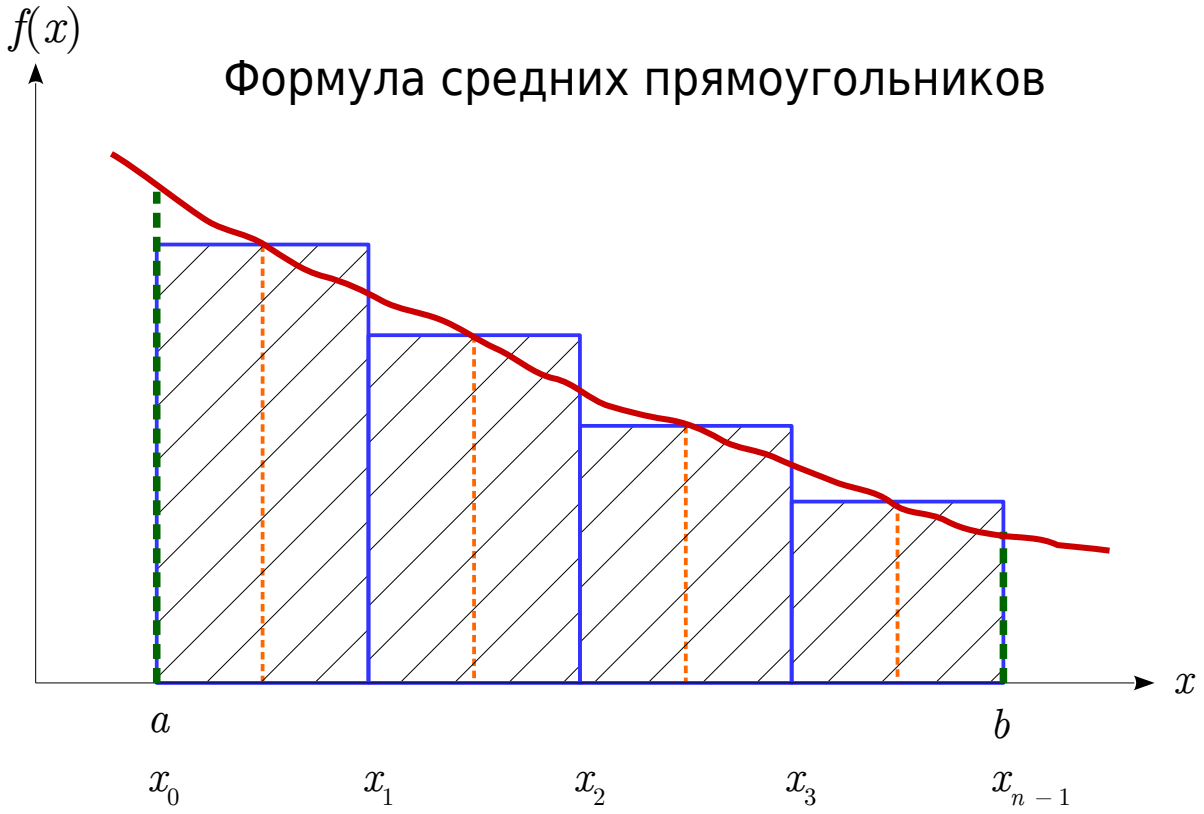
Семинар 2
Стандарт OpenMP (часть 2)

Михаил Курносов

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

Численное интегрирование (метод прямоугольников)




```
const double a = -4.0;          /* [a, b]  */
const double b = 4.0;
const int nsteps = 40000000;    /* n */

double func(double x)
{
    return exp(-x * x);
}

double integrate(double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    for (int i = 0; i < n; i++)
        sum += func(a + h * (i + 0.5));

    sum *= h;
    return sum;
}
```

$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{n} f\left(x_{i-1}+\frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i-\frac{h}{2}\right), \qquad h=\frac{b-a}{n}$$

Распараллеливание

```
const double a = -4.0;
const double b = 4.0;
const int nsteps = 40000000;

double func(double x)
{
    return exp(-x * x);
}

double integrate(double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    for (int i = 0; i < n; i++)
        sum += func(a + h * (i + 0.5));

    sum *= h;
    return sum;
}
```

- Итерации цикла for распределим между потоками
- Каждый поток вычисляет часть суммы (площади)

- Варианты распределения итераций (точек) между p потоками:

1) Разбиение на p смежных непрерывных частей

Thread 0 Thread 1 Thread 2

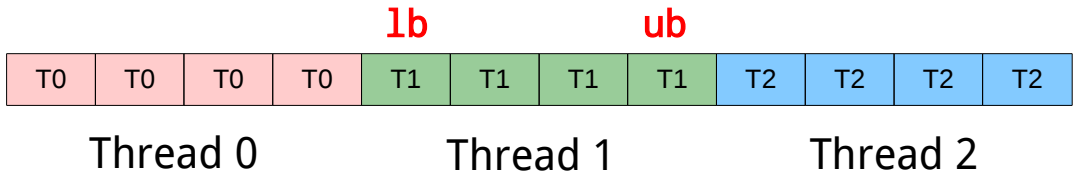
2) Циклическое распределение итераций по потокам

Thread 0, Thread1, Thread 2, Thread 0, Thread 1, Thread 2, ...

Параллельная версия

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Разбиение пространства итераций на смежные непрерывные части



Параллельная версия (продолжение)

```
const double PI = 3.14159265358979323846;

const double a = -4.0;
const double b = 4.0;
const int nsteps = 40000000;

double run_serial()
{
    double t = wtime();
    double res = integrate(func, a, b, nsteps);
    t = wtime() - t;
    printf("Result (serial): %.12f; error %.12f\n", res, fabs(res - sqrt(PI)));
    return t;
}

double run_parallel()
{
    double t = wtime();
    double res = integrate_omp(func, a, b, nsteps);
    t = wtime() - t;
    printf("Result (parallel): %.12f; error %.12f\n", res, fabs(res - sqrt(PI)));
    return t;
}

int main(int argc, char **argv)
{
    printf("Integration f(x) on [%.12f, %.12f], nsteps = %d\n", a, b, nsteps);
    double tserial = run_serial();
    double tparallel = run_parallel();

    printf("Execution time (serial): %.6f\n", tserial);
    printf("Execution time (parallel): %.6f\n", tparallel);
    printf("Speedup: %.2f\n", tserial / tparallel);
    return 0;
}
```

Компиляция и запуск

```
$ make
gcc -std=c99 -Wall -O2 -fopenmp -c integrate.c -o integrate.o
gcc -o integrate integrate.o -lm -fopenmp


$ export OMP_NUM_THREADS=4
$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.896639185158; error 0.875814665748

$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.794717040479; error 0.977736810426

$ ./integrate
Result (serial): 1.772453823579; error 0.000000027326
Result (parallel): 0.771561715425; error 1.000892135481
```

На каждом запуске разные результаты!

В чем причина?

Состояние гонки данных (race condition, data race)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Ошибка - race condition

- Несколько потоков одновременно читают и записывают переменную sum
- Значение sum зависит от порядка выполнения потоков

Состояние гонки данных (race condition, data race)

Два потока одновременно увеличивают значение переменной x на 1

(начальное значение x = 0)

```
Thread 0
x = x + 1;
```

```
Thread 1
x = x + 1;
```

Ожидаемый (идеальный) порядок выполнения потоков: первый поток увеличил x, затем второй

Time	Thread 0	Thread 1	x
0	Значение x = 0 загружается в регистр R процессора		0
1	Значение 0 в регистре R увеличивается на 1		0
2	Значение 1 из регистра R записывается в x		1
3		Значение x = 1 загружается в регистр R процессора	1
4		Значение 1 в регистре R увеличивается на 1	1
5		Значение 2 из регистра R записывается в x	**2**

Ошибки нет

Состояние гонки данных (race condition, data race)

Два потока одновременно увеличивают значение переменной x на 1
(начальное значение x = 0)

```
Thread 0
x = x + 1;
```

```
Thread 1
x = x + 1;
```

Реальный порядок выполнения потоков (недетерминированный)
(потоки могут выполняться в любой последовательности, приостанавливаться и запускаться)

Time	Thread 0	Thread 1	x
0	Значение x = 0 загружается в регистр R процессора		0
1	Значение 0 в регистре R увеличивается на 1	Значение x = 0 загружается в регистр R процессора	0
2	Значение 1 из регистра R записывается в x	Значение 1 в регистре R увеличивается на 1	1
3		Значение 1 из регистра R записывается в x	**1**

Ошибка - data race
(ожидали 2)

Устранение гонки данных

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++)
            sum += func(a + h * (i + 0.5));
    }
    sum *= h;

    return sum;
}
```

Надо сделать так,
чтобы увеличение переменной sum
в любой момент времени
выполнялось только одним потоком

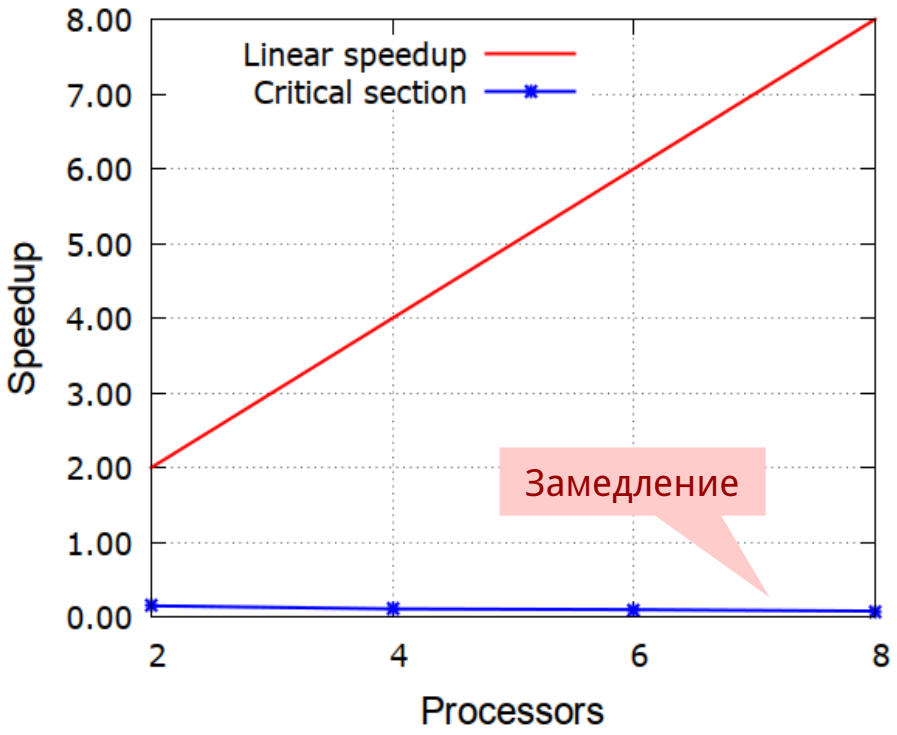
Критическая секция (critical section)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            double f = func(a + h * (i + 0.5));
            #pragma omp critical
            {
            sum += f;
            }
        }
    }
    sum *= h;
    return sum;
}
```

Критическая секция (critical section) — последовательность инструкций, в любой момент времени выполняемая только одним потоком

Критическая секция будет выполнена всеми потоками, но последовательно (один за другим)

Масштабируемость с #pragma omp critical

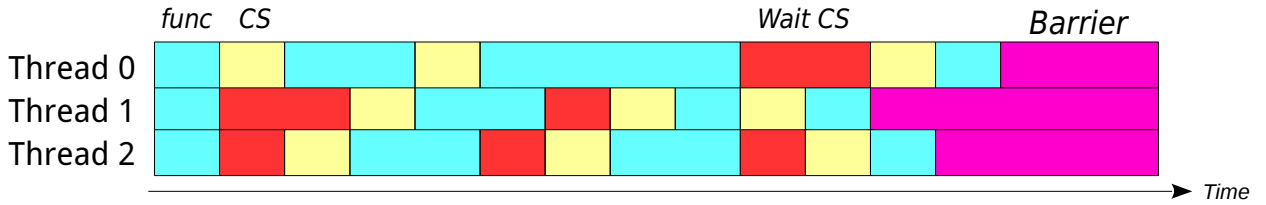



```
#pragma omp parallel
{
    /* ... */
    for (int i = lb; i <= ub; i++) {
        double f = func(a + h * (i + 0.5));
        #pragma omp critical
        {
        sum += f;
        }
    }
}
```

- Потоки ожидают освобождения критической секции другим потоком
- Код стал последовательным + накладные расходы на потоки

Вычислительный узел кластера Oak

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp



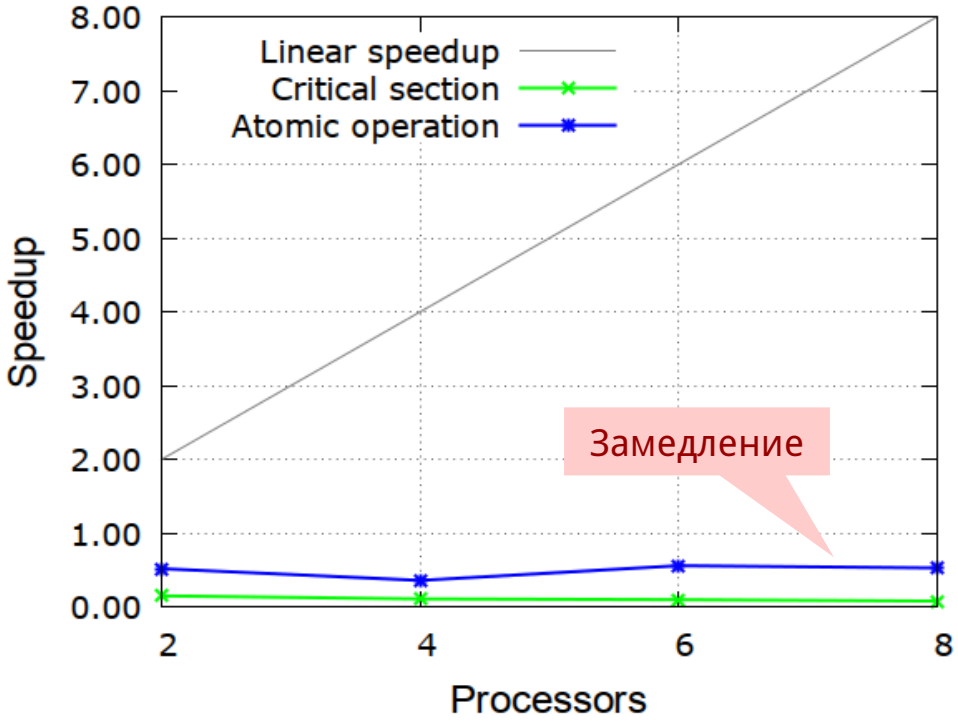
Атомарные операции (atomic operations)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            double f = func(a + h * (i + 0.5));
            #pragma omp atomic
            sum += f;
        }
    }
    sum *= h;
    return sum;
}
```

Атомарная операция (atomic operation) —
это инструкция процессора, в процессе выполнения
которой операнд в памяти блокируются
для других потоков

Операции: x++, x = x + y, x = x - y, x = x * y, x = x / y, ...

Масштабируемость с #pragma omp atomic




```
#pragma omp parallel
{

    for (int i = lb; i <= ub; i++) {
        double f = func(a + h * (i + 0.5));
        #pragma omp atomic
        sum += f;
    }
}
```

- Теперь потоки ожидают освобождения переменной sum (аппаратная CS)
- Каждый поток выполняет ub - lb + 1 захватов переменной sum

Вычислительный узел кластера Oak

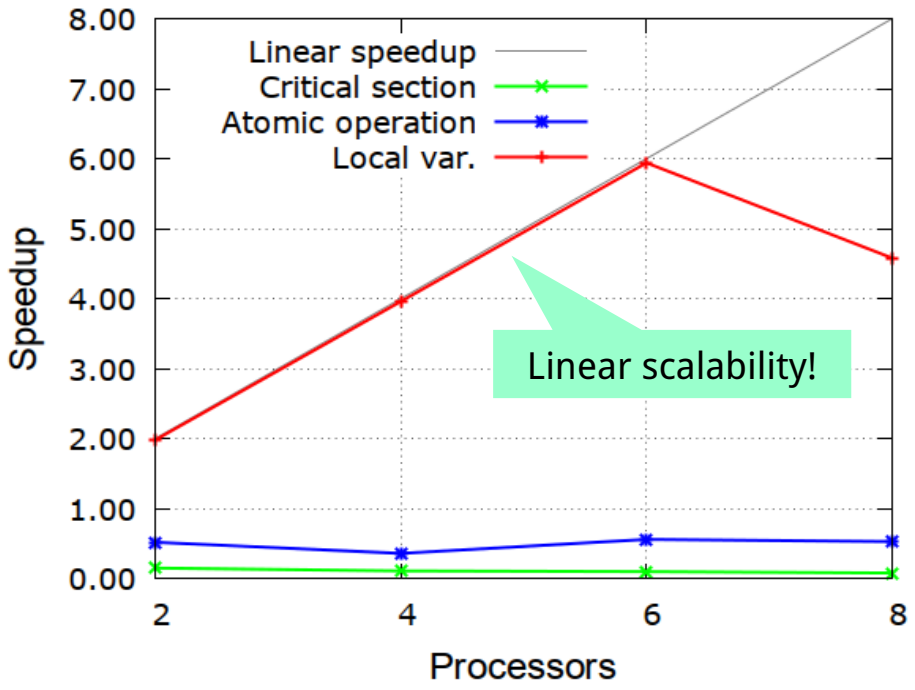
- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp

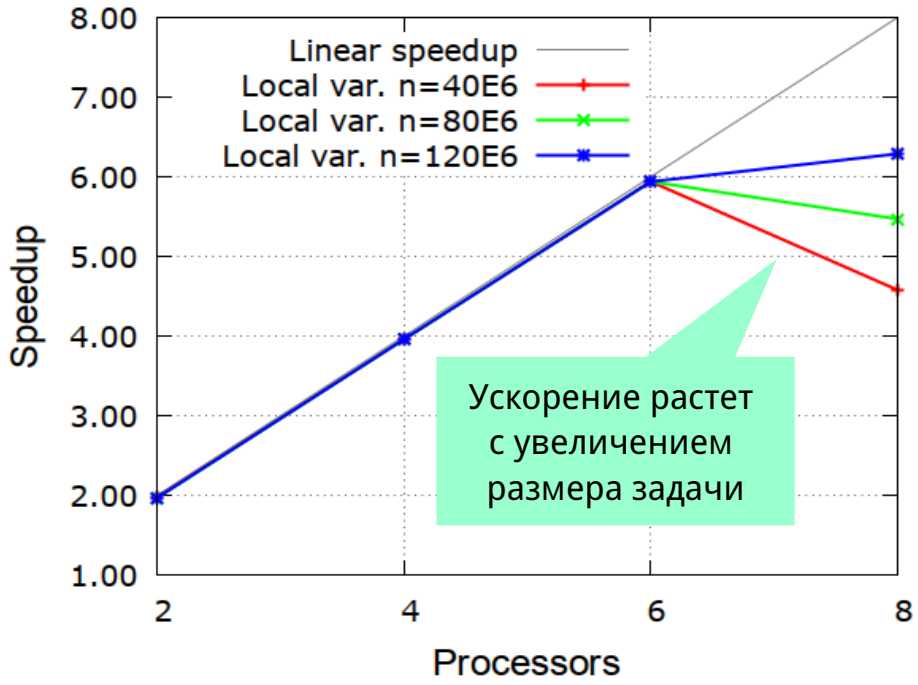
#pragma omp atomic + локальная переменная

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = n / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (n - 1) : (lb + items_per_thread - 1);
        double sumloc = 0.0;

        for (int i = lb; i <= ub; i++)
            sumloc += func(a + h * (i + 0.5));

        #pragma omp atomic
        sum += sumloc;
    }
    sum *= h;
    return sum;
}
```

- Каждый поток накапливает сумму в своей локальной переменной sumloc
- Затем атомарной операцией вычисляется итоговая сумма (всего nthreads захватов переменной sum)

#pragma omp atomic + локальная переменная





Вычислительный узел кластера Oak

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -O2 -fopenmp

Задание

Реализовать параллельную версию программы численного интегрирования методом прямоугольников:

- написать код функции integrate_omp с использованием #pragma omp atomic и локальной переменной (слайд 15)
- оценить ускорение программы на 2, 4, 6 и 8 потоках при числе точек интегрирования nsteps = 40 000 000 и 80 000 000

Шаблон программы находится в каталоге _task