Computer Science МФТИ

OpenMP API components

Субботина Анна
МФТИ

Содержание лекции

- Директивы
- Библиотечные функции/процедуры
- Переменные окружения

Формат директив

- C: директивы чувствительны к регистру
  - Синтаксис: **#pragma omp directive [clause ...]**

- Fortran: директивы не чувствительны к регистру
  - Синтаксис: **sentinel directive [clause ...]**
  - sentinel directive – одна из:
    - !$OMP или C$OMP или *$OMP

Контексты директив

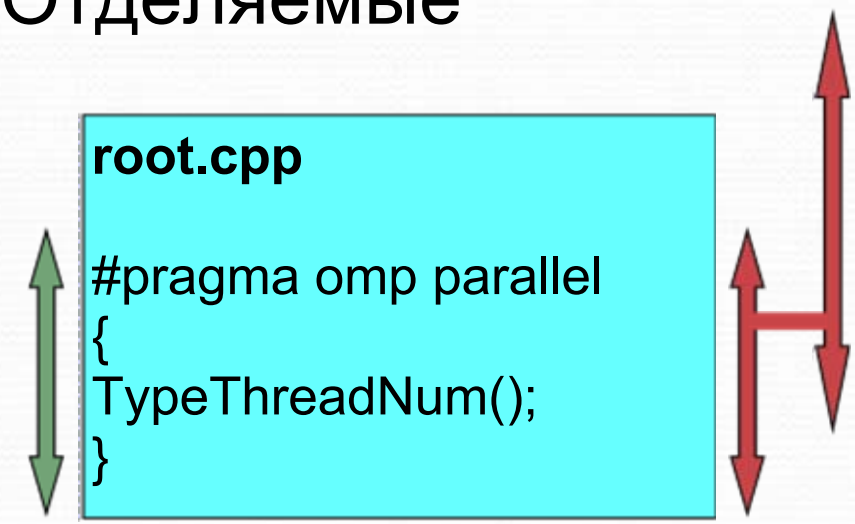
- Статические
- Динамические
- Отделяемые

root.cpp

```
#pragma omp parallel
{
TypeThreadNum();
}
```

node.cpp

```
void TypeThreadNum()
{
int num;
num = omp_get_thread_num();
#pragma omp critical
printf("Hello from %d\n",num);
}
```



Статический (лексический) контекст параллельной области

Динамический контекст параллельной области (включает статический контекст)

Отделяемые (orphaned) директивы могут появляться вне параллельной области

Типы директив

- Определение параллельной области
- Разделение работы
- Синхронизация
- Раздача заданий (tasking)
- Управление разделаемыми данными

Определение параллельной области

Параллельная область - это блок кода, исполняемый всеми потоками одновременно

```
#pragma omp parallel [clause[[,] clause] ...]
{
        "этот код выполняется параллельно"
} (подразумевается барьер)
```

```
!$omp parallel [clause[[,] clause] ...]
        "этот код выполняется параллельно"
!$omp end parallel (подразумевается барьер)
```

Разделение работы

```
#pragma omp for
{
    ........
}
!$OMP DO
....
!$OMP END DO
```

```
#pragma omp sections
{
    ........
}
!$OMP SECTIONS
........
!$OMP END SECTIONS
```

```
#pragma omp single
{
    ........
}
!$OMP SINGLE
........
!$OMP END SINGLE
```

- Работа распределяется между потоками
- Задается только в параллельной области
- Учитывается или всеми потоками в команде (team), или ниодним
- На входе синхронизация потоков (барьер) не подразумевается, только на выходе (если не указан параметр nowait)
- Директивы не создают никаких новых потоков

Директива "for" (пример)

```
#pragma omp parallel default(none)\
                    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait

      for (i=0; i<n; i++)
              d[i] = 1.0/c[i];
} /*-- конец параллельной области --*/
(подразумевается барьер)
```

Параметр "collapse"

- позволяет параллелить вложенные циклы не прибегая к вложенным параллельным областям
- параметр collaps(n), указывет сколько вложенных циклов (n) должно быть распараллелено

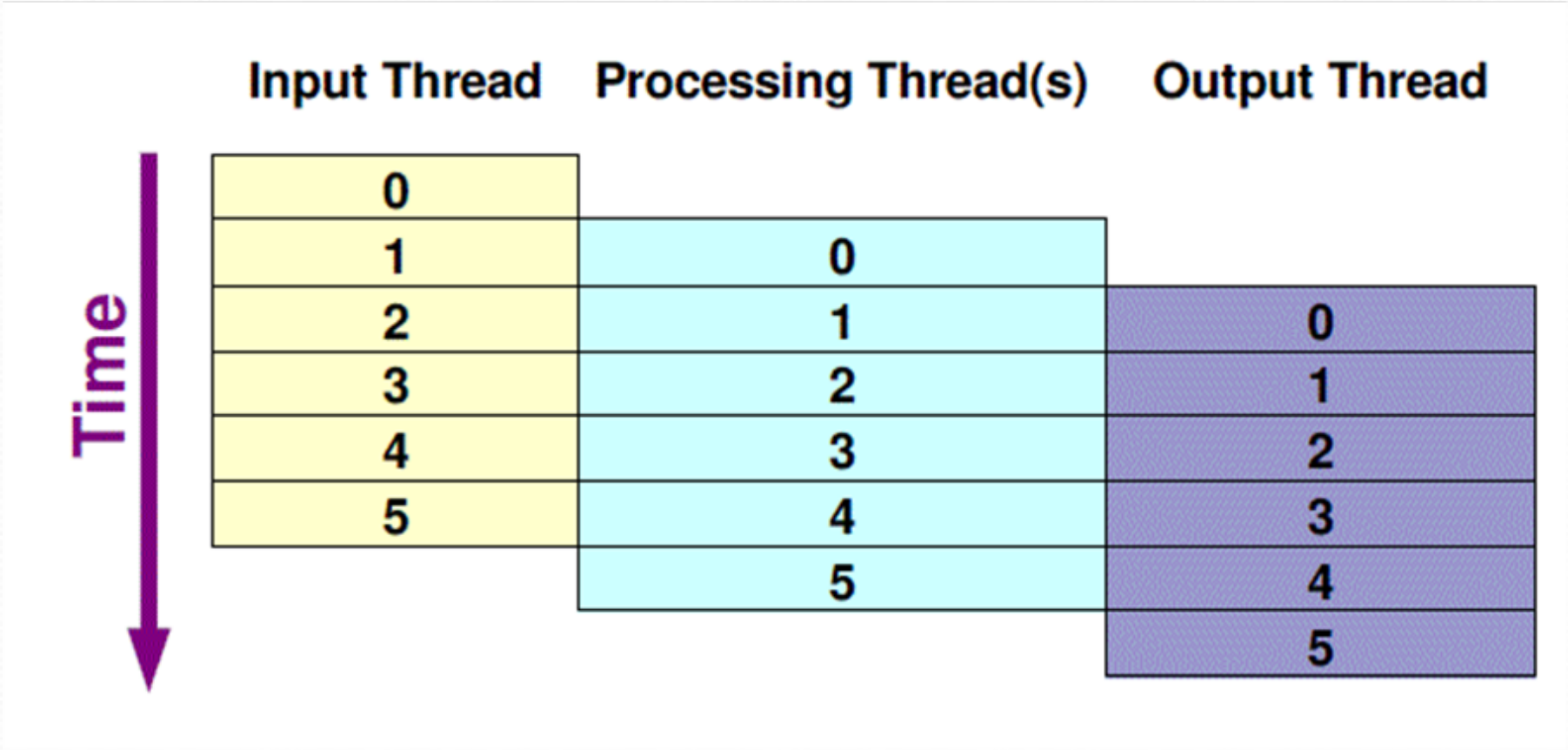
```
#pragma omp parallel for collapse(2) ...
{
        for(i = il; i < iu; i+=is) {
                for(j = jl; j < ju; j+=js) {
                        for(k = kl; k < ku; k+=ks) {
                                .....
                        }
                }
        }
}
```

  - компилятор формирует один цикл и параллелит его

Директива "section" (пример)

```
#pragma omp parallel default(none)\
                        shared(n,a,b,c,d) private(i)
{
      #pragma omp sections nowait
      {
            #pragma omp section
            for (i=0; i<n-1; i++)
                  b[i] = (a[i] + a[i+1])/2;
            #pragma omp section
            for (i=0; i<n; i++)
                  d[i] = 1.0/c[i];
      } /*-- конец секций --*/
} /*-- конец параллельной области --*/
```

Перекрытие ввода/вывода и обработки



Перекрытие ввода/вывода и обработки

```
#pragma omp parallel sections {
        #pragma omp section {
                for (int i=0; i<N; i++) {
                        (void) read_input(i);
                        (void) signal_read(i); }
        }
        #pragma omp section {
                for (int i=0; i<N; i++) {
                        (void) wait_read(i);
                        (void) process_data(i);
                        (void) signal_processed(i); }
        }
        #pragma omp section {
                for (int i=0; i<N; i++) {
                        (void) wait_processed(i);
                        (void) write_output(i); }
        }
} /*-- конец параллельной области --*/
```

Входящие потоки

Обрабатывающие потоки

Выходящие потоки

Директива "single"

Исходный код

```
.....
"read a[0..N-1]";
.....
```

"определяем A как общую переменную"

```
#pragma omp parallel
{
    .....
```

нужен ОДИН поток

```
    "read a[0..N-1]";
```

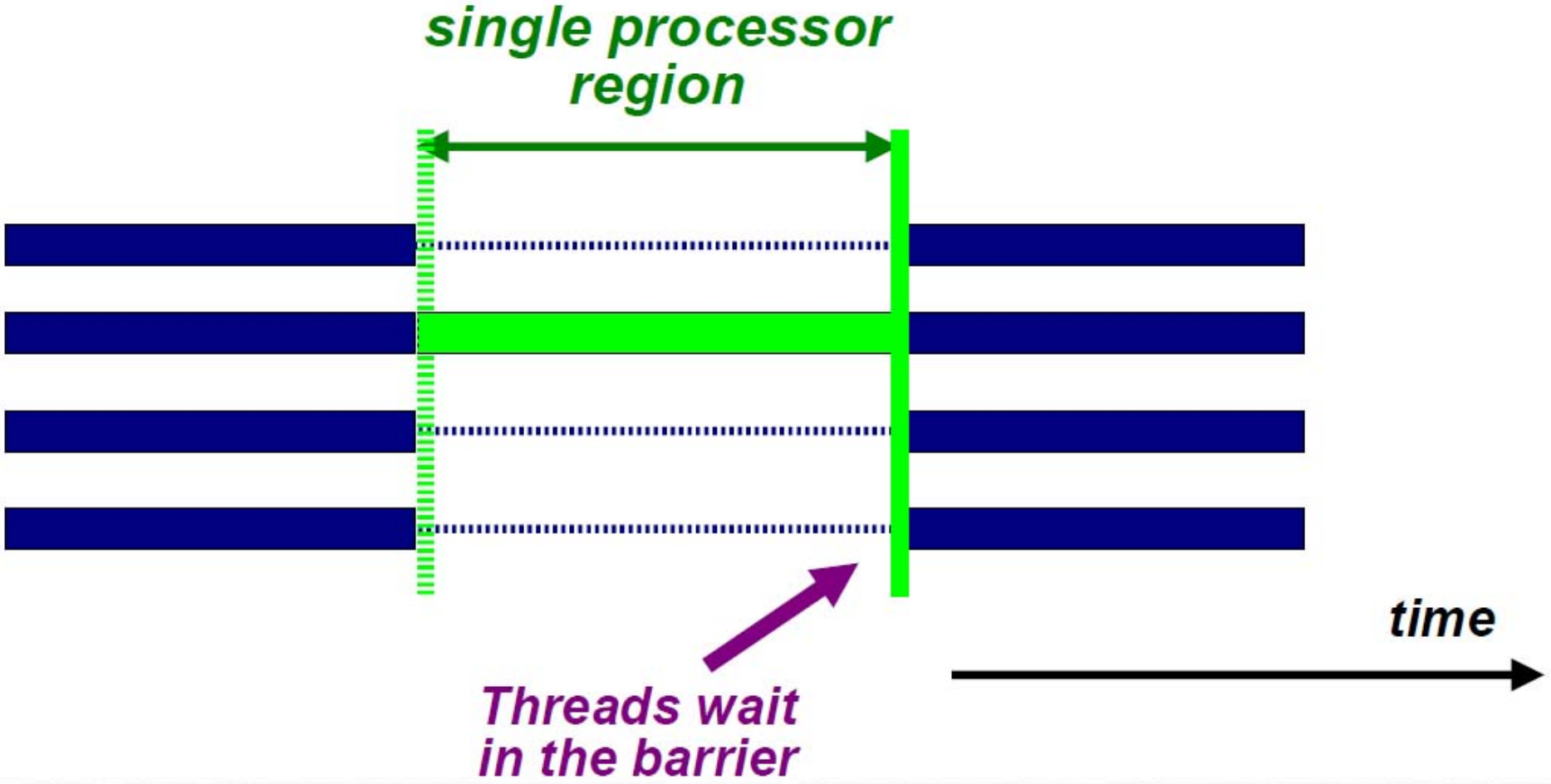
все, считали

```
    .....
}
```

Parallel Version

здесь может понадобиться барьер

Директива "single"



Директива "single"

Только ОДИН поток из команды (team) может изполнять код внутри блока

```
#pragma omp single [private][firstprivate] [copyprivate][nowait]
{
        <code-block>
}
```

```
!$omp single [private][firstprivate]

        <code-block>

!$omp end single [copyprivate][nowait]
```

Сочетание директив

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
!$omp parallel
!$omp do
        ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
        ...
!$omp end parallel do
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
        ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
        ...
!$omp end parallel workshare
```

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

```
!$omp parallel
!$omp sections
        ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
        ...
!$omp end parallel sections
```

Отделяемые директивы

```
                :
#pragma omp parallel
{
                :
        (void) dowork();
                :
}
                :
```

```
void dowork()
{
                :
        #pragma omp for
        for (int i=0;i<n;i++) {
                :
        }
                :
}
```

- Спецификация OpenMP не требует директивам разделения работы и синхронизации (omp for, omp single, critical, barrier, итд) находиться строго внутри лексического контекста параллельной области
- Эти директивы могут использоваться вне лексицеского контекста

Отделяемые директивы

```
(void) dowork();        /* последовательно */

#pragma omp parallel
{
        (void) dowork();    /* параллельно */
}
```

```
void dowork()
{
                :
        #pragma omp for
        for (int i = 0; i < n; i++) {
                :
        }
                :
}
```

Когда отделяемая директива встречается при последовательном исполнении одним потоком (и/или master-thread) вне динамического контекста какой-либо параллельной области, она игнорируется

Процедура "lock" в OpenMP

- Блокировки предоставляют бОльшую гибкость использования критической секции и атомарных операций
  - позволяют конфигурировать асинхронное выполнение
- Так называемая "lock" переменная - это встроенная переменная
  - Fortran: тип INTEGER
  - C/C++: тип omp_lock_t и omp_nest_lock_t (для вложенных блоков)
- Переменная "lock"следует использовать только через заданный интерфейс
- В противном случае, без определенной инициализации, поведение программы может быть непредсказуемым

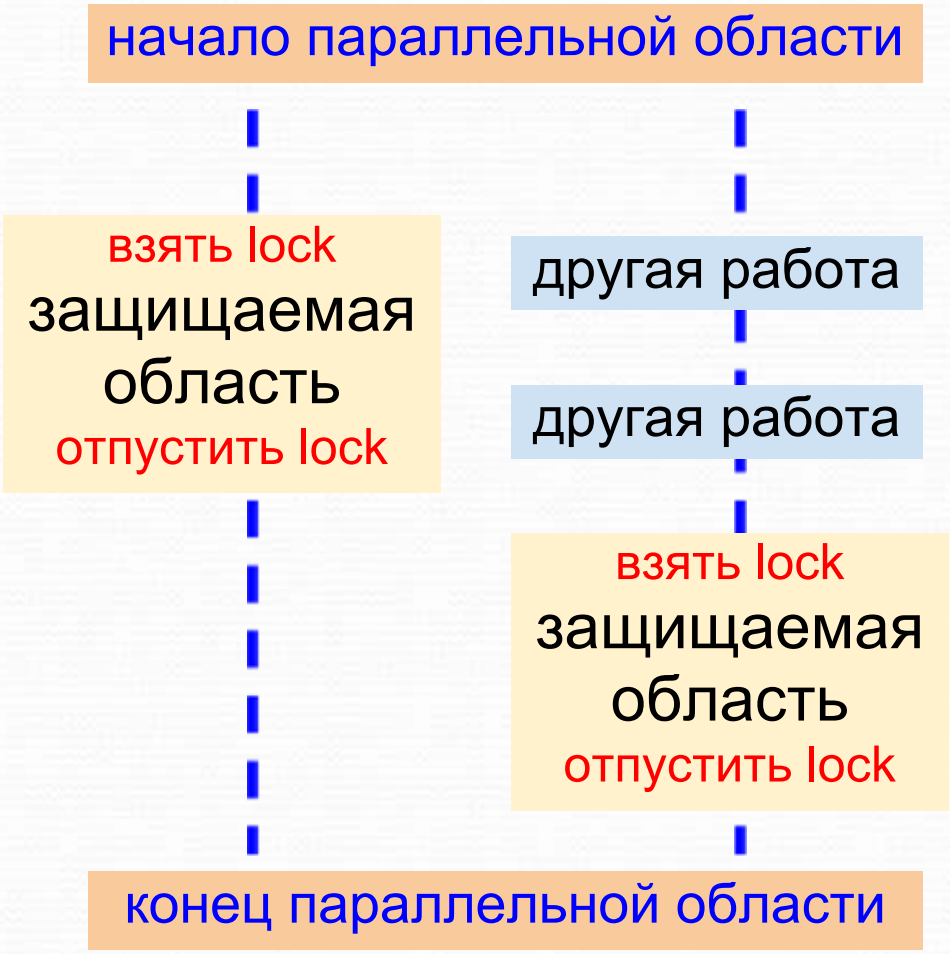
Вложенная блокировка

- Простая блокировка: не можем взять lock если уже кто-то его взял
- Вложенная блокировка: lock может быть взят много раз одним и тем же потоком до того как разблокировать
- Интерфейс функций связанных с простыми и вложенными блокировками:

Простая блокировка:
omp_init_lock
omp_destroy_lock
omp_set_lock
omp_unset_lock
omp_test_lock

Вложенная блокировка:
omp_init_nest_lock
omp_destroy_nest_lock
omp_set_nest_lock
omp_unset_nest_lock
omp_test_nest_lock

Процедура "lock" (пример)



- В защищенной блокировкой области происходит обновление общей переменной
- Один поток берет lock и вычисляет переменную
- В то же время, другой поток выполняет другую работу
- Когда первый поток снимет lock, второй получит доступ к общей переменной

Процедура "lock" (пример)

```
....
omp_init_lock (LCK)
#pragma omp parallel shared(LCK)
{
        while(! omp_test_lock (LCK)) {
                Do_Something_Else()
        }
        Do_Work()
        omp_unset_lock (LCK)
}
omp_destroy_lock (LCK)
```

инициализируем lock переменную

проверяем возможность взятия lock-а (можем -> берем lock)

отпускаем lock

удаляем переменную

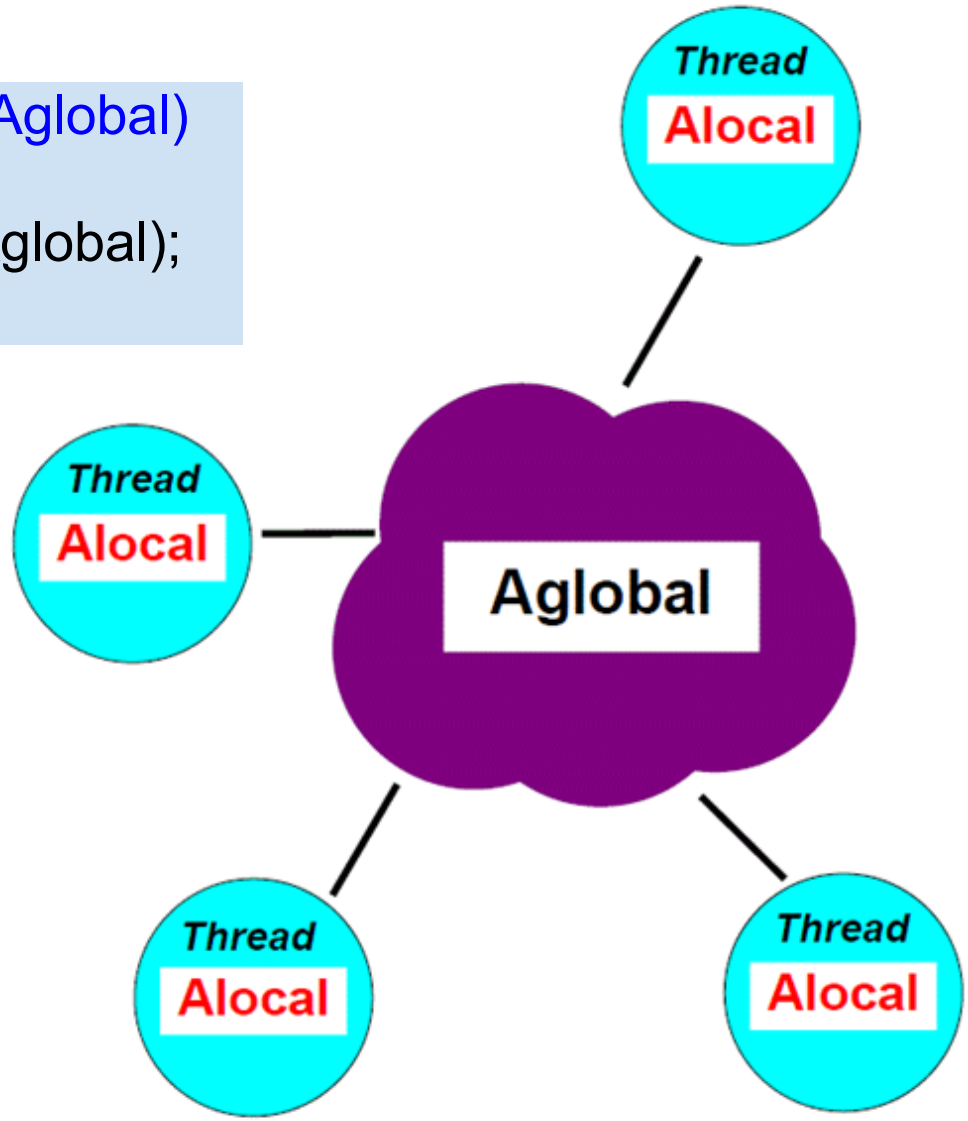
Вывод второго потока

TID: 1 at 09:07:27 => entered parallel region
TID: 1 at 09:07:27 => done with WAIT loop and has the lock
TID: 1 at 09:07:27 => ready to do the parallel work
TID: 1 at 09:07:27 => this will take about 18 seconds
TID: 0 at 09:07:27 => entered parallel region
TID: 0 at 09:07:27 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:32 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:37 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:42 => WAIT for lock - will do something else for 5 seconds
TID: 1 at 09:07:45 => done with my work
TID: 1 at 09:07:45 => done with work loop - released the lock
TID: 1 at 09:07:45 => ready to leave the parallel region
TID: 0 at 09:07:47 => done with WAIT loop and has the lock
TID: 0 at 09:07:47 => ready to do the parallel work
TID: 0 at 09:07:47 => this will take about 18 seconds
TID: 0 at 09:08:05 => done with my work
TID: 0 at 09:08:05 => done with work loop - released the lock
TID: 0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100 (проверяем результат)

Стек

```
#omp parallel shared(Aglobal)
{
        (void) myfunc(&Aglobal);
}
```

```
void myfunc(float *Aglobal)
{
        int Alocal;
        .....
}
```



Переменная Alocal находится в локальной памяти, доступна только потоку-владельцу и хранится в стеке потока

Размер стека

OMP_STACKSIZE n [B,K,M,G]

Set thread stack size in n Byte, KB, MB, or GB
Default is KByte

- У каждого потока есть свой локальный стек
- При выходе за границы стека программа падает по segfault-y
- Существуют два разных размера стека:
  - для master thread-а - задается в зависимости от OS (например, в Unix "ulimit/limit")
  - для Worker thread-ов - используется переменная окружения OMP_STACKSIZE
- Значение OMP_STACKSIZE по-умолчанию зависит от конкретной реализации

OMP_STACK_SIZE (пример)

```
#define N 2000000

void myFunc(int TID, double *check);
void main()
{
        double check, a[N];
        #pragma omp parallel private(check)
        {
                myFunc(&check);
        } /*-- конец параллельной области
}
```

Main() требуется около 16 MB стека

myFunc требуется около 8 MB стека

```
#define MYSTACK 1000000

void myFunc(double *check)
{
        double mystack[MYSTACK];
        int i;

        for (i=0; i<MYSTACK; i++)
                mystack[i] = TID + 1;
        *check = mystack[MYSTACK-1];
                        .....
}
```

OMP_STACK_SIZE (пример)

```
% setenv OMP_NUM_THREADS 1
% limit stack 10k
% ./stack.exe
Segmentation Fault (core dumped)
% limit stack 16m
% ./stack.exe
Thread 0 has initialized local data
% setenv OMP_NUM_THREADS 2
% ./stack.exe
Segmentation Fault (core dumped)
% setenv OMP_STACKSIZE 8192
% ./stack.exe
Thread 0 has initialized local data
Thread 1 has initialized local data
% setenv OMP_NUM_THREADS 4
% ./stack.exe
Thread 0 has initialized local data
Thread 2 has initialized local data
Thread 3 has initialized local data
Thread 1 has initialized local data
```

недостаточный размер стека для master thread-а

сейчас работает для одного потока

Но не работает для двух...

Увеличиваем размер стека для worker-thread-ов

И снова все работает

Управление данными

- Локальные переменные неопределены на входе и на выходе параллельной области
- Локальные переменные вне параллельной области никак не синхронизуются с теми же переменными вне области
- С помощью first/last параметров это можно изменить

Управление данными

```
main()
{
        A = 10;
#pragma omp parallel
{
        #pragma omp for private(i) firstprivate(A) lastprivate(B)...
        for (i=0; i<n; i++) {
                ....                    /* A неопределена, пока не */
                B = A + i;              /* продекларирована как firstprivate */
                ....
        }
        C = B;                          /* B неопределена, пока не */
                                        /* продекларирована как lastprivate */
}
}
```

Параметры (first/last)private

```
#pragma omp parallel firstprivate(list)
```

- Параметр firstprivate позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных

```
#pragma omp parallel lastprivate(list)
```

- Параметр lastprivate позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию)

Параметр "default"

default (none | shared)

C/C++

none

- всем переменным в параллельной области класс должен быть назначен явно

shared

- всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс shared

Параметр "reduction"

reduction (operator: list)

Параметр reduction определяет список переменных, для которых выполняется операция редукции

- перед выполнением параллельной области для каждого потока создаются копии этих переменных
- потоки формируют значения в своих локальных переменных
- при завершении параллельной области на всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных

Параметр "reduction" (пример)

```
main () { // vector dot product
        int i, n, chunk;
        float a[100], b[100], result;        /* общие переменные! */
        n = 100; chunk = 10;
        result = 0.0;
        for (i=0; i < n; i++)
                a[i] = i * 1.0; b[i] = i * 2.0;

        #pragma omp parallel for default(shared) \
                                schedule(static,chunk) reduction(+:result)
        for (i=0; i < n; i++)
                result = result + (a[i] * b[i]);
        printf("Final result= %f\n",result);
}
```

Параметр "schedule"

```
schedule ( static | dynamic | guided | auto [, chunk] )
schedule (runtime)
```

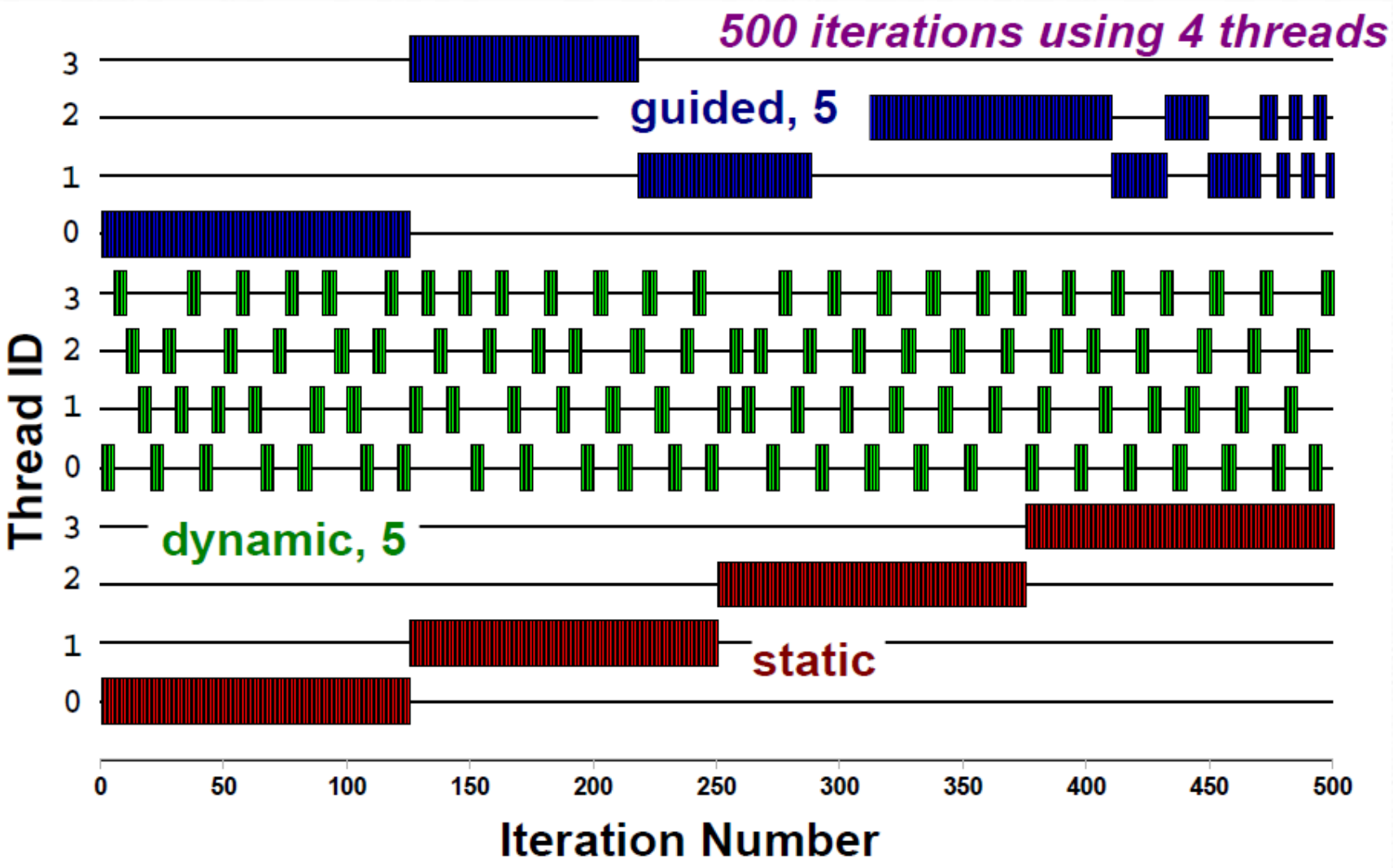
Распределяет итерации в директиве for

- static – итерации делятся по chunk итераций и статически разделяются, если chunk не определен, итерации делятся равномерно и непрерывно
- dynamic – распределение осуществляется динамически (по умолчанию chunk=1)
- guided – размер итерационного блока уменьшается экспоненциально при каждом распределении; chunk определяет минимальный размер блока (chunk=1)
- runtime – правило распределения определяется переменной OMP_SCHEDULE (при использовании runtime параметр chunk задаваться не должен)

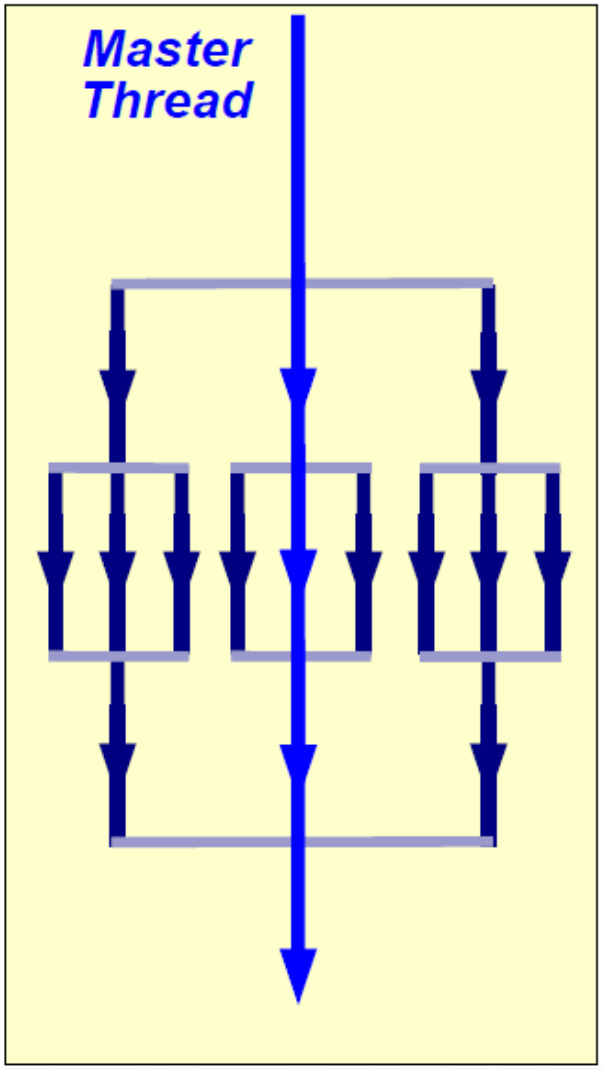
Пример static schedule

Thread	0	1	2	3
chunk не задан	1 - 4	5 - 8	9 - 12	13 - 16
chunk = 2	1 - 2 9 - 10	3 - 4 11 - 12	5 -6 13- 14	7 -8 15 - 16

Пример schedule



Вложенные циклы



3-way parallel

9-way parallel

3-way parallel

Outer parallel region

Nested parallel region

Outer parallel region

Note: nesting level can be arbitrarily deep

Вложенные циклы

- для задания максимального уровня вложенных циклов используются переменная окружения

  OMP_MAX_ACTIVE_LEVELS

  и библиотечные функции

  omp_set_max_active_levels()

  omp_get_max_active_levels()

- для задания максимального количества потоков, также используются переменная окружения

  OMP_THREAD_LIMIT

  и библиотечная функция

  omp_get_thread_limit()

Вложенные циклы

Для контроля за переменными:

omp_set_num_threads()

внутри параллельной области задает размер команды (team) для следующего уровня параллелизма

Библиотечные функции позволяют определить:

- глубину вложенных циклов

  omp_get_level()

  omp_get_active_level()

- IDs родительского/прародительского потока

  omp_get_ancestor_thread_num(level)

- размер родительской/прародительской комнды

  omp_get_team_size(level)

Директива "master"

```
#pragma omp master
```

- Определяет фрагмент кода, которыйдолжен быть выполнен только основным потоком
- Все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется)

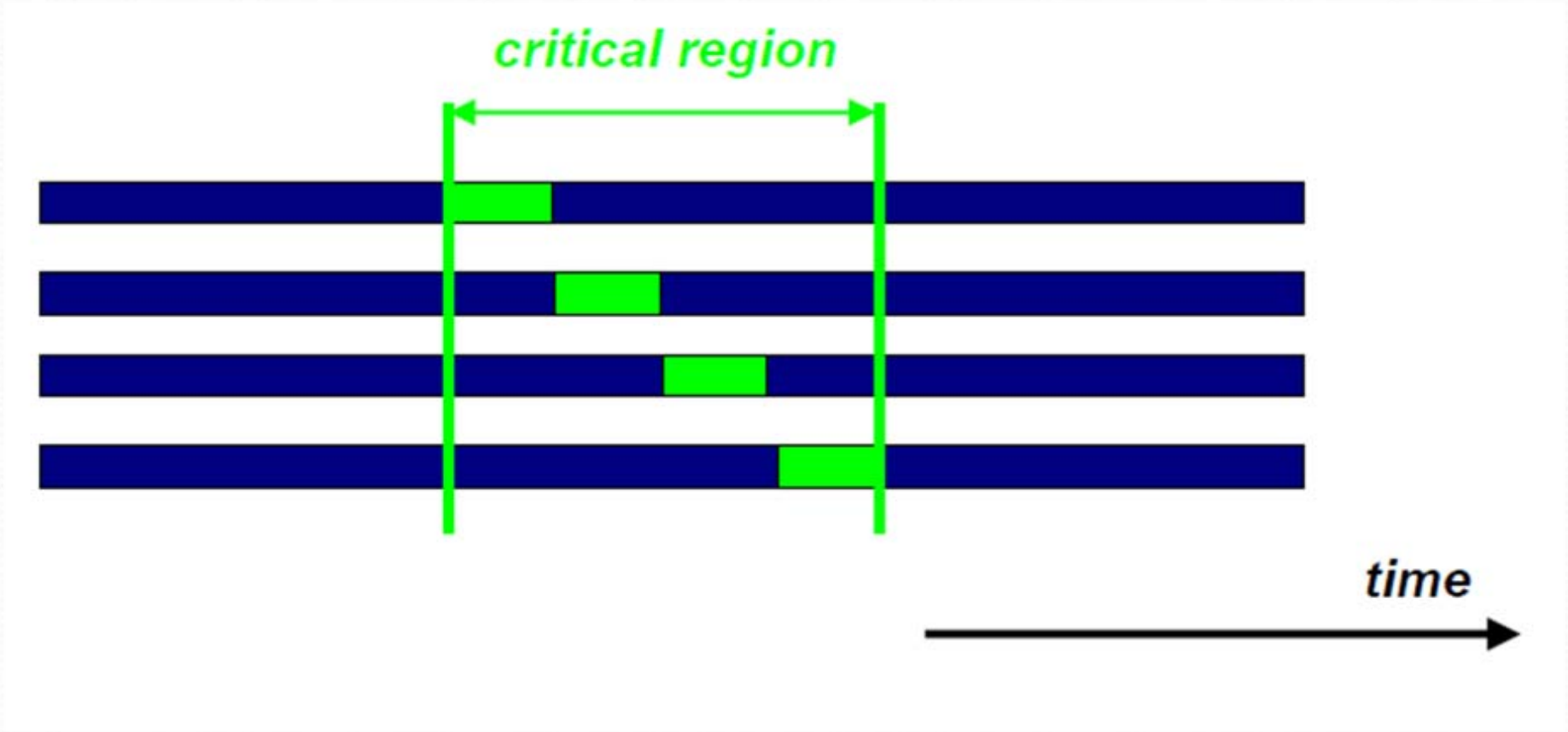
Директива "critical"

```
#pragma omp critical [name]
```

Определяет фрагмент кода, который
должен выполняться только одним потоком в каждый текущий момент времени (критическая секция)

Все неименованные критические секции условно ассоциируются с одним именем

Директива "critical"



Директива "critical"

```
#include <omp.h>

main() {
        int x;
        x = 0;
        #pragma omp parallel shared(x)
        {
                #pragma omp critical
                x = x + 1;
        }
}
```

Директива "atomic"

```
#pragma omp atomic
```

Определяет переменную, доступ к которой (чтение/запись) должна быть выполнена как неделимая операция

• Возможный формат записи выражения
x binop = expr , x++, ++x, x--, --x
- x должна быть скалярной переменной
- expr не должно ссылаться на x
- binop должна быть неперегруженной операцией вида +, -, *, /, &, ^, |, >>, <<

Директива "atomic" (пример)

```
#include <omp.h>

int main(int argc, char *argv[])
{
	int count = 0;
	#pragma omp parallel
	{
		#pragma omp atomic
			count++;
	}
	printf("Число нитей: %d\n", count);
}
```

Директива "flush"

```
#pragma omp flush (list)
```

Определяет точку синхронизации, в которой системой должно быть обеспечено единое для всех процессов состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память)

- Если указан список list, то восстанавливаются только указанные переменные
- Директива flush неявным образом присутствует в директивах barrier, critical, ordered, parallel, for, sections, single

Вопросы?