Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

Лекция 8. Технологии OpenMP.

Эффективное распределение нагрузки. Синхронизация.

The OpenMP® API specification for parallel programming http://openmp.org

Антонов А.С. Параллельное программирование с использованием технологии OpenMP. М.: Издво МГУ, 2009.

Примеры из учебника «Технологии параллельного программирования MPI и OpenMP» http://parallel.ru/tech/tech_dev/MPI%26OpenMP/examples

Примеры http://pro-prof.com/archives/1150

Демонстрация примеров http://coliru.stacked-crooked.com/

Эффективность parallel

В конце параллельного раздела потоки приостанавливаются и ждут перехода в следующий параллельный раздел. Процессы создания и приостановки потоков связаны с системными издержками

```
#pragma omp parallel for
for (i=0; i<x; i++)
fn1();
#pragma omp parallel for
// adds unnecessary overhead
for (i=0; i<y; i++)
fn2();
```

Код значительно быстрее своего аналога за счет снижения системных издержек, поскольку вход в параллельный раздел осуществляется только один раз.

```
#pragma omp parallel
{
#pragma omp for
for (i=0; i<x; i++)
fn1();
#pragma omp for
for (i=0; i<y; i++)
fn2();
}
```

Разделы программного кода, влияющие на производительность всего приложения, должны выполняться внутри параллельного раздела.

Эффективность и sections

```
#pragma omp parallel
{
#pragma omp for
for (i=0; i<x;
i++)
Fn1();
#pragma omp sections
{
#pragma omp section
{
TaskA();
}
#pragma omp section
{
TaskB();
}
#pragma omp section
{
TaskC();
}
}}
```

Организовано распределение циклов и нециклических разделов кода между потоками.

Создана группа потоков, между ними распределена обработка итераций цикла.

Потоки выполняют параллельную обработку оставшихся разделов кода. Если количество разделов программного кода будет больше числа потоков, обработка нескольких разделов будет отложена до тех пор, пока не появятся свободные потоки.

В отличие от планирования циклов, распределение нагрузки между потоками при обработке параллельных разделов кода осуществляется и контролируется OpenMP.

Программисту нужно выбрать, какие переменные будут общими, а какие – локальными.

Как эффективнее

Эффективность parallel – один вход в параллельную область.

Выбор for или sections.

Неявные барьеры (for) или nowait (например для вложенных циклов).

Сегмент параллельного раздела выполняется единственным потоком (master или single).

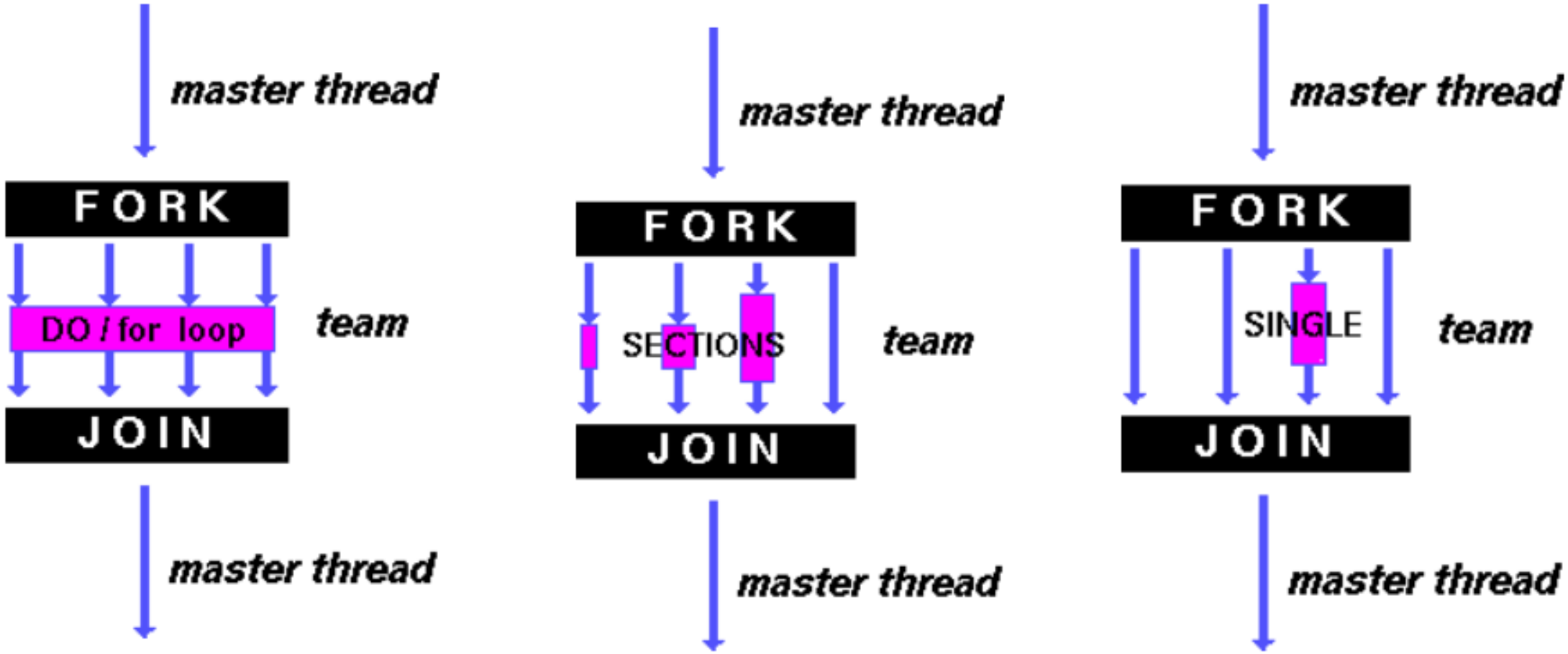
Атомарные.

Критические секции.

Операторы firstprivate и lastprivate

Используя богатые возможности OpenMP, вы сможете существенно упростить процесс организации поточной обработки в своих приложениях.

Конструкции распределения работ

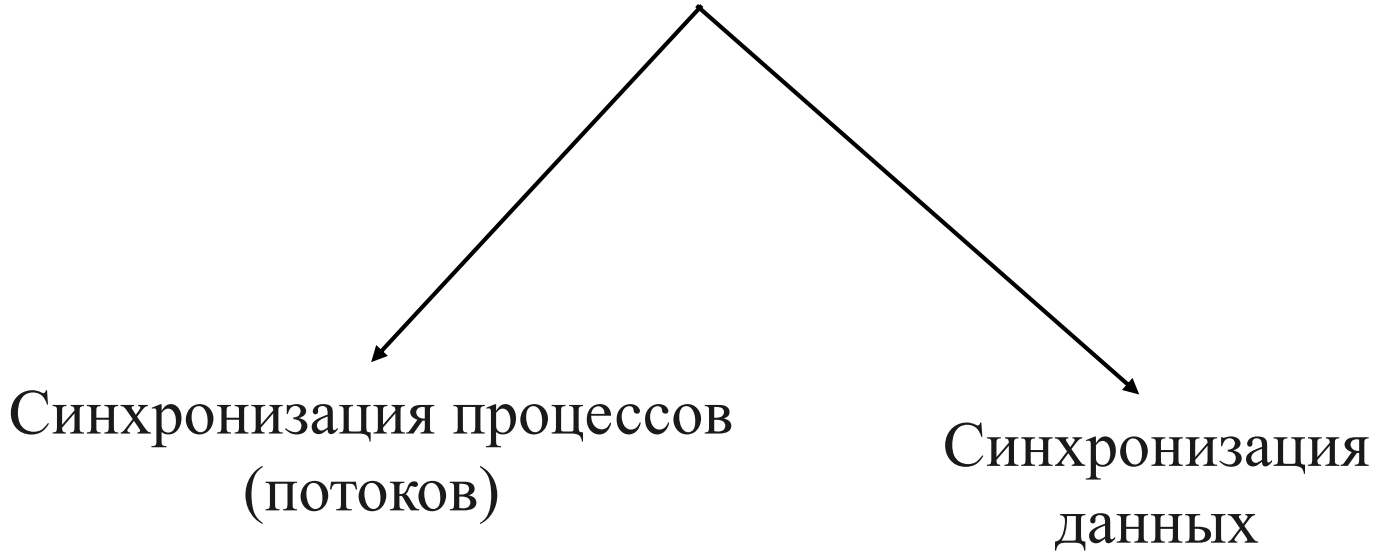


Средства синхронизации в OpenMP

Синхронизация (от греч. συνχρόνος – одновременный) – процесс приведения к одному значению одного или нескольких параметров разных объектов.

Набор директив в OpenMP, предназначенных для синхронизации работы нитей:

- Барьер
- Директива ordered
- Критические секции
- Директива atomic
- Замки
- Директива flush



Барьер

Самый распространенный способ синхронизации – барьер, директива

#pragma omp barrier

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (task).

В примере директива barrier использована для упорядочивания вывода от работающих нитей. Выдача "Сообщение 3" со всех нитей придёт строго после двух предыдущих выдач.

```
int main(int argc, char *argv[])
{
#pragma omp parallel
 {
 cout << "Сообщение 1" << endl;
 cout << "Сообщение 2" << endl;
#pragma omp barrier
 cout << "Сообщение 3" << endl;
 }
}
```

```
Сообщение 1Сообщение 1Сообщение 1

Сообщение 2
Сообщение 2

Сообщение 2
Сообщение 1
Сообщение 2
Сообщение 3
Сообщение 3
Сообщение 3
Сообщение 3
```

Использование барьеров

Для кода:

```
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, может быть неправильный ответ.

Нужна синхронизация по доступу к a[i].

Когда использовать барьеры?

Когда изменение данных происходит асинхронно и целостность данных под вопросом.

Барьеры могут привести к падению производительности и масштабируемости программы. Их использовать надо с осторожностью.

Директивы ordered

Директивы ordered определяют блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

#pragma omp ordered

```
int main(int argc, char *argv[])
{
int i, n;
#pragma omp parallel private (i, n)
    {
        n=omp_get_thread_num();
        #pragma omp for ordered
            for (i=0; i<6; i++)
            {
            cout << "Нить " << n << " итерация " << i << endl;
            //printf("Нить %d, итерация %d\n", n, i);
         #pragma omp ordered
                {
                cout << "ordered: Нить " << n << " итерация " << i << endl;
                }
            }
    }
}
```

Пример – применение директивы ordered и опции ordered.
Цикл for помечен как ordered.
Внутри тела цикла идут две выдачи – одна вне блока ordered, а вторая – внутри него.

Пример ordered

Блок операторов относится к самому внутреннему из объемлющих циклов, в параллельном цикле должна быть задана опция ordered. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации.

Может использоваться, например, для упорядочения вывода от параллельных нитей.

В результате первая выдача получается неупорядоченной, а вторая идёт в строгом порядке по возрастанию номера итерации.

```
Нить Нить Нить 1 итерация 20Нить  итерация 20 итерация 43 итерация 5

ordered: Нить 0 итерация 0
Нить 0 итерация 1
ordered: Нить 0 итерация 1

ordered: Нить 1 итерация 2
Нить 1 итерация 3
ordered: Нить 1 итерация 3

ordered: Нить 2 итерация 4
ordered: Нить 3 итерация 5
```

Критические секции

С помощью директив critical оформляется критическая секция программы.

#pragma omp critical [()]

В каждый момент времени в критической секции может находиться не более одной нити.

Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции.

Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити 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.

Директива critical

Критическая секция – область программы, которая выполняется одновременно только одним потоком.

Все потоки проходят критическую секцию по очереди.

Полезны для избавления от ошибок соревнования, чтения записи данных (неопределенный порядок).

Может привести к тому, что параллельная программа станет последовательной.

```
#pragma omp parallel
{
        //.. параллельное выполнение
#pragma omp critical
        {
                // критическая секция
        }
}
```

Пример critical

Переменная n объявлена вне параллельной области, поэтому по умолчанию является общей.

Критическая секция позволяет разграничить доступ к переменной n.

Каждая нить по очереди присвоит n свой номер и затем напечатает полученное значение.

```
int main(int argc, char *argv[])
{
    int n;
#pragma omp parallel num_threads(5)
    {

#pragma omp critical
        {
            n = omp_get_thread_num();
            cout << "Нить " << n << endl;
        }
    }
}
```

Если не указана директива critical, результат непредсказуем.

С директивой critical порядок вывода результатов произвольный, но это всегда набор одних и тех же чисел от 0 до OMP_NUM_THREADS-1.

Результат подобен, если n объявлена локальной, и каждая нить работает бы со своей копией этой переменной. Однако если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь одной нитью. Если критической секции нет, то все нити могут одновременно выполнить данный участок кода.

Директива atomic

Частым случаем использования критических секций является обновление общих переменных.

Если переменная sum является общей и оператор **sum = sum + expr** находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Избежать эту ситуацию позволяет механизм критических секций или специальная директива atomic.

#pragma omp atomic

Данная директива относится к идущему непосредственно за ней оператору присваивания, 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 нити, выполняющей операцию.

Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

Пример atomic.

Производится подсчет общего количества порожденных нитей.

Каждая нить увеличивает на единицу значение переменной count.

Для того, чтобы предотвратить одновременное изменение несколькими нитями значения переменной, стоящей в левой части оператора присваивания, используется директива atomic.

```
 6  int main(int argc, char *argv[])
 7  {
 8      int count = 0;
 9  #pragma omp parallel num_threads(5)
10      {
11      //#pragma omp atomic
12          count++;
13          cout << "Число нитей: " << count << endl;
14      }
15  }
16
```

```
Число нитей: Число нитей: 33Число нитей:

3
Число нитей: 5
Число нитей: 5
```

Атомарные операции

Атомарные операции, например, чтения-модификации-записи, загрузки (load) и сохранения (store).

Операция в общей области памяти называется **атомарной**, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти. Во время выполнения такой операции над переменной, ни один поток не может наблюдать изменение наполовину завершенным. Атомарная загрузка гарантирует, что переменная будет загружена целиком в один момент времени. Неатомарные операции не дают такой гарантии. Без подобных гарантии неблокирующее программирование было бы невозможно, поскольку было бы нельзя разрешить нескольким потокам оперировать одновременно одной переменной.

Общее правило: *В любой момент времени когда два потока одновременно оперируют общей переменной, и один из них производит запись, оба потока* ***обязаны*** *использовать атомарные операции.*
Если нарушено это правило, и каждый поток использует неатомарные операции, возникает ситуация – **состояние гонок по данным** (data race), и неопределенное поведение. Причина – операции чтения и записи разорваны (torn read/write).

Операция с памятью может быть неатомарной даже на одноядерном процессоре, т.к. используется несколько инструкций процессора. Переносимый код не использует **предположение** об атомарности отдельной инструкции.

Замки

Замки – один из вариантов синхронизации в OpenMP (locks).

В качестве замков используются общие целочисленные переменные (размер достаточен для хранения адреса). Переменные должны использоваться только как параметры примитивов синхронизации.

Замок находится в одном из 3-х состояний:

неинициализированный, разблокированный или заблокированный.

Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Только нить, захватившая замок, может его освободить, после чего замок – разблокирован.

Два типа замков: **простые замки и множественные замки**.

Множественный замок может многократно захватываться одной нитью перед его освобождением, простой замок может быть захвачен только однажды.

Для множественного замка вводится понятие коэффициента захваченности (nesting count).

Изначально nesting count = 0, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу.

Если nesting count = 0 – множественный замок считается разблокированным.

Для инициализации простого или множественного замка используются соответственно функции omp_init_lock() и omp_init_nest_lock().

void omp_init_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *lock);

Состояние замка

После выполнения функции замок переводится в разблокированное состояние.
Функции omp_destroy_lock() и omp_destroy_nest_lock() используются для переведения простого или множественного замка в неинициализированное состояние.

void omp_destroy_lock(omp_lock_t *lock);

void omp_destroy_nest_lock(omp_nest_lock_t *lock);

Для захватывания замка используются функции omp_set_lock() и omp_set_nest_lock().

void omp_set_lock(omp_lock_t *lock);

void omp_set_nest_lock(omp_nest_lock_t *lock);

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние.
Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу. Для освобождения замка используются функции omp_unset_lock() и omp_unset_nest_lock(). **void omp_unset_lock(omp_lock_t *lock);**

void omp_unset_nest_lock(omp_lock_t *lock);

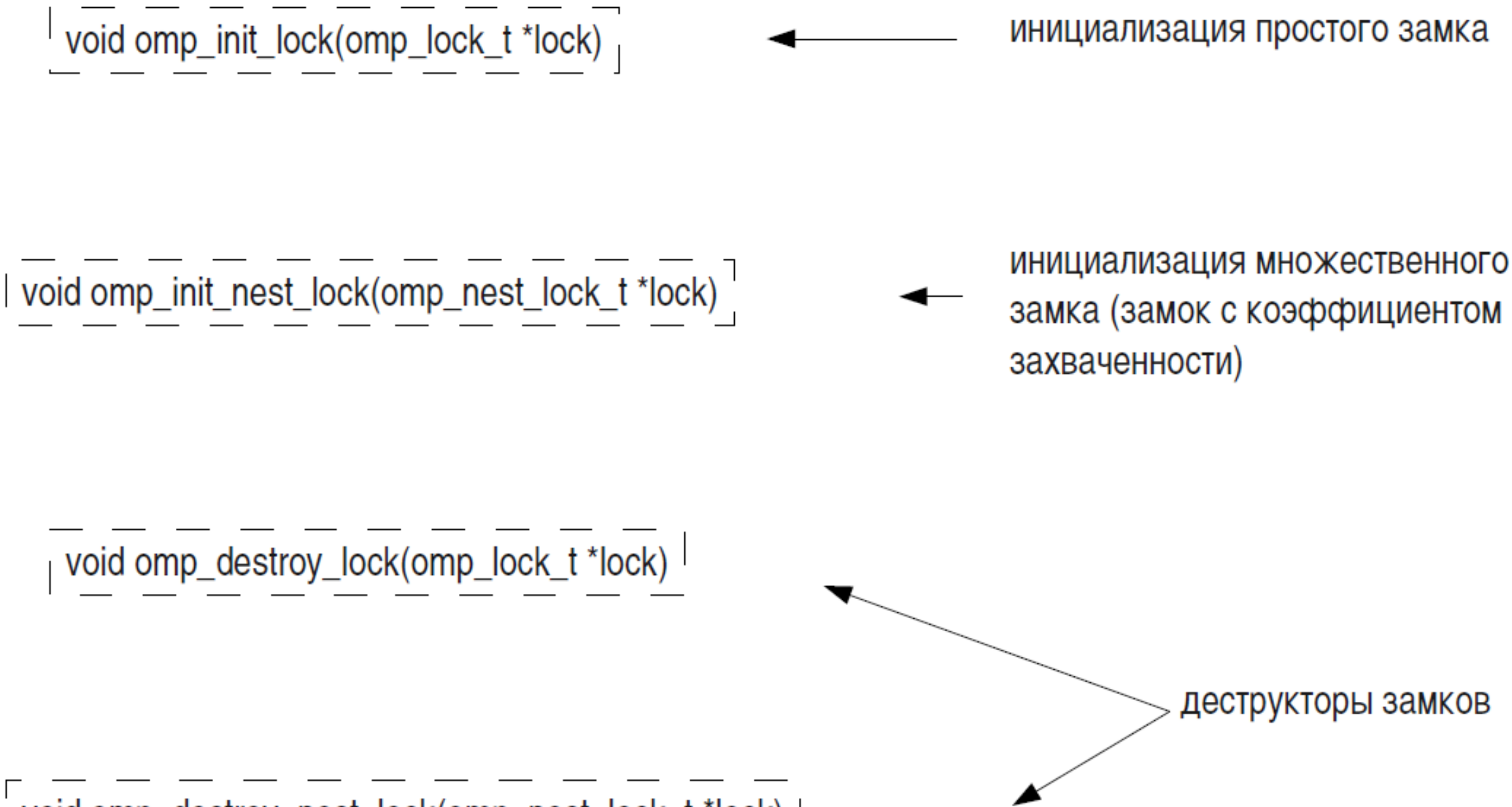
Вызов функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка – уменьшает на единицу коэффициент захваченности. Если коэффициент равен нулю, замок освобождается. Если после освобождения замка есть нити, заблокированные на операции, захватывающей данный замок, замок будет сразу же захвачен одной из ожидающих нитей.

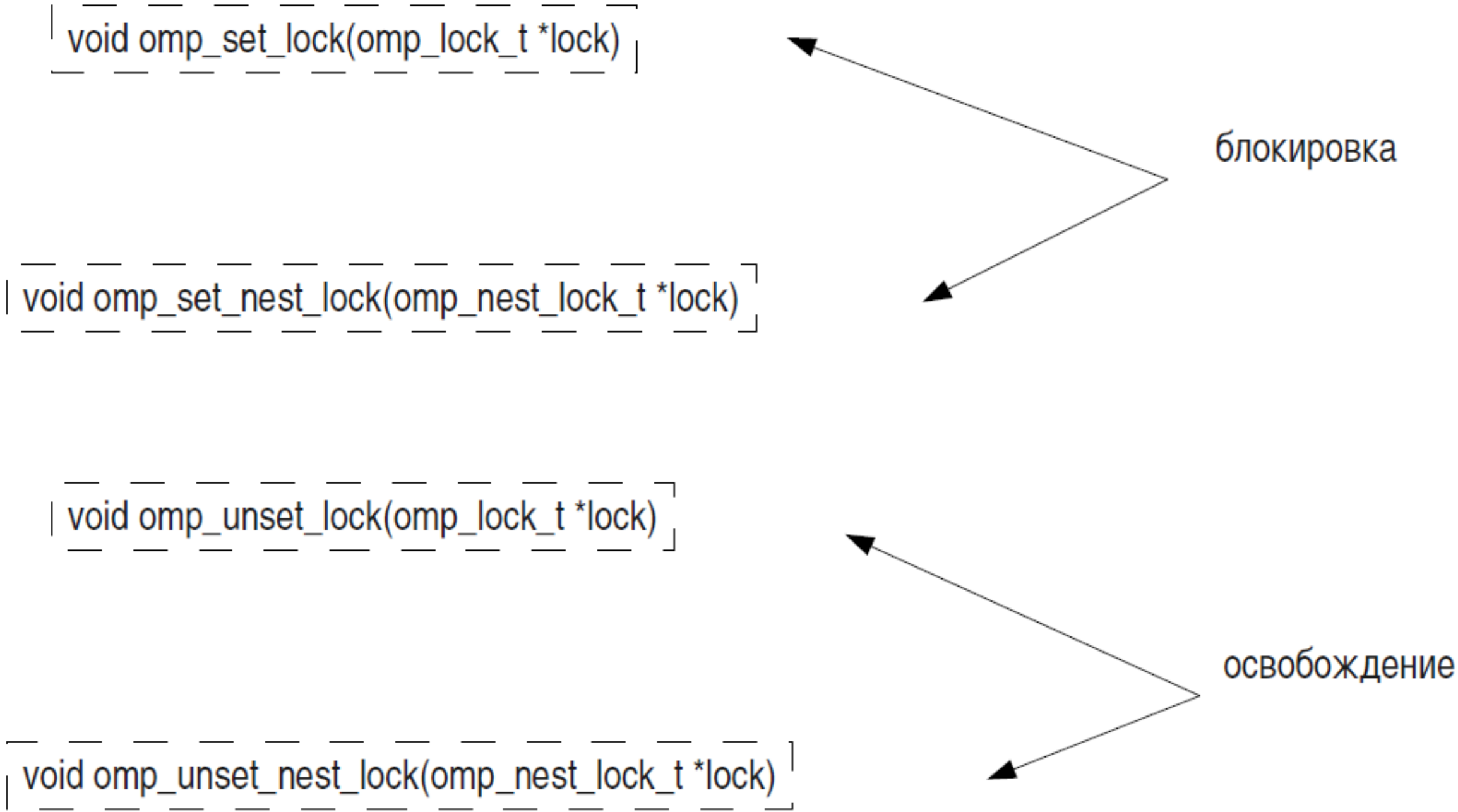
Пример lock

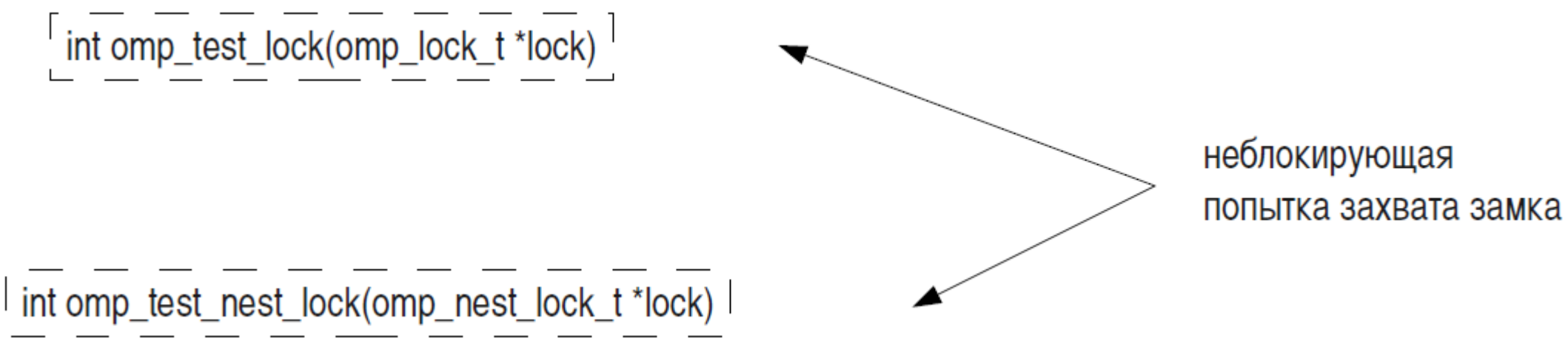
Переменная lock использована для блокировки. В последовательной области – инициализация данной переменной с помощью omp_init_lock().
В начале параллельной области каждая нить присваивает переменной n порядковый номер.
С помощью функции omp_set_lock() одна из нитей выставляет блокировку, а остальные нити ждут, пока нить, вызвавшая эту функцию, не снимет блокировку с помощью функции omp_unset_lock().
Вывод сообщений "Начало закрытой секции..." и "Конец закрытой секции...« разделено по потокам вывода.
С помощью функции omp_destroy_lock() происходит освобождение переменной lock.

```
 6  omp_lock_t lock;
 7  int main(int argc, char *argv[])
 8  {
 9   int n;
10   omp_init_lock(&lock);
11  #pragma omp parallel private (n)
12   {
13      n=omp_get_thread_num();
14      omp_set_lock(&lock);
15      cout << "  Начало закрытой секции, нить " << n << endl;
16      sleep(2);
17      cout << "Конец закрытой секции, нить " << n << endl;
18      omp_unset_lock(&lock);
19   }
20   omp_destroy_lock(&lock);
21  }
```

```
  Начало закрытой секции, нить 0
Конец закрытой секции, нить 0
  Начало закрытой секции, нить 1
Конец закрытой секции, нить 1
  Начало закрытой секции, нить 3
Конец закрытой секции, нить 3
  Начало закрытой секции, нить 2
Конец закрытой секции, нить 2
```


void omp_init_lock(omp_lock_t *lock)
инициализация простого замка
void omp_init_nest_lock(omp_nest_lock_t *lock)
инициализация множественного замка (замок с коэффициентом захваченности)
void omp_destroy_lock(omp_lock_t *lock)
деструкторы замков
void omp_destroy_nest_lock(omp_nest_lock_t *lock)


void omp_set_lock(omp_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
блокировка
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
освобождение



Для неблокирующей попытки захвата замка используются функции omp_test_lock() и omp_test_nest_lock().

int omp_test_lock(omp_lock_t *lock);

int omp_test_nest_lock(omp_lock_t *lock);

Функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. В противном – возвращается 0.

В заключении

- Использование замков является наиболее гибким механизмом синхронизации, поскольку с помощью замков можно реализовать все остальные варианты синхронизации.
- Условие nowait используется для минимизации операций синхронизации.

Последняя директива flush

Директива flush обеспечивает видимость единого согласованного образа памяти для всех нитей в необходимые пользователю моменты времени, то есть позволяет синхронизировать состояние памяти (явно записать значение переменной в общую память).

#pragma omp flush [(список)]

Выполнение данной директивы предполагает следующее:

- значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память;
- все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям;
- если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п.

Операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются.

Выполнение данной директивы связано со значительными накладными расходами, и если нужна гарантия согласованного представления не всех переменных, их можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться.

Как эффективнее

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,
- перед порождением и после завершения любой задачи (task),
- вызывается для переменной, участвующей в операции, ассоциированной с директивой atomic.

flush не применяется на входе области распределения работ, а также на входе и выходе области действия директивы master.