

# **Введение в параллельные вычисления**

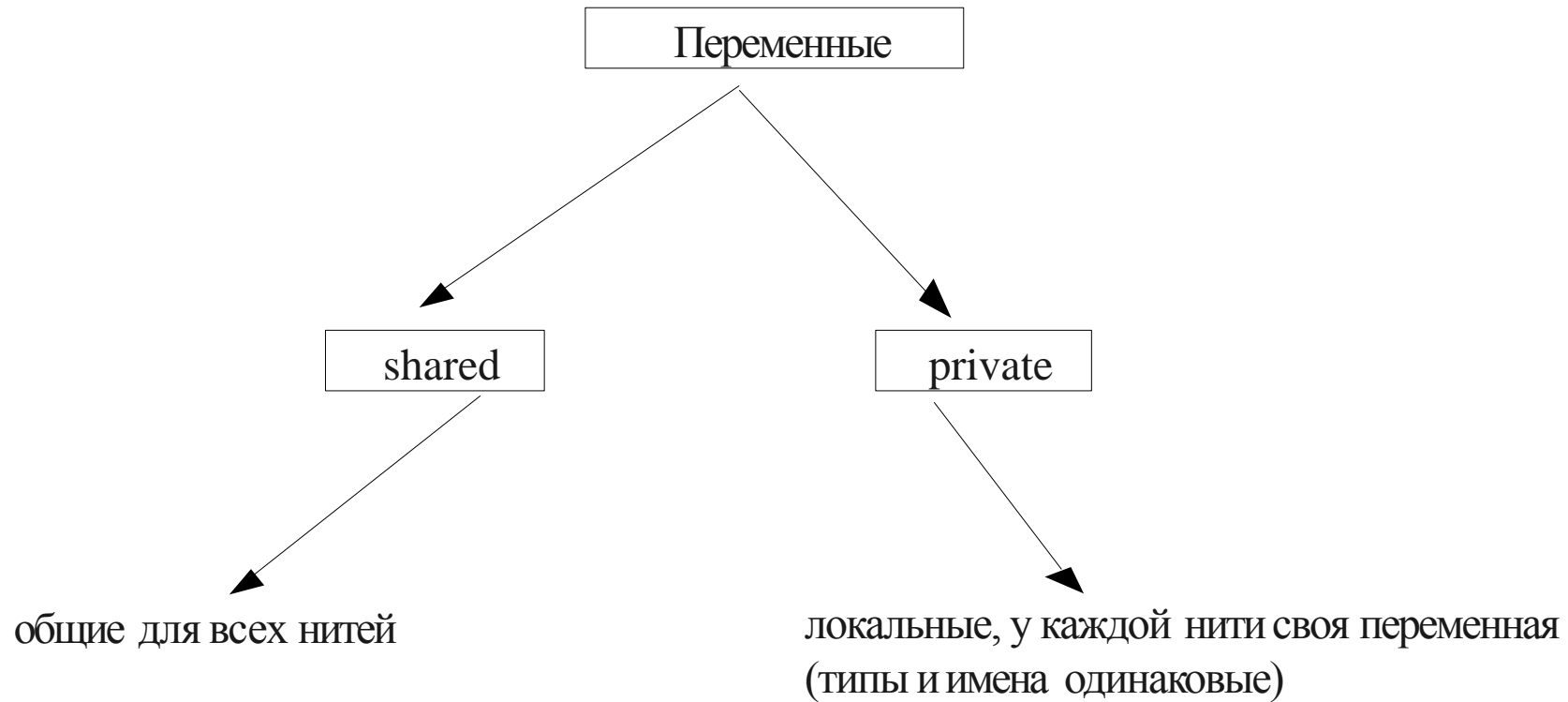
## **Лекция 4. OpenMP (часть 2)**

КС-40, КС-44  
РХТУ

Преподаватель  
Митричев Иван Игоревич, к.т.н.

2017

# Модель данных в OpenMP



По умолчанию,

- переменные, объявленные вне параллельной области – shared
- переменные, объявленные в параллельной области – private

# Пример - найти ошибку

**Найти ошибку, чтобы  
вывод мог быть в виде:**

```
Thread:0Thread:
Master:4
Thread:3
Thread:2
1
```

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int nthreads, tid;
    #pragma omp parallel
    {
        tid = omp_get_thread_num();
        cout << "Thread:" << tid << endl;
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            cout << "Master:" << nthreads << endl;
        }
    }
    return 0;
}
```

104\_01.cpp

# Опции директивы parallel

Опция	Пояснение
if (условие)	Порождать параллельные потоки, или выполнять только главный поток
num_threads (целое число)	Задаёт количество параллельных потоков
default (shared   none)	Класс переменных по умолчанию; none – класс всем переменным должен быть назначен явно
private (список)	Переменные, для которых порождается локальная копия в каждом потоке
firstprivate	То же, что и private + переменные инициализируются значениями, которые имеют аналогичные переменные в главном потоке
shared	Переменные, общие для всех потоков
copyin	Копировать значения переменных threadprivate из главного потока в переменные threadprivate дочерних
reduction (оператор: список)	По окончании параллельного блока применить оператор к переменным из списка

# Замер времени в OpenMP

<code>double omp_get_wtime()</code>	←	время в секундах с некоторого момента в прошлом
<code>double omp_get_wtick()</code>	←	разрешение таймера в секундах (минимальный интервал времени)

//пример

```
double start = omp_get_wtime();
```

```
//...
```

```
//...
```

```
double end = omp_get_wtime();
```


```
cout<<endstart<<endl;
```

**104\_03.cpp**

# Опция if

## if (условие)

*Существует следующий приоритет при определении числа потоков (от старшего правила к младшему)*

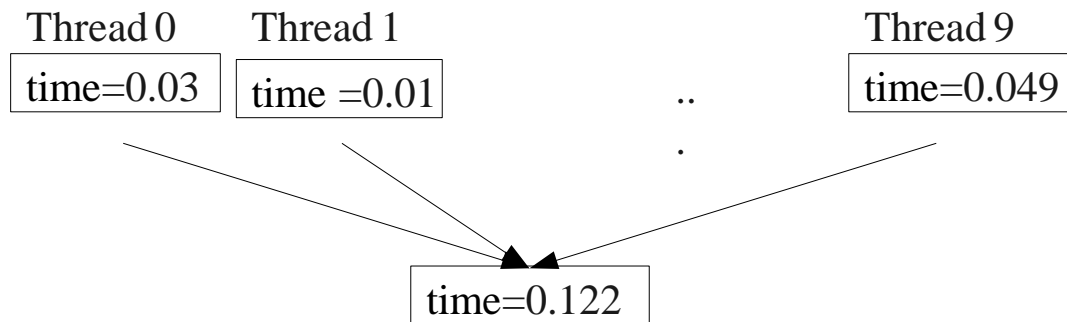
- 
- **Опция if**
  - Опция *num\_threads*
  - функция *omp\_set\_num\_threads*
  - переменная окружения *OMP\_NUM\_THREADS*
  - по умолчанию — количество процессоров/ядер на узле

104\_04.cpp

104\_05.cpp

## Пример работы опции reduction

```
int nthreads, tid;
double time=0;
#pragma omp parallel num_threads(10) reduction(+:time)
{
    cout<< "What do you think the time is? -> " <<time<<endl;
    time = omp_get_wtime();
    time=omp_get_wtime()-time;
    cout<< "wall time for thread " << tid << endl;
}
cout << "Total cpu time for all threads " << time << endl;
return 0;
```



Можно использовать операции: +, \*, , &, |, ^, ||, &&

104\_06.cpp – редукция в непустую переменную  
104\_07.cpp – редукция в переменную POD-типа

# Отличие private от firstprivate

Переменные не инициализируются

Переменные инициализируются

```
double time=999;  
#pragma omp parallel num_threads(3) firstprivate(time)  
{  
    cout<< "time variable " << time << endl;  
}  
return 0;
```

[l04\\_08.cpp](#)

[l04\\_09.cpp](#)



# Директива threadprivate

```
#include <iostream>
#include <omp.h>
using namespace std;
int n=999;

int main()
{
    omp_set_num_threads(2);
    #pragma omp threadprivate(n)

    #pragma omp parallel
    {
        cout<< "before n= " << n << endl;
        n=omp_get_thread_num();
        cout<< "after n= " << n << endl;
    }
    // значение от потока 0
    cout<<" MIDDLE n = "<<n<<endl;
    // значения сохраняются между секциями
    #pragma omp parallel
    {
        cout<< "n= " << n << endl;
    }
    return 0;
}
```

l04\_11.cpp

# Пример работы опции copyin

Copyin обновляет значение в соответствии со значением в последовательной секции (thread 0)

```
int n;  
#pragma omp threadprivate(n)  
int main(int argc, char *argv[])  
{  
    n=1;  
#pragma omp parallel copyin(n)  
    {  
        cout<< n <<endl;  
    }  
    return 0;  
}
```

**l04\_12.cpp – пример, когда copyin не нужен**

**l04\_13.cpp – не хватает copyin**

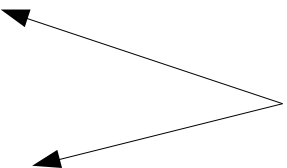
**l04\_14.cpp**

# Вложенные параллельные области

export OMP\_NESTED=true

void omp\_set\_nested( int nested )

Разрешить использование  
вложенных параллельных  
областей



```
    omp_set_nested(1);
```

```
    #pragma omp parallel num_threads(3) private (tid)
```

```
    {
```

```
        tid = omp_get_thread_num();
```

```
    //produces 15 threads
```

```
        #pragma omp parallel num_threads(5)
```

```
        {
```

```
    // critical section - one-by-one execution
```

```
        #pragma omp critical
```

```
            cout<<tid<<"=> new "<<omp_get_thread_num()<<endl;
```

```
        }
```

#pragma omp critical – критическая область (два потока не могут находиться в ней одновременно)

104\_15.cpp

# Директива single

(выполнить участок кода в параллельной секции один раз)

```
#pragma omp single опции
{
    // ...
}
```

## Пример:

```
int tid=777;
#pragma omp parallel num_threads(3) firstprivate (n)
{
    // все значения tid далее перезаписываются
    tid=omp_get_thread_num()+8;
    cout <<tid<<endl;
    #pragma omp single nowait
    // печатает 8 из потока 0 (мастера)
    cout<<"single:" << tid << endl;
}
```

nowait – отключение неявной барьерной синхронизации.

Тем не менее, все равно поток 0 выполняет в современных реализациях OpenMP код single. То есть single nowait аналогичен master, но медленнее [<https://stackoverflow.com/questions/18820471/what-is-the-benefit-of-prAGMA-omp-master-as-opposed-to-prAGMA-omp-single>]

l04\_16.cpp

# Использование опции copyprivate

```
#pragma omp parallel num_threads(3) private (n)
{
    n=omp_get_thread_num()+1;
    cout <<n<<endl;
    #pragma omp single copyprivate(n)
        n=100;
    cout <<n<<endl;
}
```

Значение n копируется во все приватные переменные

l04\_17.cpp

# Директива master

(выполнить главным потоком)

```
int n;  
#pragma omp parallel num_threads(3) private (n)  
{  
    n=omp_get_thread_num()+1;  
    cout <<n<<endl;  
    #pragma omp master  
        n=100;  
    cout <<n<<endl;  
}
```

Барьерная синхронизация не выполняется!

#pragma omp  
barrier



барьерная  
синхронизация

l04\_18.cpp

# Директива for

```
int A[10],B[10],C[10], i,n;
for (i=0;i<10;i++)
{
    A[i]=i; B[i]=2*i;C[i]=0;
}
#pragma omp parallel shared (A,B,C) private (i,n) num_threads(3)
{
    n = omp_get_thread_num();
    #pragma omp for
    for (i=0;i<10;i++)
    {
        C[i]=A[i]+B[i];
        cout<<i<<" by "<<n<<endl;
    }
}
```

[l04\\_19.cpp](#)

Если параллельная секция включает только цикл, использовать можно **parallel for**:

```
int A[10], i;
#pragma omp parallel for shared (A) private (i)
    for (i=0; i<10; i++) {
        cout<<i<<" by "<<omp_get_thread_num()<<endl;
        A[i]=5;    }
```

[l04\\_19b.cpp](#)

# Автоматическое распараллеливание циклов

## #pragma omp for опции

- private (список) – данные переменные будут свои у каждого потока
- firstprivate (список) – private + переменные инициализируются значением из главного потока
- lastprivate (список) – этим переменным присваивается значение от последней итерации цикла
- reduction (оператор: список) – применить операцию к списку переменных
- schedule (type[, chunk]) – опции распределения итераций цикла по нитям
- collapse (n) – с директивой связан не один цикл, а n циклов внутри параллельной области
- ordered – в цикле могут встречаться директивы ordered
- nowait – отменить неявную барьерную синхронизацию в конце цикла

Список переменных дается через запятую

Параллельный цикл:

- **должен иметь независимые итерации → удобно использовать в контексте параллелизма по данным**
- не должен иметь побочных выходов



# Параметры опции `schedule`

## распределение итераций цикла

**static** – блочноциклическое распределение; итерации распределяются блоками по `chunk` итераций на нить; если нити закончились, а итерации еще есть, то распределение продолжается с первой нити; если число `chunk` не указано, итерации делятся поровну между нитями;

**dynamic** – динамическое распределение; сначала каждая нить получает `chunk` итераций; затем следующую порцию по `chunk` итераций получает первая освободившаяся нить;

**guided** – динамическое распределение; изменяется размер порции: он уменьшается с начального значения до `chunk`, пропорционально количеству еще не распределенных итераций, деленному на количество нитей;

**auto** – способ распределения итераций определяется компилятором и /или средой выполнения;

**runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды `OMP_SCHEDULE`

# Пример работы опции schedule

```
int A[10],B[10],C[10], i,n;
for (i=0;i<10;i++)
{
    A[i]=i; B[i]=2*i;C[i]=11110;
}
#pragma omp parallel shared (A,B,C) private (i,n) num_threads(3)
{
    n = omp_get_thread_num();
    // on my PC - 3 is a chunk size by default
    #pragma omp for schedule (static)
    for (i=0;i<10;i++)
    {
        C[i]=A[i]+B[i];
        #pragma omp critical
        cout<<i<<" by "<<n<<endl;
    }
}
```

[l04\\_20.cpp](#)

...

[l04\\_24.cpp](#)


# Распределение итераций цикла по нитям

i	static	static,1	static,2	dynamic	dynamic,2	guided	guided,2
0	0	0	0	1	0	0	0
1	0	1	0	2	0	0	0
2	0	2	1	2	1	0	0
3	0	3	1	3	1	0	0
4	1	4	2	0	2	1	1
5	1	0	2	4	2	1	1
6	1	1	3	4	3	1	1
7	1	2	3	3	3	1	1
8	2	3	4	1	4	2	2
9	2	4	4	2	4	2	2
10	2	0	0	0	1	2	2
11	2	1	0	1	1	3	3
12	3	2	1	4	2	3	3
13	3	3	1	3	2	4	4
14	3	4	2	2	0	4	4
15	3	0	2	4	0	3	3
16	4	1	3	0	3	3	3
17	4	2	3	2	3	4	4
18	4	3	4	2	4	2	4
19	4	4	4	3	4	1	4

# Директива ordered

— ВЫПОЛНИТЬ часть цикла упорядоченно

```
int A[20],B[20],C[20], i,n;
for (i=0;i<20;i++)
{
    A[i]=i; B[i]=2*i;C[i]=11110;
}
#pragma omp parallel shared (A,B,C) private (i,n) num_threads(2)
{
    n = omp_get_thread_num();
    // on my PC - 10 initial chunk size
    #pragma omp for schedule (guided) ordered
    for (i=0;i<20;i++)
    {
        C[i]=A[i]+B[i];
        // ordered execution
        #pragma omp ordered
        cout<<i<<" by "<<n<<endl;
    }
}
```



**l04\_25.cpp**

Итерации упорядочены по возрастанию

# Управление режимом распараллеливания цикла

`export OMP_SCHEDULE="dynamic,1"` \_\_\_\_\_ при помощи переменной окружения

`void omp_set_schedule (omp_sched_t type, int chunk);`

`void omp_get_schedule (omp_sched_t * type, int * chunk);` при помощи функций OpenMP

```
typedef enum omp_sched_t{ omp_sched_static =  
    1,  
    omp_sched_dynamic = 2,  
    omp_sched_guided = 3,  
    omp_sched_auto = 4  
}omp_sched_t;
```

# Директивы sections и section — свой код для КАЖДОГО ИЗ ПОТОКОВ

```
srand(time(NULL));
int n;
#pragma omp parallel num_threads(3) private (n)
{
    n = omp_get_thread_num();
    #pragma omp sections
    {
        #pragma omp section
        {cout<<"First thread:"<<n<<endl; }
        #pragma omp section
        {
            sleep(2);
            cout<<"Second thread:"<<n<<endl;
        }
        #pragma omp section
        {
            sleep(rand()%5+3);
            cout<<"Third thread:"<<n<<endl;
        }
    }
    cout<<"Parallel area:"<<n<<endl;
}
```

Параллельные  
секции



**104\_26.cpp**

# Директивы sections + lastprivate

```
srand(time(NULL));
```

```
int n;
```

```
#pragma omp parallel num_threads(3)
```

```
{
```

```
    #pragma omp sections lastprivate (n)
```

```
    {
```

```
        #pragma omp section
```

```
        {
```

```
            n=1;
```

```
            cout<<"First thread:"<<n<<endl;
```

```
        }
```

```
        #pragma omp section
```

```
        {
```

```
            n=2;
```

```
            cout<<"Second thread:"<<n<<endl;
```

```
        }
```

```
        #pragma omp section
```

```
        {
```

```
            n=3;
```

```
            cout<<"Third thread:"<<n<<endl;
```

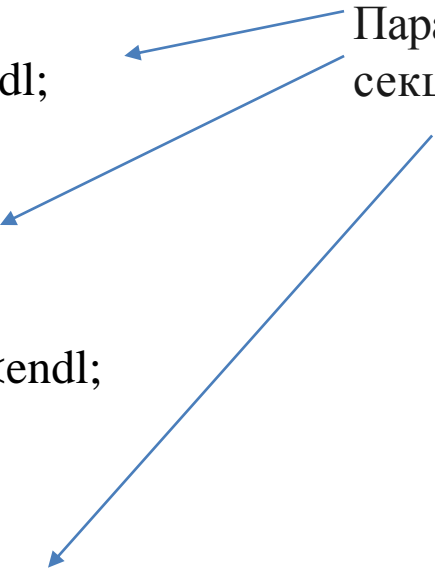
```
        }
```

```
    }
```

```
    cout<<"Parallel area:"<<n<<endl;
```

```
}
```

Параллельные  
секции



**l04\_27.cpp**

# Директивы for + lastprivate

```
int i;  
#pragma omp parallel for num_threads(10) private(i) lastprivate(n)  
    for (i=0; i<10; i++)  
        n=i;
```

[104\\_28.cpp](#)

// Можно гарантировать, что значение n после выхода из параллельной секции всегда равно 9

В примере 104\_28b – вывод 10, поскольку lastprivate объявлен сам счетчик i, а он инкрементируется (++)

[104\\_28b.cpp](#)



# Опции для директивы sections

- private (список) — локальные переменные
- firstprivate (список) — локальные переменные + инициализация значением из нитимастера
- lastprivate (список) — в итоге получают значение из последней секции
- reduction (оператор: список) — сбор результата
- nowait — отмена неявной барьерной синхронизации