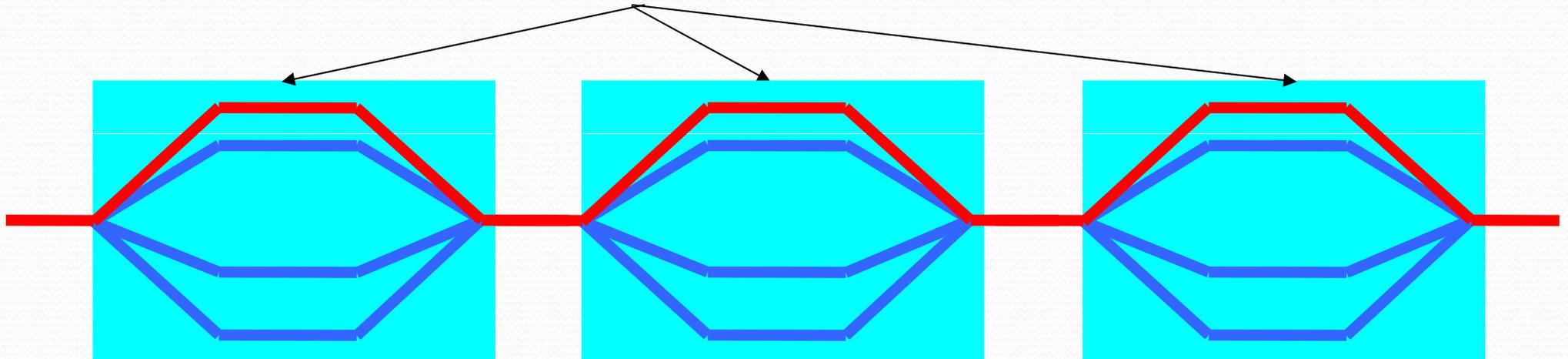




Суперкомпьютеры и параллельная обработка данных

Бахтин Владимир Александрович
*к.ф.-м.н., ведущий научный сотрудник
Института прикладной математики им М.В.Келдыша
РАН
кафедра системного программирования
факультет вычислительной математики и кибернетики
Московского университета им. М.В. Ломоносова*

Параллельная область (директива parallel)

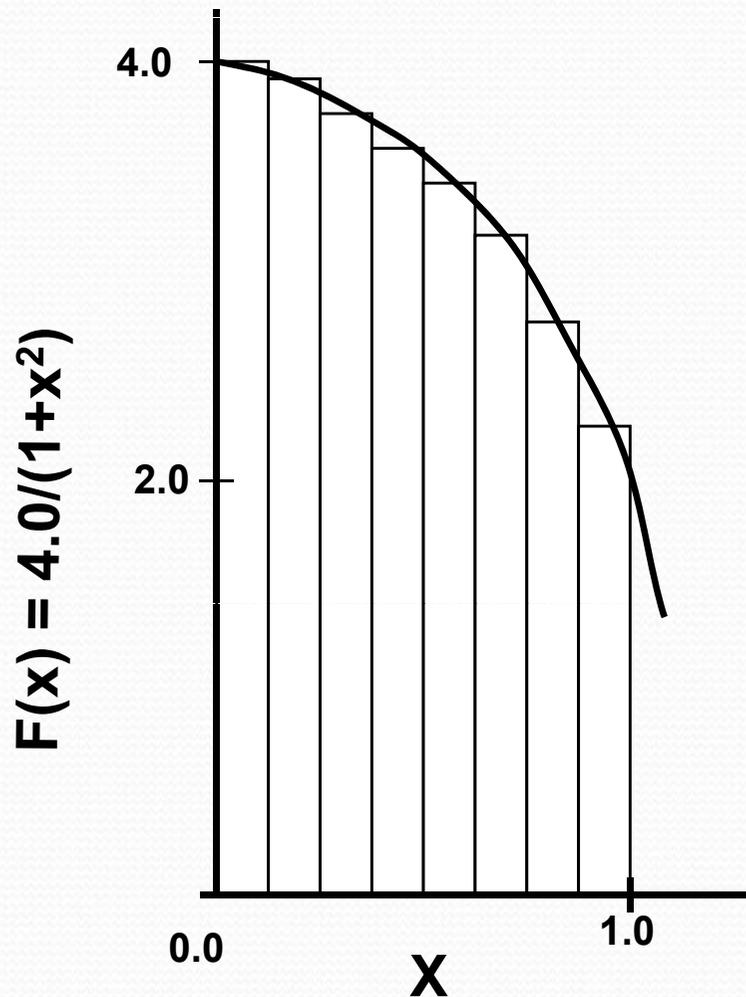


#pragma omp parallel [*клауза* [[,] *клауза*] ...]
структурный блок

где *клауза* одна из :

- **default(shared | none)**
- **private(list)**
- **firstprivate(list)**
- **shared(list)**
- **reduction(operator: list)**
- **if(scalar-expression)**
- **num_threads(integer-expression)**
- **copyin(list)**
- **proc_bind (master | close | spread)** //OpenMP 4.0

Вычисление числа π



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Мы можем
аппроксимировать интеграл
как сумму прямоугольников:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Где каждый прямоугольник
имеет ширину Δx и высоту
 $F(x_i)$ в середине интервала

Вычисление числа π . Последовательная программа

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i ++)
    {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π . Параллельная программа

```
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Конфликт доступа к данным

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

Thread0: $sum = sum + val$; && Thread1: $sum = sum + val$;

Время	Thread 0	Thread 1
1	LOAD R1,sum	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R3,sum
4	STORE R1,sum	LOAD R4,val
5		ADD R3,R4
6		STORE R3,sum

Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

Вычисление числа π . Параллельная программа

```
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt) {
            x = h * ((double)i - 0.5);
            #pragma omp critical
                sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π . Параллельная программа

```
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        double local_sum = 0.0;
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
#pragma omp critical
        sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP. Клауза reduction

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Редукционные операции

reduction(operator:list)

- ❑ Внутри параллельной области для каждой переменной из списка list создается копия этой переменной. Эта переменная инициализируется в соответствии с оператором operator (например, 0 для «+»).
- ❑ Для каждой нити компилятор заменяет в параллельной области обращения к редукционной переменной на обращения к созданной копии.
- ❑ По завершении выполнения параллельной области осуществляется объединение полученных результатов.

Оператор	Начальное значение
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0
max	Least number in reduction list item type
min	Largest number in reduction list item type

Клауза if

if(*scalar-expression*)

В зависимости от значения *scalar-expression* для выполнения структурного блока будет создана группа нитей или он будет выполняться одной нитью.

```
#include <omp.h>
int main()
{
    int n = 0;
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
    #pragma omp parallel if (n>10)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (int i = id + 1; i <= n; i=i+numt)
            func (i);
    }
    return 0;
}
```

Клауза `num_threads`

`num_threads(integer-expression)`

`integer-expression` задает максимально возможное число нитей, которые будут созданы для выполнения структурного блока

```
#include <omp.h>
int main()
{
    int n = 0;
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
    #pragma omp parallel num_threads(10)
    {
        int id = omp_get_thread_num ();
        func (n, id);
    }
    return 0;
}
```

Определение числа нитей в параллельной области

Число создаваемых нитей зависит от:

- клаузы `if`
- клаузы `num_threads`
- значений переменных, управляющих выполнением OpenMP-программы:
 - `dyn-var` (включение/отключение режима, в котором количество создаваемых нитей может изменяться динамически: **OMP_DYNAMIC, omp_set_dynamic()**)
 - `nthreads-var` (максимально возможное число нитей, создаваемых при входе в параллельную область: **OMP_NUM_THREADS, omp_set_num_threads()**)
 - `thread-limit-var` (максимально возможное число нитей, создаваемых для выполнения всей OpenMP-программы: **OMP_THREAD_LIMIT**)
 - `nest-var` (включение/отключение режима поддержки вложенного параллелизма: **OMP_NESTED, omp_set_nested()**)
 - `max-active-level-var` (максимально возможное количество вложенных параллельных областей: **OMP_MAX_ACTIVE_LEVELS, omp_set_max_active_levels()**)

Определение числа нитей в параллельной области

Пусть ***ThreadsBusy*** - количество OpenMP нитей, выполняемых в данный момент;

Пусть ***ActiveParRegions*** - количество активных параллельных областей;

if в директиве `parallel` существует клауза **if**

then присвоить переменной ***IfClauseValue*** значение выражения в клаузе **if**;

else *IfClauseValue* = true;

if в директиве `parallel` существует клауза **num_threads**

then присвоить переменной ***ThreadsRequested*** значение выражения в клаузе **num_threads**;

else *ThreadsRequested* = *nthreads-var*;

***ThreadsAvailable* = (*thread-limit-var* - *ThreadsBusy* + 1);**

Определение числа нитей в параллельной области

```
if (IfClauseValue == false)
then number of threads = 1;
else if (ActiveParRegions >= 1) and (nest-var == false)
then number of threads = 1;
else if (ActiveParRegions == max-active-levels-var)
then number of threads = 1;
else if (dyn-var == true) and (ThreadsRequested <= ThreadsAvailable)
then number of threads = [ 1 : ThreadsRequested ];
else if (dyn-var == true) and (ThreadsRequested > ThreadsAvailable)
then number of threads = [ 1 : ThreadsAvailable ];
else if (dyn-var == false) and (ThreadsRequested <= ThreadsAvailable)
then number of threads = ThreadsRequested;
else if (dyn-var == false) and (ThreadsRequested > ThreadsAvailable)
then number of threads зависит от реализации компилятора;
```

Определение числа нитей в параллельной области

```
#include <stdio.h>
#include <omp.h>
int main (void)
{
    omp_set_nested(1);
    omp_set_max_active_levels(8);
    omp_set_dynamic(0);
    omp_set_num_threads(2);
    #pragma omp parallel
    {
        omp_set_num_threads(3);
        #pragma omp parallel
        {
            ...
        }
    }
}
```

Клауза copyin

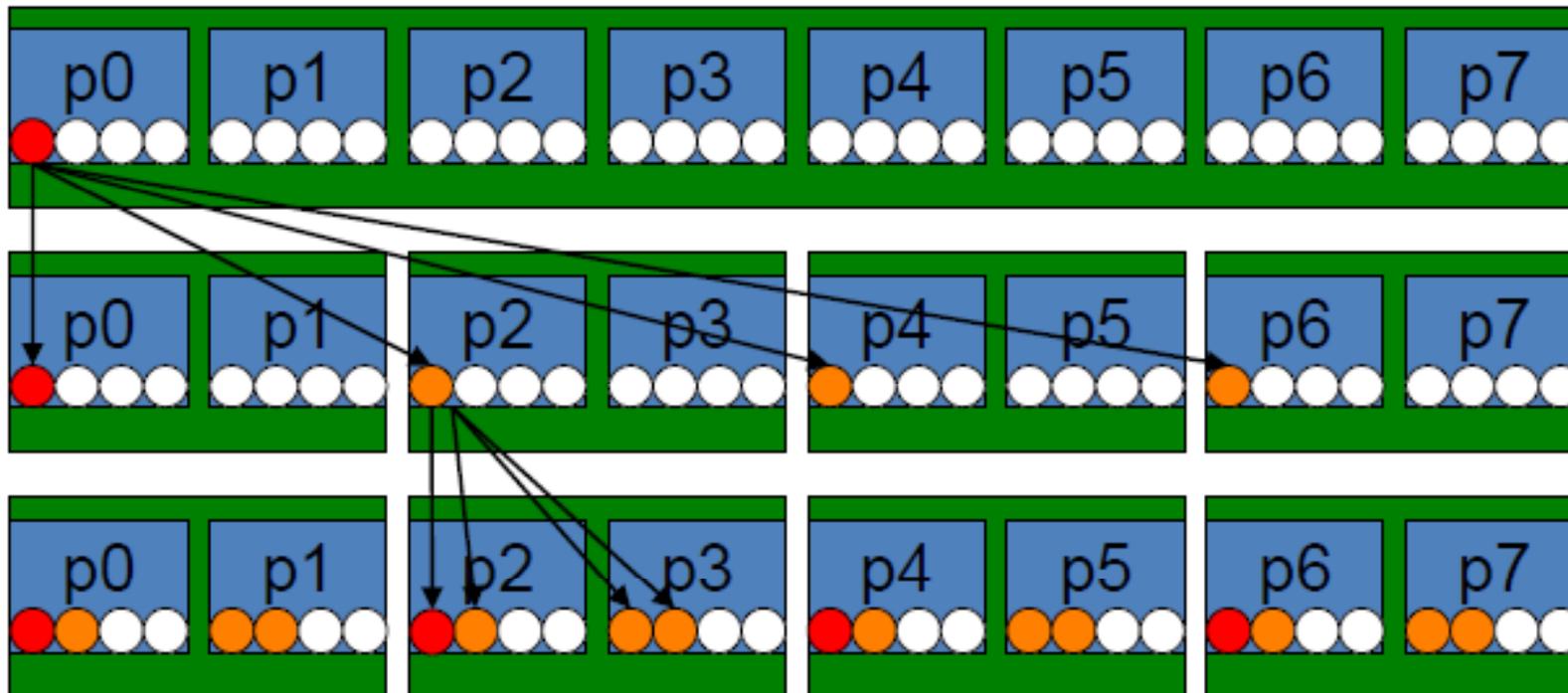
copyin(list)

Значение каждой threadprivate-переменной из списка list, устанавливается равным значению этой переменной в master-нити

```
float* work;
int size;
float val;
#pragma omp threadprivate(work,size,val)
void compute()
{
    work = (float*)malloc( sizeof(float)*size);
    for(int i = 0; i < size; ++i ) work[i] = val;
    ... // computation with work array elements
}
int main()
{
    printf("Enter the size of array and value\n");
    scanf("%d",&size);
    scanf("%f",&val);
    #pragma omp parallel copyin(val,size)
        compute();
}
```

Клауза `proc_bind` (OpenMP 4.0)

```
#pragma omp parallel proc_bind(spread) num_threads(4)  
{  
  #pragma omp parallel proc_bind(close) num_threads(4)  
  work();  
}
```



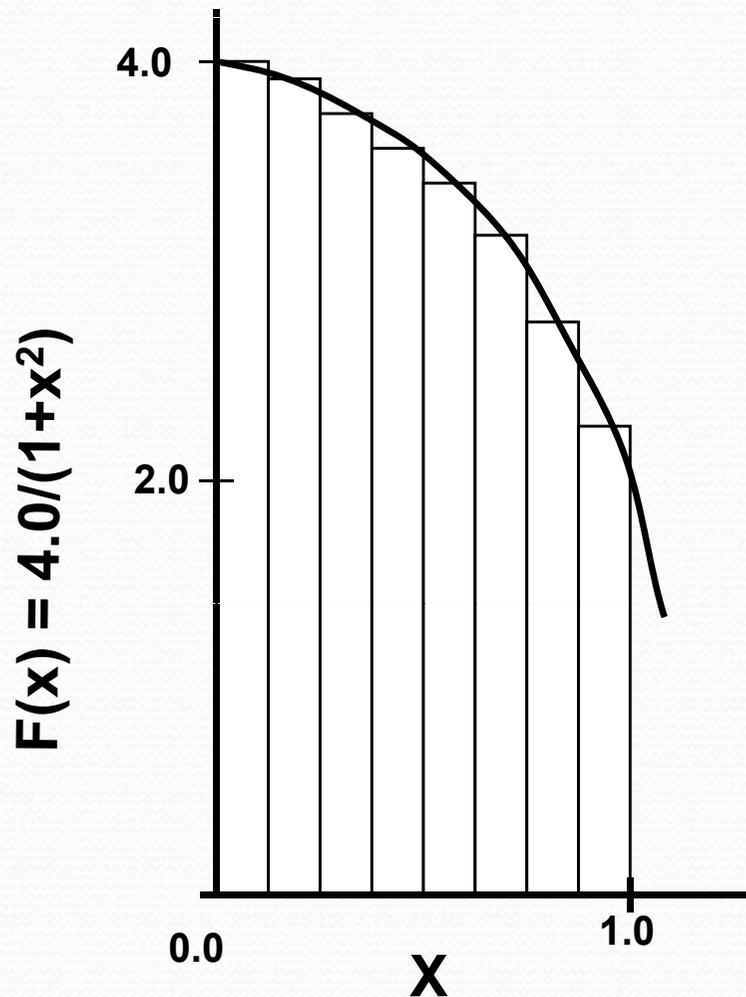
Содержание

- ❑ Тенденции развития современных вычислительных систем
- ❑ OpenMP – модель параллелизма по управлению
- ❑ Конструкции распределения работы
- ❑ Конструкции для синхронизации нитей
- ❑ Система поддержки выполнения OpenMP-программ
- ❑ Новые возможности OpenMP

Конструкции распределения работы

- Распределение витков циклов (директива for)
- Циклы с зависимостью по данным. Организация конвейерного выполнения для циклов с зависимостью по данным.
- Распределение нескольких структурных блоков между нитями (директива SECTION).
- Выполнение структурного блока одной нитью (директива single)
- Распределение операторов одного структурного блока между нитями (директива WORKSHARE)
- Понятие задачи

Вычисление числа π



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Мы можем аппроксимировать интеграл как сумму прямоугольников:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Где каждый прямоугольник имеет ширину Δx и высоту $F(x_i)$ в середине интервала

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n; i=i+numt)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        #pragma omp for schedule (static,1)
        for (i = 1; i <= n; i++)
            {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP

```
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        int iam = omp_get_thread_num();
        int numt = omp_get_num_threads();
        int start = iam * n / numt + 1;
        int end = (iam + 1) * n / numt;
        if (iam == numt-1) end = n;
        for (i = start; i <= end; i++)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        #pragma omp for schedule (static)
        for (i = 1; i <= n; i++)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Распределение витков цикла

#pragma omp for [*клауза*[[,*клауза*] ...]
for (*init-expr*; *test-expr*; *incr-expr*) *структурный блок*

где *клауза* одна из :

- **private**(*list*)
- **firstprivate**(*list*)
- **lastprivate**(*list*)
- **reduction**(*operator*: *list*)
- **schedule**(*kind*[, *chunk_size*])
- **collapse**(*n*)
- **ordered**(*n*)
- **nowait**

Распределение витков цикла

init-expr : var = loop-invariant-expr1

| *integer-type var = loop-invariant-expr1*

| *random-access-iterator-type var = loop-invariant-expr1*

| *pointer-type var = loop-invariant-expr1*

test-expr:

var relational-op loop-invariant-expr2

| *loop-invariant-expr2 relational-op var*

relational-op: <

| *<=*

| *>*

| *>=*

incr-expr: ++var

| *var++*

| *--var*

| *var --*

| *var += loop-invariant-integer- expr*

| *var -= loop-invariant-integer- expr*

| *var = var + loop-invariant-integer- expr*

| *var = loop-invariant-integer- expr + var*

| *var = var - loop-invariant-integer- expr*

var: signed or unsigned integer type

| *random access iterator type*

| *pointer type*

Parallel Random Access Iterator Loop (OpenMP 3.0)

```
#include <vector>
void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;
    #pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```

Использование указателей в цикле (OpenMP 3.0)

```
void pointer_example ()  
{  
    char a[N];  
    #pragma omp for default (none) shared (a,N)  
    for (char *p = a; p < (a+N); p++ )  
    {  
        use_char (p);  
    }  
}
```

for (char *p = a; p != (a+N); p++) - **ошибка**

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}  
void good_collapsing(int n)  
{  
    int i, j;  
    #pragma omp parallel default(shared)  
    {  
        #pragma omp for collapse (2)  
        for (i=0; i<n; i++) {  
            for (j=0; j < n; j++)  
                work(i, j);  
        }  
    }  
}
```

Клауза collapse:
collapse (*положительная целая константа*)

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}  
void error_collapsing(int n)  
{  
    int i, j;  
    #pragma omp parallel default(shared)  
    {  
        #pragma omp for collapse (2)  
        for (i=0; i<n; i++) {  
            work_with_i (i);           // Ошибка  
            for (j=0; j < n; j++)  
                work(i, j);  
        }  
    }  
}
```

Клауза collapse может быть использована только для распределения витков тесно-вложенных циклов.

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}  
void error_collapsing(int n)  
{  
    int i, j;  
    #pragma omp parallel default(shared)  
    {  
        #pragma omp for collapse (2)  
        for (i=0; i<n; i++) {  
            for (j=0; j < i; j++)      // Ошибка  
                work(i, j);  
        }  
    }  
}
```

Клауза collapse может быть использована только для распределения витков циклов с прямоугольным индексным пространством.

Использование редуционных операций

```
void reduction (float *x, int *y, int n)
{
    int i, b, c;
    float a, d;
    a = 0.0;
    b = 0;
    c = y[0];
    d = x[0];
    #pragma omp parallel for private(i) shared(x, y, n) \
        reduction(+:a) reduction(^:b) \
        reduction(min:c) reduction(max:d)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
        if (c > y[i]) c = y[i];
        d = fmaxf(d,x[i]);
    }
}
```

Реализация редуционных операций

```
#include <limits.h>
void reduction_by_hand (float *x, int *y, int n)
{
    int i, b, b_p, c, c_p;
    float a, a_p, d, d_p;
    a = 0.0f;
    b = 0;
    c = y[0];
    d = x[0];
    #pragma omp parallel shared(a, b, c, d, x, y, n) private(a_p, b_p, c_p, d_p)
    {
        a_p = 0.0f; b_p = 0; c_p = INT_MAX; d_p = -HUGE_VALF;
        #pragma omp for private(i) nowait
        for (i=0; i<n; i++) {
            a_p += x[i]; b_p ^= y[i]; if (c_p > y[i]) c_p = y[i]; d_p = fmaxf(d_p,x[i]);
        }
        #pragma omp critical
        {
            a += a_p; b ^= b_p; if (c > c_p ) c = c_p; d = fmaxf(d,d_p);
        }
    }
}
```

Редукционные операции, определяемые пользователем (OpenMP 4.0)

```
#pragma omp declare reduction (reduction-identifier : typename-list :  
combiner) [initializer(initializer-expr)]
```

- reduction-identifier** - название редукционной операции
- typename-list** – тип (типы)
- combiner** – выражение для объединения частичных результатов
- initializer** – начальное значение создаваемых частных переменных
- omp_out** refers to private copy that holds combined value
- omp_in** refers to the other private copy
- omp_priv** represents the private element
- omp_orig** represents the original variable

Использование редукционных операций, определяемых пользователем (OpenMP 4.0)

```
struct point
{
    int x;
    int y;
} points[N], minp = { MAX_INT, MAX_INT };

#pragma omp declare reduction (min : struct point : \
    omp_out.x = omp_in.x > omp_out.x ? omp_out.x : omp_in.x, \
    omp_out.y = omp_in.y > omp_out.y ? omp_out.y : omp_in.y ) \
    initializer ( omp_priv = { MAX_INT, MAX_INT })

#pragma omp parallel for reduction (min: minp)
for (int i = 0; i < N; i++)
{
    if (points[i].x < minp.x) minp.x = points[i].x;
    if (points[i].y < minp.y) minp.y = points[i].y;
}
```

Редукционные операции, определяемые пользователем (OpenMP 4.0)

```
#pragma omp declare reduction (merge : std::vector<int> :  
omp_out.insert (omp_out.end(), omp_in.begin(), omp_in.end()))
```

```
void schedule (std::vector<int> &v, std::vector<int> &filtered)  
{  
    #pragma omp parallel for reduction (merge: filtered)  
    for (std::vector<int>::iterator it = v.begin(); it < v.end(); it++)  
        if ( filter(*it) ) filtered.push_back(*it);  
}
```

Распределение витков цикла. Клауза schedule

Клауза schedule:

`schedule(алгоритм планирования[, число_итераций])`

Где алгоритм планирования один из:

- `schedule(static[, число_итераций])` - статическое планирование;
- `schedule(dynamic[, число_итераций])` - динамическое планирование;
- `schedule(guided[, число_итераций])` - управляемое планирование;
- `schedule(runtime)` - планирование в период выполнения;
- `schedule(auto)` - автоматическое планирование (OpenMP 3.0).

```
#pragma omp parallel for private(tmp) shared (a) schedule (runtime)
```

```
for (int i=0; i<N-2; i++)  
  for (int j = i+1; j< N-1; j++) {  
    tmp = a[i][j];  
    a[i][j]=a[j][i];  
    a[j][i]=tmp;  
  }
```

