойство элементов числового массива.

Общий случай (многомерные массивы)

Массивы ***Array*** и ***Mask*** *должны быть одинаковы по форме*

Функции подводят ***Итог*** (***array****=массив*, ***dim****=№измерения*, ***Mask****=свойство*)

Итог *подводят –*

- 4 функции редукции с *числовым ответом* по числовому массиву:
  - ***Sum-****сумма,* ***Product*** *– произведение,* ***MinVal*** *- минимум,* ***MaxVal*** *– максимум*
- 3 функции редукции, с *логическим или счётным* ответом по логическому массиву:
  All*-все ли* элементы со свойством *Mask,*
  Any- *есть ли* элементы со свойством *Mask,*
  Count*-количества элементов со свойством Mask;*
- 2 функции о положении ***max, min*** в форме целочисленного вектора с числом компонент, равным рангу массива *Array****:*** ***MinLoc*** *- положение минимума,* ***MaxLoc*** *- положение максимума;*
- функции, применимы к массивам от 1 до 7 измерений.

О форме массивов для итоговых функций

Дадим следующие разъяснения по форме массивов, являющихся параметрами итоговых функций:

- *Shape* - форма, вектор, составленный из размерностей в порядке следования измерений
- в общем случае **all***(Shape(Array)==* **Shape***(Mask)), такие* массивы, одинаковые по форме, называют *конформными* массивами
- в одномерных *конформных* массивах одинаковое число элементов **size***(Array)==***size***(Mask)*, а для *много*мерных этого совпадения мало для *конформности*
- *Mask=логическое выражение* относительно *A конформно A*, например: s=**sum**(*array=A*, *Mask = 0< A.and.A*<1)

Возможны 2 варианта *редукции* (сокращения числа измерений):

1. *скаляр* как итог по всему массиву (*dim* не задан);
2. ответ с рангом *меньше на одно* указанное подытоженное измерение *dim* .

Примечание: для одномерного массива (ранга 1) 2 варианта вырождаются в один единственный – *только скаляр*.

О параметрах функций - при вызове функции можно задать от 1 до 3 параметров из числа следующих, поимённо:

- *Array= числовой массив;*

Mask= логический массив, *Array и Mask* – конформны, то есть **shape***(Array)=***shape***(Mask),* если *Mask* задают как логическое выражение над *Array,* то *конформность* гарантируется scalar=**sum**(A, *Mask=A>0*)

- *dim=**номер*** сворачиваемого измерения массива

Применение функций иллюстрируется на 3-х мерном массиве A в форме параллелепипеда, состоящего из единиц
integer,dimension(1:3,1:4,1:5)::A=1
Форма массива *shape*(A)=[3,4,5]; размер *size*(A)=3*4*5=60.
Параллелепипед A содержит: 3 строки / 4 колонки / 5 таблиц.

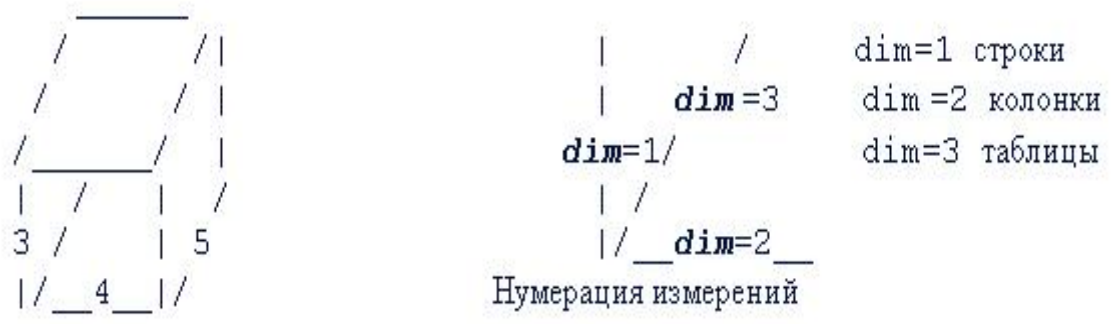


Рис.3. Геометрическая интерпретация 3-мерного массива

Выбор направления сжатия параллелепипеда осуществляется так:

- нет параметра *dim* - суммируется всё, исчезают все измерения, параллелепипед стягивается в точку - в ответе скаляр, например,

write** (2,*)* ‘сумма=’,sum**(A), &*
‘ произведение=*’,* ***product***(A)
- в ответе: сумма=60 произведение=1;

1. при *dim*=1, в направлении ⬇ суммируются и исчезают 3 строки; результат *sum(A,1)*- матрица формы *(/4,5/)*, как нижняя грань параллелепипеда;
2. при *dim*=2, в направлении ➔ суммируются и исчезают 4 колонки; результат *sum(A,2)*- матрица формы *(/3,5/)*, как правая грань параллелепипеда;
3. при *dim*=3, в направлении ⬋ суммируются и исчезают 5 таблиц; результат *sum(A,3)*- матрица формы *(/3,4/)*, как передняя грань параллелепипеда.

Сводная таблица итоговых функций

Сведем в таблицу все сведения о 9 итоговых функциях.

Параметры функций: *array*-числовой массив*, Mask*-логический массив, *dim*–сворачиваемое измерение массива.

Всегда *array* и *Mask* должны быть *конформны shape(array)==shape(Mask),* здесь *shape(*A*)-форма* массива*,* целочисленный вектор из размерностей по всем измерениям*, Mask*– логический массив, как *свойство элементов,* участвующих в подведении итогов по числовому массиву *array.* Чтобы гарантировать *конформность,* параметр *mask* часто задают в виде логического выражения для числового массива, как в примерах S=Sum(*array*=A,*mask*=A<0); C=All(2<A .and. A<3). Параметры *Array, mask* могут быть секциями, как в примере Any(A(1,1:2,:)>0).

В таблице будем пользоваться обозначениями ***F(****array*), ***F(***mask***)***, **F**(array[, mask]), **F(**mask [,dim]**),** когда будет идти речь о группе итоговых функций**.** Квадратные скобки [*аргумент*] пишут тогда, когда аргумент необязателен.

Таблица. Итоговые функции.

<table>
<tr><td rowspan="3">Имена, назначение функций и тип обязательного параметра - массива</td><td colspan="2">F(array)
array-обязателен: integer real complex</td><td colspan="2">F(mask)
Mask - обязателен: logical из {.true. .false.}</td></tr>
<tr><td>Положение экстремума</td><td>итоговые числовые операции</td><td>Итоговые логические операции
.or. .and.</td><td>Сколько штук .true.</td></tr>
<tr><td>maxloc(A)
minLoc(A)
Loc - от англ. Location-место
Ищется вектор с координатами экстремума</td><td><tt>sum</tt>(A) – сумма
<tt>product</tt>(A)– произведение
<tt>maxVal</tt>(A) - максимум
<tt>minVal</tt>(A) – минимум
от англ. Value-значение</td><td>есть ли .true.
<tt>Any</tt>(Mask)-
обобщение .or.
логического сложения

все ли .true.
<tt>All</tt>(Mask)-
обобщение .and.
логического умножения</td><td><tt>Count</tt>(Mask)
пересчитать элементы со свойством Mask</td></tr>
<tr><td>Аргументы
Скобки [ ], когда [необязателен]</td><td>F(array[, mask])
1) array
обязателен
2) [dim] для F95
3) [mask]</td><td>F(array[,dim][, mask])
1) array
обязателен
2) [dim]
3) [mask]</td><td colspan="2">F(mask [,dim])
1) mask обязателен
2) [dim]</td></tr>
<tr><td>Тип ответа</td><td>вектор
из целых чисел</td><td>того же типа, что array=A
скаляр или массив</td><td>Логический скаляр или массив</td><td>Целый скаляр или массив</td></tr>
<tr><td rowspan="2">Форма ответа

Особо
форма одно-мерного массива</td><td>вектор, с числом компонент, равным рангу Array</td><td colspan="3">dim - целое от 1 до ранга массива
Массив либо Скаляр, в зависимости от наличия dim
• ранг ответа на 1 меньше: вдоль dim подводится итог и оно исчезает
• нет dim - скаляр, итог подводится по всему массиву</td></tr>
<tr><td>не скаляр, а вектор с единственным элементом</td><td colspan="3">всегда скаляр
• задаем ли dim=1, для единственного измерения
• не задаем параметр dim вообще</td></tr>
</table>

Глава 4. Конструкции Фортрана для последовательных и параллельных вычислений

Управляющие конструкции Фортрана – это средства для самостоятельного программирования параллельно-последовательных вычислений, зависящих от каких-либо условий применения.

Скалярные управляющие конструкции Фортрана-77

Понятно, что только написание конформных выражений не дает достаточных средств для распараллеливания сложных задач. Как упоминалось ранее, по внешнему виду конформные выражения ничем не отличаются от скалярных выражений. Следующий шаг – использование конформных выражений в традиционных управляющих конструкциях. Из истории развития Фортрана можно вспомнить, что в Фортране-77 был узаконен стиль структурного программирования, опирающийся на 2 основные управляющие конструкции ***Do*** и ***If*** , которые включают, где они есть, скалярные условия, и подразумевают последовательные скалярные действия. В Фортране-90 стали возможны *конформные* присваивания под управлением конструкций ***Do If*** и ***Select case***.

Цикл **Do .. .. enddo** в 3 разновидностях с 1 блоком предполагает последовательные действия:

- **Do** ! бесконечный цикл
  ```
  .. ..
  enddo
  ```
- **Do** *параметр= начало , конец , шаг* ! цикл по параметру
  ```
  .. ..
  Enddo
  ```
- **Do while(** *условие* **)** ! итеративный цикл или цикл по скалярному условию
  ```
  .. ..
  Enddo
  ```

Ветвление ***If*** в 3 разновидностях с количеством блоков 2, 1, 0 :

- **If(** *условие* **) then ! *с*** 2 блоками
  ```
  .. .. блок ДА
  else
  .. .. блок НЕТ
  endif
  ```
- **If(** *условие* **) then ! *с*** 1 блоком
  ```
  .. .. блок ДА
  endif
  ```
- **If(** *условие* **)** *единственный_простой* ***! без*** блоков

Переключатель на много+1 направлений

```
Select case (целое)
   Case ( константа, диапазон_констант, перечисление ) ! и их сочетания
    .. ..
Case ()
    .. ..
Case default    ! в остальных случаях
    .. ..

End Select
```

Блочные конструкции ***Do If*** и ***Select case*** могут вкладываться друг в друга неограниченно.

Конформные управляющие конструкции

Конформные присваивания и формулы, как уже упоминалось ранее, по внешнему виду ничем не отличаются от скалярных присваиваний и формул. При вложении *конформных* присваиваний в скалярное управление получится управление на уровне манипуляций с векторами и матрицами или их секциями.

Если условия или циклы надо варьировать покомпонентно, то управляющие конструкции также должны стать *конформными.* Такие дополнительные управляющие конструкции оформились в Фортране-95. Конструкции реализуют *параллельные* действия над *конформными* векторами и матрицами при *конформных условиях.*

Оператор и конструкция *where*

Ветвление с количеством блоков 2,1,0 записывается как ***where*** в 3 разновидностях :

1. **where** *(конформное_условие* ***) ! с*** 2 блоками
   .. .. блок ДА *с конформными_действиями и конформными* ***where***
   else where
   .. .. блок НЕТ *с конформными_действиями и конформными* ***where***
   end where
2. ***where (****конформное_условие* ***)***
   ! с 1 блоком ***.. .. блок ДА*** *с конформными_действиями и*
   ! конформными ***where***
   end where
3. **where** *(конформное_условие* ***)*** *единственный_простой_конформный*
   ! без блоков

Нетрудно заметить внешнее сходство ***if*** и ***where*** – *оба они ветвления*. Однако есть кардинальное отличие:

- ***if*** задает *скалярное ветвление,* группируя действия над *скалярами* или *векторами целиком;*
- ***where*** - это *векторное покомпонентное ветвление,* которое группирует *параллельные* действия над *компонентами* векторов, матриц, многомерных массивов, секций с непременным требованием *конформности условия и действий.*

Имеется определенное сходство у ***where*** и ***Do*** – они оба могут обрабатывать вектора целиком. Однако имеется и принципиальное различие:

- ***Do*** обрабатывает компоненты вектора последовательно;
- ***where*** обрабатывает компоненты вектора независимо, параллельно;
- так что ***where*** - это скорее не цикл, а перечисление потенциально *параллельных процессов.*

Нетрудно заметить сходство ***if*** и ***Do*** - они могут выполняться над компонентами массивов, но только последовательно, а не покомпонентно в параллель.

Покомпонентно в параллель характерно выполнение только для ***where*** и ***forall***. Порядок действий, соответствующих ***where:***

- по комбинациям индексов, если они заданы триплетами, формируется секция;
- по секции, или по всему вектору вычисляется конформная маска;
- затем *параллельно* выполняются конформные действия – одно или несколько в последовательности заданной в блоке ***where;***
- действия выполняются только для истинных значений в маске;
- как всегда, *параллельно* – это, прежде всего независимость процессов, потенциальная способность - в том смысле, что это станет реальностью,
  - если будет использован распараллеливающий компилятор IFC
  - если процессоров будет предостаточно;
  - если процессоров будет недостаточно, то поочередно пачками
  - технические аспекты реализации передаются в ведение компилятора и операционной системы;
  - ситуация по числу свободных процессоров может меняться ввиду многозадачности и многопользовательности ОС, что затрудняет наши измерения времени решения задачи.

Традиционный пример – вычисление обратной величины *Obr* для *A*

```
real,dimension(1:10,1:10) :: A,Obr
```

в виде 2-блочной конструкции ***where***

```
where (A/=0 )     ! с 2 блоками
    Obr=1/A
else where
   Obr=0
end where
```

Можно также оформить в виде 2 операторов
Obr=0; **where** *(A/=0) Obr=1/A*

Оператор и конструкция *forall*

Иногда надо выполнить *конформные* действия *в пространстве* некой виртуальной секции. В этих случаях используют оператор ***forall***, в котором задают триплеты по ряду измерений и *маску* отбора к полученной секции по смыслу аналогичную ***where,*** но отличающейся по форме написания.

По форме записи маски для секции различаются:

- в операторе ***where –*** маска пишется в векторной форме
- в операторе ***forall*** - более гибко маска пишется в виде скаляра в форме переменной с индексами, заданной в заголовке ***forall***

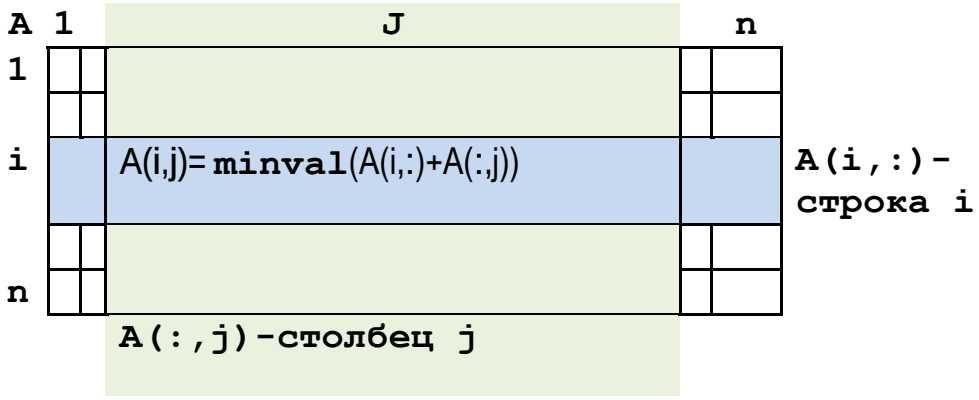
Действия над компонентами секции выполняются независимо и *параллельно*.

Триплеты, указывающие законы изменения индексов в секции, и маску для секции задают одним из 2 способов:

- или в *простом* безблочном операторе
  forall **(***индекс1имярек=триплет***, ..** *индексNимярек=триплет* [*, маска_секции*]**)** *единственный_простой_конформный оператор* **;**
- или в *составном* одноблочном операторе

forall (*индексимярек=триплет***,..** *индексимярек=триплет* [*, маска_секции*] **)**
.. .. маскированные *действия, конформные секции,* построенной по всем возможным комбинациям индексов из триплетов
end forall .

Ниже показан пример вычисления парафлоида – матрицы минимальных расстояний между вершинами ориентированного графа со взвешенными дугами. Граф задан весовой матрицей инцидентности. Схематично показано вычисление меры близости вершин графа по методу Флойда. Отметим компактность кода - единственный исполняемый оператор **forall** подпрограммы на Фортране-95. С целью упрощения на схеме не показаны условия, прописанные в этом простом операторе подпрограммы.




```
subroutine parafloid(n,A)
  integer,intent(in):: n
  integer,intent(in),dimension(1:n,1:n) :: A
  integer i,j
      forall( i=1:n, j=1:n,  i/=j.and.A(i,j)>0 ) &
          A(i,j) =  minval( A(i,:)+A(:,j),  mask=A(i,:)>0.and.A(:,j)>0 )
end subroutine parafloid
```

У ***forall*** есть аналогии и различия с другими управляющими конструкциями:

- с ***where –*** оба задают секцию, и маска строится заранее;
- *с вложенными* ***do –*** обе конструкции задают закон изменения индексов в виде арифметической прогрессии по триплету в заголовках;
- с ***do -*** и в теле оператора ***forall*** элементы массивов адресуются по индексам, заданным в заголовке ***forall ;***
- смысл адресации элементов массива в ***do*** и в ***forall*** абсолютно разный:
  - в ***do*** элементы массива выбираются *последовательно* по мере изменения индексов во вложенных ***do;***
  - в ***forall*** элементы массива выбираются *параллельно*, по предварительно построенным всем возможным комбинациям индексов;

Таким образом главное различие в назначении операторов:

- ***DO*** – основа *последовательного* выполнения повторяющихся действий;
- ***Forall*** – основа *параллельно*го выполнения повторяющихся действий.

Любое присваивание массивов из *where* можно переписать как *forall*, но некоторые *forall* не могут быть записаны через *where* только на уровне манипуляций с массивами и не выражаются другими конструкциями.
Например, **where**(A /= 0) B = 1.0 / A легко переписать в виде
forall (i=1:100, j=1:100, A(i,j)/=0) B(i,j) = 1.0 / A(i,j) .
Однако обратное не всегда справедливо – следующий оператор

forall(I= 1:N, J=1:N) H(I,J) = 1.0/(I + J - 1)

нельзя переписать через *where* на уровне манипуляций с массивами.

Forall задает следующее:

- список имен индексов, варьируемых в секции;
- каждому поименованному индексу сопоставлен триплет, определяющий закон его изменения;
- после списка задается необязательная маска, конформная секции, которая записывается в терминах варьируемых индексов;
- какие именно действия, над какими массивами или секциями, в какой последовательности задается в блоке оператора ***forall*** **.**

Порядок действий, соответствующих ***forall:***

- по заданным триплетам создаются списки всех возможных значений индексов;
- далее формируются все комбинации индексов;
- по комбинациям индексов формируется секция и конформная маска
- затем *параллельно* выполняются конформные действия в последовательности заданной в блоке;
- действия выполняются только для истинных значений в маске.

Получается, что ***forall*** - это скорее не цикл, а перечисление потенциально *параллельных процессов*.

Примеры. Произведение ненулевых элементов главной диагонали.

```
real,dimension(1:10,1:10) :: A
real diaPro; integer i
! Правильно
diaPro =1
do i=1,10
  if( A(i,i)/=0)     diaPro = diaPro* A(i,i)
enddo
```

Вариант с Forall вместо Do - неправильный, потому что Forall должен включать конформные присваивания

! в рамках определяемой ***1:10*** секции, а ***diaPro-скаляр***

diaPro =1;

Forall (i=1:10, A(i,i)/=0**)** ***diaPro = diaPro* A(i,i)***

! ошибка в левой части присваивания - ***diaPro*** не может быть скаляром.

Для ***Forall*** правильный вариант может выглядеть следующим образом: выбрать диагональ в вектор, а затем элементы перемножить.

```
real,dimension(1:10,1:10) :: A
real,dimension(1:10) :: Diagonal
 real diaPro; integer i
Diagonal =1
Forall (i=1:10, A(i,i)/=0) Diagonal(i)= A(i,i)
diaPro=product(Diagonal)
```

Синтаксис записи триплета через двоеточие ещё раз подчёркивает:

- уместность аналогии с секциями, а не циклами;
- *параллельность*, а не *последовательность;*
- чтобы потенциально *параллельная* конструкция стала реально таковой, непременными являются 2 условия:
    1. распараллеливающий компилятор, как IFC;
    2. наличие многих процессоров.

Различие скалярных и векторных конструкций

Поскольку ***forall*** не цикл, а перечисление параллельных процессов, то напрашивающаяся аналогия с циклом ограниченна. В ряде случаев ***do*** и ***forall*** одинаковы по действию, тривиальные примеры:

```
Integer, dimension(1:4) :: A
    A=0   ! параллельные действия
    forall   (i=1:4) A(i)=0 ! параллельные действия
    do   i=1,4 ! последовательные действия
      A(i)=0
    end do
```

Простой пример, из книги Немнюгина [1], иллюстрирующий разницу между ***Do*** и ***Forall :***

- цикл ***Do*** для массива

  integer, dimension(1:4) :: A=(/ 1,2,3,4 /)

```
Do i=2,3 ! последовательные действия
   A(i) = A(i-1) + A(i) +  A(i+1)
Enddo
 !  получим
A(2) = A( 1) + A2) +  A(3)=1+2+3=6
A(3) = A( 2) + A(3) +  A(4)=6+3+4=13 ! A( 2)=6 - изменившееся значение;
```

- **forall,** перечисляющий параллельные процессы над исходными значениями массива, - те же по виду действия дают другой ответ

```
integer, dimension(1:4) :: A=(/ 1,2,3,4 /)
Forall( i=2:3 ) ! параллельные действия
  A(i) = A(i-1) + A(i) +  A(i+1)  ! подставляется исходное значение A( 2)=2
   ! то же по-другому A(i)=sum(A(i-1:i+1))  - проясняет суть дела
End forall
 ! даст ответ
A(2) = A( 1) + A2) +  A(3)=1+2+3=6
A(3) = A( 2) + A(3) +  A(4)=2+3+4=9  ! A( 2)=2 исходное значение
```

Решение без циклов A(2:3) = (/ **sum**(A(1:3)) , **sum**(A(2:4)) всё разъясняет: берем старые значения, а заносим новые значения A. Ни к чему здесь **forall** – это чисто академический стиль.

Нетрудно заметить внешнее сходство ***Do*** – *цикл* и ***Forall*** – *как бы цикл*. Однако есть кардинальное отличие цикла и перечисления ***Forall***:

- ***Do*** задает *последовательность* действий, которые выполняются над скалярами либо целиком векторами;
- ***forall*** задает *параллельные* действия, которые выполняются над компонентами векторов, матриц, многомерных массивов, секциями с непременным требованием *конформности условия и действий.*

Вложение конструкций

Относительно вложения конструкций друг в друга заметим следующее:

- блочные конструкции ***Do*** и ***If*** могут друг в друга вкладываться неограниченно;
- блочные *конструкции* ***forall*** и ***where*** могут друг в друга вкладываться неограниченно, если условия и действия конформны;
- блочные *параллельные конструкции* ***forall*** и ***where*** могут вкладываться в скалярные блочные конструкции ***Do*** и ***If;***
- скалярные конструкции ***Do*** и ***If нельзя*** вкладывать в параллельные конструкции ***forall*** и ***where***, поскольку туда можно вкладывать только конформные параллельные конструкции.

В теле оператора forall может быть следующее:

- присваивание;
- WHERE – оператор или конструкция
  WHERE используют маску при присваивании для массивов;
- FORALL – оператор или конструкция ;
- разрешено вложение только *параллельных* действий над векторами, матрицами, многомерными массивами с непременным требованием *конформности условия и действий.*

Относительно вложения других операторов в ***forall*** и ***where***

- не разрешено вложение обычных скалярных операторов, в частности ***write*** и ***read;***
- не разрешено вложение обычных скалярных операторов, в частности ***Do*** и ***If*** **.**

Преобразование последовательных конструкций в параллельные

Заметим, что при определенных условиях, конструкцию ***Do***, на самом деле, можно преобразовать в параллельные. Например, простейшее A=0 соответствует циклу

```
integer,dimension(1:1000):: A
Do i=1,1000
  A(i)=0
Enddo
```

Однако, понятно, что если Вы собираетесь заниматься параллельным программированием, то написание такого цикла просто неразумно – и тут не стоит полагаться на то, что компилятор оптимизирует такие ляпы. Хотя вполне возможно, что IFC10 действительно это сделает - проверьте. Да и с точки зрения читаемости программы подобный цикл режет натренированный глаз.

Кстати сказать, если писать на “Си”, то там кроме циклов других вариантов нет. Препроцессорные примочки сути дела не меняют – последовательный цикл остается. Если же Вы станете пользоваться MPI или OMP для распараллеливания, то написание только усложнится, потому что придется добавлять дополнительно директивы на этих языках – вчитайтесь в такого рода текст на Си.

Численные методы. Ряд Тейлора для заданной функции.

Данная задача решается в форме виртуальной работы: задание получают, а отчет сдают через сайт (ЦДО), а выполняется оно самостоятельно во внеаудиторное время. Ознакомиться с

последовательными и выполнить параллельные расчёты для заданной функции:

- знакомимся с конформными вычислениями и конструкциями;
- знакомимся с операторами форматного вывода;
- аналог вычисления ряда для синуса – параллельные расчеты;
- № варианта – по официальному списку группы;
- в этой задаче нет критических по быстродействию участков, цель – знакомство с конформными вычислениями.

Глава 5. Сравнительное программирование параллельных и последовательных вычислений на Фортране-95

Выполняется сравнительное программирование параллельных и последовательных вычислений на Фортране-95. Рекомендуется опробовать программу на Си для тех же задач в плане самостоятельной работы.

Векторное произведение квадратной матрицы на вектор

Одна и та же программа на Фортране может стать и параллельной и последовательной:

- программа на Фортране для *параллельных* вычислений параллельной - если скомпилировать в IFC - Intel Fortran Compiler и выполнить на многопроцессорном компьютере;
- программа на Фортране для последовательного вычисления *станет последовательной* - если скомпилировать в MS FPS 4.0, *станет последовательной* - если даже скомпилировать в IFC, но выполнить на однопроцессорном компьютере.

Программа на Си - исключительно последовательная и не содержит никаких сведений для распараллеливания. Директивы MPI, написанные вручную, делают программу на любом языке работающей параллельно-последовательно. Надо специально думать и о решении задачи на Си, и о распараллеливании на языке MPI - не Си-программа становится параллельной, такой её делают директивы MPI.

Векторное произведение квадратных матриц - программы на фортране

Исследуется два решения одной и той же задачи:

- *последовательные* вычисления
  программа намеренно написана по-старому в виде 3 вложенных последовательных конструкций - циклов, и её не сделает параллельной даже компилятор IFC;

- *параллельно-последовательные* вычисления
  программа написана с применением перечислителя параллельных процессов ***forall***, а каждый из процессов умножает матрицу на вектор по параллельной стандартной программе ***matmul***, как в предыдущей задаче.

Вычисление синуса с использованием ряда Тейлора - программы на Фортране

Исследуется два решения одной и той же задачи:

- *последовательные* вычисления:
  1. *x* f(*x*) An(*x*) Sn(*x*) - скаляры $
  2. *по Eps, по x,* по *n* - вложенные циклы: расчет - печать для каждой пары x, *eps.*
- *параллельно-последовательные* расчеты:
  имеется цикл только по *n:*
  1. вектор X-заполнен заранее при *фиксированном Eps*
     - X f(X) An(X) Sn(X) – векторы;
     - цикл по X исчез - распараллеливание по X стало возможно;
  2. вектор X-заполнен заранее, вектор *Eps:*
     - f(X) – вектор;
     - An(X, *Eps*), Sn(X, *Eps*) – матрицы$
     - распараллеливание с помощью ***forall*** по *X* и *Eps*

  сначала расчеты - потом печать.

Скаляр x / скаляр eps - базовый расчет

Исследуется два решения одной и той же задачи:

- *последовательные* вычисления:
  - *x* f An Sn *Eps* – скаляры;
  - цикл *по n*: расчет - печать для каждого n.

```
Program var1
 Implicit None
  Real :: eps=0.00001 ! методическая погрешность для графика
   Real:: x2,x=2.5 ! конкретное число из области сходимости ряда
    Integer :: n ! номер члена ряда
     Integer :: Ng =50
! MAX допустимое значение номера члена ряда, например, 50 или 150
 Real An   ! член ряда, An - это в программе, а в математике это A0, A1, .. An, An +1
  Real Sn,f    ! сумма Sn зависит от n, а функция f(x0) не зависит от n
   Namelist /avaria/ x, An, Sn, f, eps, N, Ng
```

```
Open(1,file="Teilor.txt")     ! для просмотра  глазами
Write(1,*) 'eps=',eps ;
Write(1,*) 'n, An,  f,  Sn, abs(f-Sn)' !надписи
x2=x*x
An=x
! Выбрать стартовое значение A0  или A1 - конкретная формула для начала ряда
Sn=An  ! стартовое значение суммы ряда
f=sin(X)  ! эталон  Вашей функции для фиксированного аргумента
    Write(1,*) 0,An,f,Sn,abs(Sn-f)
 do  n=0,NG  ! суммирование ряда, начиная с n=0 или n=1
    An = -An*x2/((2*n+2)*(2*n+3))
    Sn=Sn+An  ! рекуррентная формула A(n+1) = An*Tn
      if(abs(Sn-f)<eps)  exit  ! так нужно при переменном знаке An
     Write(1,*) n+1,An,f,Sn,abs(Sn-f)
 endDo
              If( n>Ng )  then  ! при n>Ng - проблемы
                   Write(1, avaria); Stop " KARAUL!!!"
              EndIf
Write(1,*) 'fin:',x, f,Sn, abs(f-Sn), n
! предельное Sn, abs(f-Sn) и n для заданного x
End Program var1   ! бесформатный вывод
 eps=  9.9999997E-06
   n,      An           f,            Sn,          abs(f-Sn)
   0   2.500000      0.5984721       2.500000       1.901528
   1  -2.604167      0.5984721     -0.1041667      0.7026389
   2   0.8138021     0.5984721      0.7096354      0.1111633
   3 -0.1211015      0.5984721      0.5885339      9.9382401E-03
   4   1.0512283E-02 0.5984721      0.5990462      5.7405233E-04
   5 -5.9728883E-04  0.5984721      0.5984489      2.3245811E-05
 fin:  2.500000    0.5984721     0.5984728    6.5565109E-07     5
```

Скаляр x / Вектор Eps - последовательное вычисление для многих X

Исследуется два решения одной и той же задачи:

- *последовательные* расчеты:
  - имеются вложенные циклы по *Eps,* по x и по *n$*
  - x f An Sn – скаляры;
  - циклы блокируют распараллеливание;
  - печать по мере проведения расчетов.

```
Program Sin_var3    ! красивая таблица
 Implicit None
  Integer :: j,count,ke
   Real,dimension(1:6) ::   &  ! методическая погрешность
  eps=(/ 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001 /)
```

```
Real::x,Xn=0,xk=1.,xh=1./200000  ! аргумент, границы и шаг изменения
 Real An,Sn,f   !  член ряда, сумма, функция
  character*50 :: f12
   character*16 :: fn="('|',F5.2,2('|',"
    character*26 :: "),'|',I6,5x,'|',e11.1,'|')"
     character*2  :: cj2
      character*4  :: ff="F12."
       Integer :: n,Ng=50 !  номер члена ряда и его MAX  значение
Open(1,file='Teilor.txt') !  таблица для просмотра
DO ke=1,6
 j=nint(abs(log10(eps(ke))))
 Write(cj2,"(i2)") j
 f12=trim(fn) // ff // trim(adjustl(cj2)) //trim(fk)
 Write(1,*)   ' 12 format',f12
 Write(1,10) eps(ke) ! вывод "шапки" таблицы
 count=1
 Do x = Xn, Xk+ xh /2, xh
  f=sin(x); An=x; Sn=An !  стартовые значения A0 и суммы ряда
  do n=0,Ng,1 !  суммирование ряда
    An = -An*x*x / ((2*n+2)*(2*n+3));  Sn=Sn+An
    if( Abs(f-Sn)<eps(ke) ) exit
  endDo
        If(n>Ng) Stop ' KARAUL!!! '   ! при n>Ng - проблемы
  !  сколько цифр выводить после точки указано в F12.6 в соответствии с  eps(ke)
  if(mod(count,40000)==1) &
     Write(1,f12)x,f,Sn,n+2,abs(f-Sn)
! вывод строки таблицы
  count=count+1
 EndDo
 Write(1,11) !  вывод ”донышка” таблицы
ENDDO
 10 Format('Точность вычисления ряда eps=',e11.1/&
 '----------------------------------------------------'/ &
 '| x |  станд   | Тейлор   |Членов ряда|  Разница  |'/ &
 '----------------------------------------------------')
 12 Format('|',F5.2,2('|',F12.6),'|',I6,5x,'|',e11.1,'|')
 11 Format(1x,55('-'),1x)
End Program Sin_var3
```

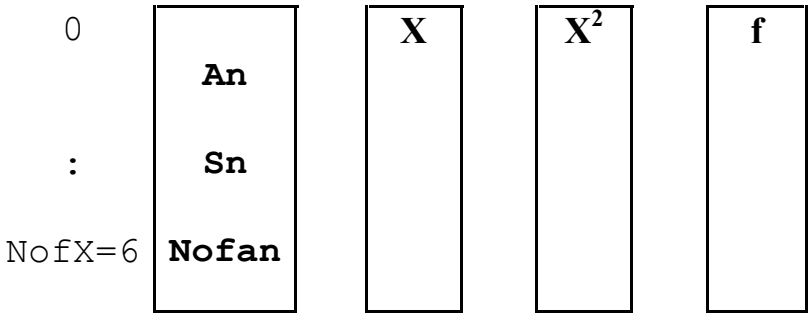
Далее приводится распечатка результатов работы программы.

```
Точность вычисления ряда eps=    0.1E+00
 ------------------------------------------------------
|  x  |   станд    |  Тейлор    |Членов ряда|  Разница  |
 ------------------------------------------------------
| 0.00|         0.0|         0.0|     2     |    0.0E+00|
| 0.20|         0.2|         0.2|     2     |    0.3E-05|
| 0.40|         0.4|         0.4|     2     |    0.9E-04|
| 0.60|         0.6|         0.6|     2     |    0.6E-03|
| 0.80|         0.7|         0.7|     2     |    0.3E-02|
| 1.00|         0.8|         0.8|     2     |    0.8E-02|
 ------------------------------------------------------
  12 format('|',F5.2,2('|',F12.2),'|',I6,5x,'|',e11.1,'|')
Точность вычисления ряда eps=    0.1E-01
 ------------------------------------------------------
|  x  |   станд    |  Тейлор    |Членов ряда|  Разница  |
 ------------------------------------------------------
| 0.00|        0.00|        0.00|     2     |    0.0E+00|
.. ..
  12 format('|',F5.2,2('|',F12.6),'|',I6,5x,'|',e11.1,'|')
Точность вычисления ряда eps=    0.1E-05
 ------------------------------------------------------
|  x  |   станд    |  Тейлор    |Членов ряда|  Разница  |
 ------------------------------------------------------
| 0.00|    0.000000|    0.000000|     2     |    0.0E+00|
| 0.20|    0.198755|    0.198755|     3     |    0.2E-08|
| 0.40|    0.389749|    0.389750|     3     |    0.3E-06|
| 0.60|    0.565163|    0.565163|     4     |    0.6E-07|
| 0.80|    0.717985|    0.717984|     4     |    0.4E-06|
| 1.00|    0.842105|    0.842105|     5     |    0.2E-08|
 ------------------------------------------------------
```

Вектор X / скаляр eps - параллельное вычисление по X

Исследуется два решения одной и той же задачи:

- *параллельно-последовательные* расчеты:
  - имеется цикл по *n –членам ряда;*
  - X f(X) An(X) Sn(X) - векторы, X-заполнен заранее;
  - цикл по X исчез - распараллеливание по X стало возможно;
  - сначала расчеты - потом печать



```
Program paraSin_X
 Implicit None
  Real :: eps=1e-6    ! методическая погрешность для графика
   Integer,parameter :: NofX=6, Ng=20
! Ng-допустимое значение n члена ряда
    Integer :: n, p, b=NofX+1, cnt,cnt1 ! n-номер члена ряда
     Real,save,dimension(0:NofX) ::An, &
         Sn,X2,f,X=(/(p*(1./NofX),p=0,NofX)/)
     Integer*1,save,dimension(0:NofX) :: NofAn
Open(1,file="Teilor.txt")
f=sin(X); Nofan=1; cnt=0; An=x; Sn=An; X2=X*X
do n=0,NG
  where(abs(Sn-f)>eps) ! участвуют конформные векторы и скаляры
    An = -An*X2 / ((2*n+2)*(2*n+3));  Sn = Sn+An; NofAn=n+1
  endwhere
  cnt1=sum(int(Nofan)); if( cnt==cnt1 ) exit
  cnt=cnt1
enddo
call printresX
                contains
subroutine printresX
  Write(1,4) eps
  Write(1,6)
  Write(1,1) '     x   ', x
  Write(1,6)
if(n<=NG)   then ! успешно
 Write(1,2) '     n   ', NofAn
 Write(1,1) '     f   ', f
 Write(1,1) '     Sn  ', Sn
 Write(1,3) '  |f-Sn| ', abs(f-Sn)
 Write(1,6)
 Write(1,5) maxval(Nofan)
Else ! не успешно
  Write(1,1) ' не успешно: n>Ng :  ', An
endif
1 format('|',a,'|',<b>(g7.5,'|'))
2 format('|',a,'|',<b>(i7, '|'))
3 format('|',a,'|',<b>(e7.1, '|'))
4 format('eps=',e7.1)
5 format(67('-')/'Nmax=',i2)
6 format(67('-'))
end subroutine printresX

End Program paraSin_X
```

Далее приводится распечатка результатов работы программы.

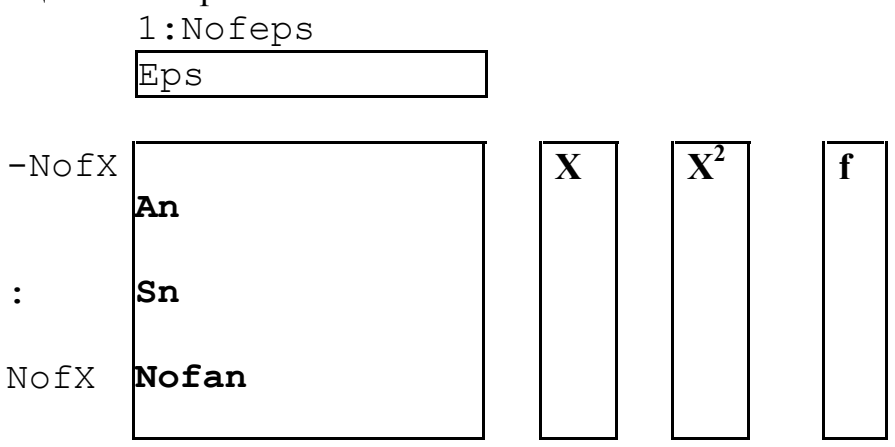
```
eps=0.1E-05
-------------------------------------------------------------------
|   x    | 0.0000|0.16667|0.33333|0.50000|0.66667|0.83333| 1.0000|
-------------------------------------------------------------------
|   n    |      0|      2|      2|      3|      3|      3|      4|
|   f    | 0.0000|0.16590|0.32719|0.47943|0.61837|0.74018|0.84147|
|   Sn   | 0.0000|0.16590|0.32719|0.47943|0.61837|0.74018|0.84147|
||f-Sn|  |0.0E+00|0.0E+00|0.9E-07|0.3E-07|0.1E-06|0.5E-06|0.0E+00|
-------------------------------------------------------------------
Nmax= 4
```

Вектор X вектор Eps - параллельное вычисление по X по Eps

Исследуется два решения одной и той же задачи:

- *параллельно-последовательные* расчеты
  - имеется цикл только по *n*
  - X f(X) - векторы, X-заполнен заранее
  - An(X, *Eps*), Sn(X, *Eps*) – матрицы
  - распараллеливание с помощью ***forall*** по *X* и *Eps*
  - сначала расчеты - потом печать

Самое простое решение по написанию, но совсем не обязательно самое эффективное по быстодействию - это параллелизация при помощи *forall* по секции X на Eps




```
Program paraSin_X_EPS
 Implicit None
  Integer,parameter :: Nofeps=6,ng=20,NofPrint=11
Integer,parameter::NofX=10000,stepOfPrint=(2*NofX+1)/(NofPrint-1.)
! по {Eps}:
Real,save,dimension(1:Nofeps):: &
eps=(/0.1,0.01,1e-3,1e-4,1e-5,1e-6/)    ! по {X}:f(X),X2(X)
Real,save,dimension(-NofX:NofX)::f,x2, &
      X=(/(p*(1./NofX),p=-NofX,NofX)/)
! по {X,Eps}: an(X,Eps),sn(X,Eps),Nofan(X,Eps) Eps-методическая погрешность
```

```
Real,save,dimension(-NofX:NofX,1:Nofeps) :: an,sn
Integer*1,save,dimension(-NofX:NofX,1:Nofeps)::Nofan
Integer :: p,n, ke,kx, cnt,cnt1
 Open(1,file='Teilor.txt') ! таблица для просмотра Nofan, An,Sn
 x2=x*x; f=sin(x); Nofan=0; cnt=0
 An=SPREAD(X,DIM=2,NCOPIES=Nofeps); Sn=An ! копии по столбцам
 DO n=0,ng  ! последовательные итерации по n-членам ряда
 ! впараллель в пространстве двумерной секции {X,Eps} по маске, где |An|>eps
  forall (kx=-NofX:NofX,ke=1:Nofeps, abs(An(kx,ke))>eps(ke) )
      An(kx,ke) = -An(kx,ke)*x2(kx) / ((2*n+2)*(2*n+3))
      sn(kx,ke)=sn(kx,ke)+an(kx,ke) ! результаты запоминаются
      Nofan(kx,ke)=n+1 ! по каждой паре {X,Eps}: своё n
  endforall
  cnt1=sum(int(Nofan)); if( cnt==cnt1 ) exit
  cnt=cnt1
 ENDDO
 call  printResults  ! печать после параллелей

 contains

 subroutine printResults ! процедура печати после параллелей
  Integer i,j
  where(Nofan==0) Nofan=1
  do i=1,Nofeps
   Write(1,10) eps(i) ! вывод "шапки" красивой таблицы с циклом по eps
   do  j=-NofX,NofX,stepOfPrint ! вывод строк таблицы
     Write(1,12) &
       x(j),f(j),Sn(j,i),Nofan(j,i),abs(f(j)-Sn(j,i))
    enddo    Write(1,11)  ! вывод "донышка" таблицы
  enddo
  Write(1,13) n
10 Format('Точность вычисления ряда eps=',e11.1/ &
'----------------------------------------------------'/ &
'| x |  станд  | Тейлор  | Член. ряда |  Разница  |'/ &
'----------------------------------------------------')
12 Format('|',F5.2,2('|',F12.6),'|',I6,5x,'|',e11.1,'|')
11 Format(1x,55('-'),1x)
13 Format(/'Nmax=',i3)
 End subroutine printResults
End Program paraSin_X_EPS
```

Далее приводится распечатка результатов работы программы.

```
Точность вычисления ряда eps=    0.1E+00
 ----------------------------------------------------
```

```
|  x  |   станд    |  Тейлор    |Член. ряда |  Разница  |
 ---------------------------------------------------------
|-1.00|   -0.841471|   -0.841667|     2     |    0.2E-03|
|-0.80|   -0.717356|   -0.714667|     1     |    0.3E-02|
.. ..
| 0.80|    0.717356|    0.714667|     1     |    0.3E-02|
| 1.00|    0.841471|    0.841667|     2     |    0.2E-03|
 ---------------------------------------------------------
Точность вычисления ряда eps=    0.1E-01
 ---------------------------------------------------------
|  x  |   станд    |  Тейлор    |Членов ряда|  Разница  |
 ---------------------------------------------------------
|-1.00|   -0.841471|   -0.841667|     2     |    0.2E-03|
.. ..
Точность вычисления ряда eps=    0.1E-05
 ---------------------------------------------------------
|  x  |   станд    |  Тейлор    |Членов ряда|  Разница  |
 ---------------------------------------------------------
|-1.00|   -0.841471|   -0.841471|     5     |    0.0E+00|
|-0.80|   -0.717356|   -0.717356|     4     |    0.6E-07|
.. ..
| 0.80|    0.717356|    0.717356|     4     |    0.6E-07|
| 1.00|    0.841471|    0.841471|     5     |    0.0E+00|
 ---------------------------------------------------------
Nmax=  5
```

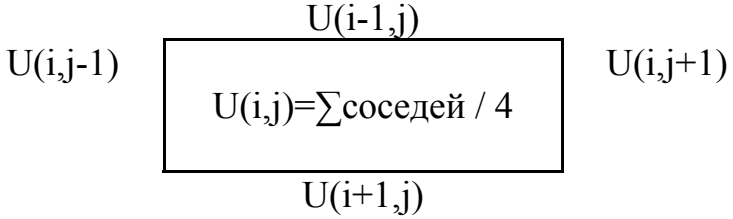
Метод Зейделя для решения уравнений в частных производных

Программы на Фортране для расчетов по матрицам n на n .

Разница в просмотре точек сечения:

- *последовательный* – просмотр всех с последующей оценкой невязки в итерации;
- *параллельный* – просмотр только точек с большой невязкой.

Сложность не в уравнениях системы – они простые



Сложность – в количестве этих уравнений n*n=287*287=82369 и в большом числе итераций 23935 для достижения заданной точности 0.003 . Степень дискретизации n влияет на достижимую точность.

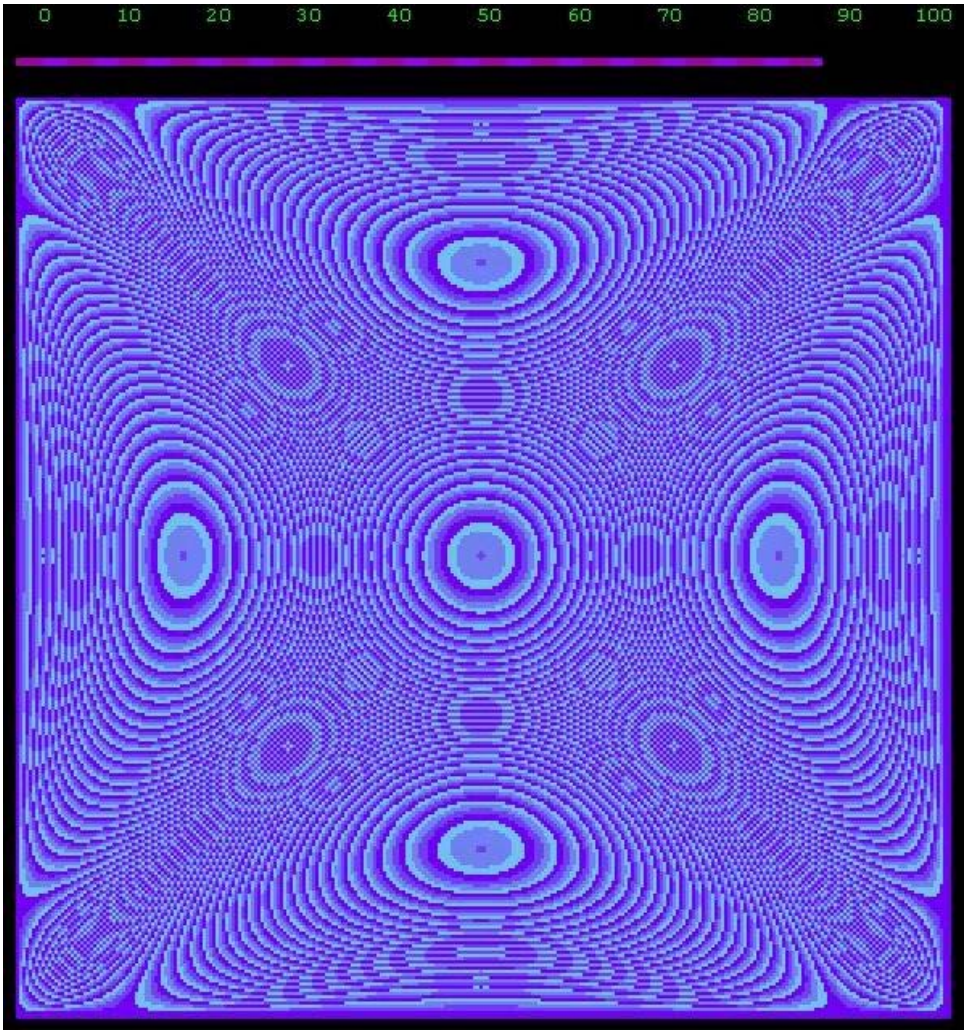


Рис. 3. Распределение температур в результате расчетов
Всего итераций: 23935 для точности 0.3E-02 n= 287 время 39.79сек.

Программа на Фортране для расчетов по методу Зейделя.

```
program ParallelvZedel
 ! внутри поначалу 0, а границы по 100 / расчет: параллельный n=287
 !  100 100 100
 !  100  0  100
 !  100 100 100
  USE IFQWIN
   implicit none
    integer,parameter::N=287,NofIterations=100*n
```

```
     real*8,save,dimension(0:n+1,0:n+1):: U
      integer i,j,c
       real :: startTimer, Timer, eps=0.003
! "Расчеты по Зейделю для дифуравнений в частных производных"
 open(6,file="parazedel.txt")
startTimer=tima()       ! стартовое время
U=0 ! внутри поначалу 0      ~[(100.*i/(n+1.),i=0,n+1)]
 U(0,:)=100;  U(n+1,:)=100;  U(:,0)=100;  U(:,n+1)=100
! а каёмочка по 100
   DO c=1,NofIterations ! итерации до eps
    forall(i=1:n,j=1:n, &
     abs(0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1))-U(i,j))>eps)  &
                  u(i,j)=0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1))
    if(all(abs(0.25*(U(0:n-1,1:n)+U(2:n+1,1:n)+ &
          U(1:n,0:n-1)+U(1:n,2:n+1))-U(1:n,1:n))<=eps))exit
   ENDDO
Timer=tima()-startTimer   ! финишное время
    if(c<100*n ) then
     write(6,1) c,eps,n,Timer
  1 format(' Всего итераций: ',i5, &
     ' для точности ',e7.1/'n=',i4,' время ',f8.2,'сек.')
     call printTemp(U,n)
    else
     write(6,*) 'итераций: ',c,'>',100*n
    endif
 close(6)

contains

subroutine printTemp(U,n)
 USE IFQWIN
  implicit none
   integer,intent(in)::n
   real*8,dimension(0:n+1,0:n+1):: U
    integer,parameter ::steper=#4000 ! шкала цветная
    integer,dimension(0:1023) :: Tcol
     integer :: i,j,d4,b
     integer*2 :: x0=10,y0=60,x1,y1,x2,y2,r=2,dummy2
  Tcol=[(ior(steper*i,#770077),i=0,1023)]
  b=n+2
  do i=0,n+1   ! матрица температур
   y1=y0+i*r; y2=y1+2*r
   do j=0,n+1
```

```
     x1=x0+j*r; x2=x1+2*r
     d4=setcolorrgb(Tcol(nint(U(i,j)*10)))
    ! цветом - до десятых
     dummy2= rectangle($GFILLINTERIOR,x1,y1,x2,y2)
    enddo
  enddo
end subroutine printTemp

real function tima(  )
 USE IFPORT
 implicit none
  INTEGER*2 hour,hminuit,hsec,h100
   call GETTIM( hour,hminuit,hsec,h100)
    tima = hour*3600+hminuit*60 + hsec    + h100*0.01
END function tima

end program ParallelvZedel
```

Программа исследования зависимости времени расчета от размерности задачи

```
program ParaSeqvZedel
! 2 расчета: 1)параллельный до n=5611 2)последовательный  до n=295
 implicit none
  integer,parameter::NofDima=27,sota=10
   integer,parameter::seqvNofDima=15,seqvsota=18 ! 15 18
    integer,allocatable,dimension(:)::Dima
     real*8 ,allocatable,dimension(:):: Timer
      integer:: k
       real*8 :: eps=0.001
 !  1)параллельный до n=287
 allocate(Dima(1:NofDima),Timer(1:NofDima))
 Dima=(/(k,k=NofDima,NofDima+NofDima*sota,sota)/)
 open(6,file="parazedel.txt")
 do k=1,NofDima
  call   paracalc(Dima(k))
 enddo
 write(6,"(i7,f8.2)") ( Dima(k), Timer(k),k=1,NofDima)
 deallocate(Dima,Timer)
 close(6)

! 2) последовательный  до n=287
 allocate(Dima(1:seqvNofDima),Timer(1:seqvNofDima))
```

```
Dima=(/(k,k=seqvNofDima,seqvNofDima+seqvNofDima*seqvsota,seqvsota)/)
 open(7,file="seqvzedel.txt")
 do k=1,seqvNofDima
  call   seqvcalc(Dima(k))
 enddo
 write(7,"(i7,f8.2)") ( Dima(k), Timer(k),k=1,seqvNofDima)
 deallocate(Dima,Timer)
 close(7)

contains

subroutine paracalc(n) ! параллельные расчеты для одной размерности n
 implicit none
  integer,intent(in)::n
   real*8,save,allocatable,dimension(:,:):: U
   logical*1,save,allocatable,dimension(:,:):: logo
   real*8 dmax, origin
     integer i,j,c
      real startTimer
allocate(U(0:n+1,0:n+1),logo(1:n,1:n))
! внутри поначалу 0, а границы по 100
 !  100 100 100
 !  100  0  100
 !  100 100 100
U=0
U(0,:)=100; U(n+1,:)=100; U(:,0)=100; U(:,n+1)=100
   startTimer=tima()    ! стартовое время
 DO c=1,10*n
  logo=.false.
  forall(i=1:n,j=1:n,abs(0.25*(U(i-1,j)+U(i+1,j) &
    +U(i,j-1)+U(i,j+1))-U(i,j))>eps)
     u(i,j)=0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1))
     logo(i,j)=.true.
  end forall
  if( all(logo) )exit
 ENDDO
   Timer(k)=tima()-startTimer ! финишное время
   !call printTemp(U,n)
 deallocate(U,logo)
end subroutine paracalc

subroutine seqvcalc(n)
```

```
! последовательные расчеты для одной размерности n
 implicit none
  integer,intent(in)::n
   real*8,allocatable,dimension(:,:):: U
    real*8 :: dmax,origin
     integer i,j,c
      real startTimer
allocate(U(0:n+1,0:n+1))
! внутри поначалу 0, а границы по 100
U=0; U(0,:)=100; U(n+1,:)=100; U(:,0)=100
 U(:,n+1)=100;   startTimer=tima() ! стартовое время
   DO c=1,100*n
    dmax=0
    do i=1,n
     do j=1,n
      origin=U(i,j)
    U(i,j)=0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1))
      dmax=max(abs(origin-U(i,j)),dmax)
     enddo
    enddo
    if(dmax<=eps) exit
   ENDDO
   Timer(k)=tima()-startTimer  ! время = финиш - старт
 deallocate(U)
end subroutine seqvcalc
 real function tima(  )
 USE IFPORT
 implicit none
  INTEGER*2 our,minut,sec,h100
   call GETTIM( our,minut,sec,h100)
    tima = our*3600+minut*60 + sec + h100*0.01
END function tima
end program ParaSeqvZedel
```

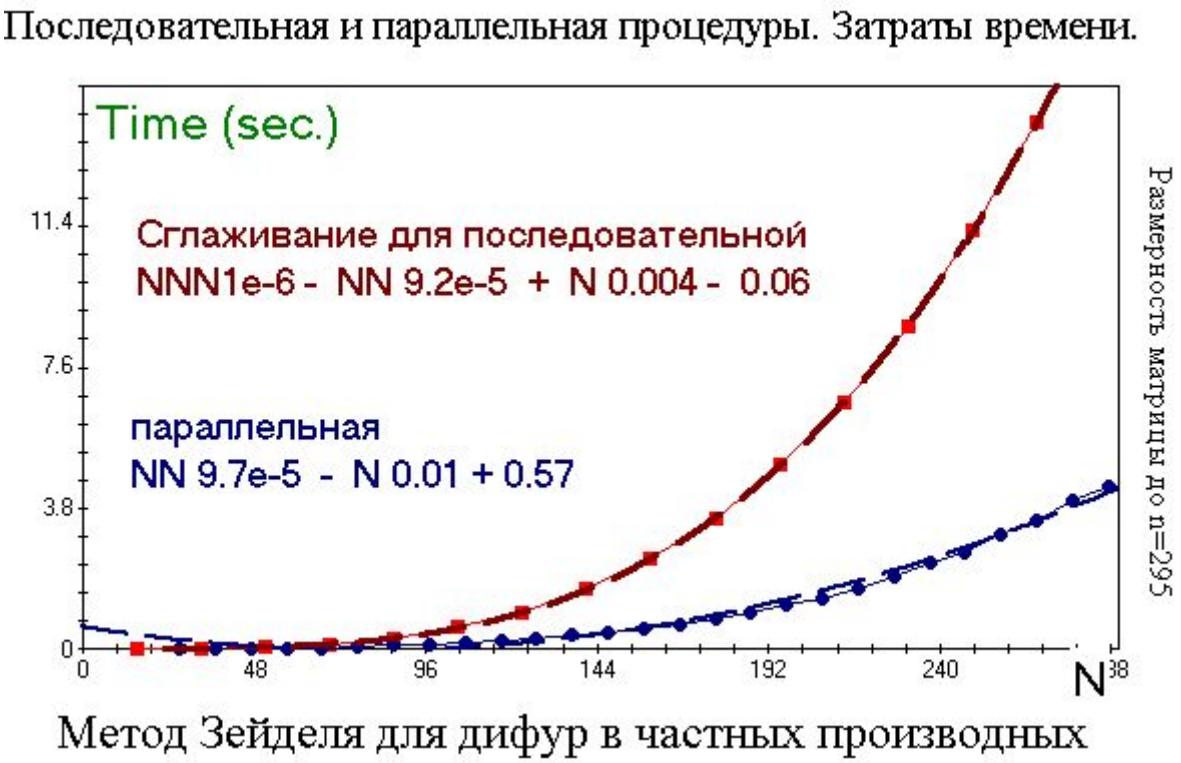


Рис.4. Сравнение времен расчета параллельной и последовательной процедур в зависимости от размерности задачи.

Глава 6. Практика параллельного программирования.

Инженер, ориентированный на инновации, может не найти готового алгоритма и многое вынужден программировать сам. Студент, специализирующийся в области фотоники и оптоинформатики, также вынужден сам осваивать многое. Практика требует работы с интегрированными оболочками типа MS Visual Studio, с интенсивной помощью на английском. Задачи 3D-моделирования заставляют пользоваться не только многоядерным ПК, но и кластерами, которые в основном работают в среде Linux, а не Windows. Работа в Linux подразумевает написание скриптов на языке Bash и использование Make-технологии. Ускорение в решении задач на кластере достигается при создании параллельных программ. Параллельное программирование подразумевает не только знание языков программирования Фортран или “C”, но и использование технологии распараллеливания OpenMP и MPI. Для ведения сложных проектов современный Фортран предлагает централизованно описывать данные, используя модульное программирование.

Кафедра ФОИ предлагает следующее, уникальное для ИТМО оборудование - кластер PHOIF, состоящий из

- 11 модулей по две материнской платы в каждом

- 22 узла (nodes) с ОС Gentoo Linux, из них 5 узлов с Windows Server (виртуально)
- 44 4-ядерных процессоров Intel Quad Xeon
- 4x44 = 176 ядер, 88 пар, каждая пара с отдельным кэшем
- 44x6+32 = 164Gb – оперативной памяти, общая шина InfiniBand 200Gb/s
- 18x200+4x250 Gb = 4,6 Tb – Дисковое хранилище
- Glasterfs – распределенная файловая система на кластере.

Работа в разнородных сетях. Удаленный доступ к кластеру

Прежде всего, сообщим структуру сети. Компьютерный класс ФОИ входит в одноранговый домен RUNNET, в котором адреса Linux-кластера (194.85.163.135) и Windows-сервера – статические, у других ПК – адреса динамические. Доступ к кластеру организован по безопасному протоколу SSH. Для удаленного доступа к командной оболочке Linux кластера можно использовать PuTTY. Windows варианты этого Telnet и SSH клиента, а также SFTP и SCP клиенты можно скачать на страничке PuTTY Download Page.

Для обмена файлами с кластером из Windows используйте, FileZilla Client (http://www.altware.ru/soft/filezilla-client), который поддерживает безопасный протокол передачи данных SFTP, или WinSCP под SCP (http://winscp.net/eng/docs/lang:ru). Для организации графического доступа к кластеру используйте GPL Free NX Client, который реализует оконный интерфейс XWindows под MS Windows, скачать его можно по ссылке http://www.nomachine.com/download.php.

Особенностью кластера PHOIF является наличие в его составе отдельных Windows серверов. Подключение к удаленному рабочему столу Windows Server вы найдете в программном меню Стандартные/Связь. Зная IP адрес сервера, и учетную запись (аккаунт) вы сможете работать на своем компьютере как на многопроцессорном сервере. Для обмена файлами с Windows сервером используйте любой FTP клиент, например, Windows Commander, единственно IP адрес с которого вы передаете файлы должен знать сервер, поскольку простой FTP протокол передачи данных не безопасен. На одном из Windows серверов, в директории public находятся материалы по курсу и подборка литературы, в основном по Fortran, C++, Java, Linux, и параллельному программированию. Там же вы найдете ссылки для запуска прикладных программ по выполнению виртуальных лабораторий.

Работа в MS Visual Studio. Компиляция, сборка и отладка

Обучение прикладному программированию будет проходить в процессе создания научных проектов в области фотоники и оптоинформатики. Microsoft Visual Studio - это универсальная

интегрированная оболочка, она предоставляет удобный текстовый редактор, интегрированный с компилятором для быстрого поиска ошибок. VS позволяет собирать приложение из программ, написанных на разных языках, например, Си и Fortran. Именно на этих языках написано большинство научных библиотек со свободным доступом. Объектно-ориентированное программирование на Java, C#, C++ не затрагивается.

Работая с Visual Studio Вы приобретаете опыт отладки

- F5 продвигает отладку кода до следующей точки останова
- F9 или мышка создают ее
- F10 выполняет одну строчку кода
- F11 - вход внутрь выполняемой функции или процедуры
- Shift-F11 выход из нее
- создание условные точки останова (Condition.. в контекстном меню останова по условию)
- проверка значений переменных и массивов в процессе отладки - навести курсор на простые переменные в точке останова
- просматривать значения массивов в окнах Watch (меню Debug/Windows/Watch в VS 2005)

Типы проектов Intel Visual Fortran Compiler. Ключи компилятора

Компилятор Fortran 95 от фирмы Intel встраивается в MS Visual Studio. Он поддерживает простые типы проектов, не использующие прикладные библиотеки. Прежде всего, это проект Console Application (*.exe), который ограничивается стандартом языка, имеет единственную точку входа PROGRAM (процедура, запускаемая при загрузке приложения в оперативную память), и не использует графику или диалоги для ввода входных параметров (ввод только из текстового файла). Это просто, полностью переносимо, но неудобно для больших проектов. Тип проекта Static Library (*.lib) – это архив откомпилированных функций, который использует только сборщик, при загрузке приложения не используется. Динамически загружаемые библиотеки отличаются от простых загружаемых (.exe) модулей множественностью точек входа (исполняемых процедур, специально выделенных для линковщика с помощью директив препроцессора), причем процесс загрузки (запуск основной точки входа DllMain) здесь отделен от исполнения, а исполняемые процедуры запускают внешние программы, используя процесс динамической сборки. В отличие от статической библиотеки, сборщик подчищает код динамической библиотеки, от всех процедур, которые не зависят функционально от точек входа. Тип проекта Win32 Application использует Win32 интерфейс, это наиболее богатый по возможностям тип проекта под Windows. Однако эти возможности не так просто изучить, и, кроме того, они не переносятся под Linux. Исполняемый (.exe) модуль сам является динамически загружаемым модулем, но имеет другую главную

точку входа, WinMain. При использовании этого типа проекта необходимо выносить в отдельные dll-модули, ту часть, которая не зависит от платформы, чтобы ее было легче переносить под Linux. В рамках Win32 приложения можно использовать не зависящую от платформы библиотеку диалогов (процедуры, начинающиеся с префикса DLG, например, DLGModal), чтобы организовать ввод параметров задачи в процессе работы загруженной программы.

Наконец, проекты QuickWin Application имеют удобный, переносимый под Linux оконный интерфейс, специально доработанный для использования научной графики в Fortran. Единственное его неудобство – невозможность использовать диалоговую библиотеку.

Важной в MS Visual Studio является возможность создавать разные конфигурации проектов в рамках Windows - под 32-разрядную (Win32) и под 64-разрядную (x64) ОС. Поскольку на рабочих станциях в компьютерном классе и на сервере установлены Windows с разной разрядностью, то нам это понадобится. Компилятор позволяет создавать 64 разрядные приложения на 32 битовой машине в классе, но исполняться они могут только на сервере. Другая возможность – изменение конфигурации. Помимо стандартных конфигураций Release, предназначенной для компиляции отлаженного и оптимизированного модуля, и Debug, необходимой для запоминания отладочной информации в модуле, мы будем изменять длину чисел с плавающей точкой, заданную операторами описания типа по умолчанию (real, double precision, complex, double complex). Это важно для научных расчетов, поскольку позволяет оперативно менять точность всех операций сразу. Для того чтобы не менять программу в зависимости от используемой конфигурации необходимо ограничить использование операторов описания типа с явно заданной длиной, например, real(4) :: *var*. Для обозначения дополнительных конфигураций для разных платформ используется постфикс r32 – для конфигурации с точностью, стандартной для Фортрана, и r64 - для конфигурации с удвоенной точностью (причем double precision в этом случае будет иметь четверную точность, и занимать 16 байтов). Кроме этого, в конфигурациях используются постфиксы OMP для использования директив распараллеливания потоков OpenMP, и MPI – для использования библиотеки MPI.

Точность переменных по умолчанию, как и тип платформы, задаются с помощью ключей компилятора, которые передает ему Visual Studio при запуске в командной оболочке. Набор ключей для заданной конфигурации можно посмотреть в свойствах проекта (Project/Properties/Fortran/Command Line). Основные ключи необходимо знать, поскольку их же нужно будет использовать в сценариях по запуску компилятора под Linux. Выполните `ifort -help`, чтобы узнать все флаги компилятора под Linux.

Операционная система Linux.

Работа в коммандной строке

“Линуксоидами” сегодня становятся все большее количество молодых людей. И дело тут не только в открытости кода бесплатной ОС, и в отсутствии “сюрпризов” логики от Микрософт. Linux - наиболее надежная, и масштабируемая система. Linux сегодня устанавливают на ноутбуки, чтобы иметь более разнообразный набор бесплатного ПО, которым можно управлять по своему усмотрению. Linux используется для поддержки сайта президента. Без Linux сегодня невозможен многопроцессорный кластер с быстрой связью на основе шины InfiniBand. Обратная сторона повышенной надежности и масштабируемости – сложная, и затратная по стоимости специалистов настройка и поддержка. Однако научный работник должен уметь быстро овладевать смежной специальностью. Выигрыш не только в моделировании научных задач любой сложности, но и профессиональная работа с базами данных, или с современными инструментами разработки объектно-ориентированных коммерческих приложений.

Для пользователя кластера надо знать основные команды Unix:
ls – сведения о директории (аналог dir под Windows),
cat – замена copy, rename, type под Windows,
ps – сведения о запущенном процессе,
top – script, написанный на основе ps, о его id,
kill – остановить, или приостановить, запущенный процесс,
kill –STOP id; kill –TERM id; kill –KILL id; **(**killall – не надо**)**
screen **–** запуск терминалов (окон с очередью команд),
с возможностью выхода и последующего продолжения,
ssh n4 – вход на машину 4-го узла (всего 21 + HEAD)
exit **–** выход из режима терминала.

С помощью них, и Midnight Commander (команда mc) вы сможете проверять свои файлы, запускать и снимать задания. Синтаксис, флаги и предназначение этих, и всех других команд ОС вы сможете узнать из системы помощи Unix. Запускайте man *ssh*, чтобы получить справку по команде ssh. Большая часть команд Linux эмулирована по Windows, см. например, *.exe файлы в директории /usr/bin/ на Windows сервере. Сделав их общедоступными, вы сможете писать сценарии для Linux под Windows.

Язык сценариев командной оболочки Bash

Bash называют и язык написания сценариев (скриптов), и командную оболочку (Shell из Unix). Bash намного богаче и функциональнее, чем Batch под Windows, хотя в гибридных сетях

приходится использовать оба языка. Скриптовый язык предназначен для запуска команд интерпретатора, внешних командных строк Linux. Тип всех переменных – строковый. Сценарий при запуске тоже является командой, и у него есть аргументы, поэтому в языке определены позиционные параметры:

$1, $2, ..$9 - аргументы по номеру,
$#, $@ - число аргументов и все аргументы сценария списком,

Другой пример специального типа переменных – это локальные переменные, т.е. переменные с областью видимости внутри процедуры, определенной в сценарии, или во всем сценарии, если переменные там и определены (в это смысле сам сценарий можно понимать как процедуру с внешним вызовом).

Вызываемому сценарию передаются переменные командной оболочки. Их называют переменными окружения. Переменные задаются по именам, а их значения подставляются с помощью прямых ссылок:

var=’some string’ # определение локальной переменной,
$var, ${var}, ${PATH} # прямые ссылки на var и на переменную окружения,

Ссылки также бывают косвенными, когда внутри фигурных скобок пишется ссылка на переменную:

args=$# ; lastarg=${!args} # косвенная ссылка на последний аргумент.

Внутри фигурных скобок можно выполнять команды, так, например, если вы не знаете, определена ли у вас в сценарии переменная username, вы можете написать ссылку на нее со значением по умолчанию, которое подставляется при выполнении внешней команды Unix whoami:

echo ${username-`whoami`} – параметр со значением по умолчанию.

Как же работать с числами в Bash? Очень просто, надо применять внешнюю команду let:

v=2; let “v+=2 ”; echo “v=$v” # => (v=4): let - команда арифметики

Как видите, эта команда проводит разбор (parsing) арифметических выражений, вычисляет их, и преобразует значения всех переменных обратно в строковые.

Теперь об условных конструкциях в Bash.

```
if comm; then  tComm   # comm, tComm, fComm – это функции, или
else fComm             #  команды, внешние или внутренние
fi
```

В данной конструкции comm – это, как правило, вызов внешней команды, которая должна возвращать ‘0’, если завершилась успешно. Таким образом, в данной конструкции истина кодируется ‘0’, что надо учитывать при написании скриптов. Чтоб улучшить понимание таких конструкций, в Bash пошли на изменение синтаксиса вызова наиболее употребительной внешней команды test (руководство по ней доступно по man test). Команду test заменяют квадратными скобками. Два эквивалентных вызова команды с флагом:

```
[ -z ‘name’ ];  test –z ‘name’  # замена  test -  команды проверки строк,
[[ $a == z* ]]    -  команда test с расширенными возможностями сравнения,
```

Два распространенных примера ее использования с целью проверки, определена ли переменная ‘name’:

```
[ -n ‘name’ ]  &&  echo ‘Info: variable $name is defined’ # условная печать
if [ -z ‘name’ ]; then  tComm $name   # условная конструкция test
fi
```

Аналогично строится синтаксическая замена команды let, причем оба последующих оператора возвращают значение ‘0’ (истина), даже если арифметическое значение выражения - не ноль (в известном смысле двойные круглые скобки противоположны двойным квадратным):

```
let “$var+7”; (($var+7 ))  #  замена  let -  команды вычисления выражений,
```

Наконец, в обеих командах, test и let, можно использовать операции сравнения, строковые и арифметические соответственно:

```
 [ "$a" -le "$b" ];  (($var <= 7 ))   # операции сравнения.
```

Написание собственных сценариев - это лучший путь к изучению языка сценариев**.** Изложенного материала достаточно, чтобы разобраться в приведенном ниже, полезном сценарии. Подробности Bash, и другие примеры можно найти в
http://www.opennet.ru/docs/RUS/bash_scripting_guide/

```
#!/bin/bash
# приведем лучший метод:
#
# find "somedir" -type l -print0|\
# xargs -r0 file|\
```

```
# grep "broken symbolic"|
# sed -e 's/^\|: *broken symbolic.*$/"/g'

#Если скрипт не получает входных аргументов,
# то каталогом поиска является текущая директория
#В противном случае, каталог поиска задается из командной строки
####################
[ $# -eq 0 ] && sdir=`pwd` || sdir=$@

#Функция linkchk проверяет каталог поиска на наличие
# в нем ссылок на несуществующие файлы, и выводит их имена.
#Если анализируемый файл является каталогом,
# то он передается функции linkcheck рекурсивно.
##########
linkchk () {
 for element in $1/*; do
  [ -h "$element" -a ! -e "$element" ] && echo
\"$element\"
  [ -d "$element" ] && linkchk $element
# '-h' проверяет символические ссылки, '-d' -- каталоги.
 done
        }
#Вызов функции linkchk для каждого аргумента
# командной строки, если он является каталогом.
#Иначе выводится сообщение об ошибке
# и информация о порядке пользования скриптом.
################
for directory in $sdir; do
    if [ -d $directory ]
    then linkchk $directory
    else
        echo "$directory не является каталогом"
        echo "Порядок использования: $0 dir1 dir2 ..."
    fi
done
exit 0
```

Внешние и встроенные в интерпретатор, команды Bash отличаются друг от друга, даже если они имеют одинаковое имя и предназначение. В скриптах лучше использовать внутренние команды, поскольку они быстрее работают. Приведем здесь самые важные команды Bash, знать которые необходимо:

```
echo "внутренняя команда \"echo\".“
/bin/echo “Внешняя команда /bin/echo.“

# read; printf – интерактивный Вввод и Вывод на консоль
read var < commands.sh; printf var > text.log # переопределение
ввода/вывода

cd dir; pwd            # переход в каталог, текущий каталог
pushd; popd; dirs   # работа со стеком последних каталогов

set; unset  # установка и сброс внутренних переменных Bash
export      # установка переменных, доступных родительским оболочкам.

source file.sh;  . file.sh  # это два способа вызова команды.
# Здесь ‘.’ – это команда, запуск сценария из файла в той же оболочке
exec file.sh   # замещение текущего сценария новым (дочерним)
# (последняя команда тек. сценария, например, перезапуск после
засыпания)

Внешние команды Bash в Linux (доступны под Windows в C:\usr\bin):

/bin/echo  # полный путь для вызова одноименных команд; поиск в $Path,

ls;   ls -R  # вывод "списка" файлов, в том числе рекурсивно,
cat file1 file2 > file;   tac … # вывод файлов в прямом и обратном порядке,

cp file1 file2; mv; rm  # копирование, перемещение и удаление файлов,
mkdir; rmdir  # создание и удаление каталогов (удаляет только пустой),

ln; ln -s  # создает жесткую и мягкую (soft) ссылку (отличаются при уд-и).
           # Позволяет задавать несколько имен одному и тому же файлу.

chmod   # Изменяет атрибуты доступа (rwx) к файлу (владелец,группа,все)
chmod +x file     # доступен для исполнения всем пользователям,
chmod u+s file   #  для всех - привилегии владельца файла,

 chmod 644 file  # право на чтение/запись–владельцу, на чтение–группе,
всем
                  # (rw_,r__,r__) =>(110,100,100) =>(644) – восьмеричное
число
                  # (см. меню Файл/Права в Midnight Commander)
# Самые распространенные внешние команды:
```

find dir # поиск в файловой системе, начиная с каталога dir
grep <рег_выражение> file1… # поиск в файлах вхождений регулярного выражения

Команда chmod под Linux очень важна, поскольку она управляет правами доступа к файлам, а по умолчанию право на запись вновь создаваемых файлов – только у владельца. Когда вам захочется передать ваши файлы для использования, вам придется применить эту команду к каждому файлу, разрешив его изменение группе, или всем пользователям. Чтобы делать это автоматически для всех файлов в директории (и рекурсивно во всех поддиректориях) нужно написать сценарий. Советую сделать это самостоятельно, используя, например, сценарий проверки битых ссылок, приведенный выше.

Технология управления проектом Make

В Windows компиляция, сборка, отладка выполняется в интегрированной среде разработки (IDE), прежде всего Visual Studio. В Linux традиционно используют не графические оболочки, а текстовые интерпретаторы команд. Если вы будете писать универсальные сценарии только с этой целью, то зря потеряете время. Технология GNU Make позволяет не только выполнять названные рутинные задачи, но и другие, более продвинутые. Это более надежно, и оставляет большую свободу выбора. Например, используя внешние инструменты разбора синтаксиса, можно построить свой объектный браузер, позволяющий, например, менять имя всех вызовов метода данного класса. А можно просто поддерживать свои собственные рутинные операции, например: выборочная очистка, копирование, сохранение (Backup), передача по сети, в том числе с посылкой почты.

Make интерпретирует правила исполнения команд, и выполняет их при соблюдении определенных условий. Правило состоит из цели, пререквизитов, и списка команд. Цель – это обычно файл (или файлы, в случае множественной цели), который является результатом выполнения команд, пререквизиты – это файлы из которых строится цель, они появляются как аргументы в списке исполняемых команд. Список команд правила выполняется только тогда, когда цель устарела по сравнению с пререквизитами, то есть время последнего изменения цели – меньше, чем время последнего изменения одного, или нескольких пререквизитов. Названное правило уменьшает количество запусков команд. Например, явное правило

```
fname.obj : fname.f90  fname.fd
            ifort  -c -o: fname.obj  fname.f90
```

задает команду компиляции фортрановского файла fname.f90 с помощью компилятора Intel, причем команда будет выполняться, только если объектный модуль fname.obj отсутствует (точнее, если интерпретатор не может его найти), или вы изменили один из файлов, fname.f90 или fname.fd, после последней компиляции fname.f90. Синтаксис правила понятен из примера, главное, о чем необходимо постоянно помнить: в пределах строк с командами действует синтаксис написания команд, и строчки с командами начинаются с невидимого символа \Tab, табуляция, в пределах остальных строчек, в том числе тех, которые начинаются с пробелов, действует синтаксис интерпретатора. Команды должны соответствовать операционной системе, в которой они запускаются. Это могут быть и Linux, и Windows, в которой также существует вариант GNU make (см. файл make.exe в /usr/bin/, который не надо путать с nmake.exe от фирмы Microsoft).

Приведем и другое явное правило, для сборки проекта, которое использует предопределенные макросы:

```
name.exe : fname1.obj  fname2.obj
        link  -o:$@    $^
# $^  заменяется на список пререквизитов, # $@ заменяется на цель
```

Если однотипных явных правил становится слишком много правильнее применить обобщающее, неявное правило. Приведем неявное правило с множественными целями, описывающее в Фортране команду трансляции файла с определением модуля:

```
%.obj  %.mod  :  %.f90  %.fi  %.fd   # $<  первый пререквизит из списка
        ifort  -c -o:$@    $<
```

Оно демонстрирует применение шаблона %, который заменяет собой любое имя файла. Неявное правило применяется, только если не найдено подходящее явное правило. Цели подчиненных правил могут использоваться как пререквизиты основных. В результате определения таких зависимостей получается цепочка правил, которая должна быть деревом, т.е. в ней не должно быть рекурсий. Все дерево, начиная с начальной цели, проверяется на предмет необходимости применения правила. Если такая необходимость есть, выполняются команды последнего из проверенных правил, после чего проверка повторяется. Если такой необходимости нет – не делается ничего. Начальных целей, вместе с деревьями зависимых правил, может быть несколько. Начальная цель для проверки правил задается в аргументе команды запуска интерпретатора make. По умолчанию, в отсутствии такого аргумента, начальной целью является цель первого определенного правила. Обычно это абстрактная цель all. Абстрактные цели, в отличие от других, не являются именами

файлов, и не проверяются на предмет наличия таких файлов на диске. Команды правила с абстрактной целью выполняются всегда, если они появляются в цепочке зависимости от начальной цели. Абстрактные цели должны быть определены как пререквизиты правила определения абстрактных целей (.PHONY). Приведем пример абстрактных правил для командной оболочки Windows:

```
.PHONY: all echo warn
all : echo warn      # повтор цели приводит к накоплению
пререквизитов
all : name.lib name.exe      # порядок важен !

clean :              # общепринятая цель для перекомпиляции
        del *.obj    # команда оболочки Batch в Windows (rm в Linux)

echo :               # цель для вывода на консоль в целях отладки
@echo name = $(name)   # команда оболочки echo
```

Наконец, о синтаксисе языка интерпретатора make. Это полноценный скриптовый язык с набором операторов, похожим, но не совпадающим с Bash. В частности, ссылка на переменную здесь ограничивается круглыми, а не фигурными скобками. Есть много возможностей языка make, которые отсутствуют в Bash, с ними можно ознакомиться на страничке http://www.gnu.org/software/make/manual/. Здесь отметим лишь два типа переменных, которые задаются разным типом присваиваний. Сравним три несвязанные последовательности операторов, приведенные в столбцах таблицы

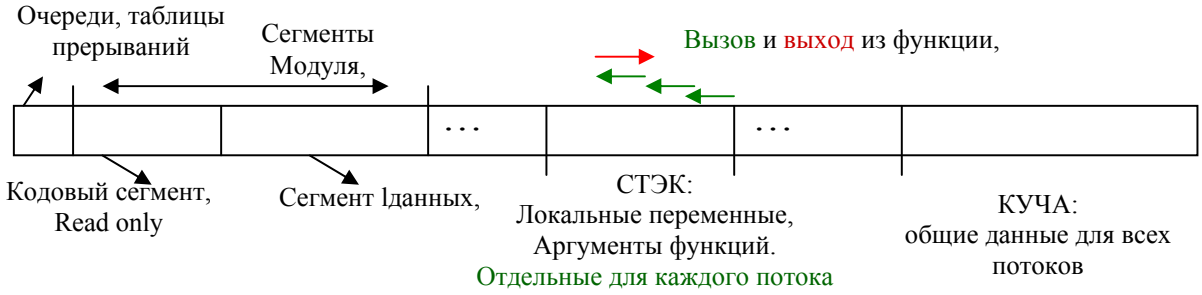
x = N.o T.c y = $(x) x = N.a T.c echo $(y) # N.a T.c	x = N.o T.c y = $(x) x = N.a T.c echo $(y: %.a=%.o) # N.o T.c	x := N.o T.c y := $(x) x := N.a T.c echo $(y) # N.o T.c

В обычном операторе присваивания ‘=’ ссылки на переменные в запоминаемом выражении в правой части оператора не подставляются. Они запоминаются как ссылки, или как ссылки с правилами подстановки. Поэтому в выводе оператора echo в первом столбце – значение переменной x, присвоенное в третьей строчке. Во втором столбце вывод изменился, поскольку мы применили в ссылке подстановку с использованием шаблона %. Другой оператор присваивания ‘:=’ подставляет все ссылки в момент присваивания, поэтому результат echo в третьем столбце более привычен.

Часто встречается еще одна форма присваивания переменной, это присваивание с проверкой. Оператор name?=value выполняется, только если переменной name не было присвоено значение до этого. Такое присваивание удобно выполнять, когда не нужно переопределять значение переменной окружения, или переменной, заданной в аргументе команды make, но, все же, необходимо задать значение по умолчанию, если переменная не определена.

Глава 7. Методики распараллеливания задач.

Прежде, чем разбираться с методиками распараллеливания задач, нужно четко понимать, как распределяется оперативная память задачи, и кому она логически принадлежит. Где в оперативной памяти расположены разные типы переменных. Знание этого очень помогает и в отладке программ. На рисунке представлена схема распределения виртуального адресного пространства модуля, с адресацией слева направо.



Процесс, как программа, выполняющаяся в собственном адресном пространстве, обладает всеми своими элементами: и кучей, где располагаются глобальные переменные, и стеком, где находятся локальные переменные, и формальные параметры вызова функции. На виртуальное адресное пространство индивидуального процесса проецируются и общие таблицы системных вызовов.

Поток запускается внутри процесса, в том числе главный поток процесса, начинающий и оканчивающий его исполнение. Потоки разделяют общее адресное пространство процесса, размещают, и очищают данные в общей куче, но имеют собственный стек вызовов исполняемых процедур, куда записываются локальные переменные, и аргументы процедуры при каждом ее вызове.

Из следующей таблицы можно понять, где размещаются массивы, заданные в Фортране, в зависимости от типа описания и области видимости. В последней колонке таблицы показано, когда очищается память, отведенная под массивы, если она не очищается явно, с помощью *deallocate*.

типы массивов,	область видимости,	размещение,	очистка
явно заданные, Arr(-3:10)	глобальная (модуль)	сегмент данных	после выхода из программы
явно заданные, Arr(-3:10)	локальная (функция)	СТЭК	после выхода из функции
Неявно заданные, allocatable:: Arr(:)	локальная (функция)	КУЧА	после выхода из функции
Неявно заданные, pointer:: Arr(:)	глобальная (модуль) локальная (функция)	КУЧА	после выхода из программы

Разные модели параллельного программирования классифицируют в зависимости от способа организации доступа к кодовому сегменту, и сегменту данных. В прикладных задачах распараллеливают наиболее критичные части кода, в этом смысле каждая параллельная нить исполнения имеет дело с одним и тем же кодом, поэтому можно выбирать из двух моделей распараллеливания: Single Instructions, Multiple Data (SIMD), или Single Instructions, Single Data (SISD).

Рассматриваются две основные методики распараллеливания задач. Технология OpenMP сейчас используется для распараллеливания потоков, которые в процессе своего исполнения имеют дело с общими данными, если только каждый из потоков не завел свои собственные, т.е. это в основном SISD модель. Библиотека вызовов интерфейса MPI предназначена для распараллеливания процессов, каждый из которых имеет собственные данные, поэтому это распараллеливание по модели SIMD.

Распараллеливание в рамках Win32 API

Обучение практическому распараллеливанию удобнее начинать в рамках Win32 API. Основные вызовы kernel32 модуля можно найти в справке MSDN, но лучше почитать о них в первоисточнике [Джеффри Рихтер. «Windows для профессионалов (Программирование в Win32 API) » Microsoft Press, M.1995]. Разберем пример на языке Си, в котором вычисляется число π в нескольких потоках:

```
#include <windows.h>
#include <stdio.h>
#include "omp.h"

const int gNumSteps = 200000000, gNumThreads = 8;
double gStep = 0.0, gPi = 0.0;
CRITICAL_SECTION gCS;

DWORD WINAPI threadFunction(LPVOID pArg)
```

```
{
    int myNum = *((int *)pArg);
    double partialSum = 0.0, x;   // local to each thread

    for ( int i=myNum; i<gNumSteps; i+=gNumThreads )
    {    //compute partial sums at each thread
        x = (i + 0.5f) / gNumSteps;
        partialSum += 4.0f / (1.0f + x*x);
    }
    // add partial sum to global final answer
    EnterCriticalSection(&gCS);
      gPi += partialSum * gStep;
    LeaveCriticalSection(&gCS);

    return 0;
}

int main()
{
    HANDLE threadHandles[gNumThreads];
    int tNum[gNumThreads];
    double time_begin = omp_get_wtime();

    printf("Computed value of Pi: ");

    InitializeCriticalSection(&gCS);
    gStep = 1.0 / gNumSteps;
    for ( int i=0; i<gNumThreads; ++i )
    { tNum[i] = i;
      threadHandles[i]=CreateThread(NULL,0,// Stack size
        threadFunction,   // Thread function
        (LPVOID)&tNum[i],// Data for thread func()
        0, NULL);         // Returned thread ID
    }
    WaitForMultipleObjects(
        gNumThreads,threadHandles, TRUE,INFINITE);
    DeleteCriticalSection(&gCS);

    printf("%12.9f\n", gPi );
    printf_s("time = %.16g\n",omp_get_wtime()-
time_begin);
}
```

Не вдаваясь в подробности математического алгоритма, обратим лишь внимание на то, что здесь вычисляется конечный ряд с `gNumSteps` членами. Для того, чтобы его вычислить запускаются `gNumThreads` потоков, в которых вычисляются частичные ряды с шагом `gNumThreads`. Потоки создаются и запускаются с помощью Win32 функции `CreateThread`, которой передаются ссылки на функцию запуска `threadFunction` и ее аргумент - номер потока и члена ряда. Далее, с помощью Win32 функции `WaitForMultipleObjects` основной поток исполнения останавливается, и ждет пока не закончится исполнение функций вычисления частей ряда во всех запущенных потоках с индетефикаторами, расположенными в массиве `threadHandles`. Функция вычисления ряда `threadFunction` в каждом из потоков вычисляет ряд в своей локальной переменной `partialSum`. До тех пор пока изменяются только локальные переменные, потоки не мешают друг другу, даже учитывая, что они используют общую память для чтения значений глобальной переменной `gNumSteps`. Как только потоки собираются изменить глобальную переменную `gPi` (чтобы суммировать частичные суммы ряда) необходимо применить средства синхронизации потоков.

Средства синхронизации потоков. Возможность взаимоблокировок

Смысл синхронизация потоков – в организации последовательного доступа к общему ресурсу, будь это отрезок кода, который невозможно логически распараллелить (например, отрезок чужого кода, черный ящик), или последовательное устройство чтения/записи. Когда один из конкурирующих потоков занял ресурс, другие потоки, претендующие на него должны останавливаться, и ждать, пока кто-то не сигнализирует, что ресурс свободен. Это означает необходимости поддержки очередей процессов, и обмена сообщениями между ними. В Windows это делается опосредованно, через вызовы Win32 API. Однако, несмотря на наличие посредника, тупиковые ситуации все же возможны (такой уж это посредник). Тупиковая ситуация (deadlock) – это когда один поток ждет сигнала от другого потока, а тот, в свою очередь, ждет сигнала от него, а посредник не знает, кто должен быть первым. Наличие тупиковых ситуаций существенно усложняет отладку параллельного кода, хотя уже есть и продвинутые отладчики, например, Intel VTune Perfomance Analyzer.

Критические секции. События. Семафоры. Mutex

Кратко перечислим классические инструменты синхронизации потоков, которые используются в Wni32 API. Более подробно, с примерами кода, теорию параллельного программирования можно

прочитать в упомянутой выше книжке. Самый простой способ защиты кода от одновременного использования – поместить его в критическую секцию, как это делается в конце функции `threadFunction` выше. При этом, перед вызовом `EnterCriticalSection` система организует очередь потоков, а вызов `LeaveCriticalSection` сигнализирует, что система может продолжить исполнение следующего потока. Саму структуру критической секции, которую использует система, необходимо инициализировать перед, и очистить после исполнения всех потоков (см. вызовы в `main`).

При использовании событий уже сам программист определяет место и условия приостановки и возобновления выполнения потоков. При этом один из потоков использует функцию WaitForSingleObject(event), ожидая выполнения события в другом потоке. Таким событием может быть, например, асинхронная обработка реакции пользователя. Поток, в котором это событие произошло, вызывает функцию PulseEvent(event). При этом система лишь посылает сообщения всем потокам, открывшим это событие с помощью OpenEvent(event), возобновляя их исполнение.

Семафоры – наиболее общее средство синхронизации потоков. Они лишь ограничивают одновременный доступ к ресурсу. В вызове CreateSemaphore(num) аргумент задает количество потоков для одновременного исполнения на этом семафоре. Сами потоки, вызывая функцию WaitForSingleObject(semaphor), проверяют открыт или закрыт семафор, вставая в очередь на исполнение. После окончания обработки кода, привязанного по смыслу к семафору, поток должен вызвать ReleaseSemaphore(semaphor), освобождая очередь следующему. За смысловой нагрузкой синхронизации потоков по светофору должен следить программист.

Mutex – это одноместный семафор, наиболее употребляемый инструмент синхронизации. Mutex синхронизирует процессы, и защищает от взаимоблокировок. Он предназначен для обеспечения последовательного доступа к ресурсу. Таким ресурсом может выступать, например, массив, от значения которого зависит функция запуска потока. Структуру данных mutex, как и других инструментов синхронизации, обычно называют по названию ресурса для того, чтобы было удобнее понимать смысл вызовов Win32 API, предназначенных для управления им:

1. CreateMutex – ассоциация с ресурсом по имени.
2. WaitForSingleObject(mutex) - завладеет ресурсом или нет.
3. ReleaseMutex(mutex) – освобождение занятого ресурса.

Стандарт OpenMP для распараллеливания между потоками в процессе

Для распараллеливания в рамках стандарта *Open Multi-Processing* можно использовать как вызовы библиотечных процедур, так и специальные директивы препроцессора. Последний способ предпочтительнее, поскольку оставляет возможность оттранслировать основной текст программы без использования библиотеки OpenMP. В небольшом фрагменте кода мы используем обе возможности.

```
nProcs=OMP_GET_NUM_PROCS() !number of processes available
nMax = OMP_GET_MAX_THREADS()  ! max number of threads
print*,'nProcs=',nProcs,',nMax=',nMax,',numThrs=',num

if(nProcs==nMax) call OMP_SET_NUM_THREADS(3)
nProcs=OMP_GET_NUM_PROCS()
nMax = OMP_GET_MAX_THREADS()
num = OMP_GET_NUM_THREADS()  ! number of threads executing
print*,'nProcs=',nProcs,', nMax=',nMax,',numThrs= ',num

sum = 0; time_begin = OMP_GET_WTIME()

!$OMP PARALLEL ! num_threads(4)
!$OMP DO private(x)reduction(+:sum)
do i=0,numSteps-1
   x=(i+0.5d0)*step/numSteps; sum=sum+4/(1d0 + x*x)
end do
num = OMP_GET_NUM_THREADS();
print*,'num  Thrs = ',num ! threads executing
!$OMP END PARALLEL

pi = sum / numSteps; time_end = OMP_GET_WTIME()
print *, 'pi    = ',pi
print *,'wallclock time = ', time_end - time_begin, '
seconds'
```

Функция **OMP_GET_NUM_PROCS** возвращает число процессоров, доступных программе, точнее общее число ядер, поскольку ядро процессора, с выделенным кэшем для исполнения команд, операционной системой воспринимается как отдельный процессор. Так, на кластере внутри одной машины работает два четырехядерных прцессора, итого восемь процессоров с точки зрения системы. Функция

OMP_GET_MAX_THREADS возвращает максимальное число потоков в процессе, доступных для параллельного исполнения. Число потоков логически не связано с числом процессоров, его можно установить любым с помощью процедуры **OMP_SET_NUM_THREADS**. Другой способ – задавать переменную окружения **OMP_NUM_THREADS**, тогда ваша программа не будет явно зависить от библиотеки OpenMP. По умолчанию, если не задана переменная окружения, число потоков равно числу доступных процессоров, поэтому в тексте примера использован условный вызов процедуры **OMP_SET_NUM_THREADS**. Это позволяет и здесь использовать переменную окружения. Функция **OMP_GET_NUM_THREADS** возвращает число потоков, исполняющихся под управлением OpenMP в данном отрезке кода. На Intel Core 2 Duo вывод программы выглялит так

```
nProcs =        2 , nMax =        2 , num Thrs =       11
nProcs =        2 , nMax =        3 , num Thrs =        1
num  Thrs =       3
num  Thrs =       3
num  Thrs =       3
pi   =   3.14159265358986
wallclock time =   3.625000     seconds
```

Как видите, число потоков исполнения изменилось только внутри секции кода, ограниченной директивой **!$OMP PARALLEL**. Так и задается отрезок кода, предназначенный для распараллеливания. Директива **PARALLEL**, как и все другие директивы OpenMP, может иметь аргументы. С помощью аргумента `num_threads(2)`, закомментированного в тексте, можно еще одним способом изменить число потоков, которые порождаются в начале, и заканчиваются в конце секции **PARALLEL**. До тех пор пока внутри секции не появляются дополнительные директивы, во всех потоках выполняется один и тот же код (Single Instructions), что не имеет большого смысла. Чтобы явно задать разные куски кода для разных потоков используйте директиву

!$OMP SECTIONS, в тексте используется директива **!$OMP DO**. Она позволяет возложить на процедуры библиотеки всю рутинную работу по распараллеливанию цикла **DO**. Первый аргумент директивы **private(x)** описывает переменную *x*, как локальную переменную потока. При этом, в начале директивы значение переменной в основном потоке (master thread) копируется в локальную память потока, а при выходе из секции локальные значения теряются. Второй аргумент директивы **reduction(+:sum)** предназначен для объявления переменных в основном потоке, которые доступны для записи из потоков параллельного исполнения. Как мы объясняли здесь, именно такие ситуации требуют синхронизации потоков.

В данном случае библиотека сама создает критическую секцию для всех операторов, требующих суммирования в переменной *sum*. Есть и другие способы синхронизации потоков в OpenMP. Директива **Critical** явно задает критическую секцию для последовательного исполнения кода. Директива **Barrier** синхронизует заданную часть потоков, приостанавливая их, пока к точке кода, где она распложена не подойдут все назначенные потоки. После этого они продолжают параллельное асинхронное исполнение.

Распараллеливание с помощью директив OpenMP очень удобно, безопасно, и имеют перспективу дальнейшего развития, поскольку этот стандарт внедряется фирмой Intel. Другие инструменты Intel важны на этапе оптимизации программы с параллельными исполнением. Перечислим их:

1. Обнаружение ошибок с помощью Intel Thread Checker;
2. Оптимизация исполнения с помощью Intel Thread Profiler;
3. Отладка и оптимизация параллельного исполнения потоков с помощью Intel VTune.

Главный недостаток технологии OpenMP в том, что она рассчитана на использование только внутри одной машины, точнее в системах с общей оперативной памятью. Для того чтобы программировать для кластера, пользуясь эффективностью массового распараллеливания, необходимо использовать многопроцессорный интерфейс MPI.

Библиотека MPI для распараллеливания между процессами в кластере

Суть массового распараллеливания в том, чтобы процессы с почти одинаковым кодом работали на разных машинах. По большей части они обладают не только своим виртуальным адресным пространством, но и физическая оперативная память у них разная. Любой вычислительный процесс состоит из итераций, поэтому периодически параллельные процессы должны обмениваться обновленными частями оперативной памяти. В физике такие информационные зависимости выделить просто. Например, поле излучения, распределенное в поперечном сечении пучка, дифрагирует в процессе своего распространения. При этом, первоначально узкий пучок излучения расплывается, занимая всю область счета. Оперативная память каждого из процессов занимает только часть области счета, но на каждой из итераций при моделировании процесса распространения процессы должны обмениваться обновленными значениями поля с соседними областями.

Интерфейс передачи сообщений между параллельно исполняемыми процессами (*Message Passing Interface*, MPI) описывает соглашения о вызове процедур передающих эти сообщения. Стандарт MPI 1.1 содержит более 120 функций. Рассмотрим только ту часть из них, которые необходимы для написания простейших программ. Все имена констант и функций на-

чинаются с префикса `MPI_`, и для того, чтобы использовать их и в Си, и в Fortran необходимо их подключить при помощи **`include`** `'mpif.h'` .
.

Группы процессов, коммуникаторы.
Обмен сообщениями между процессами

Все процессы в MPI разбиваются на группы, способные обмениваться сообщениями с заданным идентификатором. Идентификатор такой группы, или даже сама группа называется коммуникатором. Чтобы послать конкретному процессу сообщение необходимо знать два его основных атрибута: *коммуникатор (группа) и номер процесса в коммуникаторе (группе), от 0 до n – 1*. Каждый процесс может одновременно входить в разные коммуникаторы. Существует один коммуникатор, определенный по умолчанию с идентификатором **`MPI_COMM_WORLD`**, это группа всех процессов, доступных MPI библиотеке в рамках одного запуска группы процессов из командной оболочки командой mpirun.

Сообщение, посланное от одного процесса к другому, обладает атрибутами сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Для работы с атрибутами сообщений введен массив **`MPI_STATUS`**, с его помощью можно, например, задать идентификатор процесса, от которого ожидается сообщение. В последнем аргументе (в Си – возвращаемое значение функции) большинство функций MPI возвращают информацию об успешности завершения. В примерах вызовов здесь используется целая переменная `IERR`. В случае успешного выполнения функция вернет значение **`MPI_SUCCESS`**, иначе — код ошибки.

Инициализация параллельной части программы осуществляется функцией **`MPI_INIT`**. Все другие функции MPI могут быть вызваны только после вызова **`MPI_INIT.`** Единственная MPI-функция, которую можно вызвать до вызова **`MPI_INIT`** – это **`MPI_INITIALIZED`**`(FLAG, IERR)`, которая в аргументе `FLAG` возвращает `.TRUE.`, если вызвана из параллельной части приложения, и `.FALSE.` в противном случае. Инициализация параллельной части для каждого приложения должна выполняться только один раз. Завершение параллельной части приложения осуществляется функцией **`MPI_FINALIZE`**. Все последующие обращения к любым MPI-функциям, в том числе к **`MPI_INIT`**, запрещены. К моменту вызова **`MPI_FINALIZE`** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены. Код, расположенный до вызова **`MPI_INIT`** и после **`MPI_FINALIZE`** является последовательной частью программы, но в отличие от OpenMP он

исполняется во всех запущенных процессах, главного процесса не существует. Поскольку исполняется один и тот же код, то и содержание сегментов данных во всех процессах перед входом в параллельную часть одинаково.

Для того, чтобы не повторять одних и тех же вычислений во всех процессах, необходимо научиться различать процессы в параллельной части программы. Функция **MPI_COMM_SIZE**(COMM, SIZE, IERR) с целыми аргументами возвращает в аргументе SIZE число параллельных процессов в коммуникаторе COMM. Функция **MPI_COMM_RANK** (COMM, RANK, IERR) возвращает в аргументе RANK номер процесса в коммуникаторе COMM в диапазоне от 0 до SIZE-1. Минимальная MPI программа с выводом сообщений от каждого из запущенных процессов выглядит так:

```
PROGRAM EXAMPLE
INCLUDE 'mpif.h'
INTEGER IERR, SIZE, RANK
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
PRINT *, 'PROCESS ', RANK,' SIZE ', SIZE
CALL MPI_FINALIZE(IERR)
END
```

Процедура
MPI_SEND(BUF,COUNT,DATATYPE,DEST,MSGTAG,COMM,IERR)
реализует блокирующую посылку массива BUF с идентификатором сообщения MSGTAG, состоящего из COUNT элементов типа DATATYPE, процессу с номером DEST в коммуникаторе COMM. Тип всех аргументов функции целый, кроме массива, или структуры данных BUF. Все элементы посылаемого сообщения должны быть расположены подряд в буфере BUF. Операция посылки сообщения начинается независимо от того, была ли инициализирована соответствующая процедура приема. При этом сообщение может быть скопировано как непосредственно в буфер приема, так и помещено в некоторый системный буфер (если это предусмотрено в реализации MPI). Значение COUNT может быть нулем. Процессу разрешается передавать сообщение самому себе, однако это может привести к тупиковой ситуации, особенно при блокирующей посылке и приеме в одном потоке ☺. Тип передаваемых элементов должен указываться в целом параметре DATATYPE с помощью предопределенных констант типа, перечисленных для языка Фортран в следующей таблице

Тип данных в MPI	Тип данных в Fortran
MPI_INTEGER	**INTEGER**
MPI_REAL	**REAL**
MPI_DOUBLE_PRECISION	**DOUBLE PRECISION**
MPI_COMPLEX	**COMPLEX**
MPI_LOGICAL	**LOGICAL**
MPI_CHARACTER	**CHARACTER(1)**
MPI_BYTE	8 бит, используется для передачи нетипизированных данных.
MPI_PACKED	Тип для упакованных данных

Блокировка гарантирует корректность повторного использования всех параметров после возврата из процедуры. Это означает, что после возврата из **MPI_SEND** можно использовать любые присутствующие в вызове переменные без опасения использовать передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу DEST, остается за разработчиками конкретной реализации MPI. Такая неопределенность может не устроить пользователя, поэтому MPI предоставляет следующие модификации процедуры передачи данных с блокировкой **MPI_SEND**:

MPI_BSEND: Передача сообщения с *буферизацией.* Если прием посылаемого сообщения еще не был инициализирован процессом – получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение данной процедуры никак не зависит от соответствующего вызова процедуры приема сообщений. Тем не менее, процедура может вернуть код ошибки, если места под буфер недостаточно. О выделении массива для буферизации должен заботиться пользователь. Назначайте массив для использования при посылке с буферизацией с помощью функции **MPI_ATTACH**, и освобождайте эти массивы с помощью функции **MPI_DETTACH**.

MPI_SSEND: Передача сообщения с *синхронизацией.* Выход из данной процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизацией может замедлить выполнение программы, но позволяет избежать наличия в системе большого количества не принятых буферизованных сообщений.

MPI_RSEND: Передача сообщения *по готовности.* Данной процедурой можно пользоваться только в том случае, если процессор-получатель уже инициировал прием сообщения. В противном случае вызов процедуры,

вообще говоря, является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры **MPI_RSEND** можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов (например, **MPI_BARRIER**, или **MPI_SSEND**). Во многих реализациях процедура **MPI_RSEND** сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

MPI_RECV(BUF,COUNT, DATATYPE, SOURCE,MSGTAG, COMM,STATUS,IERR)
Реализует блокирующий прием в буфер BUF не более COUNT элементов типа DATATYPE сообщения с идентификатором MSGTAG, от процесса с номером SOURCE в коммуникаторе COMM с заполнением массива атрибутов приходящего сообщения STATUS. Если число реально принятых элементов меньше значения COUNT, то гарантируется, что в буфере BUF изменяются только элементы, соответствующие элементам принятого сообщения. Если количество элементов в принимаемом сообщении больше значения COUNT, то возникает ошибка переполнения. Чтобы избежать этого, можно сначала определить структуру приходящего сообщения при помощи процедуры **MPI_PROBE** или **MPI_IPROBE**. Если нужно узнать точное число элементов в принимаемом сообщении, то можно воспользоваться процедурой **MPI_GET_COUNT**. Блокировка гарантирует, что после возврата из процедуры **MPI_RECV** все элементы сообщения будут приняты и расположены в буфере BUF.

В следующем примере каждый процесс с четным номером посылает сообщение своему соседу с номером на единицу большим. Дополнительно поставлена проверка для процесса с максимальным номером, чтобы он не послал сообщение несуществующему процессу. Значения переменной *b* изменятся только на процессах с нечетными номерами.

```
PROGRAM EXAMPLE2
INCLUDE 'mpif.h'
INTEGER IERR, SIZE, RANK, a, b
INTEGER status(MPI_STATUS_SIZE)
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD,SIZE,IERR)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD,RANK,IERR)
 a = RANK; b = -1
 IF(MOD(RANK,2) .EQ. 0) THEN
  IF(RANK+1 .LT. SIZE) & ! посылают все кроме последнего
    CALL MPI_SEND(a,1,MPI_INTEGER,&
      RANK+1,5,MPI_COMM_WORLD,IERR)
 ELSE
    CALL MPI_RECV(b,1,MPI_INTEGER, &
```

```
      RANK-1,5,MPI_COMM_WORLD,STATUS,IERR)
 ENDIF
 CALL MPI_FINALIZE(IERR)
END
```

При приеме сообщения вместо аргументов SOURCE и MSGTAG можно использовать следующие предопределенные константы:

MPI_ANY_SOURCE - признак того, что подходит сообщение от любого процесса,

MPI_ANY_TAG - признак того, что подходит сообщение с любым идентификатором.

При одновременном использовании этих двух констант будет принято сообщение с любым идентификатором от любого процесса. Обратим внимание на некоторую несимметричность операций посылки и приема сообщений. С помощью константы MPI_ANY_SOURCE можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

Реальные атрибуты принятого сообщения всегда можно определить по соответствующим элементам массива STATUS. В Фортране массив STATUS является целочисленным массивом размера MPI_STATUS_SIZE. Константы MPI_SOURCE, MPI_TAG, MPI_ERROR, являются индексами для доступа к значениям соответствующих полей данного массива:
STATUS(MPI_SOURCE) – номер процесса отправителя;
STATUS(MPI_TAG) – идентификатор сообщения;
STATUS(MPI_ERROR) – код ошибки.
В языке Си параметр STATUS является структурой предопределенного типа MPI_Status с полями MPI_SOURCE, MPI_TAG, MPI_ERROR.

MPI_PROBE(SOURCE, MSGTAG, COMM, STATUS, IERR)

Получение в массиве STATUS информации о структуре ожидаемого сообщения с идентификатором MSGTAG от процесса с номером SOURCE в коммуникаторе COMM с блокировкой. Возврата из процедуры не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Следует особо обратить внимание на то, что процедура определяет только факт прихода сообщения, но реально его не принимает. Если после вызова **MPI_PROBE** вызывается **MPI_RECV** с такими же параметрами, то будет принято то же самое сообщение, информация о котором была получена с помощью вызова **MPI_PROBE**.

Синхронизация процессов в MPI. Передача сообщений без блокировки

В MPI предусмотрен набор процедур для осуществления *асинхронной передачи данных*. В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без какой-либо остановки работы процессов. На фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной передачи. Данная возможность исключительно полезна для создания эффективных программ. В самом деле, программист знает, что в некоторый момент ему потребуется массив, который вычисляет другой процесс. Он заранее выставляет в программе асинхронный запрос на получение данного массива, а до того момента, когда массив реально потребуется, он может выполнять любую другую полезную работу. Для завершения асинхронного обмена требуется вызов дополнительной процедуры, которая проверяет, завершилась ли операция, или дожидается ее завершения. Только после можно использовать буфер посылки для других целей без опасения запортить отправляемое сообщение.

MPI_ISEND(BUF,COUNT,DATATYPE, DEST, MSGTAG, COMM, REQUEST, IERR)

Неблокирующая посылка из буфера BUF COUNT элементов сообщения типа DATATYPE с идентификатором MSGTAG процессу DEST коммуникатора COMM. Возврат из процедуры происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере BUF. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации, подтверждающей завершение данной посылки. Определить тот момент времени, когда можно повторно использовать буфер BUF без опасения испортить передаваемое сообщение, можно с помощью возвращаемого параметра REQUEST и процедур семейств **MPI_WAIT** и **MPI_TEST**. Параметр REQUEST имеет в языке Фортран тип **integer** (в языке Си – предопределенный тип **MPI_Request**) и используется для идентификации конкретной неблокирующей операции.

Аналогично трем модификациям процедуры **MPI_SEND**, предусмотрены три дополнительных варианта процедуры **MPI_ISEND**:

MPI_IBSEND: неблокирующая передача сообщения с буферизацией;

MPI_ISSEND: неблокирующая передача сообщения с синхронизацией;

MPI_IRSEND: неблокирующая передача сообщения по готовности.

К изложенной выше семантике работы этих процедур добавляется отсутствие блокировки.

MPI_IRECV(BUF,COUNT,DATATYPE,SOURCE,MSGTAG,COMM,REQUEST, IERR)

Неблокирующий прием в буфер BUF не более COUNT элементов сообщения типа DATATYPE с идентификатором MSGTAG от процесса с

номером DEST в коммуникаторе COMM без заполнения массива STATUS. В отличие от блокирующего приема, возврат из процедуры происходит сразу после инициализации процесса приема без ожидания получения всего сообщения и его записи в буфер BUF. Окончание процесса приема можно определить с помощью параметра REQUEST и процедур семейств **MPI_WAIT** и **MPI_TEST**.

Сообщение, отправленное любой из процедур **MPI_SEND**, **MPI_ISEND** и любой из трех их модификаций, может быть принято любой из процедур **MPI_IRECV** и **MPI_RECV**.

MPI_IPROBE(SOURCE, MSGTAG, COMM, FLAG, STATUS, IERR)
Получение в массиве STATUS информации о структуре ожидаемого сообщения с идентификатором MSGTAG от процесса с номером SOURCE в коммуникаторе COMM без блокировки. В параметре FLAG с типом **LOGICAL** возвращается значение .TRUE., если сообщение с подходящими атрибутами уже может быть принято (в этом случае действие процедуры полностью аналогично **MPI_PROBE**), и значение .FALSE., если сообщения с указанными атрибутами еще нет.

MPI_WAIT(REQUEST, STATUS, IERR)
Ожидание завершения асинхронной операции, ассоциированной с идентификатором REQUEST и запущенной вызовом процедуры **MPI_ISEND** или **MPI_IRECV**. Пока асинхронная операция не будет завершена, процесс, выполнивший процедуру **MPI_WAIT**, будет заблокирован. В случае ожидания операции неблокирующего приема определяется параметр STATUS. После выполнения процедуры идентификатор неблокирующей операции REQUEST устанавливается MPI_REQUEST_NULL.

MPI_WAITALL(COUNT, REQUESTS, STATUSES, IERR)
Ожидание завершения COUNT асинхронной операций, ассоциированных с идентификаторами массива REQUESTS(COUNT). Для операций, перечисленных в REQUESTS с типом неблокирующего приема определяются соответствующие параметры в массиве STATUSES. Если во время одной или нескольких операций обмена возникли ошибки, в элементах массива STATUSES будет установлено соответствующее значение. После выполнения процедуры соответствующие элементы параметра REQUESTS устанавливаются в значение MPI_REQUEST_NULL.

Ниже показан пример фрагмента программы, в которой все процессы обмениваются сообщениями с ближайшими соседями в соответствии с топологией кольца при помощи неблокирующих операций. Заметим, что

использование для этих целей блокирующих операций могло привести к возникновению тупиковой ситуации.

```
PROGRAM EXAMPLE3
INCLUDE 'mpif.h'
INTEGER IERR, RANK, SIZE, prev, next, regs(4), buf(2)
INTEGER stats(MPI_STATUS_SIZE,4)
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD,SIZE,IERR)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD,RANK,IERR)
 prev = RANK - 1; IF(RANK.eq.0) prev = SIZE - 1
 next = RANK + 1; IF(RANK.eq.SIZE - 1) next = 0
 CALL MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, 5, &
  MPI_COMM_WORLD, regs(1), IERR)
 CALL MPI_IRECV(buf(2), 1, MPI_INTEGER, next, 6, &
  MPI_COMM_WORLD, regs(2),IERR)
 CALL MPI_ISEND(RANK,1, MPI_INTEGER, prev, 6, &
  MPI_COMM_WORLD, regs(3), IERR)
 CALL MPI_ISEND(RANK,1, MPI_INTEGER, next, 5, &
  MPI_COMM_WORLD, regs(4), IERR)
 CALL MPI_FINALIZE(IERR)
END
```

Глава 8. Распараллеливание программы для моделирования нелинейного параболического уравнения.

Цель данного курса – научить студента распараллеливанию на практике. Это предполагает написание своих алгоритмов, которые могут пригодиться в дальнейшей работе. С самого начала необходимо ориентироваться на реальные научные задачи. Мы предлагаем студентам самостоятельно написать, и распараллелить разными способами, алгоритм решения двумерного параболического уравнения, которое используется для моделирования диссипативных солитонов в лазере с насыщающимся поглотителем. Ознакомиться с этой темой из физики лазеров вы можете по материалам спецкурса ИТМО [*Розанов Н.Н.* “Обработка информации оптическими методами” – учебное пособие СПбГУ ИТМО, 2008]

Диссипативные солитоны в лазерах или лазерные солитоны, как локализованные островки генерации с размером порядка зоны Френеля, обладают частицеподобными свойствами, и могут быть использованы как биты информации для организации памяти, или как параллельные каналы чтения/записи оптической информации с помощью лазеров микронного размера. Однако, условия для существования и устойчивости

диссипативных солитонов довольно жесткие. Для появления заметного интервала бистабильности необходимо выбирать активную среду с эффективной интенсивностью насыщения концентрации носителей как минимум на порядок больше эффективной интенсивности насыщения пассивной среды

Солитоны в лазерах, в отличие от от диссипативных солитонов в интерферометрах, оказались намного более разнообразными и более подвижными. Спектр их связанных и возбужденных состояний охватывает все степени свободы, что позволило изучить все их частицеподобные свойства [*Н.Н. Розанов, С.В. Федоров, А.Н. Шацев* “Численный анализ комплексов двумерных солитонов в лазерных схемах класса А”, в сб. статей “Проблемы когерентной и нелинейной оптики” под ред. И.П.Гурова и С.А.Козлова, СПб: СПбГУ ИТМО, с.133-176, 2006]. Солитоны в лазерах представляют собой локализованные вдоль апертуры стационарные потоки энергии света, порожденные в активной, и исчезающие в пассивной среде. Такие потоки могут вращаться, осциллировать, и делиться. Излучение лазера может содержать локализованную дислокацию волнового фронта, что представляет собой локализованный вихрь энергетических потоков. Вихри с любым зарядом (кратностью дислокации) устойчивы в лазере, они могут образовывать устойчивые комплексы с сильной, или слабой связью, они могут двигаться вдоль апертуры, в том числе криволинейно. Отличие сильной от слабой связи между лазерными солитонами очень существенно, и определяется бифуркациями энергетических потоков. Наличие сильной связи может приводить к быстрому самодвижению асимметрично связанных комплексов. Были также изучены различные сценарии неупругих столкновений движущихся комплексов, что может быть полезным при проектировании логических операций при обработке оптической информации.

В самом простом случае безинерционной среды, лазерные солитоны описываются одним комплексным параболическим уравнением для амплитуды поля излучения:

$$\frac{\partial E}{\partial t} - (i+d)\nabla_{\perp}^2 E = f(I)E, \quad f(I) = -1 + \frac{g_0}{1+I/I_g} - \frac{a_0}{1+I/I_a}. \qquad (*)$$

Здесь $\nabla_{\perp}^2 = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)$ - двумерный Лапласиан по координатам области счета в перпендикулярном сечении пучка излучения;

$I = |E|^2$ - интенсивность излучения, аргумент функция нелинейности $f(I)$;

$i = \sqrt{-1} = \mathrm{dCmplx}(0,1)$ - мнимая единица, обеспечивающая перед Лапласианом эффект дифракции излучения;

d - коэффициент диффузии света, связанный с ограниченностью контура линии усиления/поглощения.

Коэффициенты усиления слабого сигнала в активной среде, g0, и поглощения в пассивной среде, a0, нормированы на линейные потери света в резонаторе. Чисто диссипативная нелинейность света задается функцией $f(I)$, в которую входит, наряду с единичным коэффициентом потерь в резонаторе, насыщающийся коэффициент усиления $g = g_0/(1 + I/I_g)$, и коэффициент поглощения $a = a_0/(1 + I/I_a)$, где *Ig* и *Ia* – интенсивности насыщения активной и пассивной среды соответственно. Диссипативные лазерные солитоны существуют и устойчивы в узком интервале бистабильной генерации на все апертуре. Для отладки алгоритма моделирования солитонов выбираем следующие параметры сред:

d = 0.06 - малый коэффициент диффузии поля,
Ig =10 - интенсивноть насыщения активной среды,
Ia = 1 - интенсивноть насыщения пассивной среды,
g0 =2.11 - коэффициент усиления слабого сигнала,
a0 =2.00 - коэффициент поглощения слабого сигнала

В этом случае бистабильность генерации существует на интервале коэффициентов усиления 2.29 < g0 < 3.0. Ориентировочно, интервал устойчивости основного состояния двумерного солитона 2.10 < g0 < 2.12. Причем на нижней границе интервала локальный островок генерации (солитон) исчезает, а на верхней границе – он начинает осциллировать (бифуркация Хопфа), и, при дальнейшем увеличении g0, превращается в расширяющуюся радиально симметричную волну переключения, которая заполняет однородной генерацией всю апертуру.

Для того, чтобы образовался солитон, как стационарное решение, необходимо численно решать уравнение (*) с помощью итераций по эволюционной координате *t* (время, нормированное на время жизни фотона в резонаторе). Начальное условие достаточно выбрать в виде пучка с супергауссовым распределением:

$$E(t=0) = \sqrt{I_0}\,\mathrm{Exp}\left[-(x/\mathrm{wx})^4 - (y/\mathrm{wy})^4\right],$$

где wx = wy = 12, I0 = 7.0,
и для выбранной рабочей точки можно выбрать следующие параметры области счета и пучка:
Nx = Ny = 128 - число точек по пространственным координатам,
Ax = Ay = 50 - величина области счета, dx=Ax/(Nx-1) , dx=Ay/(Ny-1).
Наконец, организуя итерации по времени с помощью явного разностного метода, необходимо выбрать достаточно малый шаг итерации dt = 0.001.

Наиболее быстрый из последовательных алгоритмов для решения параболического уравнения с нелинейностью – метод амплитудно-

фазового экрана на основе быстрого преобразования Фурье (FFT splitting method). В этом методе одна итерация разбивается на два шага. На одном шаге решается линейное уравнение дифракции в Фурье представлении, на другом – нелинейное обыкновенное дифференциальное уравнение среды без дифракции. В этом методе шаг одной итерации можно увеличить до ста раз: dt = 0.1. Однако, такой метод распараллеливается хуже, чем явный разностный метод. Поэтому, вам предлагается написать алгоритм решения уравнения (*) на основе метода Зейделя, представленного в Главе 4.

Для того, чтобы написать код для решения уравнения необходимо выписать его разностную схему. Мы это сделаем на примере вещественного линейного уравнения теплопроводности из Главы 4, оставляя уравнение (*) студенту для лабораторной и/или самостоятельной работы:

$$\frac{\partial T}{\partial t} = \nabla_{\perp}^2 T = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) T.$$

Заменяя производные на их разностные аналоги, имеем

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\mathrm{dt}} = \frac{1}{\mathrm{dx}} \left(\frac{T_{i+1,j}^{n} - T_{i,j}^{n}}{\mathrm{dx}} - \frac{T_{i,j}^{n} - T_{i-1,j}^{n}}{\mathrm{dx}} \right) + \frac{1}{\mathrm{dy}} \left(\frac{T_{i,j+1}^{n} - T_{i,j}^{n}}{\mathrm{dy}} - \frac{T_{i,j}^{n} - T_{i,j-1}^{n}}{\mathrm{dy}} \right) \Rightarrow$$

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \mathrm{dt} \frac{T_{i+1,j}^{n} + T_{i-1,j}^{n} - 2T_{i,j}^{n}}{\mathrm{dx*dx}} + \mathrm{dt} \frac{T_{i,j+1}^{n} + T_{i,j-1}^{n} - 2T_{i,j}^{n}}{\mathrm{dy*dy}}.$$

Здесь мы используем индексы, нумерующие узлы области счета по пространственным координатам: $i = 1,..Nx;\ j = 1,..Ny$, и n - индекс итераций по времени. Как видите, итерация свелась к линейному преобразованию поля температур. В этом случае мы можем выбрать соотношение между шагом по времени и пространственным шагом так, чтобы матрица линейного преобразования имела единичную норму (наиболее эффективный алгоритм):

dt=dx*dx/4=dy*dy/4, тогда формула для итерации упрощается:

$$T_{i,j}^{n+1} = \left(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n} \right) / 4.$$

Отметим, что выбор соотношения шага по времени к шагу по пространству в случае нелинейного уравнения не так однозначен, поэтому его нужно подбирать экспериментально, исходя из эффективности алгоритма.

Выписав разностную схему, и упростив ее, мы можем приступать к написанию алгоритма. При этом, необходимо помнить, о необходимости егл распараллеливания. Для сравнения мы приведем здесь два фрагмента кода для решения уравнения теплопроводности. Первый предназначен для автоматического распараллеливания, поскольку использует оператор **forall**. Второй фрагмент эквивалентен первому, но выписан с использованием оператора циклов **DO**. Продумать место вставки директив

распараллеливания по технологии OpenMP, а также какие переменные, или массивы в алгоритме должны быть объявлены как локальные переменные параллельных потоков предоставляется студенту. В случае распараллеливания на весь кластер пример необходимо изменить так, чтобы правильно распределить основные и дополнительные массивы по адресным пространствам процессов, выбрав виртуальную топологию адресации процессов в коммуникаторе (в данном случае необходимо выбирать декартову топологию). Этот выбор, как и организацию процесса приема-передачи сообщений по обмену памятью также предоставляется студенту в качестве самостоятельной работы.

Отрывок из шаблонного примера для автоматического распараллеливания

```
DO iter = 1,iterMax
  logo =.true.
  Forall (  i=1:n, j=1:n, &
   abs(0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1)) - T(i,j)) > eps    )
   T(i,j)=0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
   logo(i,j) =.false.
  end forall
  if( all(logo) )  exit
 ENDDO
```

Отрывок из шаблонного примера для распараллеливания вручную

```
double precision :: Ti1,Tj1(n),Tnew
double precision,dimension(1:n) :: Tj1new
  ! Tj1(:)=T(:,j-1), Ti1=T(i-1,j)
  DO iter = 1,iterMax
   Tj1(:)=T(:,0); logo =.true.
   DO j=1,n
     Ti1=T(0,j)
     DO i=1,n
        Tnew = 0.25*(Ti1+T(i+1,j)+Tj1(i)+T(i,j+1))
        Ti1 = T(i,j); Tj1new(i) = Tnew
        logo(i,j) = abs(T(i,j)-Tnew)<= eps
     ENDDO
     Tj1(:)=T(:,j); T(:,j)=Tj1new(:)
   ENDDO
   if( all(logo) )  exit
ENDDO
```

Термины

Внешние, внутренние и модульные подпрограммы в языке Фортран.

Высокопроизводительные вычисления чаще всего достигаются при параллельных вычислениях.

Кластер – электронная вычислительная машина, обеспечивающая параллельные вычисления со многими CPU.

Клиент–Сервер – технология и программы.

Метод Зейделя - типичная итерационная схема в численном моделировании уравнений в частных производных.

Модули, функции и процедуры – виды программных единиц в языке Фортран.

Параболическое уравнение – линейное и нелинейное.

Параллельное программирование – создание приложений для кластеров и многоядерных ПК.

Параллелизация автоматическая характерна для Фортрана при использовании компиляторов ifc9, ifc10.

Параллельные вычисления – вычисления со многими CPU.

Последовательные вычисления – традиционные вычисления с одним CPU.

Протокол FTP(file transfer protocol) - протокол, предназначенный для передачи файлов в компьютерных сетях.

Протокол SSH (Secure Shell — «безопасная оболочка») — сетевой протокол, позволяющий производить удалённое управление операционной системой и передача файлов с шифрованием.

Распараллеливание последовательного приложения при помощи MPI или OMP с целью сделать его параллельным.

Удаленный доступ – возможность работы с компьютером на расстоянии через посредство сети, используя терминальный ПК.

Фортран-95 – язык для Параллельного программирования.

Ускорение оценивает во сколько раз параллельные вычисления быстрее последовательных вычислений.

Функции нелинейности среды.

Intel Visual Fortran Compiler (ifc9, ifc10, ifc11) - компилятор Фортран-95.

Linux - операционная система, чаще других используемая для кластеров.

MIMD – Multiple Instructions – Multiple Data, много данных – много программ - один из подходов к организации параллельных вычислений.

MPI (The Message Passing Interface) - интерфейс передачи сообщений является распространённым стандартом интерфейса обмена данными в параллельном программировании**.**

OMP (OpenMP - Open Multi-Processing) - «главный» поток создает набор подчиненных потоков и задача распределяется между ними. Потоки выполняются параллельно на машине с несколькими процессорами.
SISD – Single Instructions–Single Data, одна программа - одни данные, случай обычных последовательных вычислений.

SIMD – Single Instructions – Multiple Data, одна программа - много данных, наиболее часто встречающийся на практике подход к организации параллельных вычислений (Фортран-95 и выше).

Visual Studio **-** среда проектирования приложений – рабочее место программиста.

Windows 2003 Server – 64-разрядная операционная система.

Windows XP - 32-разрядная операционная система.

Термины и их толкование сведены в специальный электронный документ этого курса – “ Глоссарий – Параллельные вычисления в оптике и оптоинформатике ”. Интернет-проект “Википедия” также может помочь в толковании терминов.

Электронный глоссарий и Википедия

Электронный глоссарий (ЭГ) – это наиболее оперативное погружение в терминологию курса. ЭГ имеется в электронном виде на сайте *de.ifmo.ru* , и его можно использовать как при подготовке к лекциям, так и в практической работе по всем видам самостоятельной деятельности. ЭГ построен по словарному принципу, когда по первой букве термина Вы получаете ссылку на энциклопедическую статью для этого термина.

Википедия *http://ru.wikipedia.org/wiki* в Интернете также поможет разобраться в терминологии, её словарный запас более богатый.

Электронная и бумажная литература

Доступные Интернет-ресурсы

Сайты:
de.ifmo.ru - центр дистанционного обучения
twcad.ifmo.ru – личный сайт Звягина В.Ф.

parallel.ru – сайт лаборатории параллельных вычислений Московского университета
www.software.unn.ac.ru//ccam– сайт лаборатории параллельных вычислений Нижегородского университета
phoi.ifmo.ru - сайт кафедры фотоники и оптоинформатики

Электронные пособия

1. http://parallel.ru/cluster/beginner_guide.html страничка для начинающих пользователей вычислительных кластеров
2. Немнюгин С., Стесик О. Современный Фортран. Самоучитель.djvu
3. Иллюстрированный самоучитель по Visual Studio.Net
4. Intel.Методика разработки многопоточных приложений.xml.htm
5. Andrew Binstock. Выбор между OpenMP и методами явной многопоточности.htm
6. Открытый стандарт OpenMP,
7. А.А.Самарский. Введение в численные методы.
8. Антонов А.С. ПП с использованием технологии MPI.pdf (Fortran)
9. Антонов. Введение в ПП (методическое пособие).pdf (C++)
10. Материалы курса “Обработка информации оптическими методами”,
11. IMSL Math Library.pdf files (Fortarn/IMSL-Help)

Литература

Базовые учебники

12. Бартеньев О.В. ФОРТРАН для студентов. - М.:" Диалог МИФИ", 1999. - 400 с.
13. Немнюгин С.А., Стесик О.Л. Современный Фортран - Спб, БХВ, 2003 – 496с.

Базовые учебно-методические пособия

14. *Данные настоящего документа:* " Параллельные вычисления в оптике и оптоинформатике, Учебное пособие " - Звягин В.Ф., Федоров С.В. - Санкт-Петербургский государственный университет информационных технологий, механики и оптики, СПб, 2009, 108с.
15. " Параллельные вычисления в оптике и оптоинформатике. Лабораторный практикум, Учебное пособие″ - Звягин В.Ф., Федоров

С.В. - Санкт-Петербургский государственный университет информационных технологий, механики и оптики, СПб, 2009, 31с.

16. " Параллельные вычисления в оптике и оптоинформатике. Методические материалы по управлению самостоятельной работой студентов (УСРС), Учебное пособие" - Звягин В.Ф., Федоров С.В. - Санкт-Петербургский государственный *университет* информационных технологий, механики и оптики, СПб, готовится к печати, 29с.

Основная литература по дисциплине

17. "Практикум по информатике" - Голыничев В.Н., Звягин В.Ф., Фрейман И.А., Щупак Ю.А., Яньшина Н.А. - Санкт-Петербургский государственный институт точной механики и оптики, 2001 - 94с.

18. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие –М: Изд-во МГУ, 2004. 71 с.

19. Richard M. Stallman и Roland McGrath. GNU Make. Программа управления компиляцией. Апрель 2000, перевод (С) Владимир Игнатов, 2000. (http://www.gnu.org/software/make/manual/)

20. Mendel Cooper. Advanced Bash-Scripting Guide. Искусство программирования на языке сценариев командной оболочки. Перевод Андрей Киселев, 2007. http://www.opennet.ru/docs/RUS/bash_scripting_guide/

21. А.А.Самарский. «Задачи и упражнения по численным методам Издание 3» М.2007. ISBN 5484009286

22. А.А.Самарский. Введение в численные методы. Изд-во “Лань”, М.2005. ISBN: 5-8114-0602-9

23. “Обработка информации оптическими методами” – Розанов Н.Н., учебное пособие Санкт-Петербургского государственного университета информационных технологий, механики и оптики, 2008

Дополнительная литература

24. Рыжиков Ю.И. Современный Фортран - Спб, КОРОНА-принт, 2004. - 288с.

25. Бартеньев О.В. Современный Фортран-М.:Диалог МИФИ,1999,400 с.

26. Мак-Кракен Д., Дорн У. Численные методы и программирование на Фортране. 2-е изд.: Пер. с англ.,М.: Мир, 1977. 584 с.
27. Антонов А.С. Введение в параллельные вычисления (методическое пособие). – М: Изд-во МГУ, 2002. 69 с.
28. Andrew Binstock. Выбор между OpenMP и методами явной многопоточности (http://www.intel.ru)

Рекомендации по использованию Интернет-ресурсов и других электронных информационных источников

29. Немнюгин С., Стесик О. Современный Фортран. Самоучитель.djvu
30. Страничка для начинающих пользователей вычислительных кластеров.htm
31. Andrew Binstock. Выбор между OpenMP и методами явной многопоточности.htm
32. http://twcad.ifmo.ru/?rub=1stlect/allprn.zip, "Практикум по Фортрану-90" в курсе Информатики - рукопись, Голыничев В.Н., Звягин В.Ф., Яньшина Н.А., Учебное пособие. - 2008- 116с.
33. http://www.gnu.org/manual,
34. http://www.opennet.ru/docs,
35. http://phoif.ifmo.ru/rosanovteam.
36. http://computers.plib.ru/programming/Visual_C_7/index.html Иллюстрированный самоучитель по Visual Studio.Net
37. Intel.Методика разработки многопоточных приложений.xml.htm (http://www.rsdn.ru/?article/baseserv/RUThreadingMethodology.xml)
38. GNU Make. Программа управления компиляцией. (http://www.gnu.org/software/make/manual/)
39. http://parallel.ru/cluster/beginner_guide.html
40. Mendel Cooper. Язык сценариев командной оболочки. (http://www.opennet.ru/docs/RUS/bash_scripting_guide/)
41. Антонов А.С. ПП с использованием технологии MPI.pdf (Fortran)
42. IMSL Math Library.pdf files (Fortarn/IMSL-Help)
43. Антонов. Введение в ПП (методическое пособие).pdf (C++)

Оглавление

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007–2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация программы создала основу дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

КАФЕДРА ФОТОНИКИ И ОПТОИНФОРМАТИКИ

Кафедра "Фотоники и оптоинформатики" была создана летом 2002 года. Одной из ее задач является подготовка специалистов по *оптоинформатике* – стремительно развивающейся новой области науки и техники, в которой разрабатываются оптические технологии сверхбыстрой передачи, обработки и записи информации. Исследования кафедры - в русле приоритетных направлений развития российской науки, техники и технологий. Среди научных подразделений кафедры –лаборатория компьютерного моделирования и параллельных вычислений, в которой имеется единственный в ИТМО суперкомпьютер с производительностью 1Тфлопс, введенный в эксплуатацию в конце 2007 года. Среди студентов и аспирантов кафедры – стипендиаты Президента и Правительства Российской Федерации, победители конкурсов научных работ, проводимых Российской Академией наук, крупнейшими мировыми научными обществами, такими как INTAS (Фонд научно-исследовательских работ Европейского сообщества), SPIE (Международное общество инженеров-оптиков), CRDF (Американский фонд гражданских исследований и развития), OSA (Оптическое общество Америки). Кафедра постоянно занимает призовые места по итогам конкурсов научно-педагогических коллективов университета. На призовые премии дооснастила собственный вычислительный учебный класс Красного Домика, где с 2009 года, студенты с первого курса обучаются языкам, а затем и параллельному программированию.

В.Ф. Звягин, С.В. Фёдоров

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
В ОПТИКЕ И ОПТОИНФОРМАТИКЕ
Учебное пособие

В авторской редакции
Дизайн В.Ф. Звягин
Верстка В.Ф. Звягин
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати 06.11.09
Заказ № 2154
Тираж 100
Отпечатано на ризографе