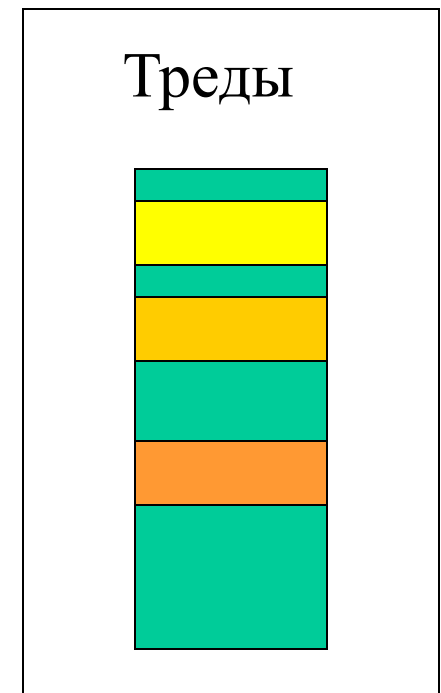
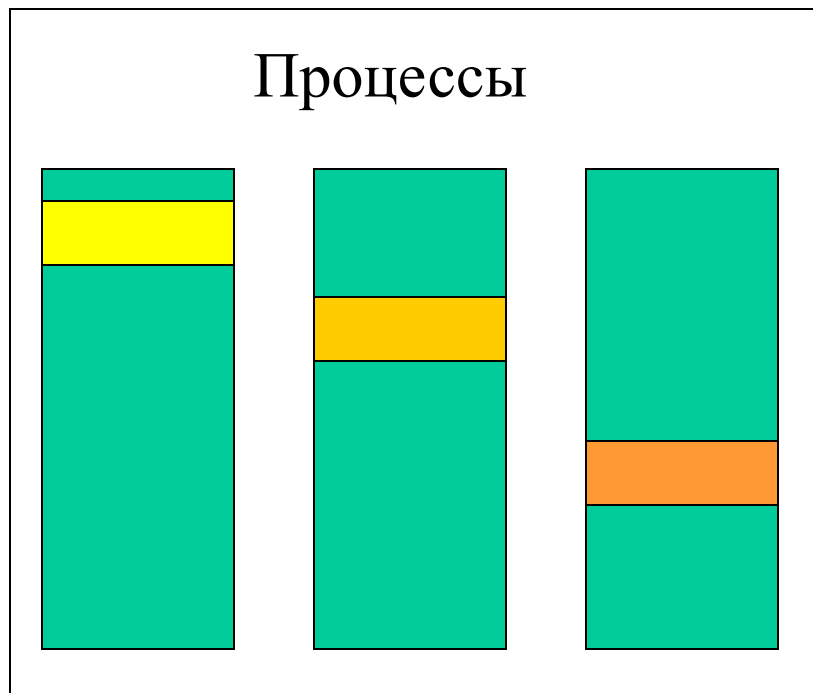


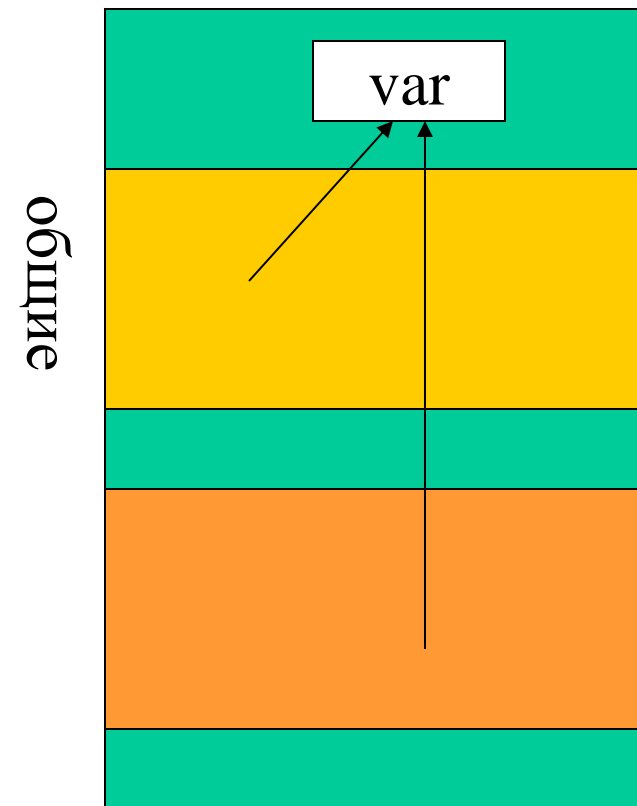
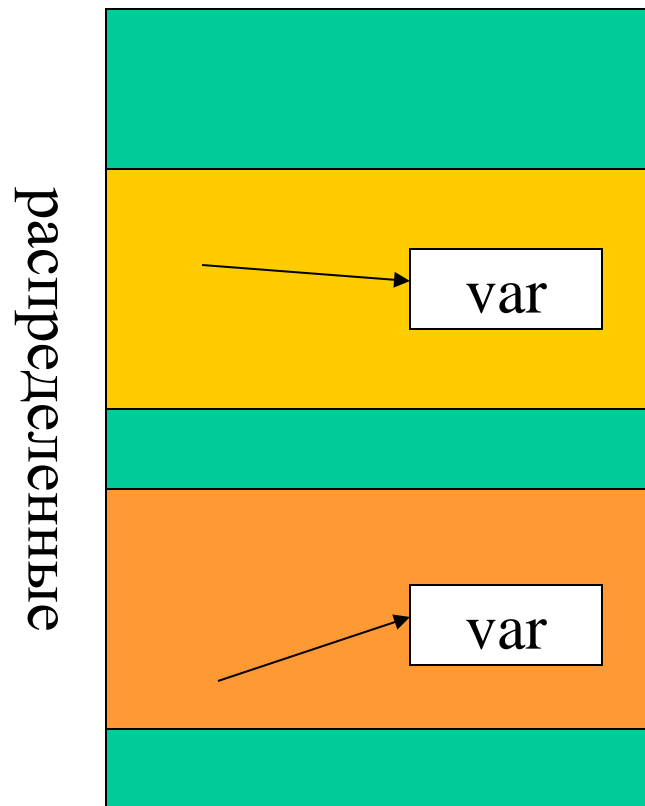
OpenMP



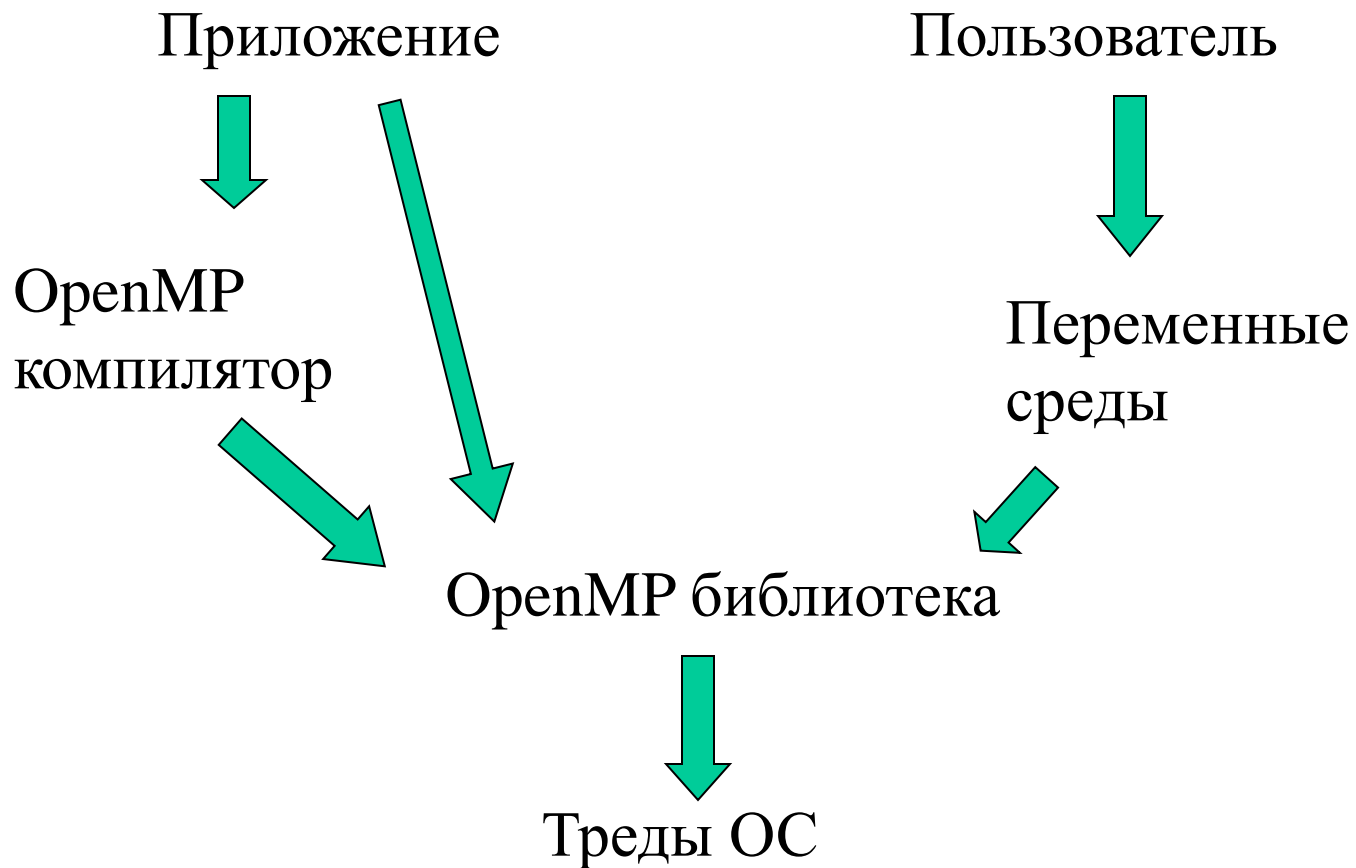
Различие между тредами и процессами



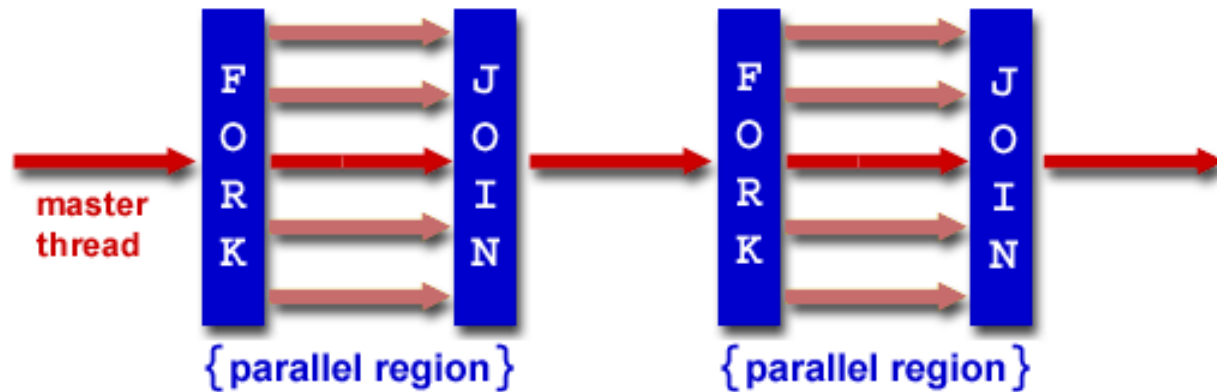
Общие и распределенные данные



Архитектура OpenMP



Модель выполнения OpenMP приложения



Работа с вычислительным пространством — число тредов

- Мастер-тред имеет номер 0
- Число тредов, выполняющих работу определяется:
 - переменная окружения **OMP_NUM_THREADS**
 - вызов функции **omp_set_num_threads()** (может вызываться перед параллельным участком, но не внутри этого участка)
- Определение числа процессоров в системе:
omp_get_num_procs()



Работа с вычислительным пространством – динамическое определение числа тредов

В некоторых случаях целесообразно устанавливать число тредов динамически в зависимости от загрузки имеющихся процессоров.

Включить данную опцию можно с помощью переменной среды

OMP_DYNAMIC [TRUE, FALSE]

или с помощью функции

omp_set_dynamic(int flag) (может вызываться перед параллельным участком, но не внутри этого участка)
Если $\text{flag} \neq 0$, то механизм включается, в противном случае – выключается.



Определение числа процессоров, тредов и своих координат в системе

int omp_get_num_procs() возвращает количество процессоров в системе;

int omp_get_num_threads() возвращает количество тредов, выполняющих параллельный участок (меняется только на последовательных участках);

int omp_get_thread_num() возвращает номер вызывающего тредда.




```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
```

```
int main()
{
```

```
    int np = omp_get_num_procs();
    printf("Total number of processors is %d\n", np);
    omp_set_num_threads(np);
```

```
#pragma omp parallel
    printf("Hello, World from thread %d of %d\n",
           omp_get_thread_num(), omp_get_num_threads());
}
```



Результат выполнения

```
Total number of processors is 4  
Hello, World from thread 1 of 4  
Hello, World from thread 0 of 4  
Hello, World from thread 3 of 4  
Hello, World from thread 2 of 4  
Для продолжения нажмите любую клавишу . . .
```

Запущено четыре потока: каждый печатает сообщение.



Общий синтаксис директив OpenMP

#pragma omp directive_name [clause[clause ...]] newline

Действия, соответствующие директиве, применяются непосредственно к структурному блоку, расположенному за директивой. Структурным блоком может быть любой оператор, имеющий единственный вход и единственный выход.

Если директива расположена на файловом уровне видимости, то она применяется ко всему файлу.



Директива parallel

#pragma omp parallel (опции)

где опции

if(scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(reduction-identifier :list)
proc_bind(master | close | spread)



Определение числа потоков

- Если `if(условие)` принимает значение `false`, то выполняется один поток
- В противном случае, используется ранее установленное значение или значение опции `num_threads`



Пример задания числа потоков

```
#include <iostream>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int fl = atoi(argv[1]);
    int nt = atoi(argv[2]);
#pragma omp parallel if(fl) num_threads(nt)
    std::cout << "hello\n";

    return 0;
}
```



Результат выполнения

```
admin@irbis8:~/course$ c++ -o omp1 omp1.cpp
admin@irbis8:~/course$ ./omp1 0 4
hello
admin@irbis8:~/course$ ./omp1 1 4
hello
admin@irbis8:~/course$
```

Необходимо «включить» поддержку OpenMP
(опция компилятора -fopenmp)



Результат выполнения

```
admin@irbis8:~/course$ ./omp1 1 4  
hello  
hello  
hello  
hello  
admin@irbis8:~/course$ ./omp1 0 4  
hello  
admin@irbis8:~/course$
```



Переменные в OpenMP-программе

```
extern int A;
void f(int c)
{
    static double z;
    int x;
}

main()
{
    double y;
    #pragma omp parallel
    {
        int a;
        f(5);
    }
}
```

По отношению к
параллельному
участку следующие
переменные являются
общими:

A, z, y

остальные

переменные:

a, c, x

являются

индивидуальными.



Опции для данных

Опция **private**

Данные, видимые в области, объемлющей блок параллельного исполнения, являются общими (**shared**). Переменные, объявленные внутри блока п.и. считаются распределенными (**private**).

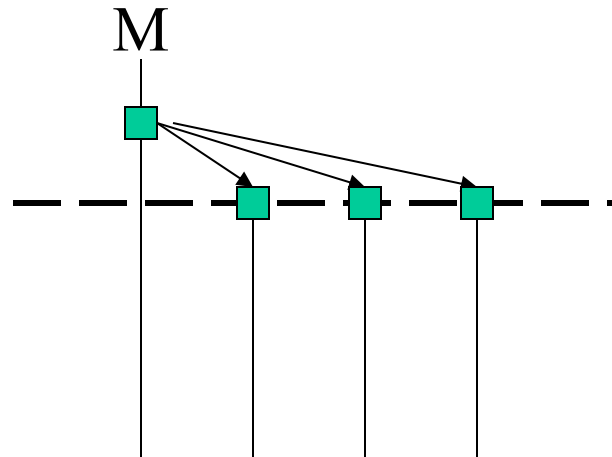
Опция **private** задает список распределенных переменных.

Только shared-переменные в объемлющей параллельном блоке могут быть аргументами опции **private**



Опция **firstprivate**

Опция **firstprivate** обладает той же семантикой, что и опция **private**. При этом, все копии переменной инициализируются значением исходной переменной до входа в блок на мастер-треде.



Опция **default**

Опция **default** задает опцию по-умолчанию для переменных. Пример:

```
#pragma omp parallel default(private)
```

Опция **shared**

Опция **shared** задает список общих переменных.

```
#pragma omp parallel default(private) shared(x)
```



```
#include <omp.h>

main () {

int nthreads, tid;


#pragma omp parallel private(nthreads, tid)
{

    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }

}
```



Глобальные общие данные

Проблема: опция `private` «работает» только для статически-видимых ссылок в пределах параллельного участка:

```
static int a;
```

```
f() {  
    printf(“%d\n”, a);  
}
```

значение a
неопределено

```
main() {  
    #omp parallel private (a)  
    {  
        a = omp_num_thread();  
        f();  
    }  
}
```

кроме того, отсутствует возможность передачи данных между параллельными участками



Директива threadprivate

#omp threadprivate (список глобальных и статических переменных) переменные становятся общими для всех тредов:

```
static int a;

#omp threadprivate(a)

f() {
    printf("%d\n", a);
}

main() {
#omp parallel
{
    a = omp_num_thread();
    f();
}
}
```



Ограничения для **threadprivate**

- Директива **threadprivate** должна следовать после объявления переменной, но предшествовать ее первому использованию
- Директива **threadprivate** должна располагаться на том же уровне видимости, что и объявление переменной
- Значение переменной на мастер-потоке сохраняется всегда между параллельными участками, а на остальных потоках только если не используется вложенный параллелизм и динамическое изменение числа тредов (требуется явно указать `set_omp_dynamic(0)`)



Опция `copyin`

Опция **`copyin`** директивы **`parallel`** определяет порядок инициализации **`threadprivate`**-переменных: эти переменные инициализируются значением на **`master`**-треде в начале параллельного участка.



```
static int a = -1;  
static int b = -1;
```

```
#pragma omp threadprivate(b)
```

```
void f() {  
#pragma omp critical  
    {  
        std::cout << "a = " << a << "\n";  
        std::cout << "b = " << b << "\n";  
    }  
}
```

```
int main(int argc, char* argv[]) {
```

```
#pragma omp parallel private (a) num_threads(4)  
    {  
        a = omp_get_thread_num();  
        b = omp_get_thread_num();  
        f();  
    }  
}
```



Результат выполнения

```
admin@irbis8:~/course$ ./thprv  
a = -1  
b = 2  
a = -1  
b = 1  
a = -1  
b = 0  
a = -1  
b = 3  
admin@irbis8:~/course$
```



Управление распределением вычислений

Для распределения вычислений применяются конструкции:

- **for**
- **sections**
- **single**



Директива **for**

#pragma omp for [clause ...]

clause:

schedule (type [,chunk])

ordered

private (list)

firstprivate (list)

lastprivate (list)

reduction (operator: list)

nowait



Директива предшествует циклу **for** канонического типа:

for(init-expr; var logical_op b; incr_expr)

init_expr ::= var = expr

logical_op >, <, >=, <=



`incr_expr ::= var ++`
`++ var`
`var --`
`-- var`
`var += incr`
`var -= incr`
`var = incr + var`
`var = var + incr`
`var = var - incr`

`var` переменная целого типа

`incr, b` инварианты цикла целого типа



```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(int argc, char* argv[])
{
    int n, iters, t, i, j;
    double *a, *b, alpha = 0.1;
    n = atoi(argv[1]);
    iters = atoi(argv[2]);

    a = (double*)malloc(n * sizeof(double));
    b = (double*)malloc(n * sizeof(double));
    t = time(NULL);
    for(i = 0; i < iters; i ++) {
        #pragma omp parallel for
        for(j = 0; j < n; j ++) {
            a[j] = a[j] + alpha * b[j];
        }
    }
    t = time(NULL) - t;
    printf("parallel loop: %d seconds\n", t);
}
```

Сложение (с умножением)
векторов – параллельный
вариант.



Результаты эксперимента

Компьютер: 2 x 64-разрядный
процессор Intel® Itanium-2® 1.6 ГГц.



Размерность	Число итераций	1 CPU	2 CPU
20000	200000	8 сек	4 сек



ИНФОРМАЦИОННЫЕ ЗАВИСИМОСТИ И ИХ УСТРАНЕНИЕ

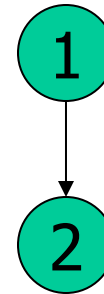


Информационные зависимости

Зависимость по данным:

1: $a = 1;$

2: $b = a;$



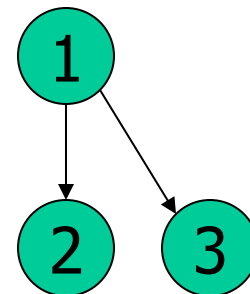
Зависимость по управлению:

1: $\text{if}(a) \{$

2: $x = c + d;$

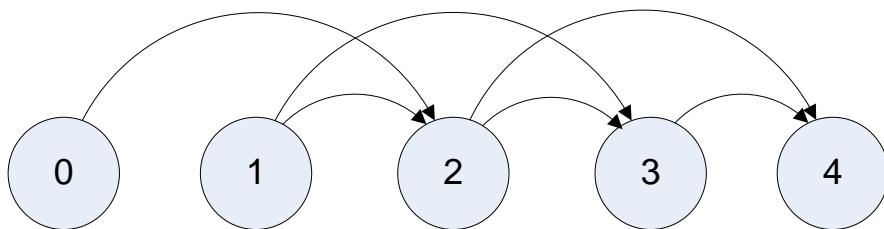
3: $y = 1;$

4: $\}$



Информационные зависимости в цикле

```
for(j = 0; j < n; j++)  
    a[j] = a[j - 1] + a[j - 2];
```



Численное интегрирование

```
double func(double x)
{
    double rv;
    int i;
    const double alpha = 100;
    const int n = 100;
    rv = 0.;
    for(i = 0; i < n; i++) {
        rv += cos(i * x) + sin(i * x) +
log(i * x + 1);
    }
    return rv;
}
```

```
main()
{
    double A, B, v;
    int N;

    A = 0.;
    B = 100.;
    N = 10000000;
    v = integr(A, B, N, func);
    printf("%lf\n", v);
}
```



Численное интегрирование

```
double integr(double a, double b, int n, double (*g)(double))
{
    int i;
    double s, h;

    s = 0.;
    h = (b - a) / n;
    #pragma omp parallel for
    for(i = 0; i < n; i++){
        s += g(a + i * h + 0.5 * h);
    }
    return s * h;
}
```



Численное интегрирование

Результаты работы

2 x Intel Xeon 53xx (Clovertown) – 8 core:

Последовательный вариант:

41 с.

$v = 71747.34$

Параллельный вариант:

5 с.

$v = 56234.44$

не совпадают !

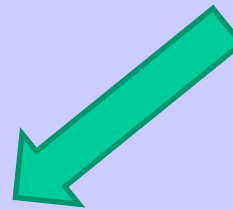


Численное интегрирование

```
double integr(double a, double b, int n, double (*g)(double))
{
    int i;
    double s, h;

    s = 0.;
    h = (b - a) / n;
    #pragma omp parallel for
    for(i = 0; i < n; i++){
        s += g(a + i * h + 0.5 * h);
    }
    return s * h;
}
```

На каждой
итерации
изменяется
значение
переменной s



опция **reduction**

Опция **reduction** определяет что на выходе из параллельного блока переменная получит комбинированное значение. Пример:

#pragma omp for reduction(+ : x)

Допустимы следующие операции: +, *, -, &, |, ^, &&, ||



опция **reduction**

Опция **reduction** определяет значение переменных, входящих в список ее аргументов, на главном потоке после завершения параллельного участка как результат выполнения редуктивной операции. На каждом из потоков, выполняющих параллельный участок, переменная инициализируется значением, соответствующим редуктивной операции.

Пусть параллельный участок выполнялся n потоками, и до него переменная имела значение v . Если в конце выполнения параллельного участка локальные копии переменной a имели значения v_1, \dots, v_n , то после параллельного участка переменная a на главном потоке получит значение, равное $(v \bowtie v_1 \bowtie v_2 \bowtie \dots \bowtie v_n)$.



Опция reduction

операция	значение для инициализации
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0



Численное интегрирование (правильный вариант)

```
double integr(double a, double b, int n, double (*g)(double))
{
    int i;
    double s, h;

    s = 0.;
    h = (b - a) / n;
    #pragma omp parallel for reduction(+:s)
    for(i = 0; i < n; i++){
        s += g(a + i * h + 0.5 * h);
    }
    return s * h;
}
```



Численное интегрирование

Результаты работы
2 x Intel Xeon 53xx (Clovertown) – 8 core:

Последовательный вариант:

41 с.

$v = 71747.34$

Параллельный вариант:

5 с.

$v = 71747.34$

совпадают !



Опция **schedule** директивы **for**

Опция **schedule** допускает следующие аргументы:

static - распределение осуществляется статически (порциями);

dynamic - распределение осуществляется динамически (тред, закончивший выполнение, получает новую порцию итераций);

guided - аналогично **dynamic**, но на каждой следующей итерации размер распределяемого блока итераций равен примерно общему числу оставшихся итераций, деленному на число исполняемых тредов, если это число больше заданного значения **chunk**, или значению **chunk** в противном случае (крупнее порция – меньше синхронизаций)

runtime - распределение осуществляется во время выполнения системой поддержки времени выполнения (параметр **chunk** не задается) на основе переменных среды



Особенности опции **schedule** директивы **for**

- аргумент `chunk` можно использовать только вместе с типами `static`, `dynamic`, `guided`
- по умолчанию `chunk` считается равным 1
- распараллеливание с помощью опции `runtime` осуществляется используя значение переменной `OMP_SCHEDULE`

Пример.

```
setenv OMP_SCHEDULE "guided,4"
```



Директива **sections**

```
#pragma omp sections [clause ...]
```

```
structured_block
```

```
clause:                private (list)
                        firstprivate (list)
                        lastprivate (list)
                        reduction (operator: list)
                        nowait
```

```
{
```

```
    #pragma omp section
```

```
        structured_block
```

```
    #pragma omp section
```

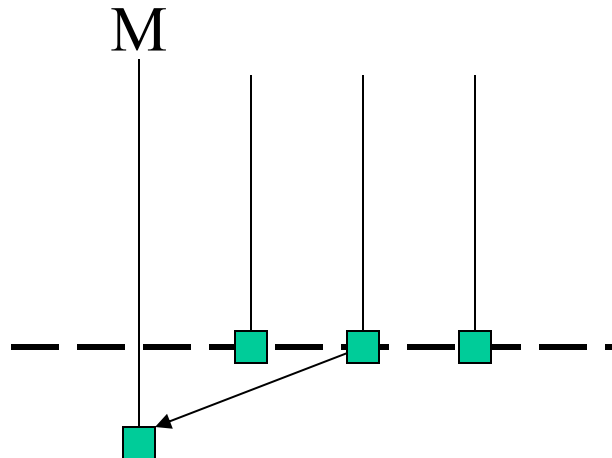
```
        structured_block
```

```
}
```



Опция **lastprivate**

Опция **lastprivate** обладает той же семантикой, что и опция **private**. При этом, значение переменной после завершения блока параллельного исполнения определяется как ее значение на последней итерации цикла или в последней секции для work-sharing конструкций (с точки зрения последовательного выполнения).



```
#include <omp.h>
#include <stdio.h>

int main() {
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    {
        printf ("id = %d\n",
omp_get_thread_num());
    }

    #pragma omp section
    {
        printf ("id = %d \n",
omp_get_thread_num());
    }
}
}
```



```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 13
```

```
id = 1
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 15
```

```
id = 1
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 1
```

```
id = 11
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 1
```

```
id = 9
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 3
```

```
id = 9
```

```
admin@irbis8:~/exp/lectures/openmp$ █
```



```
#include <omp.h>
#include <stdio.h>

int main() {
    int a = 0;
    #pragma omp parallel
    #pragma omp sections lastprivate(a)
    {
        #pragma omp section
        {
            a = 1;
            printf ("id = %d\n", omp_get_thread_num());
        }

        #pragma omp section
        {
            a = 2;
            printf ("id = %d \n", omp_get_thread_num());
        }
    }
    printf("a = %d\n", a);
}
```





admin@irbis8: ~/exp/lectures/openmp

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 12
```

```
id = 3
```

```
a = 2
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 11
```

```
id = 1
```

```
a = 2
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 0
```

```
id = 9
```

```
a = 2
```

```
admin@irbis8:~/exp/lectures/openmp$ ./sections
```

```
id = 1
```

```
id = 11
```

```
a = 2
```

```
admin@irbis8:~/exp/lectures/openmp$ █
```



Директива **single**

```
#pragma omp single [clause ...]  
structured_block
```

Директива `single` определяет что последующий блок будет выполняться только одним тредом



Директивы синхронизации

- master
- critical
- barrier
- atomic
- flush
- ordered



#pragma omp master

определяет секцию кода, выполняемого только master-тредом

#pragma omp critical [(name)]

определяет секцию кода, выполняемого только одним тредом в данный момент времени

#pragma omp barrier

барьерная синхронизация



Директива flush

```
#pragma omp flush [memory-order-clause] [(list)] new-line
```

Конструкция flush выполняет операцию синхронизации временного локального представления памяти потока с общей памятью и обеспечивает порядок операций с памятью.

Применяется ко всем переменным, обрабатываемым в потоке либо к списку операции переменных, явно указанных или подразумеваемых.

Атомарные операции в OpenMP

```
#pragma omp atomic [clause[[[,] clause] ... ] [,]] atomic-clause  
[[[,] clause [[[,] clause] ... ]]
```

expression-stmt

or

```
#pragma omp atomic [clause[,] clause] ... ] new-line
```

expression-stmt

or

```
#pragma omp atomic [clause[[[,] clause] ... ] [,]] capture  
[[[,] clause [[[,] clause] ... ]]
```

structured-block

atomic_clause

- read – атомарное чтение
- write – атомарная запись
- update – атомарное изменение (вариант по умолчанию, когда clause не указан)
- capture – группа операций

Синтаксические ограничения

atomic-clause	expr-stmt
read	v = x;
update (или не указано)	x++; x--; ++x; --x; binop= expr; x = x binop expr; x = expr binop x;

Синтаксические ограничения

atomic-clause	expr-stmt
capture	$v = x++;$ $v = x--;$ $v = ++x;$ $v = --x;$ $v = x \text{ binop} = \text{expr};$ $v = x = x \text{ binop} \text{ expr};$ $v = x = \text{expr binop} x;$

Синтаксические ограничения

atomic-clause	stmt
capture	<pre>{ v = x; x binop= expr; } { x binop= expr; v = x; } { v = x; x = x binop expr; } { v = x; x = expr binop x; } { x = x binop expr; v = x; } { x = expr binop x; v = x; } { v = x; x = expr; } { v = x; x++; } { v = x; ++x; } { ++x; v = x; } { x++; v = x; } { v = x; x--; } { v = x; --x; } { --x; v = x; } { x--; v = x; }</pre>

Используемые операции

- Neither of v and expr (as applicable) may access the storage location designated by x .
- Neither of x and expr (as applicable) may access the storage location designated by v .
- expr is an expression with scalar type.
- binop is one of $+$, $*$, $-$, $/$, $\&$, $^$, $|$, $<<$, or $>>$.
- binop , $\text{binop} =$, $++$, and $--$ are not overloaded operators.

atomic capture

- В этом случае гарантируется, что производится атомарное чтение и запись переменной **x** и изменение значения переменной **v** в соответствии с семантикой C++.
- Атомарность вычисления выражений и записи **v** не гарантируется.

Например, конструкция

v = x++;

приводит к чтению оригинального значения переменной **x**, записи его в переменную **v** и изменению значения переменной **x**

Пример atomic

```
#include <omp.h>
```

```
float work1(int i)  
{  
    return 1.0 * i;  
}
```

```
float work2(int i)  
{  
    return 2.0 * i;  
}
```

```
void atomic_example(float *x, float *y, int *index, int n)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        #pragma omp atomic update
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```

атомарная

неатомарная

OpenMP может провести оптимизацию если записываются по факту разные ячейки массива **x** — в этом случае не включать синхронизацию.

```
int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;
    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }
    for (i = 0; i < 1000; i++)
        x[i] = 0.0;
    atomic_example(x, y, index, 10000);
    return 0;
}
```

atomic read/write

```
int atomic_read(const int *p)
{
    int value;
    #pragma omp atomic read
    value = *p;
    return value;
}

void atomic_write(int *p, int value)
{
    #pragma omp atomic write
    *p = value;
}
```

atomic capture

```
struct locktype {  
    int ticketnumber;  
    int turn;  
};  
void do_locked_work(struct locktype *lock)  
{  
    int atomic_read(const int *p);  
    void work();  
    int myturn = fetch_and_add(&lock->ticketnumber);  
    while (atomic_read(&lock->turn) != myturn)  
        work();  
    #pragma omp flush  
    fetch_and_add(&lock->turn);  
}
```


«Наивный» fetch_and_add

```
int fetch_and_add(int *p)
{
    #pragma omp atomic
    (*p)++;
    return *p;
}
```

«Наивный» fetch_and_add

```
int fetch_and_add(int *p)
{
    #pragma omp atomic
    (*p)++;
    return *p;
}
```

значение по *p может
измениться



Использование atomic capture

```
int fetch_and_add(int *p)
{
    int old;
    #pragma omp atomic capture
    {
        old = *p;
        (*p)++;
    }
    return old;
}
```


Директива task

#pragma omp task [clause[[,] clause] ...] new-line
structured-block

Генерируется задание для структурного блока. Поток, который выполняет конструкцию task либо сразу выполняет задание, либо откладывает выполнение — тогда задание может быть выполнено любым потоком из группы.

```
#include <iostream>
#include <omp.h>
int main(){
#pragma omp parallel num_threads(4)
{
#pragma omp task
{
    std::cout << "[ " << omp_get_thread_num();
    std::cout << "-hello ]";
}
}
}
```

Вывод (3 разных запуска):

```
[ [ 2-hello ]1-hello ][ 1-hello ][ 3-hello ]
[ [ 2-hello ][ 1-hello ]0-hello ][ 3-hello ]
[ [ 2-hello ][ 1-hello ]0-hello ][ 3-hello ]
```

Обращение с данными в tasks

- shared-переменные в запускающем потоке остаются shared
- private-переменные становятся firstprivate

Правила по умолчанию для переменных в tasks

- In a task generating construct, if no default clause is present, a variable for which the data-sharing attribute is not determined by the rules above and that in the enclosing context is determined to be **shared** by all implicit tasks bound to the current team is **shared**.
- In a task generating construct, if no default clause is present, a variable for which the data-sharing attribute is not determined by the rules above is **firstprivate**.

Правила «приватизации» переменных в tasks

- For a firstprivate clause on a parallel, task, taskloop, target, or teams construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered unless otherwise specified.
- For a firstprivate clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered unless otherwise specified.

shared-переменные

```
#include <iostream>
#include <omp.h>
int main(){
int a = 0;
#pragma omp parallel num_threads(4)
{
#pragma omp task
{
#pragma omp atomic write
a = omp_get_thread_num() + 1;
std::cout << "[ " << a << "-hello ]";
}
}
std::cout << "a = " << a << std::endl;
}
```

Вывод: [2-hello][2-hello][3-hello][3-hello]a = 3

Вывод: [[3-hello]3-hello][1-hello][2-hello]a = 2

private-переменные

```
#include <iostream>
#include <omp.h>
int main(){
int a = 0;
#pragma omp parallel num_threads(4)
{
#pragma omp task private(a)
{
a = omp_get_thread_num() + 1;
std::cout << "[ " << a << "-hello ]";
}
}
std::cout << "a = " << a << std::endl;
}
```

Вывод: [1-hello][1-hello][4-hello][2-hello]a = 0

Вывод: [4-hello][1-hello][3-hello][2-hello]a = 0

private-переменные

```
#include <iostream>
#include <omp.h>
int main(){
#pragma omp parallel num_threads(4)
{
    int a = 41;
    #pragma omp task
    {
        std::cout << "taska = " << a << "\n";
        a = omp_get_thread_num() + 1;
        std::cout << "[ " << a << "-hello ]";
    }
    std::cout << "a = " << a << std::endl;
}
}
```


private-переменные

Вывод:

input

[4-hello]taska = 41

[4-hello]a = 41

taska = a = 41

41

taska = 41

[4-hello][3-hello]a = 41

Директива taskwait

`#pragma omp taskwait`
директива task

Используется для синхронизации заданий, созданных в текущем участке (тот же уровень вложенности) до директивы taskwait.

```

#include <iostream>
#include <omp.h>
int main(){
    constexpr int nthr = 4;
    constexpr int ntask = 8;
#pragma omp parallel num_threads(nthr)
    {
#pragma omp master
        {
            for(int i = 0; i < ntask; i++)
#pragma omp task
                {
                    sleep(i);
                    std::cout << "[ " << omp_get_thread_num() << "-T " << i << "]\n";
                }

#pragma omp taskwait

            std::cout << "\n<*** " << omp_get_thread_num() << " ***>\n";
        }
    }
}

```

Вывод на экран при разных запусках:

[1-T 0]

[1-T 1]

[3-T 2]

[2-T 3]

[1-T 4]

[0-T 7]

[3-T 5]

[2-T 6]

[2-T 0]

[2-T 1]

[3-T 2]

[1-T 3]

[2-T 4]

[0-T 7]

[3-T 5]

[1-T 6]

[3-T 0]

[3-T 1]

[1-T 2]

[2-T 3]

[3-T 4]

[0-T 7]

[1-T 5]

[2-T 6]

<*** 0 ***>

<*** 0 ***>

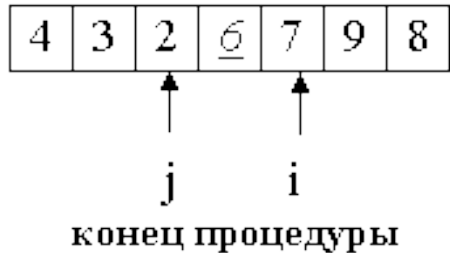
<*** 0 ***>

Быстрая сортировка

На входе массив $a[0]...a[N]$ и опорный элемент p (обычно средний элемент в массиве), по которому будет производиться разделение.

1. Введем два указателя: i и j . В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент $a[i] \geq p$. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден $a[j] \leq p$.
3. Далее, если $i \leq j$, меняем $a[i]$ и $a[j]$ местами и продолжаем двигать i, j по тем же правилам...
4. Повторяем шаг 3, пока $i \leq j$.

Быстрая сортировка



Быстрая сортировка (послед. вариант)

```
void quickSort(float* a, const long n) {  
    long i = 0, j = n;  
    float pivot = a[n / 2];  
    do {  
        while (a[i] < pivot) i++;  
        while (a[j] > pivot) j--;  
        if (i <= j) {  
            std::swap(a[i], a[j]);  
            i++; j--;  
        }  
    } while (i <= j);  
    if (j > 0) quickSort(a, j);  
    if (n > i) quickSort(a + i, n - i);  
}
```

Быстрая сортировка – параллельный вариант

```
void quickSort(float* a, const long n) {  
    long i = 0, j = n;  
    float pivot = a[n / 2]; // опорный элемент  
    do {  
        while (a[i] < pivot) i++;  
        while (a[j] > pivot) j--;  
        if (i <= j) {  
            std::swap(a[i], a[j]);  
            i++; j--;  
        }  
    } while (i <= j);  
}
```


Быстрая сортировка – параллельный вариант

```
if (n < 100) { // если размер массива меньше 100
    // сортировка выполняется в текущем потоке
    if (j > 0) quickSort(a, j);
    if (n > i) quickSort(a + i, n - i);
    return;
}
#pragma omp task shared(a)
    if (j > 0) quickSort(a, j);
#pragma omp task shared(a)
    if (n > i) quickSort(a + i, n - i);
#pragma omp taskwait
}
```

Запуск потоков

```
#pragma omp parallel
{
    #pragma omp single
    {
        quickSort(a, n - 1);
    }
}
```

РЕЗЮМЕ

- OpenMP – удобное современное высокоуровневое средство программирования систем с общей памятью
`int a = 0;`
- Сочетает минимальные изменения в коде программы с высокой производительностью



Директива **task**

#pragma omp task

```
{  
    // выполняемый код задания  
}
```

Вызывается из параллельного участка

#pragma omp taskwait

Ожидание созданных заданий



```
#include <omp.h>
#include <stdlib.h>
#include <iostream>

int t;

int fibs(int n) {
    if(n > 2)
        return fibs(n-1) + fibs(n-2);
    else
        return n;
}

int fibp(int n) {
    int i = 0, j = 0;
    if(n < t)
        return fibs(n);
#pragma omp task firstprivate(n) shared (i)
    i = fibp(n - 1);
#pragma omp task firstprivate(n) shared(j)
    j = fibp(n - 2);
#pragma omp taskwait
    return i + j;
}
```



```
int main(int argc, char* argv[]) {  
    int n = atoi(argv[1]);  
    if(argc == 2) {  
        std::cout << fibs(n);  
    } else {  
        t = atoi(argv[2]);  
#pragma omp parallel  
        {  
#pragma omp single  
            std::cout << fibp(n);  
        }  
    }  
  
    return 0;  
}
```



```
admin@irbis8:~/exp/lectures/openmp$ time ./taskex 45
1836311903
real    0m14.616s
user    0m14.593s
sys     0m0.000s
admin@irbis8:~/exp/lectures/openmp$ time ./taskex 45 25
1836311903
real    0m2.510s
user    0m20.749s
sys     0m0.000s
admin@irbis8:~/exp/lectures/openmp$ time ./taskex 45 20
1836311903
real    0m2.226s
user    0m23.101s
sys     0m0.052s
admin@irbis8:~/exp/lectures/openmp$ time ./taskex 45 15
1836311903
real    0m15.770s
user    2m30.545s
sys     0m4.448s
admin@irbis8:~/exp/lectures/openmp$
```