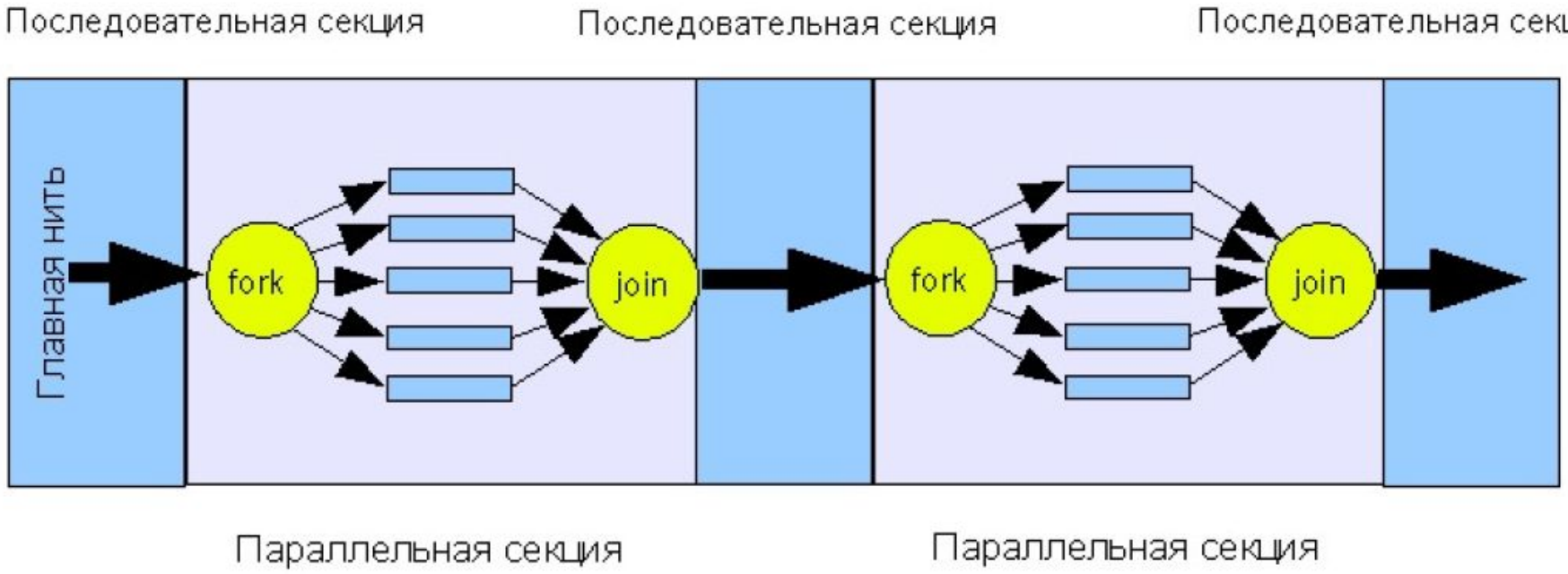
Технология **OpenMP**

- OpenMP - стандарт программного интерфейса приложений для параллельных систем с общей памятью. Поддерживает языки C, C++, Фортран.
- Первая версия появилась в 1997 году, предназначалась для языка Fortran. Для C/C++ версия разработана в 1998 году. В 2008 году вышла версия OpenMP 3.0.
- Есть аналог в .NET - Библиотека параллельных задач (TPL)

Модель программы в ***OpenMP***

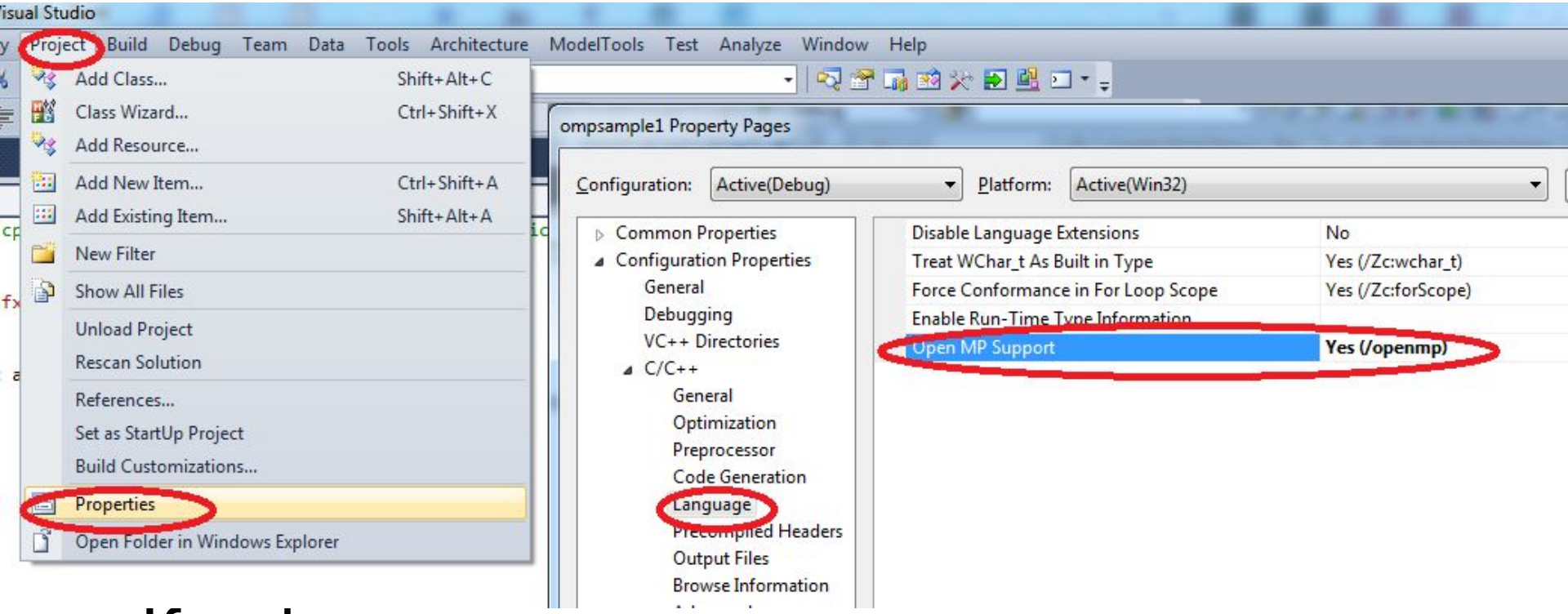


- Программа состоит из последовательных и параллельных секций.
- В начальный момент времени создается главная нить, выполняющая последовательные секции программы.
- При входе в параллельную секцию выполняется операция *fork*, порождающая семейство нитей. Каждая нить имеет свой уникальный числовой идентификатор (главной нити соответствует 0). При распараллеливании циклов все параллельные нити исполняют один код. В общем случае нити могут исполнять различные фрагменты кода.
- При выходе из параллельной секции выполняется операция *join*. Завершается выполнение всех нитей, кроме главной.

Компоненты OpenMP

- Директивы компилятора - используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются в исходный текст программы.
- Подпрограммы (функции) библиотеки времени выполнения - используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются в исходный текст программы.
- Переменные окружения - используются для управления поведением параллельной программы. Переменные окружения задаются для среды выполнения параллельной программы соответствующими командами.

Подключение



- Конфигурацию рекомендуется установить в release

Сборка программы

- `gcc -fopenmp -o test test.c`
- `icc -openmp -o test test.c`

Программа «Hello World»

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
      printf("Hello World\n");
    }

    return 0;
}
```

Директивы OpenMP

#pragma omp задает границы параллельной секции программы. С данной директивой могут

- использоваться следующие операторы:
- private;
- shared;
- default;
- firstprivate;
- reduction;
- if;
- copyin;
- num_threads.

#pragma omp for

Задает границы цикла, исполняемого в параллельном режиме.

```
#pragma omp parallel {
    #pragma omp for
    for(int i = 1; i < size; ++i)
     x[i] = (y[i-1] + y[i+1])/2;
}
```

С данной директивой могут использоваться следующие операторы:

- private;
- firstprivate;
- lastprivate;
- reduction;
- schedule;
- ordered;
- nowait.

Зависимости данных и гонки в циклах

- При распараллеливании цикла не должно быть зависимости между итерациями цикла

```
void simple(int n, float *a, float *b)
{
   #pragma omp parallel for
   for (int i=1; i<n; i++){
      a[i+1] = i * a[i];
   }
}
```

```
#pragma omp parallel {
   for(int i = 1; i < size; ++i)
    x[i] = (y[i-1] + y[i+1])/2;
} // каждый поток выполнит полный цикл for,
   // проделав много лишней работы


//сокращенный способ записи
#pragma omp parallel for
for(int i = 1; i < size; ++i)
   x[i] = (y[i-1] + y[i+1])/2;
```

- При распараллеливании циклов вы должны убедиться в том, что итерации цикла не имеют зависимостей. Если цикл не содержит зависимостей, компилятор может выполнять цикл в любом порядке, даже параллельно. Соблюдение этого важного требования компилятор не проверяет — вы сами должны заботиться об этом. Если вы укажете компилятору распараллелить цикл, содержащий зависимости, компилятор подчинится, что приведет к ошибке.
- Кроме того, OpenMP налагает ограничения на циклы for, которые могут быть включены в блок #pragma omp for или #pragma omp parallel for block. Циклы for должны соответствовать следующему формату:

  for([целочисленный тип] i = инвариант цикла; i {<,>,=,<=,>=} инвариант цикла; i {+,-}= инвариант цикла)

Общие и частные данные

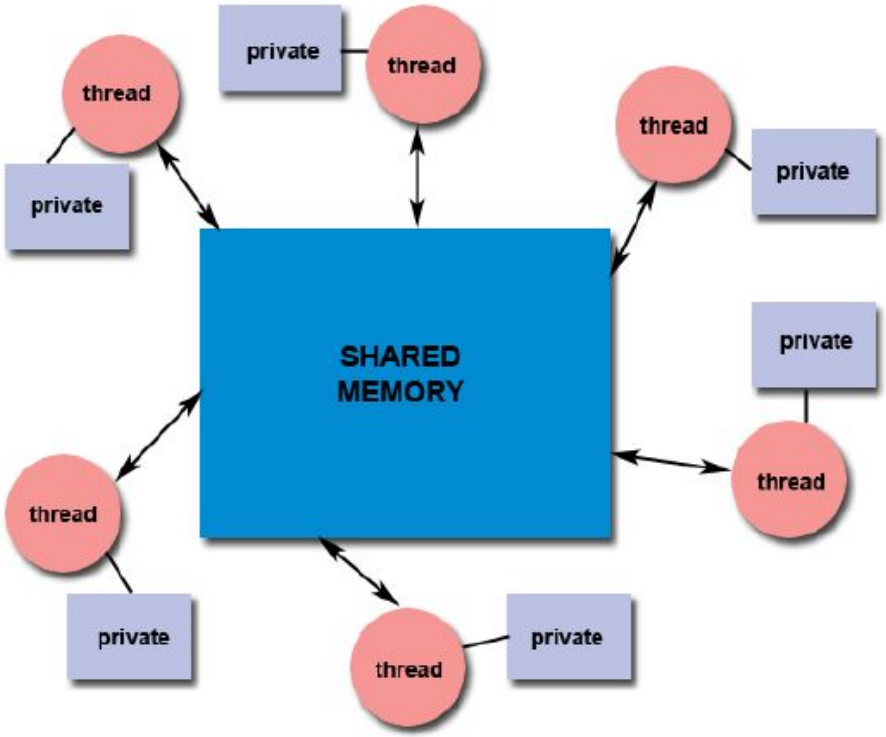
- Разрабатывая параллельные программы, вы должны понимать, какие данные являются общими (shared), а какие частными (private), — от этого зависит не только производительность, но и корректная работа программы. В OpenMP это различие очевидно, к тому же вы можете настроить его вручную.
- Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельном регионе. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.
- По умолчанию все переменные в параллельном регионе — общие, но из этого правила есть три исключения.

О данных

- В параллельныз программах все данные имеют "метки":
  - Метка "Private" ⇨ видима только одному потоку
    - Изменения в переменной локальны и не видны другим потокам
    - Пример — локальная переменная в функции, которая исполняется параллельно
  - Метка "Shared" ⇨ видима всем потокам
    - Изменения в переменной видны всем потокам
    - Пример — глобальные данные

Модель с разделяемой памятью

- Все потоки имеют доступ к глобальной разделяемой памяти
- Данные могут быть разделяемые и приватные
- Разделяемые данные доступны всем потокам
- Приватные — только одному
- Синхронизация требуется для доступа к общим данным



- Во-первых, частными являются индексы параллельных циклов for.
- Во-вторых, частными являются локальные переменные блоков параллельных регионов.
- В-третьих, частными будут любые переменные, указанные в разделах private, firstprivate, lastprivate и reduction.

Параллелизация цикла с помощью OpenMP

```
#pragma omp parallel shared(a,b)
{
#pragma omp for private(i)
  for(i=0; i<10000; i++)
    a[i] = a[i] + b[i];
}
```

```
float sum = 10.0f;
MatrixClass myMatrix;
/*Переменная j по умолчанию не является частной, но явно сделана
    таковой через раздел firstprivate.*/
int j = myMatrix.RowStart();
int i;
#pragma omp parallel {
    /*i, j и sum сделаны частными для каждого потока из группы,
    т. е. каждый поток будет располагать своей копией каждой из этих
    переменных.
    */
    #pragma omp for firstprivate(j) lastprivate(i) reduction(+: sum)
    for(i = 0; i < count; ++i) {
     /* переменная doubleI, потому что она объявлена в параллельном
    регионе. Любые нестатические и не являющиеся членами класса
    MatrixClass переменные, объявленные в методе myMatrix::GetElement,
    будут частными.*/
     int doubleI = 2 * i;
     for(; j < doubleI; ++j){
           sum += myMatrix.GetElement(i, j);
     }
    }
}
```

- Раздел **private** говорит о том, что для каждого потока должна быть создана частная копия каждой переменной из списка. Частные копии будут инициализироваться значением по умолчанию (с применением конструктора по умолчанию, если это уместно). Например, переменные типа int имеют по умолчанию значение 0.
- У раздела **firstprivate** такая же семантика, но перед выполнением параллельного региона он указывает копировать значение общей переменной в каждый поток, используя конструктор копирования, если это уместно.
- Семантика раздела **lastprivate** тоже совпадает с семантикой раздела private, но при выполнении последней итерации цикла или раздела конструкции распараллеливания значения переменных, указанных в разделе lastprivate, присваиваются переменным основного потока. Если это уместно, для копирования объектов применяется оператор присваивания копий (copy assignment operator).

Условие first/last private

- firstprivate (list)
  - Всем приватным переменным в списке присваивается значение исходных переменных до начала параллельного региона
- lastprivate (list)
  - Переменным присваивается значение того потока, который бы последним исполнялся последовательно.

- раздел reduction, но он принимает переменную и оператор. Поддерживаемые этим разделом операторы перечислены в таблице , а у переменной должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д.).
- Переменная раздела reduction инициализируется в каждом потоке значением, указанным в таблице. В конце блока кода оператор раздела reduction применяется к каждой частной копии переменной, а также к исходному значению переменной.

Операторы раздела reduction

Оператор раздела reduction	Инициализированное (каноническое) значение
+	0
*	1
-	0
&	~0 (FF...F)
|	0
^	0
&&	1
||	0

nowait - Отменяет барьерную синхронизацию при завершении выполнения параллельной секции.

schedule - По умолчанию в OpenMP для планирования параллельного выполнения циклов for применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если n — число итераций цикла, а T — число потоков в группе, каждый поток выполнит n/T итераций (если n не делится на T без остатка, ничего страшного). Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование в период выполнения (runtime scheduling) и управляемое планирование (guided scheduling).

ordered - обеспечивает сохранение того порядка выполнения итераций цикла, который соответствует последовательному выполнению программы.

Некоторые OpenMP условия

Об OpenMP условиях

- Большинство OpenMP директив поддерживают условия
- Служат для задания дополнительной информации директивам
- Например, **private(a)** для директивы **for**:
  - **#pragma omp for *private(a)***

Условия if/private/shared

- if (скалярное выражение)
  - Исполнить параллельно, если выражение истино
  - В противном случае - последовательно
- private (list)
  - Переменные не связаны с исходным объектом
  - Все переменные локальны
  - При входе и выходе значение переменных не определено
- shared (list)
  - Данные доступны всем потокам в группе
  - Все потоки имеют доступ к одним и тем же адресам

Пример

```
#omp parallel for private(i,j) \
    shared(a,b,c) if(M>100)
{
    for(i=0; i<M; i++)
        for(j=0; j<100; j++)
            a[i] = b[i][j]*c[j];
}
```

Параллельная обработка в конструкциях, отличных от циклов

- OpenMP поддерживает параллелизм и на уровне функций. Этот механизм называется секциями OpenMP (OpenMP sections).

Синтаксис:

```
#pragma omp parallel sections
{
   #pragma omp section
    //Вызов первой функции
   #pragma omp section
    //Вызов второй функции
}
```

Основные директивы OpenMP

- Распределение задач в потоках текущей группы

#pragma omp sections {section1, section2, …}

#pragma omp section *structured-block*

```
#pragma omp parallel {
  #pragma omp sections {
    #pragma omp section
       Task1();
    #pragma omp section
       Task2();
  }
}
```

```
void QuickSort (int numList[], int nLower, int nUpper)
 {
    if (nLower < nUpper)
    {
    // Разбиение интервала сортировки
    int nSplit = Partition (numList, nLower, nUpper);
     #pragma omp parallel sections
     {
          #pragma omp section
          QuickSort(numList, nLower, nSplit - 1);
          #pragma omp section
          QuickSort (numList, nSplit + 1, nUpper);
     }
    }
}
```

Синхронизация

- Барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Используется **#pragma omp barrier.**
- Критическая секция - директива **#pragma omp critical [имя].** Если используется именованная критическая секцию, тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем неименованным критическим секциям является взаимоисключающим.

Барьер

- Предположим мы выполняем следующий код:

```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

- Если циклы выполнять параллельно, то может быть неправильный ответ
- Нужна синхронизация по доступу к a[i]

Barrier

- Каждый поток ждет, пока все потоки достигнут определенную точку:
  - #pragma omp barrier

Критические секции

- Если sum разделяемая переменная, то цикл нельзя исполнять параллельно

```
for (i=0; i < N; i++){
        .....
      sum += a[i];
        .....
   }
```

- Можно использовать критическую секцию:

```
for (i=0; i < N; i++){
        .....
  //one at a time can proceed
     sum += a[i];
  //next in line, please
        .....
   }
```

Критическая секция

- Полезны для избавления от ошибок соревнования, чтения записи данных (неопределенный порядок)
- Может привести к тому, что параллельная программа станет последоовательной
- Все потоки исполняют код, но не одновременно:
  - #pragma omp critical [(name)]
    {<code-block>}
  - #pragma omp atomic
    <statement>

Пример работы директивы critical

```
void critical_example(float *x)
{
   int ix_next;
   #pragma omp parallel shared(x) private(ix_next)
   {
      #pragma omp critical (critical_section_name)
         ix_next = dequeue(x);
      doSomething(ix_next);
    }
}
```

- В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, — например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива **#pragma omp single.**
- Иногда возможностей директивы single недостаточно. В ряде случаев требуется, чтобы блок кода был выполнен основным потоком, — например, если этот поток отвечает за обработку GUI и вам нужно, чтобы какую-то задачу выполнил именно он. Тогда применяется директива **#pragma omp master.** В отличие от директивы single при входе в блок master и выходе из него нет никакого неявного барьера.
- Чтобы завершить все незавершенные операции над памятью перед началом следующей операции, используйте директиву **#pragma omp flush**

Функции OpenMP

void omp_set_num_threads(int threads);

Задает количество потоков (threads) при выполнении параллельных секций программы.

int omp_get_num_threads(void);

Возвращает количество потоков, используемых для выполнения параллельной секции.

int omp_get_max_threads(void);

Возвращает максимальное количество потоков, которые можно использовать для выполнения параллельных секций программы.

int omp_get_thread_num(void);

Возвращает идентификатор нити, из которой вызывается данная функция.

- **int omp_get_num_procs(void);**

Возвращает количество процессоров, доступных в данный момент программе.

- **int omp_in_parallel(void);**

Возвращает не ноль при вызове из активной параллельной секции программы.

- **void omp_set_dynamic(int threads);**

Включает или отключает возможность динамического назначения количества потоков при выполнении параллельной секции. По умолчанию эта возможность отключена.

- **int omp_get_dynamic(void);**

Возвращает не ноль, если динамическое назначение количества потоков разрешено.

Переменные окружения ***OpenMP***

- OMP_NUM_THREADS

Задает количество нитей при выполнении параллельных секций программы.

- OMP_SCHEDULE

Задает способ распределения итераций циклов между нитями. Возможные значения: static; dynamic; guided.

- OMP_DYNAMIC

Если этой переменной присвоено значение false, динамическое распределение итераций циклов между нитями запрещено, если true – разрешено.

- OMP_NESTED

Если этой переменной присвоено значение false, вложенный параллелизм запрещен, если true – разрешен.

- **void omp_set_nested(int nested);**

Разрешает или запрещает вложенный параллелизм. По умолчанию вложенный параллелизм запрещен.

- **int omp_get_nested(void);**

Определяет, разрешен ли вложенный параллелизм.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main()
{
const int SIZE= 1000;
int sumSec=0;
int arr[SIZE];
int max =0;
for(int i=0;i<SIZE; i++) {
        arr[i]=rand()%100;
        sumSec+=arr[i];
        if(arr[i]>max){
          max=arr[i];
        }
}
int sumPar=0;
int parMax[2];

#pragma omp parallel
{
        int localMax = 0;
        #pragma omp for reduction(+: sumPar)
        for(int i=0;i<SIZE; i++){
          sumPar+=arr[i];
          if( arr[i]>localMax )
                    localMax=arr[i];
        }
        parMax[omp_get_thread_num()] = localMax;
}
        printf("%d    %d" , sumSec , sumPar );
        printf("%d    %d    %d" , max , parMax[0] , parMax[1] );

}
```