Вариантом, на который согласились бы многие пользователи параллельных вычислительных систем, является использование традиционных последовательных языков программирования. Вариант не идеальный, но преимуществ и в самом деле много: сохраняется весь уже созданный программный багаж, программист продолжает мыслить в привычных для него терминах, а всю дополнительную работу по адаптации программы к архитектуре параллельных компьютеров выполняет компилятор. На первый взгляд все кажется вполне реальным, однако попробуем оценить сложность задачи *автоматического распараллеливания программ* компилятором.

Предположим, что мы хотим получить эффективную программу для параллельного компьютера с распределенной памятью. Мы специально сделали акцент на эффективности программы, поскольку именно в этом и состоит основная задача. Получить какой-нибудь исполняемый код не составляет никакой проблемы: для этого достаточно, например, просто скомпилировать программу для одного процессора. Но тогда зачем использовать параллельный компьютер, если при запуске на любой конфигурации всегда реально задействуется лишь один процессор, и время работы программы не уменьшается?

Чтобы полностью использовать потенциал данной архитектуры, необходимо решить три основные задачи:

- найти в программе ветви вычислений, которые можно исполнять параллельно;

- ❒ распределить данные по модулям локальной памяти процессоров;
- ❒ согласовать распределение данных с параллелизмом вычислений.

Если не решить первую задачу, то бессмысленно использовать многопроцессорные конфигурации компьютера. Если решена первая, но не решены последние две задачи, то все время работы программы может уйти на обмен данными между процессорами. В таком случае про масштабируемость программы можно забыть.

Указанные задачи в самом деле крайне сложны. Построить эффективные алгоритмы их решения даже для узкого класса программ очень не просто, в чем читатель легко убедится сам, если рассмотрит несколько реальных примеров.

Попробуем немного упростить ситуацию, и будем рассматривать параллельные компьютеры с общей памятью. Казалось бы, остается лишь задача определения потенциального параллелизма программы, но все ли так просто? Рассмотрим следующий фрагмент программы, состоящий всего из трех (!) строк:

```
      DO 10 i = 1, n
        DO 10 j = 1, n
10        U(i + j) = U(2*n - i - j + 1) * q + p
```

Какие итерации данной циклической конструкции являются независимыми, и можно ли фрагмент исполнять в параллельном режиме? Несмотря на исключительно маленький размер исходного текста, вопрос совсем не тривиальный. Можно изобразить информационную структуру данного фрагмента (какие элементы массива определяют какие) и по ней понять, что ни один из двух циклов параллельным не является. И даже с этой структурой сложно определить, что этот фрагмент можно преобразовать к эквивалентному, в котором циклы уже будут параллельными:

```
      DO 10 i = 1, n
        DO 20 j = 1, n - i
20        U(i + j) = U(2*n - i - j + 1)*q + p
        DO 30 j = n - i + 1, n
30        U(i + j) = U(2*n - i - j + 1)*q + p
10    continue .
```

Если исходный фрагмент (n = 1000) выполнить на векторно-конвейерном компьютере Cray C90, то его производительность составит около 20 Мфлопс при пиковой производительности почти в 1 Гфлопс. Основная причина низкой производительности заключается в том, что компилятор не может самостоятельно найти такую эквивалентную форму фрагмента, в которой все итерации внутреннего цикла были бы независимы, и, следовательно, он не может векторизовать фрагмент. Одновременно заметим, что производительность этого же компьютера на преобразованном фрагменте уже составит около 420 Мфлопс. И фрагмент состоит всего из трех строк, и все индексные выражения и границы циклов заданы явно, но определить параллельные свойства совсем не просто...

Рассмотрим теперь такой фрагмент:

```
      DO 10 i = 1, n
10      U(i) = Func(U, i)
```

где Func — это функция пользователя. Чтобы ответить на вопрос, являются ли итерации цикла независимыми, компилятор должен определить, не используются ли функцией Func элементы массива U. Если в теле данной функции где-то используется, например, элемент U(i - 1), то итерации зависимы. Если сама функция Func явно не использует массив U, но в ее теле где-то стоит вызов другой функции, которая в свою очередь и использует элемент U(i - 1), то итерации цикла опять-таки будут зависимы. В общем случае, компилятор должен уметь анализировать цепочки вызовов произвольной длины и выполнять полный межпроцедурный анализ. В некоторых случаях это сделать можно, но в общем случае задача крайне сложна.

А какой вывод может сделать компилятор о независимости итераций такого фрагмента:

```
      DO 10 i = 1, n
10      U(i) = A(i) + U(IU(i))
```

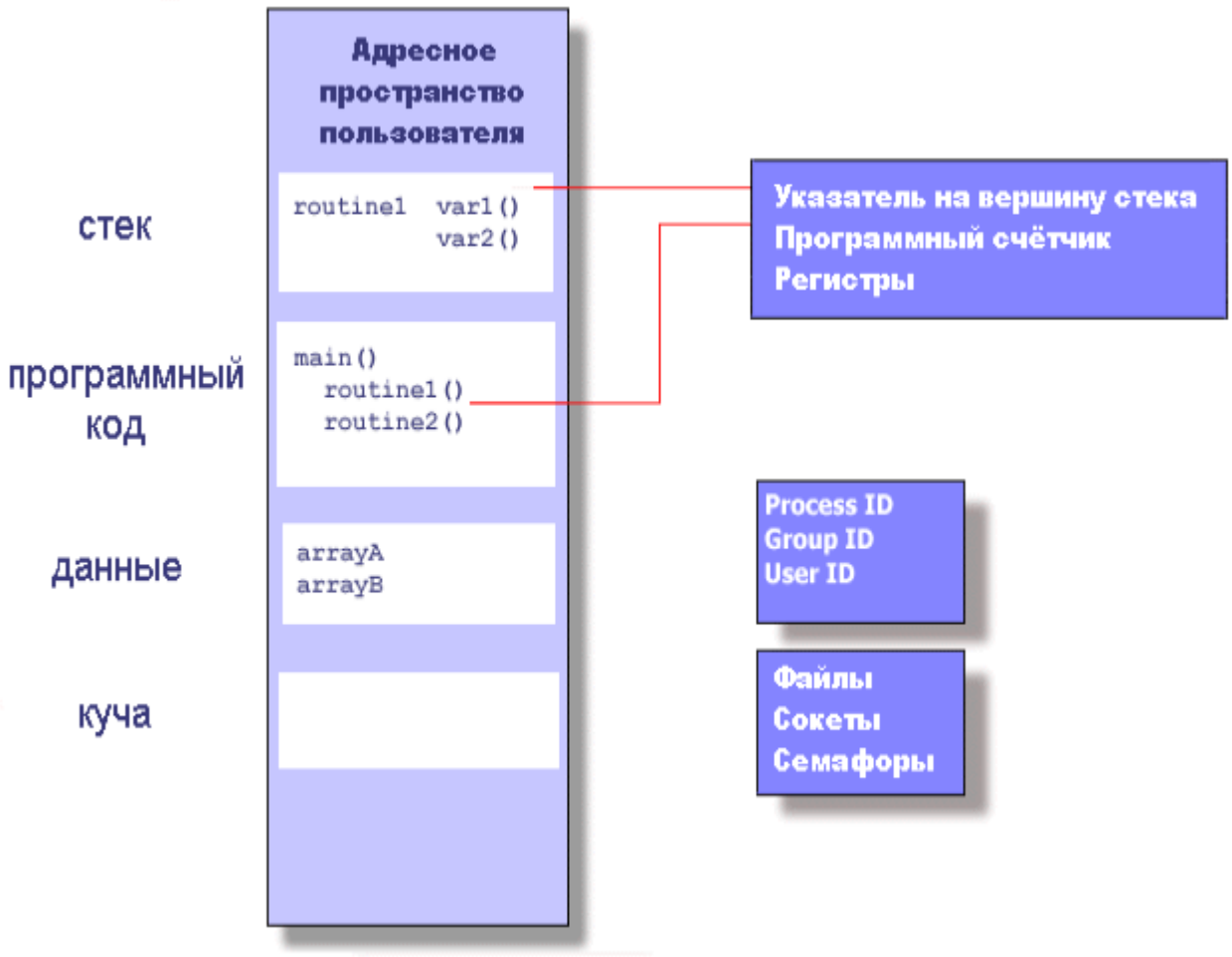
Если нет никакой априорной информации о значениях элементов массива косвенной адресации IU, а чаще всего ее нет, то ничего определенного сказать нельзя. Во всяком случае, поскольку параллельная реализация может привести к изменению результата, то компилятор "для надежности" сгенерирует последовательный код. Компилятор плохой? Нет, просто в данном случае он ничего иного в принципе сделать не может.

Подобных примеров, когда компиляторам сложно определить истинную структуру фрагмента, а значит и сложно получить его эффективную параллельную реализацию, можно привести много. Наверное, не стоит сильно винить компиляторы в неспособности решения возникающих проблем. Даже в теории полностью проанализировать произвольный фрагмент, записанный в соответствии с правилами языка программирования, невозможно. Если архитектура компьютеров не очень сложна, то компиляторы вполне в

состоянии сгенерировать эффективный код и с обычных последовательных программ. В противном случае компилятору необходимы "подсказки", содержащие указания на те или иные свойства программ.

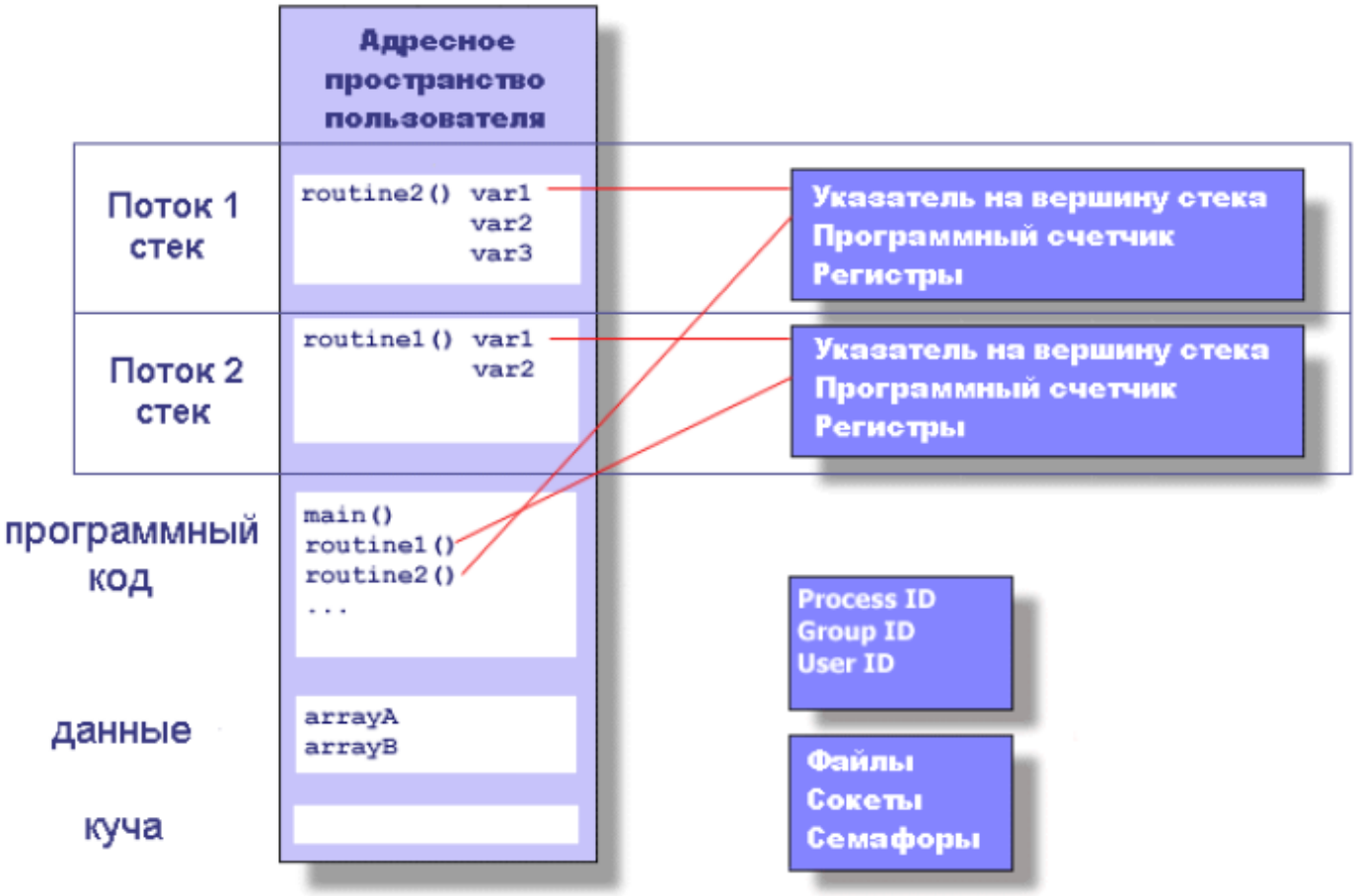
Подсказки компилятору могут быть выражены в разной форме. В одних случаях используются специальные директивы, записанные в комментариях, в других случаях в язык вводятся новые конструкции, часто используются дополнительные служебные функции или предопределенные переменные среды окружения. Типичная связка: традиционный последовательный язык + какая-либо комбинация из только что рассмотренных способов. На использовании специальных директив в комментариях к тексту программы основано и одно из самых известных расширений языка Fortran для работы на параллельных компьютерах — High Performance Fortran (HPF). В середине 90-х годов прошлого столетия с HPF были связаны очень большие надежды, поскольку язык был сразу ориентирован на разработку *переносимых* параллельных программ. Появление HPF по времени совпало с периодом бурного развития компьютеров с массовым параллелизмом, и проблема переносимости программ стала исключительно актуальной. Однако на этом пути не удалось найти приемлемого решения. Сложность конструкций HPF оказалась непреодолимым препятствием на пути создания по-настоящему эффективных компиляторов, что, естественно, предопределило отказ от него со стороны пользователей.

Процессом по определению называется среда выполнения задачи, т.е. программы:



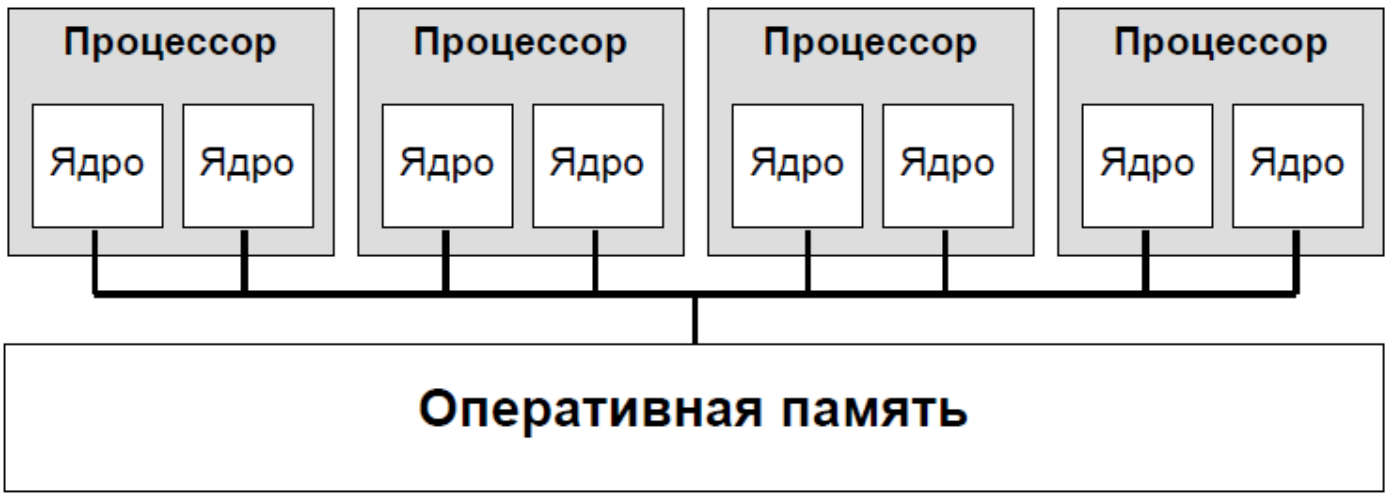
Процесс создаётся операционной системой и содержит информацию обо всех программных ресурсах и текущем состоянии выполнения программы.

В то же время, поток есть «облегчённый процесс»:



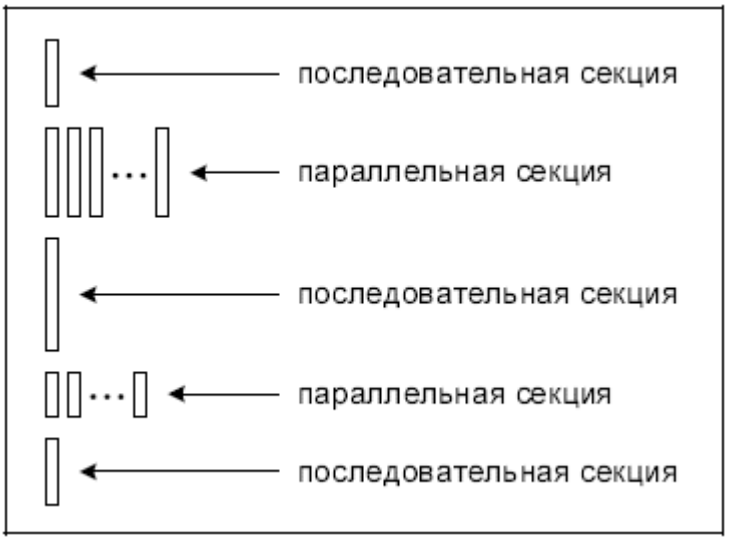
Поток создаётся в рамках процесса, но имеет свой поток управления, разделяет ресурсы создавшего его процесса с другими потоками, которые создал он же, и автоматически завершается при завершении родительского процесса.

Программирование на MPI предусматривалось для создания параллельных программ для систем с распределённой памятью, так как каждый процесс в нём занимал своё собственное адресное пространство и обмен данными происходил с помощью сообщений. В то же время для систем с общей памятью используется многопоточное программирование.



Одним из наиболее популярных средств программирования компьютеров с общей памятью, построенных на подобных принципах, в настоящее время является технология OpenMP. За основу берется последовательная программа, а для создания ее параллельной версии *пользователю предоставляется набор директив, процедур и переменных окружения.* Стандарт OpenMP разработан для языков Fortran (77, 90, и 95), C и C++ и поддерживается практически всеми производителями больших вычислительных систем. Реализации стандарта доступны как на многих UNIX-платформах, так и в среде Windows NT. Поскольку все основные конструкции для этих языков похожи, то рассказ о данной технологии мы будем вести на примере только одного из них, а именно на примере языка Fortran.

Как в рамках правил OpenMP пользователь должен представлять свою параллельную программу? Весь текст программы разбит на последовательные и параллельные области (рис. 5.2). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу оговориться, почему вместо традиционного для параллельного программирования термина "процесс" появился новый термин — "нить" (*thread*, легковесный процесс, иногда "поток"). Технология OpenMP опирается на понятие общей памяти, и поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.



Основная нить и только она исполняет все последовательные области программы. Для поддержки параллелизма используется схема FORK/JOIN. При входе в параллельную область нить-мастер порождает дополнительные нити (выполняется операция FORK). После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все порожденные нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она (выполняется операция JOIN).

В параллельной области все переменные программы разделяются на два класса: *общие* (SHARED) и *локальные* (PRIVATE). Общие переменные всегда существуют лишь в одном экземпляре для всей программы и доступны всем нитям под одним и тем же именем. Объявление локальных переменных вызывает порождение своего экземпляра каждой переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По существу, только что рассмотренные понятия областей программы и классов переменных определяют общую идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляются параллельными областями; именно эти области и только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное — это конкретизация деталей и описание особенностей реализации данной идеи на практике.

Модель Fork-join позволяет явно указывать параллельные секции программы, а также предоставляет поддержку вложенного параллелизма и динамических потоков.

OpenMP является стандартом интерфейса для многопоточного программирования над общей памятью. Он содержит в себе набор средств для языков C/C++ и Fortran: директивы компилятора, библиотечные подпрограммы и переменные окружения.

Порядок создания параллельных программ с использованием OpenMP следующий:

1.Написать и отладить последовательную программу

2.Дополнить программу директивами OpenMP

3.Скомпилировать программу компилятором с поддержкой OpenMP

4.Задать переменные окружения

5.Запустить программу

Технология OpenMP нацелена на то, чтобы пользователь имел один вариант программы для параллельного и последовательного выполнения. Однако возможно создавать программы, которые работают корректно только в параллельном режиме или дают в последовательном режиме другой результат.

Более того, из-за накопления ошибок округления результат вычислений с использованием различного количества потоков может в некоторых случаях различаться.

Модель параллельной программы в OpenMP можно сформулировать следующим образом:

Программа состоит из последовательных и параллельных секций. В начальный момент времени создается главный поток, выполняющий последовательные секции программы. При входе в параллельную секцию выполняется операция fork, порождающая семейство потоков. Каждый поток имеет свой уникальный числовой идентификатор (главному потоку соответствует 0). При распараллеливании циклов все параллельные потоки исполняют один код. В общем случае потоки могут исполнять различные фрагменты кода. При выходе из параллельной секции выполняется операция join. Завершается выполнение всех потоков, кроме главного. В программе может быть любое количество параллельных и последовательных областей. Кроме того, параллельные области могут быть также вложенными друг в друга. В отличие от полноценных процессов (используемых, например, в MPI), порождение потоков является относительно быстрой операцией, поэтому частые порождения и завершения потоков не так сильно влияют на время выполнения программы. MPI, правда, решает эту проблему тем, что все процессы создаются только один раз в начале выполнения программы – однако, этот подход намного менее гибок.
Для написания эффективной параллельной программы необходимо, чтобы все потоки, участвующие в обработке программы, были равномерно загружены полезной работой. Это достигается тщательной балансировкой загрузки, для чего предназначены различные механизмы OpenMP.
Существенным моментом является также необходимость синхронизации доступа к общим данным. Само наличие данных, общих для нескольких потоков, приводит к конфликтам при одновременном несогласованном доступе. Поэтому значительная часть функциональности OpenMP предназначена для осуществления различного рода синхронизаций работающих потоков.
Однако, OpenMP не выполняет синхронизацию доступа различных потоков к одним и тем же файлам. Если это необходимо для корректности программы, пользователь должен явно использовать директивы синхронизации или соответствующие библиотечные функции. При доступе каждого потока к своему файлу никакая синхронизация, очевидно, не требуется.
OpenMP составляют следующие компоненты:
Директивы компилятора - используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются в исходный текст программы.
Подпрограммы 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 (например, командами оболочки в операционных системах UNIX).
Для привязки к языку C в программах все директивы, имена функций и переменных окружения OMP начинаются с omp, omp или OMP. Формат директивы:

#pragma omp директива [оператор_1[, оператор_2, ...]]

Директивы реализуют значительную часть функциональности OpenMP. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в программах на языке С реализуются указаниями препроцессору, начинающимися с #pragma omp. Ключевое слово omp используется для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами.
Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных, разделяемых запятыми. Все директивы OpenMP можно разделить на 3 категории: определение параллельной области, распределение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов – опций (clause). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив.
Все функции, используемые в OpenMP, начинаются с префикса omp_. Если пользователь не будет использовать в программе имён, начинающихся с такого префикса, то конфликтов с объектами OpenMP заведомо не будет. В языке Си, кроме того, является существенным регистр символов в названиях функций. Названия функций OpenMP записываются строчными буквами.
В OpenMP-программе используется заголовочный файл omp.h. Для проверки того, что компилятор поддерживает какую-либо версию OpenMP, достаточно написать директивы условной компиляции #ifdef или #ifndef.

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
 printf("OpenMP is supported!\n");
#endif
}
```

```
#include <math.h>
#define N 10000
float x[N];

int main()
{ int i;
  float k = 2*3.14159265/N;


  for (i=0;i<N;i++) x[i]=sinf(k*i);

  return 0;
}
```

```
#include <math.h>
#define N 10000
float x[N];

int main()
{ int i;
  float k = 2*3.14159265/N;

  #pragma omp parallel for
  for (i=0;i<N;i++) x[i]=sinf(k*i);

  return 0;
}
```

Объявление параллельной секции:

```
#include <omp.h>
int main()
{
  // последовательный код
  #pragma omp parallel
  {
    // параллельный код
  }
  // последовательный код

  return 0;
}
```

Условное объявление параллельной секции: опция if (условие). Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

```
#include <omp.h>
int main()
{
  // последовательный код
  #pragma omp parallel if (expr)
  {
    // параллельный код
  }
  // последовательный код

  return 0;
}
```

Пример (hello, world):

```c
#include <stdio.h>
#include <omp.h>
int main()
{ printf("Hello, World!\n");
  #pragma omp parallel
  { int i,n;
    i = omp_get_thread_num();
    n = omp_get_num_threads();
    printf("I'm thread %d of %d\n",i,n);
  }
  return 0;
}
```

- **Компиляция:**
  `> gcc -fopenmp -o hello hello.c`
- **Запуск:**

```
> OMP_NUM_THREADS=4 ./hello
Hello, World!
I'm thread 1 of 4
I'm thread 0 of 4
I'm thread 3 of 4
I'm thread 2 of 4
```

Задание числа потоков осуществляется тремя способами:

1) Переменная окружения OMP_NUM_THREADS:
   > OMP_NUM_THREADS=4; ./a.out

2) Функция omp_set_num_threads(int):
   omp_set_num_threads(4);
   #pragma omp parallel
   {
   …
   }

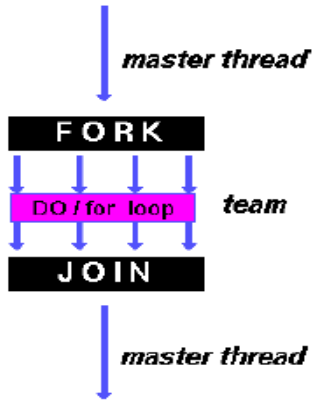
3) Параметр num_threads директивы объявления параллельной секции: явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции omp_set_num_threads(), или значение переменной OMP_NUM_THREADS;
   #pragma omp parallel num_threads(4)
   {
   …
   }

Получение числа потоков осуществляется функцией omp_get_num_threads().

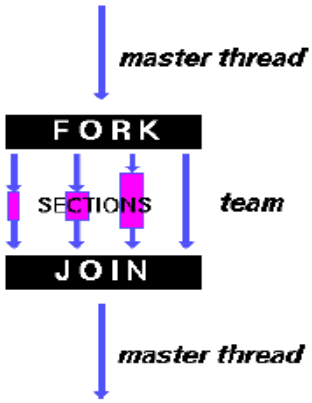
```c
#pragma omp parallel
{ …
  int n = omp_get_num_threads();
  …
}
```

При входе в параллельную область порождаются новые OMP_NUM_THREADS-1 потоков, каждый поток получает свой уникальный номер, причём порождающий поток получает номер 0 и становится основным потоков группы («мастером»). Остальные потоки получают в качестве номера целые числа с 1 до OMP_NUM_THREADS-1. Количество потоков, выполняющих данную параллельную область, остаётся неизменным до момента выхода из области. При выходе из параллельной области производится неявная синхронизация и уничтожаются все потоки, кроме породившего. Все порождённые потоки исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в системе потоки будут распределены по различным процессорам (однако это, как правило, находится под контролем операционной системы).
Директиве объявления параллельной секции можно передавать и другие параметры, например, способ разделения работы между потоками:

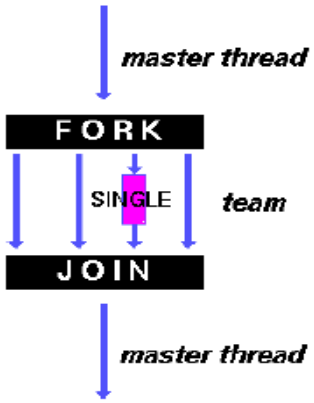
```
#pragma omp for
  for (i=0;i<N;i++)
  {
    // code
  }
```




```
#pragma omp sections
{
 #pragma omp section
 // code 1
 #pragma omp section
 // code 2
}
```




```
#pragma omp single
{
  // code
}
```



Возможные опции для оператора for: private, firstprivate, lastprivate, reduction, schedule, collapse, ordered, nowait. О них позже.
Директиву omp for можно использовать любым из двух способов (сокращённая запись); применим этот способ только в том случае, если внутри параллельной области содержится только один параллельный for-цикл. Такую же запись можно использовать, если в секции есть только одна конструкция sections (тогда она записывается как #pragma omp parallel sections).

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0;i<1000;i++)
      printf("%d ",i);
  }
  return 0;
}
```

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;

  #pragma omp parallel for
    for (i=0;i<1000;i++)
      printf("%d ",i);

  return 0;
}
```

Директива omp sections позволяет распределить работу на явно не связанные секции для задания конечного (неитеративного) параллелизма:

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel sections private(i)
  {
    #pragma omp section
    { printf("1st half\n");
      for (i=0;i<500;i++) printf("%d ",i);
    }
    #pragma omp section
    { printf("2nd half\n");
      for (i=501;i<1000;i++) printf("%d ",i);
    }
  }
  return 0;
}
```

Возможные опции – private, firstprivate, lastprivate, reduction, nowait. Внутри секции sections директива #pragma omp section задаёт участок кода для выполнения одним потоком. Перед первым участком кода в блоке sections директива section не обязательна. Какие именно потоки будут задействованы для выполнения какой секции, не специфицируется. Если количество потоков больше количества секций, то часть потоков для выполнения данного блока секций не будет задействована. Если количество потоков меньше количества секций, то некоторым (или всем) потокам достанется более одной секции.

Директива omp single обозначает среди параллельного кода секцию, которую необходимо выполнить последовательно, на одном потоке и единожды. Так же, как и

для объявления параллельной секции, для неё допустимы параметры private и firstprivate, задающие список локальных для потока переменных.

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp single
      printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

Объявленная таким образом секция также будет осуществлять синхронизацию потоков: она является барьером, поэтому первый цикл в 1000 элементов выполнится до сообщения “I’m thread %d\n”, а второй после, потому что выделенная с помощью single секция производит неявную барьерную синхронизацию потоков. Если не нужно объявлять её как барьер, существует параметр nowait:

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp single nowait
      printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for                          нет барьера
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

Какой именно поток будет выполнять выделенный single участок программы, не специфицируется. Можно объявить single-секцию, исполняемую именно главным потоком, с помощью ключевого слова master. В таком случае она тоже не будет являться барьером: неявной синхронизации данная директива не предполагает, и все остальные потоки пропустят данный участок и продолжат работу с оператора, расположенного следом за ним.

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp master
      printf("I'm Master!\n")
    #pragma omp for                                    нет барьера
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

Директива task применяется для выделения отдельной независимой задачи.
#pragma omp task {options}
Текущий поток выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.
Возможные опции: if, default, private, firstprivate, shared.
Для гарантированного завершения в точке вызова всех запущенных задач используется директива taskwait: #pragma omp taskwait. Поток, выполнивший данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данным потоком независимые задачи.

Для параллельных секций существуют параметры, позволяющие задать способы разделения работы между потоками:

- **schedule** - распределения итераций цикла между потоками
  - **schedule(static, n)** – статическое распределение. В начале цикла все итерации делятся на части по n штук и каждая определяется конкретному потоку.
  - **schedule(dynamic, n)** – динамическое распределение. В начале цикла итерации делятся на части по n штук и каждый поток получает по одной, по окончании которой запрашивает следующую.
  - **schedule(guided, n)** – управляемое распределение. Похоже на предыдущее, но части, которые получает поток после завершения текущих, постепенно уменьшаются в размере от n до 1.
  - **schedule(runtime)** – определяется переменной окружения OMP_SCHEDULE. Помимо задания из командной строки, эту переменную также можно задать с помощью вызова функции
    void omp_set_schedule(omp_sched_t type, int chunk).
    Типы omp_sched_t заданы в omp.h и представляют собой: omp_sched_static, omp_sched_dynamic, omp_sched_guided и omp_sched_auto.
    Также можно получить значение переменной с помощью
    void omp_get_schedule(omp_sched_t* type, int* chunk).

    - **schedule(auto)** – способ распределения итераций выбирается компилятором или системой выполнения автоматически
- **nowait** – отключение синхронизации в конце цикла, или, как уже было упомянуто в single, отключение «барьерности», то есть потоки будут продолжать работу дальше по завершении своей части параллельного кода
- **ordered** – выполнение итераций в последовательном порядке, т.е. так же, как они бы выполнялись при последовательной программе.
- **collapse(n)** – объединить n вложенных циклов в одно итерационное пространство. Эта опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между потоками; если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;

  #pragma omp parallel private(i)
  {
    #pragma omp for schedule(static,10) nowait
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp for schedule(dynamic,1)
      for (i='a';i<='z';i++) printf("%c ",i);
  }
  return 0;
}
```

Все переменные в OpenMP обладают разными областями видимости в зависимости от места, где они были объявлены, а также вызовов функций и директив и прочих параметров: модель памяти предполагает наличие как общей для всех потоков области памяти, так и локальной области для каждого потока. Переменные в параллельных областях программы разделяются на два основных класса: общие shared – все потоки видят одну и ту же переменную; и локальные private – каждый поток видит свой экземпляр переменной.

Общая переменная всегда существует лишь в одном экземпляре для всей области действия и доступна всем потокам под одним и тем же именем. Объявление локальной переменной вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждого потока. Изменение потоком значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других потоках.

Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум один поток читает значение общей переменной и как минимум один поток записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой «гонки данных» (data race), при которой результат выполнения программы непредсказуем. Именно поэтому при распараллеливании циклов программист должен сам убеждаться в том, что итерации данного цикла не имеют информационных зависимостей. Если цикл не содержит зависимостей, его итерации можно выполнять в любом порядке, в том числе

параллельно. Соблюдение этого важного требования компилятор не проверяет, вся ответственность лежит на программисте. Если дать указание компилятору распараллелить цикл, содержащий зависимости, компилятор это сделает, но результат работы программы может оказаться некорректным.
По умолчанию, все переменные, порождённые вне параллельной области, при входе в эту область остаются общими (shared). Исключение составляют переменные, являющиеся счетчиками итераций в цикле, по очевидным причинам. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными (private). Явно назначить класс переменных по умолчанию можно с помощью опции default. Не рекомендуется постоянно полагаться на правила по умолчанию, для большей надёжности лучше всегда явно описывать классы используемых переменных, указывая в директивах OpenMP опции private, shared, firstprivate, lastprivate, reduction.
Переменные, объявленные внутри параллельного блока, являются локальными для потока:

```
#pragma omp parallel
{
  int num = omp_get_thread_num();
  printf("Thread %d\n",num);
}
```

Переменные, объявленные вне параллельного блока, по умолчанию являются общими для всех потоков:

```
int n = 10;
#pragma omp parallel
{
  for (int i=0;i<n;i++)
    printf("%d\n",i);
}
```

Это может привести к возможным конфликтам и состояниям гонки, когда несколько потоков осуществляют запись и чтение одной и той же общей переменной:

```
int num;
#pragma omp parallel
{
  num = omp_get_thread_num();
  printf("Thread %d\n",num);
}
```

Для разрешения подобных конфликтов можно использовать параметры директив определения параллельных секций, которые обозначают области видимости всех объявленных вне параллельного блока переменных:

- private: каждая переменная для своего потока является локальной. Задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено

```
int num;
#pragma omp parallel private(num)
{
  num = omp_get_thread_num();
  printf("%d\n",num);
}
```

- firstprivate: задаёт список переменных, для которых порождается локальная копия в каждом потоке; локальные копии переменных инициализируются значениями этих переменных в потоке-мастере;

```
int num = 5;
#pragma omp parallel \
            firstprivate(num)
{
  printf("%d\n",num);
}
```

- lastprivate: переменная является локальной, но после завершения параллельной секции её последнее значение сохраняется для использования в следующей последовательной секции

```
int i,j;
#pragma omp parallel for \
            lastprivate(j)
for (i=0;i<100;i++) j = i;

printf("Последний j = %d\n",j);
```

- shared: переменная является общей, разделяемой между всеми потоками

```
int i,j;

#pragma omp parallel for \
            shared(j)
for (i=0;i<100;i++) j = i;

printf("j = %d\n",j);
```

- default(shared|none): всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс shared; none означает, что всем переменным в параллельной области класс должен быть назначен явно

```
int i,k,n = 2;
#pragma omp parallel shared(n)
  default(none) private(i,k)
{
  i = omp_get_thread_num() / n;
  k = omp_get_thread_num() % n;
  printf("%d %d %d\n",i,k,n);
}
```

- reduction: используется для того, чтобы обозначить переменную как хранящую результат некой редукционной операции. Принимает в качестве аргумента, помимо имени переменной, символ самой операции. (+, *, -, &, ^, |, &&, или ||)
  В данном примере все потоки инициализируют переменную s нулём и отдельно добавляют в неё свои значения, а на выходе все значения из каждого потока суммируются.

```
int i,s = 0;
#pragma omp parallel for \
            reduction(+:s)
  for (i=0;i<100;i++)
    s += i;

printf("Sum: %d\n",s);
```

- threadprivate: объявляет глобальную переменную локальной для потоков. Отличается от private тем, что private делает переменную более не видимой после выхода из локального блока; более того, в случае private все потоки будут пользоваться копией переменной, в то время как в threadprivate основной поток будет продолжать пользоваться её оригиналом, а копии будут создаваться лишь на второстепенных

```
int x;
#pragma omp threadprivate(x)
int main()
{ . . .
  #pragma omp parallel
  . . .
}
```

- copyin: то же, что и предыдущее, но с инициализацией начального значения, как в firstprivate.

```
int x;
#pragma omp threadprivate(x)
int main()
{ . . .
  #pragma omp parallel copyin(x)
  . . .
}
```

Помимо неявных объявлений барьеров с помощью single-секций, в OpenMP присутствуют и другие директивы, предназначенные для синхронизации потоков.

- master: выполнение кода только основным потоком

```
#pragma omp parallel
{
  //code
  #pragma omp master
  {
    // critical code
  }
  // code
}
```

- barrier: аналогичен MPI_Barrier в MPI. Потоки, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все потоки не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (task).

```
#pragma omp parallel
{
  printf("Hello!\n");

  #pragma omp barrier
  printf("I am thread %d\n",
         omp_get_thread_num());
}
```

- critical: обозначает критическую секцию. В каждый момент времени в критической секции может находиться не более одного потока. Если критическая секция уже выполняется каким-либо потоком, то все другие потоки, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедший поток не закончит выполнение данной критической секции. Как только работавший поток выйдет из критической секции, один из заблокированных на входе потоков войдет в неё. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные потоки продолжают ожидание.
- Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются

единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены. В каком-то смысле объявление критической секции аналогично захвату мьютекса.
Если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь каким-либо одним потоком. Остальные потоки, даже если они уже подошли к данной точке программы и готовы к работе, будут ожидать своей очереди. Если критической секции нет, то все потоки могут одновременно выполнить данный участок кода. С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность.

```
int i,idx[N],x[M];

#pragma omp parallel for
for (i=0;i<N;i++)
{
  #pragma omp critical
  {
    x[idx[i]] += count(i);
  }
}
```

```
int i,idx[N],x[M];

#pragma omp parallel for
for (i=0;i<N;i++)
{ int j = idx[i];
  int c = count(i);
  #pragma omp critical
  {
    x[j] += c;
  }
}
```

Так быстрее!

- atomic: обозначает секцию, которая должна выполняться атомарно, без возможностей вмешательства со стороны других потоков.
Они являются частым случаем использования критических секций на практике – обновлением общих переменных. Например, если переменная sum является общей и оператор вида sum=sum+expr находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими потоками можно получить некорректный результат. Чтобы избежать такой ситуации, можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой atomic. Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме потока, выполняющего операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

```
int i,idx[N],x[M];

#pragma omp parallel for
for (i=0;i<N;i++)
{
  #pragma omp atomic
    x[idx[i]] += count(i);
}
```

- flush: директива, которая используется для согласования значений одной и той же переменной между разными потоками. Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени все потоки будут видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива flush: её выполнение предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущего потока, будут занесены в основную память; все изменения переменных, сделанные потоком во время работы, станут видимы остальным потокам; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшего потока; данные, изменявшиеся другими потоками, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться.
  Неявно flush без параметров присутствует в директиве barrier, на входе и выходе областей действия директив parallel, critical, ordered, на выходе областей распределения работ, если не используется опция nowait, в вызовах функций omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), если при этом замок устанавливается или снимается, а также перед порождением и после завершения любой задачи (task). Кроме того, flush вызывается для переменной, участвующей в операции, ассоциированной с директивой atomic. Заметим, что flush не применяется на входе области распределения работ, а также на входе и выходе области действия директивы master.

```
int x = 0;
#pragma omp parallel sections
{
  #pragma omp section
  { x = 1;
    #pragma omp flush(x)
  }
  #pragma omp section
    while (!x);
}
```

- ordered: используется для выделения в параллельном цикле некого одного упорядоченного блока, который будет идти по итерациям так же, как последовательный цикл. Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция ordered. Поток, выполняющий первую итерацию цикла, выполняет операции данного блока. Поток, выполняющий любую следующую итерацию, должен сначала дождаться выполнения всех операций блока всеми потоками, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных потоков.

```
int i,j,k;
#pragma omp parallel for ordered
  for (i=0;i<N;i++)
  { printf("No order: %d\n",i);
    #pragma omp ordered
    printf("Order: %d\n",i);
  }
```

Один из вариантов синхронизации в OpenMP реализуется через механизм замков (locks). В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.
Замок может находиться в одном из трёх состояний: неинициализированный, разблокированный или заблокированный. Разблокированный замок может быть захвачен некоторым потоком. При этом он переходит в заблокированное состояние. Поток, захвативший замок, и только он может его освободить, после чего замок возвращается в разблокированное состояние.
Есть два типа замков: простые замки и множественные замки. Множественный замок может многократно захватываться одним потоком перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие коэффициента захваченности (nesting count). Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int x[1000];
int main()
{
     int i,max;
     omp_lock_t lock;
     omp_init_lock(&lock);
     for (i=0;i<1000;i++) x[i]=rand();
     max = x[0];
     #pragma omp parallel for
     for(i=0;i<1000;i++)
     {
          omp_set_lock(&lock);
          if (x[i]>max) max = x[i];
```

```
            omp_unset_lock(&lock);
        }
        omp_destroy_lock(&lock);
        return 0;
}
```

Блокировки бывают однократными (невложенными) и многократными (вложенными). Функции для работы с блокировками:

- omp_lock_t – однократная блокировка
  - `void omp_init_lock(omp_lock_t *lock)`
  - `void omp_destroy_lock(omp_lock_t *lock)`
  - `void omp_set_lock(omp_lock_t *lock)`
  - `void omp_unset_lock(omp_lock_t *lock)`
  - `int omp_test_lock(omp_lock_t *lock)`
- omp_nest_lock_t – многократная (вложенная) блокировка
  - `void omp_init_nest_lock(omp_nest_lock_t *lock)`
  - `void omp_destroy_nest_lock(omp_nest_lock_t *lock)`
  - `void omp_set_nest_lock(omp_nest_lock_t *lock)`
  - `void omp_unset_nest_lock(omp_nest_lock_t *lock)`
  - `int omp_test_nest_lock(omp_nest_lock_t *lock)`

Для однократной блокировки: инициализация (переводит замок из неинициализированного в разблокированный), деструкция (перевод в неинициализированное состояние), захват, освобождение. test_lock пытается сделать захват, но не блокирует поток, а просто продолжает его дальше, если захват не удался (блокировка захвачена другим потоком). Возвращает 0, если блокировка не удалась, и отличное от 0 число в противном случае.
Многократная блокировка хранит в себе счётчик: изначально захвативший её поток получает владение над ней, а также устанавливает её счётчик в единицу. Этот же поток может снова вызвать set_nest_lock, увеличивая счётчик на 1 каждый раз, когда это происходит. Для освобождения блокировки удерживающему её потоку необходимо вызвать unset_nest_lock столько же раз, сколько была вызвана функция set_nest_lock, до обнуления счётчика. test_nest_lock работает так же, как и test_lock, но возвращаемое значение – 0 при неудачной блокировке и новое значение счётчика при удачной.
Использование замков является наиболее гибким механизмом синхронизации, поскольку с помощью замков можно реализовать все остальные варианты синхронизации.
Прочие функции для работы с OpenMP:

- void omp_set_num_thread(int num_threads) – задать количество потоков для последующих параллельных секций
- int omp_get_num_threads() – получить количество потоков. Этим двум функциям также соответствует задание переменной окружения OMP_NUM_THREADS
- int omp_get_max_threads() – получить максимально возможное количество потоков для использования в следующей параллельной секции на данном устройстве
- int omp_get_thread_num() – получить идентификатор текущего потока
- int omp_get_num_procs() – получить максимально возможное количество процессоров, которые можно назначить на текущую задачу в данный момент

времени. Нужно учитывать, что количество доступных процессоров может динамически изменяться

- int omp_in_parallel() – возвращает 0, если в настоящий момент код исполняется последовательно (функция вызвана не из параллельной части программы), и ненулевое значение, если параллельно (вызвана из параллельной части).
- void omp_set_dynamic(int dynamic_threads) – позволяет включать или отключать режим динамического изменения количества потоков. При передаче функции нулевого значения в качестве аргумента режим отключается, при ненулевом – включается; тогда в каждой последующей параллельной секции количество потоков может меняться для лучшего использования ресурсов компьютера
- int omp_get_dynamic() – возвращает ненулевое значение, если динамическое изменение количества потоков включено, и нулевое в противном случае. Этим двум функциям также соответствует задание переменной окружения OMP_DYNAMIC
- void omp_set_nested(int nested) – позволяет включать или отключать поддержку вложенного параллелизма. При передаче функции нулевого значения в качестве аргумента поддержка отключается, при ненулевом – включается; тогда для обработки вложенных параллельных участков кода могут задействоваться дополнительные потоки
- int omp_get_nested() – возвращает ненулевое значение, если поддержка вложенного параллелизма включена, и нулевое в противном случае. Этим двум функциям также соответствует задание переменной окружения OMP_NESTED
- double omp_get_wtime() – возвращает текущее значение встроенного таймера, считающего время с некоторого момента в прошлом, в секундах для текущего вызвавшего эту функцию потока. Разница между двумя результатами вызова такой функции, расположенных в разных местах, показывает время работы участка, расположенного между ними. Гарантируется, что момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса, однако, таймеры разных потоков могут быть не синхронизированы и выдавать разные значения
- double omp_get_wtick() – возвращает точность таймера wtime в секундах (время, проходящее между двумя его тактами).

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
double start_time, end_time, tick;
start_time = omp_get_wtime();
end_time = omp_get_wtime();
tick = omp_get_wtick();
printf("Время на замер времени %lf\n", end_time-start_time);
printf("Точность таймера %lf\n", tick);
}
```

Если целевая вычислительная платформа является многопроцессорной и/или многоядерной, то для повышения быстродействия программы нужно задействовать все доступные пользователю вычислительные ядра. Чаще всего разумно порождать по одному потоку на вычислительное ядро, хотя это не является обязательным

требованием. Например, для первоначальной отладки может быть вполне достаточно одноядерного процессора, на котором порождается несколько потоков, работающих в режиме разделения времени. Порождение и уничтожение потоков в OpenMP являются относительно недорогими операциями, однако надо помнить, что многократное совершение этих действий (например, в цикле) может повлечь существенное увеличение времени работы программы.

Для того чтобы получить параллельную версию, сначала необходимо определить ресурс параллелизма программы, то есть, найти в ней участки, которые могут выполняться независимо разными потоками. Если таких участков относительно немного, то для распараллеливания чаще всего используются конструкции, задающие конечный (неитеративный) параллелизм, например, параллельные секции или низкоуровневое распараллеливание по номеру потока.

Однако, как показывает практика, наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно каким-либо способом раздать разным процессорам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки потоков, между которыми распределяются итерации цикла.

Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какому потоку достанутся какие итерации. Однако он не учитывает текущей загруженности процессоров, соотношения времён выполнения различных итераций и некоторых других факторов. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций. Кроме того, возможно отложить решение по способу распределения итераций на время выполнения программы (например, выбирать его, исходя из текущей загруженности потоков) или возложить выбор распределения на компилятор и/или систему выполнения.

Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных потоков к общим данным. Для этого предусмотрены достаточно развитые средства синхронизации. При этом нужно учитывать, что использование излишних синхронизаций может существенно замедлить программу.

Использование идеи инкрементального распараллеливания позволяет при помощи OpenMP быстро получить параллельный вариант программы. Взяв за основу последовательный код, пользователь шаг за шагом добавляет новые директивы, описывающие новые параллельные области. Нет необходимости сразу распараллеливать всю программу, её создание ведется последовательно, что упрощает и процесс программирования, и отладку.

Программа, созданная с использованием технологии OpenMP, может быть использована и в качестве последовательной программы. Таким образом, нет необходимости поддерживать разные версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова функций OpenMP могут быть подставлены специальные «заглушки» (stubs), текст которых приведен в описании стандарта. Они гарантируют корректную работу программы в последовательном случае – нужно только перекомпилировать программу и подключить другую библиотеку.

Одним из достоинств OpenMP его разработчики считают поддержку так называемых оторванных (orphaned) директив. Это предполагает, что директивы синхронизации и

распределения работы могут не входить непосредственно в лексический контекст параллельной области. Например, можно вставлять директивы в вызываемую подпрограмму, предполагая, что её вызов произойдёт из параллельной области.
OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле.