ТЕОРИЯ РАЗРАБОТКИ КОРПОРАТИВНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ.

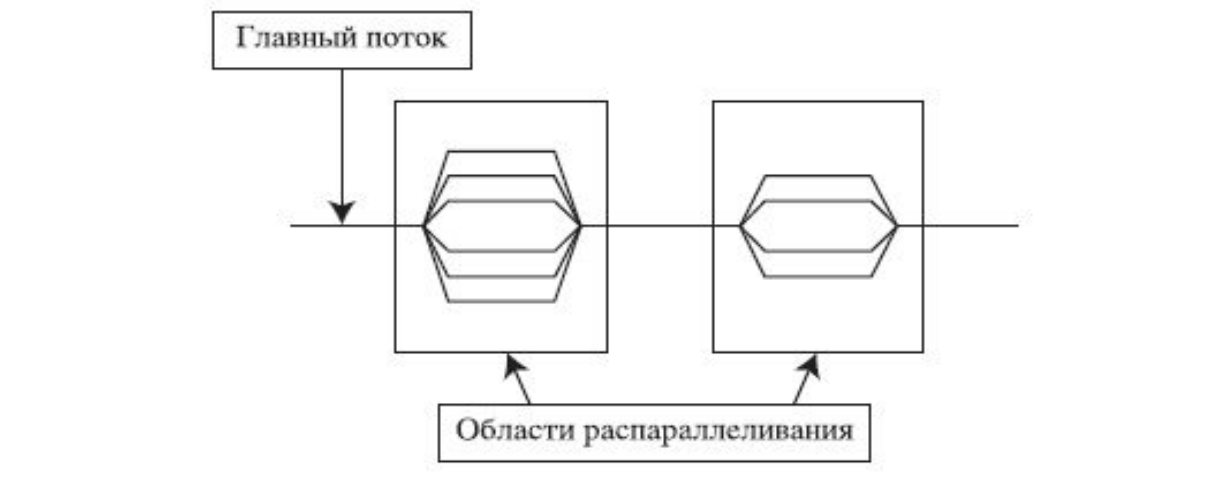
Лекция 2. Основные конструкции OpenMP.

Первый вопрос.

Принципиальная схема программирования в OpenMP

Любая программа, последовательная или параллельная, состоит из набора областей двух типов: последовательных областей и областей распараллеливания. При выполнении последовательных областей порождается только один главный поток (процесс). В этом же потоке инициируется выполнение программы, а также происходит ее завершение. В последовательной программе в областях распараллеливания порождается также только один, главный поток, и этот поток является единственным на протяжении выполнения всей программы. В параллельной программе в областях распараллеливания порождается целый ряд параллельных потоков. Порожденные параллельные потоки могут выполняться как на разных процессорах, так и на одном процессоре вычислительной системы. В последнем случае параллельные процессы (потоки) конкурируют между собой за доступ к процессору. Управление конкуренцией осуществляется планировщиком операционной системы с помощью специальных алгоритмов. В операционной системе Linux планировщик задач осуществляет обработку процессов с помощью стандартного карусельного (round-robin) алгоритма. При этом только администраторы системы имеют возможность изменить или заменить этот алгоритм системными средствами. Таким образом, в параллельных программах в областях распараллеливания выполняется ряд параллельных потоков.

Принципиальная схема параллельной программы



При выполнении параллельной программы работа начинается с инициализации и выполнения главного потока (процесса), который по мере необходимости создает и выполняет параллельные потоки, передавая им необходимые данные. Параллельные потоки из одной параллельной области программы могут выполняться как независимо друг от друга, так и с пересылкой и получением сообщений от других параллельных потоков. Последнее обстоятельство усложняет разработку программы, поскольку в этом случае программисту приходится заниматься планированием, организацией и синхронизацией посылки сообщений между параллельными потоками. Таким образом, при разработке параллельной программы желательно выделять такие области распараллеливания, в которых можно организовать выполнение независимых параллельных потоков. Для обмена данными между параллельными процессами (потоками) в OpenMP используются общие переменные. При обращении к общим переменным в различных параллельных потоках возможно возникновение конфликтных ситуаций при доступе к данным. Для предотвращения конфликтов можно воспользоваться процедурой синхронизации (synchronization). При этом надо иметь в виду, что процедура синхронизации - очень дорогая операция по временным затратам и желательно по возможности избегать ее или применять как можно реже. Для этого необходимо очень тщательно продумывать структуру данных программы.

Выполнение параллельных потоков в параллельной области программы начинается с их инициализации. Она заключается в создании дескрипторов порождаемых потоков и копировании всех данных из области данных главного потока в области данных создаваемых параллельных потоков. Эта операция чрезвычайно трудоемка - она эквивалентна примерно трудоемкости 1000 операций.

Для того чтобы получить выигрыш в быстродействии параллельной программы, необходимо, чтобы трудоемкость параллельных процессов в областях распараллеливания программы существенно превосходила бы трудоемкость порождения параллельных потоков.

После завершения выполнения параллельных потоков управление программой вновь передается главному потоку. При этом возникает проблема корректной передачи данных от параллельных потоков главному. Здесь важную роль играет синхронизация завершения работы параллельных потоков, поскольку в силу целого ряда обстоятельств время выполнения даже одинаковых по трудоемкости параллельных потоков непредсказуемо (оно определяется как историей конкуренции параллельных процессов, так и текущим состоянием вычислительной системы). При выполнении операции синхронизации параллельные потоки, уже завершившие свое выполнение, простаивают и ожидают завершения работы самого последнего потока. Естественно, при этом неизбежна потеря эффективности работы параллельной программы.

Вывод: **при выделении параллельных областей программы и разработке параллельных процессов необходимо, чтобы трудоемкость параллельных процессов была не менее 2000 операций деления.** В противном случае параллельный вариант программы будет проигрывать в быстродействии последовательной программе. Для эффективной работающей параллельной программы этот предел должен быть существенно превышен.

Второй вопрос.

Синтаксис директив в OpenMP

Основные конструкции OpenMP - это директивы компилятора или прагмы (директивы препроцессора) языка C/C++.

Общий вид директивы OpenMP

#pragma omp конструкция [предложение [предложение] …]

Большинство директив OpenMP применяется к структурным блокам.

Структурные блоки - это последовательности операторов с одной точкой входа в начале блока и одной точкой выхода в конце блока.

Пример 2.1. Пример структурного блока в программе на языке Fortran

```
c$omp parallel
10 wrk (id) = junk (id)
 res (id) = wrk (id)**2
 if (conv (res)) goto 10
c$omp end parallel
        print *, id
```

Пример 2.2. Пример неструктурного блока в программе на языке Fortran

```
c$omp parallel
10 wrk (id) = junk (id)
30 res (id) = wrk (id)**2
if (conv (res)) goto 20
goto 10c$omp end parallel
if (not_done) goto 30   20 print *, id
```

Общий вид основных директив OpenMP на языке C/C++.

```
# pragma omp parallel      \
                  private (var1, var2, …)       \
                  shared (var1, var2, …)        \
                  firstprivate (var1, var2, …) \
                  lastprivate (var1, var2, …)  \
                  copyin (var1, var2, …)        \
                  reduction (operator: var1, var 2, …) \
                  if (expression)                \
                  default (shared | none)          \
                  {   [ Структурный блок программы]}
```

Предложение OpenMP **shared** используется для описания общих переменных. Предложение **private** используется для описания внутренних (локальных) переменных для каждого параллельного процесса. Предложение **firstprivate** применяется для описания внутренних переменных параллельных процессов, однако данные в эти переменные импортируются из главного потока. Предложение **reduction** позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков. Предложение **if** является условным оператором в параллельном блоке. Предложение **default** определяет по умолчанию тип всех переменных в последующем параллельном структурном блоке программы. Предложение **copyin** позволяет легко и просто передавать данные из главного потока в параллельные.

Для продолжения длинных директив на следующих строках в программах на C/C++ применяется символ "обратный слэш" в конце строки.

Особенности реализации директив OpenMP

Количество потоков в параллельной программе определяется либо значением переменной окружения OMP_NUM_THREADS, либо специальными функциями, вызываемыми внутри самой программы.

Задание переменной окружения OMP_NUM_THREADS в операционной системе Linux осуществляется следующим образом с помощью команды

export OMP_NUM_THREADS=256

Здесь было задано 256 параллельных потоков.

Просмотреть состояние переменной окружения OMP_NUM_THREADS можно, например, с помощью следующей команды:

export | grep OMP_NUM_THREADS

Каждый поток имеет свой номер thread_number, начинающийся от нуля (для главного потока) и заканчивающийся OMP_NUM_THREADS-1.

Определить номер текущего потока можно с помощью функции **omp_get_thread_num().**

Каждый поток выполняет структурный блок программы, включенный в параллельный регион. В общем случае синхронизации между потоками нет, и заранее предсказать завершение потока нельзя. **Управление синхронизацией потоков и данных осуществляется программистом внутри программы.** После завершения выполнения параллельного структурного блока программы все параллельные потоки за исключением главного прекращают свое существование.

Пример ветвления во фрагменте программы в зависимости от номера параллельного потока

```
#pragma omp parallel
{
myid = omp_get_thread_num ( ) ;
     if (myid == 0)        do_something ( ) ;
    else        do_something_else (myid) ;}
```

В этом примере в первой строке параллельного блока вызывается функция **omp_get_thread_num**, возвращающая номер параллельного потока. Этот номер сохраняется в локальной переменной **myid**. Далее в зависимости от значения переменной **myid** вызывается либо функция **do_something()** в первом параллельном потоке с номером **0**, либо функция **do_something_else(myid)** во всех остальных параллельных потоках.

В OpenMP существуют различные режимы выполнения (Execution Mode) параллельных структурных блоков.

Возможны следующие режимы:

- **Динамический режим** (Dynamic Mode).

Этот режим установлен по умолчанию и определяется заданием переменной окружения OMP_DYNAMIC в операционной системе Linux.

Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux:

export OMP_DYNAMIC

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

omp_set_dynamic()

Отметим, что на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует.

В динамическом режиме количество потоков определяется самой операционной системой в соответствии со значением переменной окружения OMP_NUM_THREADS.

В процессе выполнения параллельной программы при переходе от одной области распараллеливания к другой эта переменная может изменять свое значение;

- **Статический режим** (Static Mode). Этот режим определяется заданием переменной окружения OMP_STATIC в операционной системе Linux. В этом случае количество потоков определяется программистом.

  Задать переменную окружения OMP_STATIC можно, например, с помощью следующей команды операционной системы Linux:

export OMP_STATIC

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

omp_set_static()

Однако на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует;

- **Параллельные структурные блоки** могут быть вложенными, но компилятор иногда по ряду причин может выполнять их и последовательно в рамках одного потока. Вложенный режим выполнения параллельных структурных блоков определяется заданием переменной окружения **OMP_NESTED=[FALSE|TRUE]** (по умолчанию задается **FALSE**) в операционной системе Linux.

  Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux:

setenv OMP_NESTED TRUE

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

omp_set_nested(TRUE | FALSE)

Директивы OpenMP

Директивы shared, private и default

Эти директивы (предложения OpenMP) используются для описания типов переменных внутри параллельных потоков.

Предложение OpenMP **shared(var1, var2, …, varN)**
определяет переменные **var1, var2, …, varN** как общие переменные для всех потоков.
Они размещаются в одной и той же области памяти для всех потоков.

Предложение OpenMP **private(var1, var2, …, varN)**
определяет переменные **var1, var2, …, varN** как локальные переменные для каждого из параллельных потоков.

В каждом из потоков эти переменные имеют собственные значения и относятся к различным областям памяти: локальным областям памяти каждого конкретного параллельного потока.

В качестве иллюстрации использования директив OpenMP shared и private рассмотрим фрагмент программы.

В этом примере переменная a определена как общая и является идентификатором одномерного массива.

Переменные myid и x определены как локальные переменные для каждого из параллельных потоков.

В каждом из параллельных потоков локальные переменные получают собственные значения. После чего при выполнении условия x<1.0 значение локальной переменной x присваивается myid -й компоненте общего для всех потоков массива a. Значение x будет неопределенным, если не определить x как переменную типа private.

Отметим, что значения private -переменных не определены до и после блока параллельных вычислений.

- Пример использования директив OpenMP shared и private в параллельной области программы
- c$omp parallel shared (a)
- c$omp& private (myid, x)
- myid = omp_get_thread_num ( )
- x = work (myid)
- if (x < 1.0) then
- a (myid) = x
- endif
- Предложение OpenMP default (shared | private | none)
- задает тип всех переменных, определяемых по умолчанию в последующем параллельном структурном блоке как shared, private или none.
- Например, если во фрагменте программы вместо private(myid, x) написать
- default(private) то определяемые далее по умолчанию переменные myid и x будут автоматически определены как private.
-

Директивы firstprivate и lastprivate

Директивы (предложения) OpenMP **firstprivate** и **lastprivate** используются для описания локальных переменных, инициализируемых внутри параллельных потоков.

Переменные, описанные как **firstprivate**, получают свои значения из последовательной части программы.

Переменные, описанные как **lastprivate**, сохраняют свои значения при выходе из параллельных потоков при условии их последовательного выполнения.

Предложение OpenMP

firstprivate(var1, var2, ..., varN)

определяет переменные **var1, var2, ..., varN** как локальные переменные для каждого из параллельных потоков, причем инициализация значений этих переменных происходит в самом начале параллельного структурного блока по значениям из предшествующего последовательного структурного блока программы.

В качестве иллюстрации рассмотрим фрагмент программы.

```
program first
   integer :: myid, c
     integer, external :: omp_get_thread_num
     c=98
!$omp parallel private (myid)
!$omp& firstprivate (c)
     myid = omp_get_thread_num ( )
    write (6, *) 'T: ', myid, 'c=', c
!$omp end parallel
     end program first
```

T:1 c=98 T:3 c=98 T:2 c=98 T:0 c=98

В этом примере в каждом параллельном потоке используется своя переменная c, но значение этой переменной перед входом в параллельный блок программы берется из предшествующего последовательного блока.

Предложение OpenMP

lastprivate(var1, var2, ..., varN)

определяет переменные **var1, var2, ..., varN** как локальные переменные для каждого из параллельных потоков, причем значения этих переменных сохраняются при выходе из параллельного структурного блока при условии, что параллельные потоки выполнялись в последовательном порядке.

В качестве иллюстрации рассмотрим фрагмент программы.
В этом примере локальная переменная **i** принимает различные значения в каждом потоке параллельного структурного блока. После завершения параллельного структурного блока переменная **i** сохраняет свое последнее значение, полученное в последнем параллельном потоке, при условии последовательного выполнения всех параллельных потоков, т. е. **n=N+1.**

```
c$omp  do shared ( x )
c$omp& lastprivate ( i)
      do i = 1, N
        x (i) = a
      enddo
     n = i
```

Директива if

Директива (предложение) OpenMP **if** используется для организации условного выполнения потоков в параллельном структурном блоке.

Предложение OpenMP

if(expression)

определяет условие выполнения параллельных потоков в последующем параллельном структурном блоке.

Если выражение expression принимает значение TRUE (Истина), то потоки в последующем параллельном структурном блоке выполняются. В противном случае, когда выражение expression принимает значение FALSE (Ложь), потоки в последующем параллельном структурном блоке не выполняются.

В качестве иллюстрации рассмотрим фрагмент программы.

```
c$omp parallel do if (n .ge. 2000)
      do i = 1, n
        a (i) = b (i)*c + d (i)
      enddo
```

В этом примере цикл распараллеливается только в том случае (n>2000), когда параллельная версия будет заведомо быстрее последовательной.

Напомним, что трудоемкость образования параллельных потоков эквивалентна примерно трудоемкости 1000 операций.

Директива reduction

Директива OpenMP **reduction** позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков последующего параллельного структурного блока.

В предложении OpenMP
reduction(operator | intrinsic: var1 [, var2,..., varN]) определяется **operator** - операции **(+, -, *, / и т. п.)** или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока.

Кроме того, определяется список локальных переменных **var1, var2, ..., varN,** в котором будут сохраняться соответствующие частичные значения. После завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.

В качестве иллюстрации рассмотрим фрагмент программы.

```
c$omp  do shared (x) private (i)
c$omp& reduction (+ : sum)
      do i = 1, N
        sum = sum + x (I)
       enddo
```

В этом примере в каждом параллельном потоке определена локальная переменная **sum** для вычисления частичных сумм. После завершения параллельных потоков все локальные переменные **sum** суммируются, а результат сохраняется в одноименной общей (глобальной) переменной **sum**.

В языке Fortran в качестве параметров operator и intrinsic в предложениях reduction допускаются следующие операции и функции:

- параметр operator может быть одной из следующих арифметических или логических операций: +, -, *, .and. , .or. , .eqv. , .neqv.;
- параметр intrinsic может быть одной из следующих функций: max, min, iand, ior, ieor.

В языке C/C++ в качестве параметров operator и intrinsic в предложениях reduction допускаются следующие операции и функции:

- параметр operator может быть одной из следующих арифметических или логических операций: +, -, *, &, ^, &&, || ;
- параметр intrinsic может быть одной из следующих функций: max, min ;
- указатели и ссылки в предложениях reduction использовать строго запрещено!

Директива copyin

Директива OpenMP **copyin** используется для передачи данных из главного потока в параллельные потоки, называемой миграцией данных. Механизмы реализации этой директивы в языках **C/C++** и **Fortran** различны. Рассмотрим далее реализации этой директивы поочередно в **C/C++ и Fortran.**

Предложение OpenMP в языке C/C++

copyin(var1 [, var2[, ...[, varN]]])

определяет список локальных переменных **var1, var2, ..., varN,** которым присваиваются значения из одноименных общих переменных, заданных в глобальном потоке.

В **Fortran** в качестве параметров директивы OpenMP **copyin** задаются имена common -блоков (cb1, cb2, ..., cbN), в которых описаны мигрирующие переменные

copyin(/cb1/ [, /cb2/ [, ...[, /cbN/]]])

т.е. такие переменные из глобального потока, которые передают свои значения своим копиям, определенным в параллельных потоках в качестве локальных переменных.

В качестве иллюстрации миграции данных рассмотрим фрагмент Fortran-программы.

В этом примере мигрирующая целочисленная переменная **x** хранится в **common -блоке mine**. Значение этой общей переменной задается в главном потоке. В каждом параллельном потоке используется своя локальная

переменная **x**, которой присваивается значение общей переменной **x**.

```
integer : : x, tid
integer, external :: omp_get_thread_num ( )
common/mine/ x
! $omp threadprivate (/mine/)
      x=33
call omp_set_num_threads (4)
! $omp parallel private (tid) copyin (/mine/)
      tid=omp_get_thread_num ( )
      print *, 'T: ',tid, ' x = ', x
! $omp end parallel
```

Результаты работы программы:

T: 1 x = 33
T: 2 x = 33
T: 0 x = 33
T: 3 x = 33

Директива for

Директива OpenMP **for** используется для организации параллельного выполнения петель циклов в программах, написанных на языке C/C++.

Предложение OpenMP в программе на C/C++

#pragma omp for

означает, что оператор **for**, следующий за этим предложением, будет выполняться в параллельном режиме, т. е. каждой петле этого цикла будет соответствовать собственный параллельный поток.

В качестве иллюстрации использования директивы OpenMP **for** рассмотрим фрагмент программы на языке C.

В этом примере петли цикла оператора **for** реализуются как набор параллельных потоков.

По умолчанию в конце цикла реализуется функция синхронизации **barrier** (барьер).

Для отмены функции **barrier** следует воспользоваться предложением OpenMP **nowait.**

```
#pragma omp parallel shared (a, b) private (j)
{   #pragma omp for
    for (j=0; j<N; j++)
      a [j] = a [j] + b [j]; }
```