Физический факультет МГУ имени М.В. Ломоносова

Параллельное программирование
для ресурсоёмких задач численного моделирования в физике

В.О. Милицин, Д.Н. Янышев, И.А. Буткарев

Лекция № 3

Содержание лекции

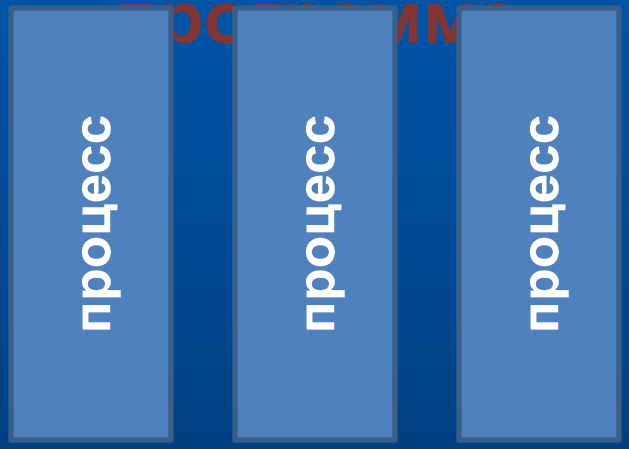
- Структура, области применения, этапы разработки и характеристики производительности параллельной программы
- Основные принципы OpenMP

Параллельная программа

Последовательная программа

один поток управления

Параллельная программа

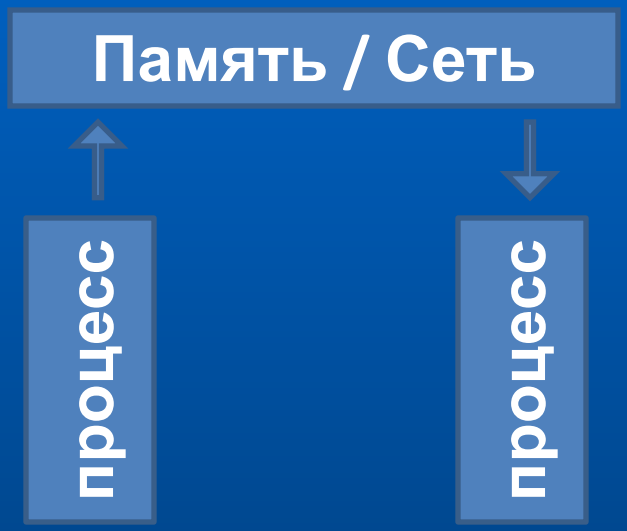


несколько потоков управления

Процесс = последовательная программа

= последовательность операторов

Взаимодействие процессов



Разделяемые переменные

один процесс осуществляет запись в переменную, считываемую другим процессом

Пересылка сообщений

один процесс отправляет сообщение, которое получает другой

Классы приложений

Многопоточные системы

Распределенные системы

Синхронные параллельные вычисления

Многопоточные системы

Примеры приложений

оконные системы на персональных компьютерах или рабочих станциях

многопроцессорные операционные системы и системы с разделением времени

системы реального времени, управляющие электростанциями, космическими аппаратами и т.д.

Признаки

число процессов (потоков) больше числа процессоров

Распределенные системы

Примеры приложений

- файловые серверы в сети
- системы баз данных для банков, заказа авиабилетов и т.д.
- Web-серверы сети Internet
- предпринимательские системы, объединяющие компоненты производства
- отказоустойчивые системы, которые продолжают работать независимо от сбоя в компонентах

Признаки

- компоненты выполняются на машинах, связанных локальной или глобальной сетью

Синхронные параллельные вычисления

Примеры приложений

научные вычисления, которые моделируют и имитируют такие явления, как глобальный климат, эволюция солнечной системы или результат действия нового лекарства

графика и обработка изображений, включая создание спецэффектов в кино

крупные комбинаторные или оптимизационные задачи, например, планирование авиаперелетов или экономическое моделирование

Признаки

количество процессов (потоков) равно числу процессоров

процессы выполняют одни и те же действия, но с собственной частью данных (параллельность по

Основные классы научных приложений

Сеточные вычисления для приближенных решений дифференциальных уравнений в частных производных

Точечные вычисления для моделирования систем взаимодействующих тел

Матричные вычисления для решения систем линейных уравнений

Этапы разработки параллельной программы

Последовательная программа

выбор наилучшего алгоритма

оптимизация

Параллельная программа

коррекция алгоритма
(наилучший параллельный алгоритм ≠ наилучший последовательный алгоритм)

распределение вычислений между процессами
(***производительность определяться временем работы наиболее загруженного процессора***)

дизайн взаимодействий и синхронизации между процессами
(***необходимо избегать состояния гонок и взаимных блокировок***)

Закон Амдала

T_1 – время выполнения последовательной программы

T_n – время выполнения параллельной программы на n процессорах

$$S = \frac{T_1}{T_n}$$ – ускорение параллельной программы

τ_s – время выполнения последовательной части алгоритма

τ_n – время выполнения параллельной части алгоритма

$$T_1 = \tau_s + \tau_n$$

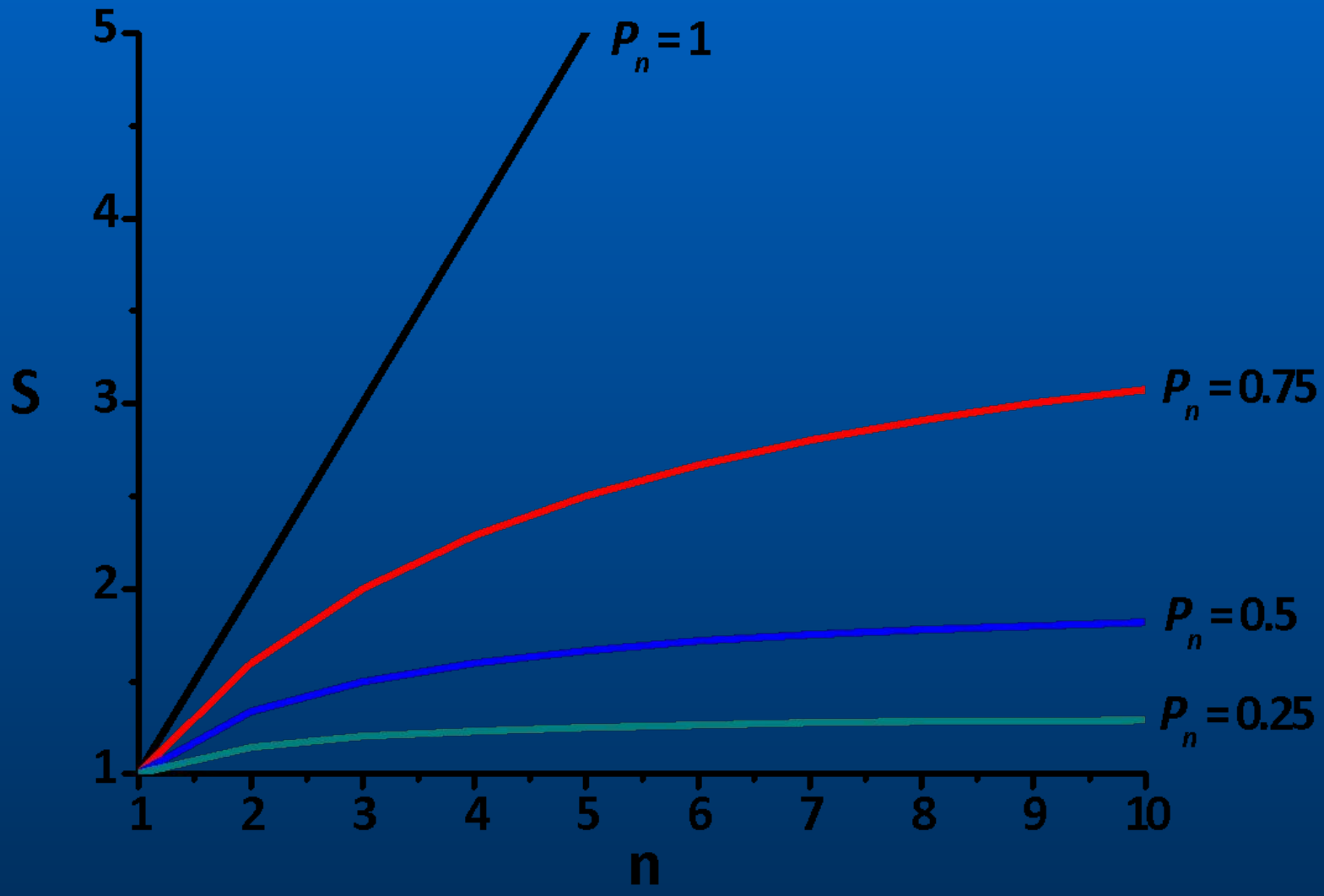
$$T_n = \tau_s + \frac{\tau_n}{n}$$

$$\Rightarrow \quad S = \frac{\tau_s + \tau_n}{\tau_s + \frac{\tau_n}{n}} = \frac{1}{P_s + \frac{P_n}{n}}$$

$$P_s = \tau_s / (\tau_s + \tau_n)$$ – доля последовательной части алгоритма

$$P_n = \tau_n / (\tau_s + \tau_n)$$ – доля параллельной части алгоритма

Закон Амдала



Сетевой закон Амдала

τ_c – время затрачиваемое на

- создание процессов и их диспетчеризацию
- взаимодействие процессов
- синхронизацию

$$T_n = \tau_s + \frac{\tau_n}{n} + \tau_c$$

– время выполнения параллельной программы на n процессорах

$$S = \frac{\tau_s + \tau_n}{\tau_s + \frac{\tau_n}{n} + \tau_c} = \frac{1}{P_s + \frac{P_n}{n} + P_c}$$

$$P_c = \tau_c / \left(\tau_s + \tau_n\right)$$

– доля времени приходящегося на обслуживание параллельной программы

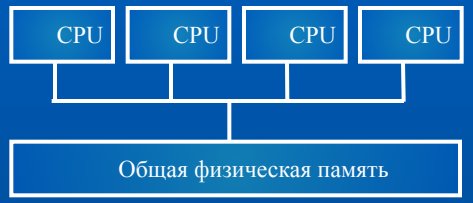
OpenMP

Параллельные архитектуры

Системы с общей памятью

- Многопоточное программирование с помощью нитей (threads)
- OpenMP – организация нитей с помощью директив компилятора
- Обмен данными между процессами через обмен сообщениями (Message Passing Interface - MPI)

SMP – архитектура

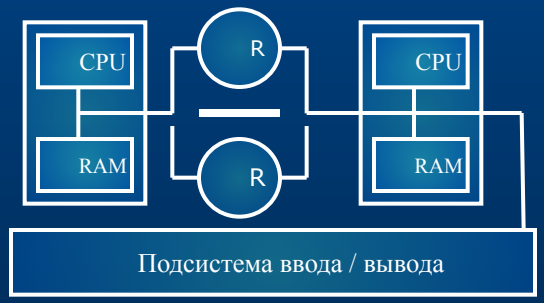


SMP ccNUMA – архитектура

Системы с распределенной памятью

- Обмен данными между процессами через обмен сообщениями (Message Passing Interface - MPI)
- Библиотеки обмена сообщениями (Message Passing Libraries - MPL):
  - Message Passing Interface – MPI
  - Parallel Virtual Machine – PVM
  - Shmem от Cray и SGI

MPP – архитектура



Основные принципы OpenMP

- OpenMP – это интерфейс прикладного программирования для создания многопоточных приложений в вычислительных системах с общей памятью
- OpenMP состоит из набора директив компиляторов и библиотек специальных функций
- OpenMP позволяет легко и быстро создавать многопоточные приложения на алгоритмических языках FORTRAN, C, C++
- Стандарты OpenMP разрабатывались в течении последних 15 лет, применительно к архитектурам с общей памятью
- OpenMP поддерживается следующими основными разработчиками
  - Оборудования: Intel, HP, SGI, Sun, IBM, …
  - Системного программного обеспечения: Intel, KAI, PGI, PSR, APR, ...
  - Прикладного программного обеспечения: ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, ТЕСИС, ЦГЭ, …

Стандарт OpenMP

- **директивы**
- **процедуры**
- **переменные среды**

  - OpenMP ARB (ARchitecture Board)
  - www.openmp.org
  - для языков Fortran и C/C++ - 1997/98 гг.
  - концепции
    - X3H5 (ANSI стандарт, 1993) - MPP
    - POSIX Threads (*Ptheads*), IEEE 1003.1c, 1995 – Unix

Стандарт OpenMP

- **Версии 1.0-2.5 (1997 – 2005)**

  внедрение и развитие потокового распараллеливания циклов
- **Версии 3.0, 3.1 (2008 – 2011)**

  добавление и развитие поддержки независимых задач
- **Версия 4.0 (2013), 4.5 (2015)**

  векторизация циклов (SIMD),
  поддержка ускорителей (target),
  …
- **Версия 5.0 (2018)**

  task reductions, != в циклах,

  полная поддержка специализированных аппаратных ускорителей

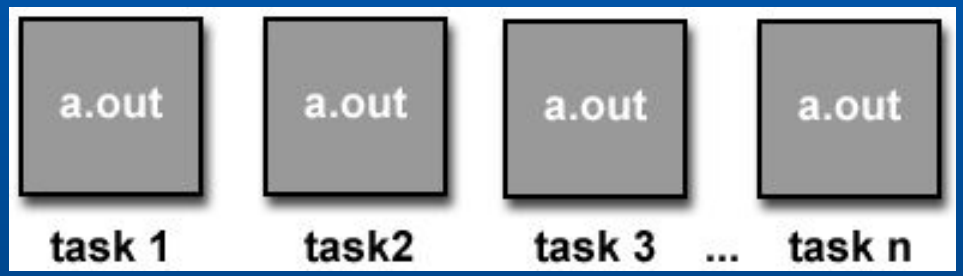
  поддержка C++11, C++14, and C++17 features

  …

Достоинства OpenMP

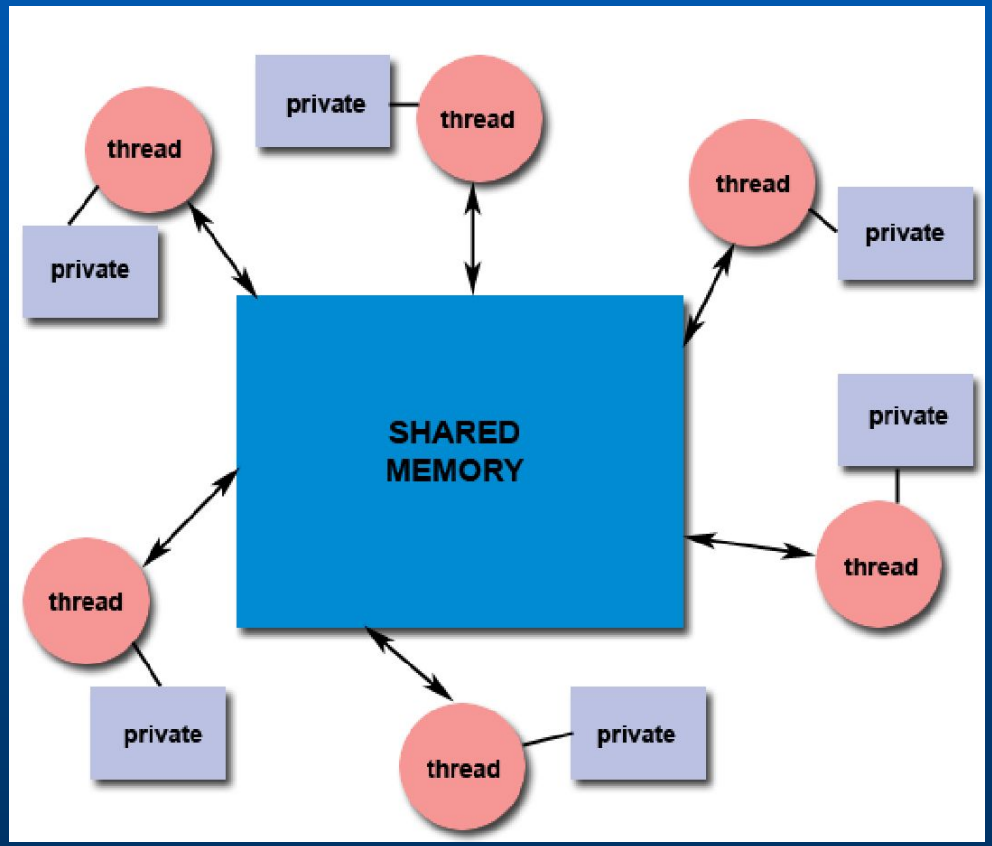
Single Program Multiple Data

```
program parallel
    ...
          if (процесс=мастер) then
        master
    else
        slave
    end if
    ...
end
```



- **инкрементальное распараллеливание**
- **гибкость контроля и единственность разрабатываемого кода**
- **эффективность**
- **стандартизованность**

Модель с разделяемой (общей) памятью



Начала программирования в OpenMP

- В моделях с общей памятью для обмена данными между потоками следует использовать общие переменные
- При этом возможен конфликт при доступе к данным
- Для предотвращения таких конфликтов следует использовать процедуру синхронизации (synchronization)
- Следует иметь ввиду, что процедура синхронизации очень дорогая операция и ее желательно избегать или применять как можно реже

Структура параллельной программы

- **Набор директив компилятора**
  - определение параллельной области
  - разделение работы
  - синхронизация
- **Библиотека функций**
- **Набор переменных окружения**

Формат записи директив

- **Формат**

  C/C++

  #pragma omp имя_директивы [clause,…]

  FORTRAN

  c$omp имя_директивы [clause,…]

  !$omp имя_директивы [clause,…]

  ***$omp имя_директивы [clause,…]**

- **Пример**

  #pragma omp parallel default(shared) private(beta,pi)

Основные конструкции OpenMP

- Большинство директив OpenMP применяется к структурным блокам. Структурные блоки – это последовательность операторов с одной точкой входа в начале блока и одной точкой выхода в конце блока.

Структурный блок

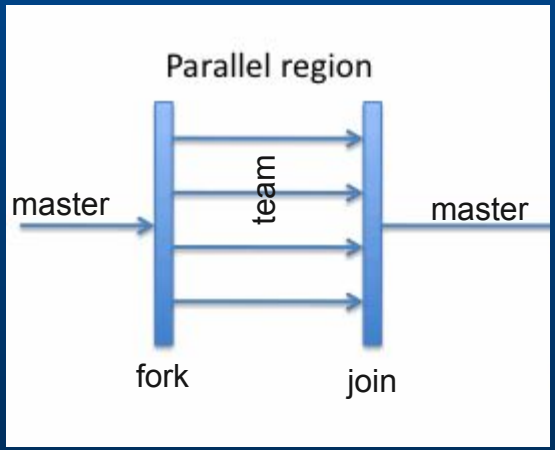
```
c$omp parallel

10     wrk(id) = junk(id)
       res(id) = wrk(id)**2
       if (conv(res)) goto 10

c$omp end parallel

       print *, id
```

Неструктурный блок

```
c$omp parallel

10     wrk(id) = junk(id)
30     res(id) = wrk(id)**2
       if (conv(res)) goto 20
       goto 10

c$omp end parallel

       if (not_done) goto 30
20     print *, id
```

Порождение нитей

- **PARALLEL [clause,…] … END PARALLEL**
  (основная директива OpenMP)
  - создается набор (*team*) из N потоков (входной поток является основным потоком этого набора (*master thread*) и имеет номер 0)
  - Код области дублируется или разделяется между потоками для параллельного выполнения
  - в конце области синхронизация потоков – выполняется ожидание завершения вычислений всех потоков; дальнейшие вычисления продолжает выполнять только основной поток.



Порождение нитей

- FORTRAN

```
c$omp parallel
c$omp& shared(var1, var2, …)
c$omp& private(var1, var2, …)
c$omp& firstprivate(var1, var2, …)
c$omp& reduction(operator|intrinsic:var1, var2, …)
c$omp& if(expression)
c$omp& default(private|shared|none)

      [ Структурный блок программы]

c$omp end parallel
```

- C/C++

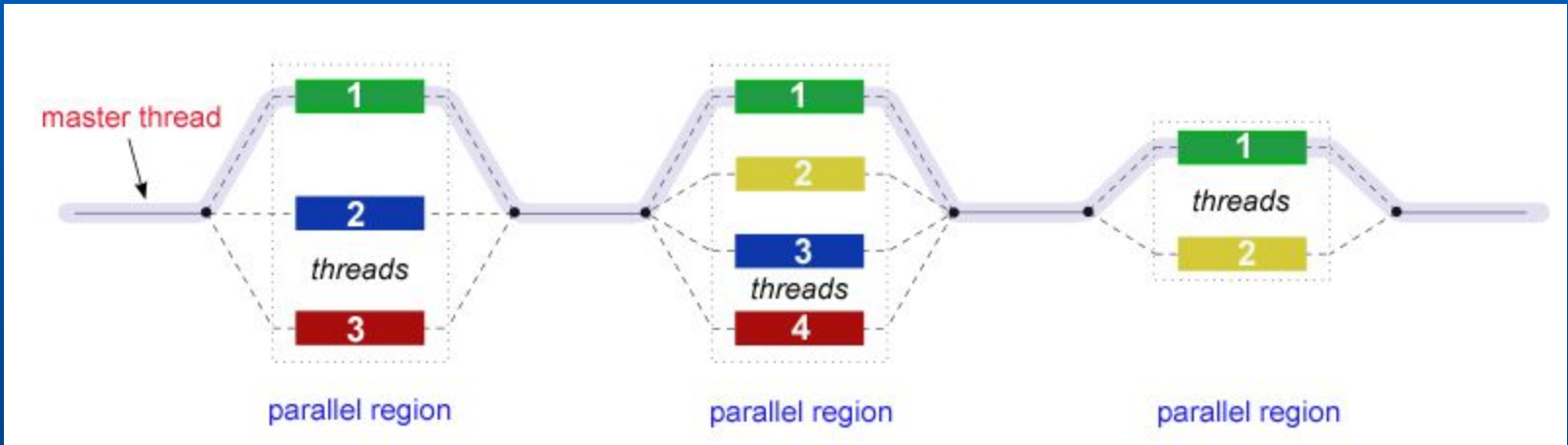
```
#pragma omp parallel                                \
        private (var1, var2, …)                     \
        shared (var1, var2, …)                      \
        firstprivate(var1, var2, …)                 \
        copyin(var1, var2, …)                       \
        reduction(operator:var1, var2, …)           \
        if(expression)                              \
        default(shared|none)                        \
{

   [ Структурный блок программы]

}
```

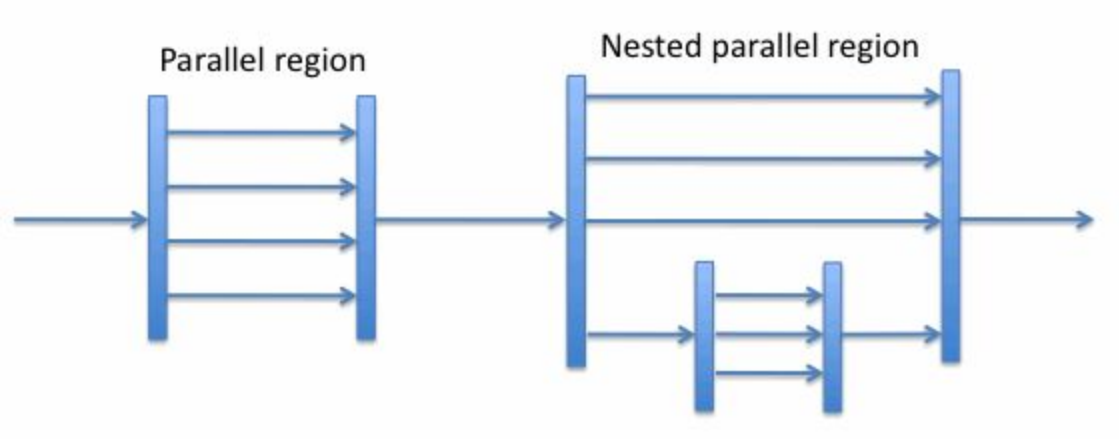
Определение параллельной области

- Количество потоков определяется
  - переменной окружения **OMP_NUM_THREADS**
  - функцией **omp_set_num_threads()**
- Каждый поток имеет свой номер thread_number
  - определяется функцией omp_get_thread_num() - от нуля (главный поток) и до OMP_NUM_THREADS-1)
- Каждый поток выполняет структурный блок программы, включенный в параллельный регион
- В общем случае синхронизации между потоками нет. После завершения параллельного блока все потоки за исключением главного прекращают свое существование

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

Модель выполнения





Определение параллельной области

- **Режимы выполнения (Execution Mode) параллельных блоков**
  - динамический (Dynamic Mode) – количество потоков определяется ОС

  переменная окружения OMP_DYNAMIC

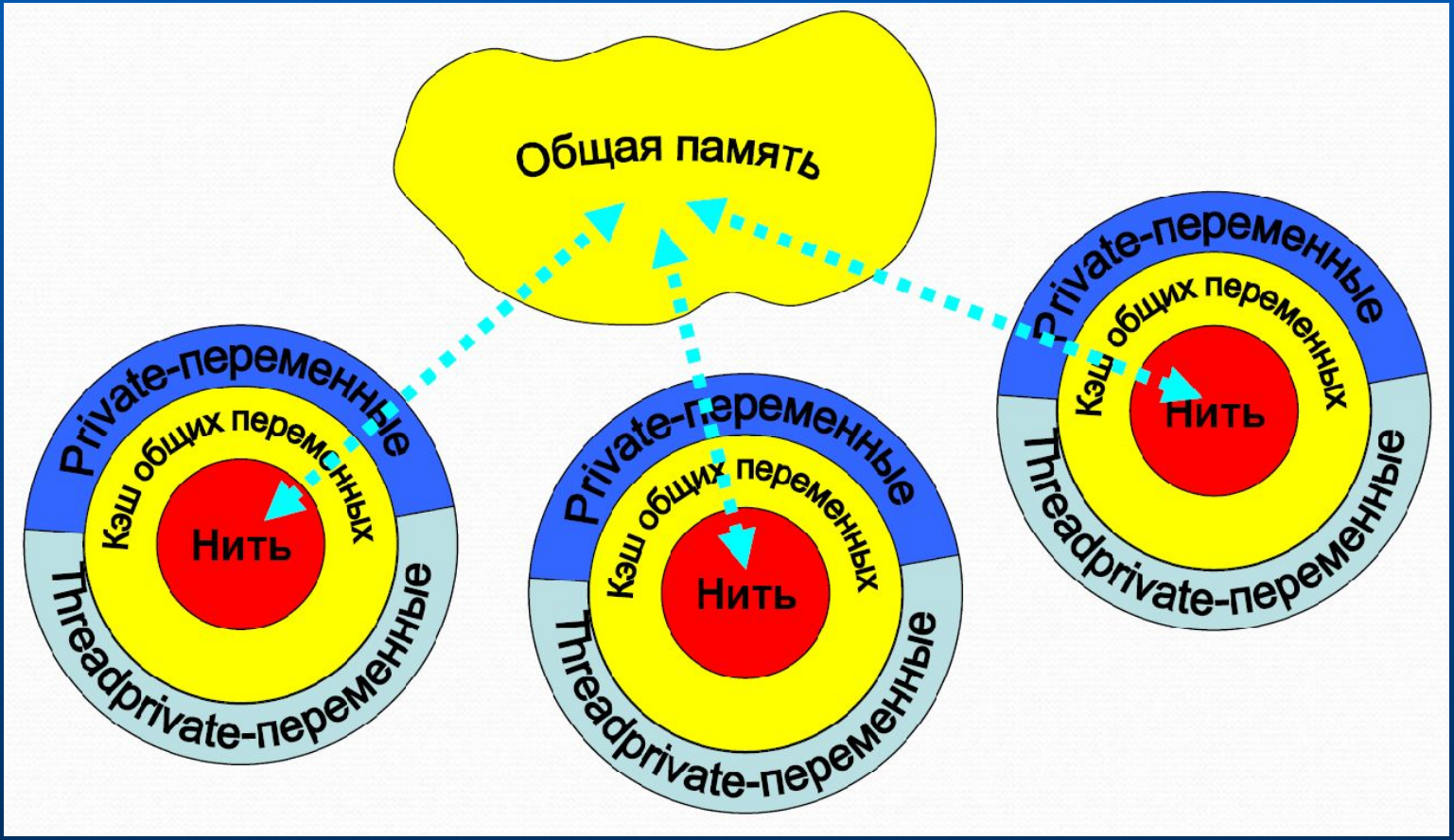
  функция omp_set_dynamic()
  - статический (Static Mode) – количество потоков определяется программистом

  переменная окружения OMP_STATIC

  функция omp_set_static()
- **Возможна вложенность параллельных структурных блоков (во многих реализациях не обеспечивается)**

  по умолчанию во вложенной области создается один поток управление - функция omp_set_nested() или переменная OMP_NESTED

Модель памяти



*Технология параллельного программирования OpenMP. © Бахтин В.А

Директивные предложения (clauses) OpenMP

- **shared(var1, var2, …)**

  переменные var1,… являются общими для всех потоков и относятся к одной области памяти
- **private(var1, var2, …)**

  переменные var1, var2,… имеют собственные значения внутри каждого потока и относятся к разным областям памяти
- **firstprivate(var1, var2, …)**

  переменные имеют тип private и инициализируются в начале структурного блока
- **lastprivate(var1, var2, …)**

  переменные имеют тип private и сохраняют свои значения при выходе из структурного блока
- **if(condition)**

  следующий параллельный блок выполняется только в том случае, если condition=TRUE
- **default(shared|private|none)**

  определяет по умолчанию тип всех переменных определяемых по умолчанию в следующем параллельном структурном блоке
- **schedule (type [,chunk])**

  определяет распределение петель циклов по потокам
- **reduction (operator | intrinsic: var1, var2, …)**

  гарантирует безопасное вычисление разностей, сумм, произведений и т.д. (operator задает операцию) по петлям циклов

Примеры реализации предложений

private

```
c$omp  parallel shared(a)
       private(myid,x)
      myid=omp_get_thread_num()
      x = work(myid)
      if (x < 1.0) then
        a(myid) = x
      end if
```

- Каждый поток имеет свою собственную копию переменных “x” и “myid”
- Значение “x” будет неопределенным, если не определить “x” как private
- Значения private-переменных не определены до и после блока параллельных вычислений

default

```
      parallel do default(private)
       shared(a)
      …
```

- Описание default автоматически определяет переменные “x” и “myid” как private

Пример реализации предложения firstprivate

```
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 каждом параллельном потоке используется своя переменная “c”, но значение этой переменной перед входом в параллельный блок программы берется из предшествующего последовательного блока

Пример реализации предложения lastprivate

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

      n = i
```

- В этом случае переменная “i” определена для каждого потока в параллельном блоке
- После завершения параллельного блока переменная “i” сохраняет последнее значение, полученное в параллельном блоке, при условии его последовательного выполнения, т.е. n=N+1

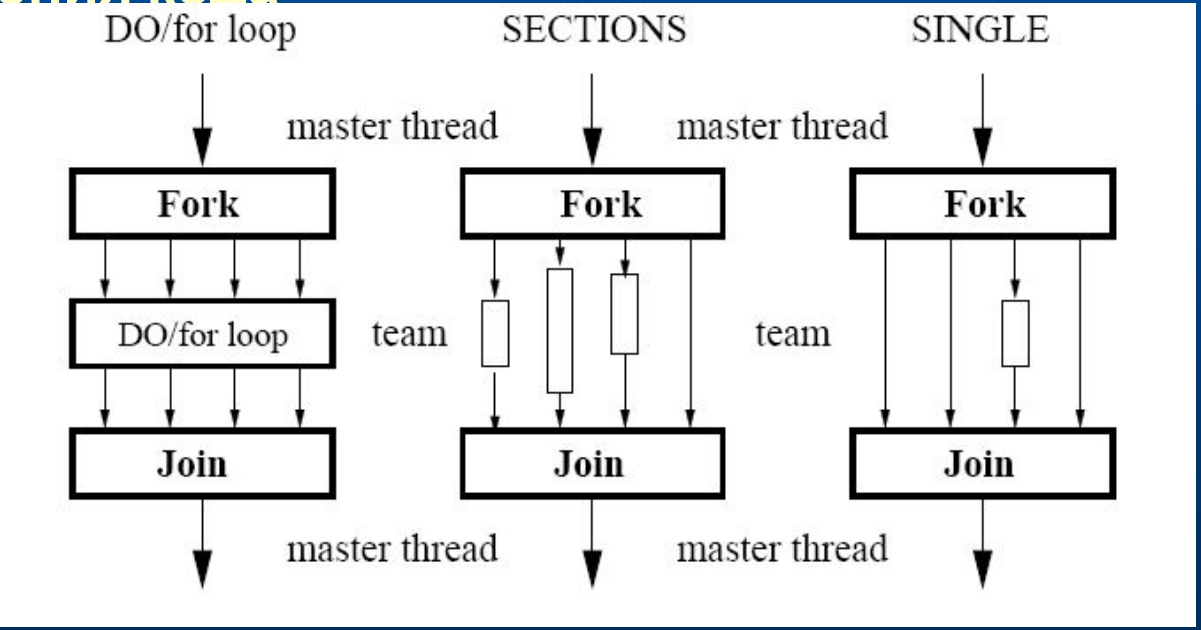
Пример реализации предложения if

```
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 !!!
- Трудоемкость образования потоков ~ 1000 операций деления!!!

Разделение работы (work-sharing constructs)

- Do/for - распараллеливание циклов (параллелизм данных)
- Sections - функциональное распараллеливание
- Single - директива для указания последовательного выполнения кода



Конструкции разделения работы

- to be continued

Переменные окружения OpenMP

- OMP_NUM_THREADS - определяет число нитей для исполнения параллельных областей приложения.
- OMP_SCHEDULE - определяет способ распределения итераций в цикле, если в директиве DO использовано предложение SCHEDULE(RUNTIME).
- OMP_DYNAMIC - разрешает или запрещает динамическое изменение числа нитей.
- OMP_NESTED - разрешает или запрещает вложенный параллелизм.
  - Пример

```
$ export OMP_NUM_THREADS=16
$ setenv OMP_SCHEDULE "guided,4"
$ export OMP_DYNAMIC=false
$ setenv OMP_NESTED TRUE
```

Некоторые функции OpenMP

- (void) omp_set_num_threads(int num_threads) – задает число потоков в области параллельных вычислений
- int omp_get_num_threads() – возвращает количество потоков в текущий момент
- int omp_get_max_threads() – возвращает максимальное количество потоков, которое может быть возвращено функцией omp_get_num_threads
- int omp_get_thread_num() – возвращает номер потока от 0 до количество потоков - 1
- int omp_get_num_procs() – возвращает количество процессоров доступных программе
- (int/logical) omp_in_parallel() – возвращает TRUE из параллельной области программы и FALSE из последовательной
- (void) omp_set_dynamic(TRUE | FALSE) – определяет или отменяет динамический режим
- (int/logical) omp_get_dynamic() – возвращает TRUE, если режим динамический, и FALSE в противном случае
- (void) omp_set_nested(TRUE | FALSE) – определяет или отменяет вложенный режим для параллельной обработки
- (int/logical) omp_get_nested() – возвращает TRUE, если режим вложенный, и FALSE в противном случае

- http://www.openmp.org/specifications/

Пример разделения работы

```
int sum_openmp (int *data, int n)
{
        int res = 0;                                    // результат
        int max_num_th = omp_get_max_threads();  // максимальное число
        потоков
        int* sub_sum = new int[max_num_th]();       // массив частичных сумм
        int id = 0;

        #pragma omp parallel shared(sub_sum, n) private (id)
        {
           id = omp_get_thread_num();                // уникальный номер
           int num_th = omp_get_num_threads();  //реальное число потоков

           int portion = n/num_th;                   // порция на поток
           int i_start= portion *id,  i_end = i_start + portion;
           int modulus = n%num_th;
           // коррекция порции
           if (id < modulus ) {
                i_start += id, i_end += id + 1;
           }  else {
                i_start += modulus , i_end += modulus ;
           }
           // расчёт частичных сумм
           for (int i = i_start; i < i_end; i++)  sub_sum[id] += data[i];
        }
        // расчёт полной суммы
        for (id = 0; id < max_num_th; id++) res +=  sub_sum[id] ;

        delete[] sub_sum;
        return res;
}
```