Лекция 6
Многопоточное программирование
Стандарт OpenMP

Курносов Михаил Георгиевич

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

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Весенний семестр, 2020

Показатели эффективности параллельных алгоритмов и программ

Разработка параллельного алгоритма

- **Поиск параллелизма в известном последовательном алгоритме, его модификация или создание нового алгоритма:** определения уровня распараллеливания – уровень инструкций (мелкозернистый параллелизм, fine grained), потоков/процессов (крупнозернистый параллелизм, coarse grained)
- Выбор класса целевой ВС: с общей или распределенной памятью
- Разработка алгоритма в терминах одной из моделей программирования целевой ВС:
  - Системы с общей памятью (SMP/NUMA): fork/join model, CSP, Actor model, передача сообщений
  - Системы с распределенной памятью (кластеры, MPP): явная передача сообщений (message passing: односторонние/двусторонние/коллективные обмены), BSP – Bulk synchronous parallel, MapReduce
- **Параллельная версия самого эффективного последовательного алгоритма решения задачи необязательно будет самой эффективной параллельной реализацией**

Реализация параллельного алгоритма (программы)

- Выбор инструментальных средств (MPI, OpenSHMEM; OpenMP, POSIX Threads, Cilk)
- Распределение подзадач между процессорами (task mapping, load balancing)
- Организация взаимодействия подзадач (message passing, shared data structures)
- Учет архитектуры целевой вычислительной системы
- Запуск, измерение и анализ показателей эффективности параллельной программы
- Оптимизация программы

Показатели эффективности параллельных алгоритмов

- Коэффициент ускорения (Speedup)
- Коэффициент эффективности (Efficiency)
- Коэффициент накладных расходов
- Показатель равномерности загруженности параллельных ветвей (процессов, потоков)

Коэффициент ускорения (Speedup)

- Введем обозначения:
  - $\boldsymbol{T(n)}$ – время выполнения последовательной программы (sequential program)
  - $\boldsymbol{T_p(n)}$ – время выполнения параллельной программы (parallel program) на *p* процессорах
- **Коэффициент $S_p(n)$ ускорения** параллельной программ (Speedup):

$$\boldsymbol{S_p(n) = \frac{T(n)}{T_p(n)}}$$

- Коэффициент ускорения $S_p(n)$ показывает во сколько раз параллельная программа выполняется на *p* процессорах быстрее последовательной программы при обработке одних и тех же входных данных размера *n*
- Как правило

$$S_p(n) \leq p$$

Коэффициент ускорения (Speedup)

- Введем обозначения:
  - $T(n)$ – время выполнения последовательной программы (sequential program)
  - $T_p(n)$ – время выполнения параллельной программы (parallel program) на p процессорах
- **Коэффициент $S_p(n)$ ускорения** параллельной программ (Speedup):

$$S_p(n) = \frac{T(n)}{T_p(n)}$$

- **Цель распараллеливания – достичь линейного ускорения на максимально большом числе процессоров**

$$S_p(n) \approx p \quad \text{или} \quad S_p(n) = \Omega(p) \quad \text{при} \quad p \to \infty$$

Коэффициент ускорения (Speedup)

- **Какое время брать за время выполнения последовательной программы?**
  - Время лучшего известного алгоритма (в смысле вычислительной сложности)?
  - Время лучшего теоретически возможного алгоритма?

- **Что считать временем выполнения $T_p(n)$ параллельной программы?**
  - Среднее время выполнения потоков программы?
  - Время выполнения потока, завершившего работу первым?
  - Время выполнения потока, завершившего работу последним?

Коэффициент ускорения (Speedup)

- **Какое время брать за время выполнения последовательной программы?**
  - **Время лучшего известного алгоритма или время алгоритма, который подвергается распараллеливанию**

- **Что считать временем выполнения $T_p(n)$ параллельной программы?**
  - **Время выполнения потока, завершившего работу последним**

Коэффициент относительного ускорения (Rel. speedup)

- **Коэффициент относительного ускорения** (Relative speedup) – отношения времени выполнения параллельной программы на *k* процессорах к времени её выполнения на *p* процессорах (*k* < *p*)

$$S_{Relative}(k, p, n) = \frac{T_k(n)}{T_p(n)}$$

- Коэффициент эффективности (Efficiency) параллельной программы

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T(n)}{pT_p(n)} \in [0, 1]$$

- Коэффициент накладных расходов (Overhead)

$$\varepsilon(p, n) = \frac{T_{Sync}(p, n)}{T_{Comp}(p, n)} = \frac{T_{Total}(p, n) - T_{Comp}(p, n)}{T_{Comp}(p, n)}$$

- $T_{Sync}(p, n)$ – время создания, синхронизации и взаимодействия *p* потоков
- $T_{Comp}(p, n)$ – время вычислений в каждом из *p* потоков

Виды масштабируемости программ

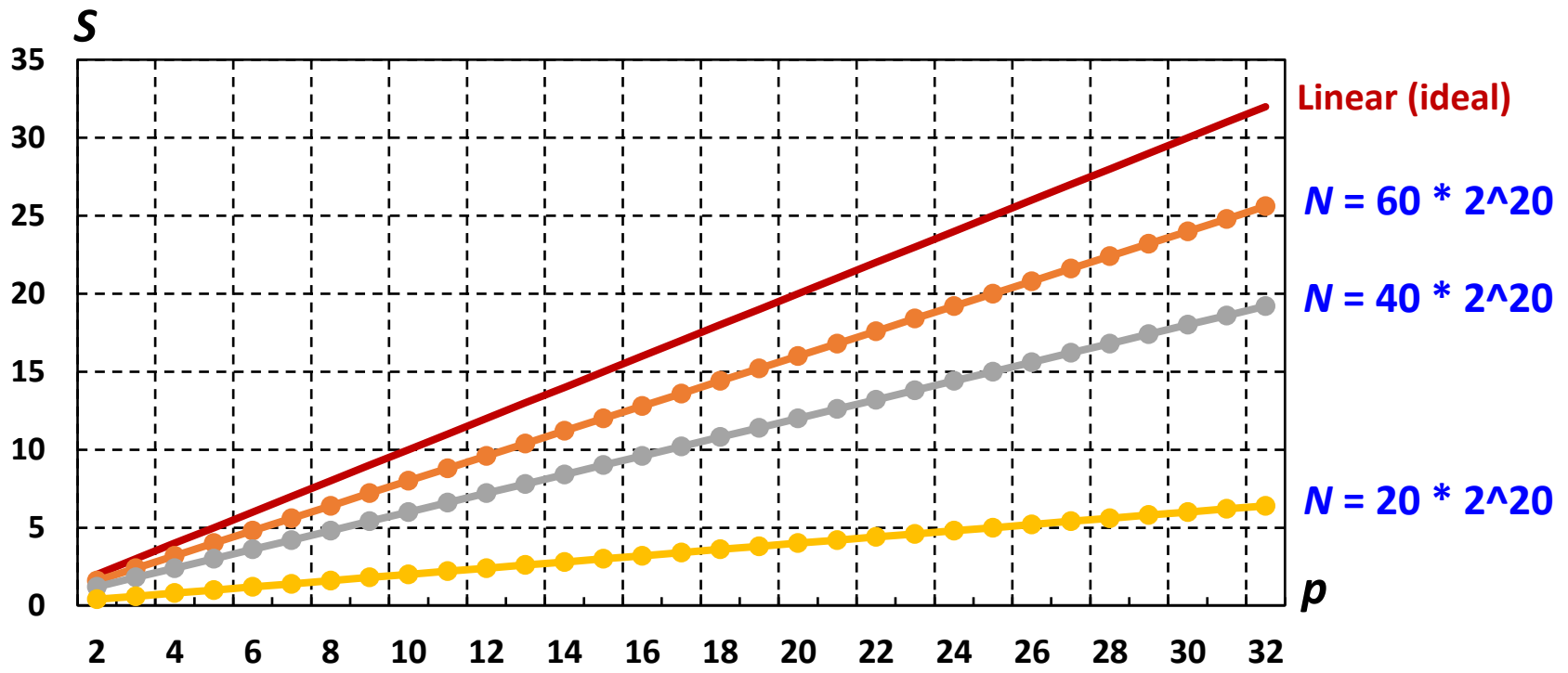
- **Масштабируемость параллельной программы** (scalability) – характеристика программы, показывающая как изменяются ее показатели производительности при варьировании числа параллельных процессов на конкретной ВС

- **Строгая/сильная масштабируемость** (strong scaling) – зависимость коэффициента ускорения от числа p процессов **при фиксированном размере *n*** входных данных (n = const)
  - Показывает как растут накладные расходы с увеличением p
  - Цель – минимизировать время решения задачи фиксированного размера

- **Слабая масштабируемость** (weak scaling) – зависимость коэффициента ускорения параллельной программы от числа процессов **при фиксированном размере входных данных на один процессор (**n / p = const**)**
  - Цель – решить задачу наибольшего размера на ВС

- Параллельная программа (алгоритм) коэффициент ускорения, которой линейной растет с увеличением p называется линейно масштабируемой или просто ***масштабируемой*** (scalable)

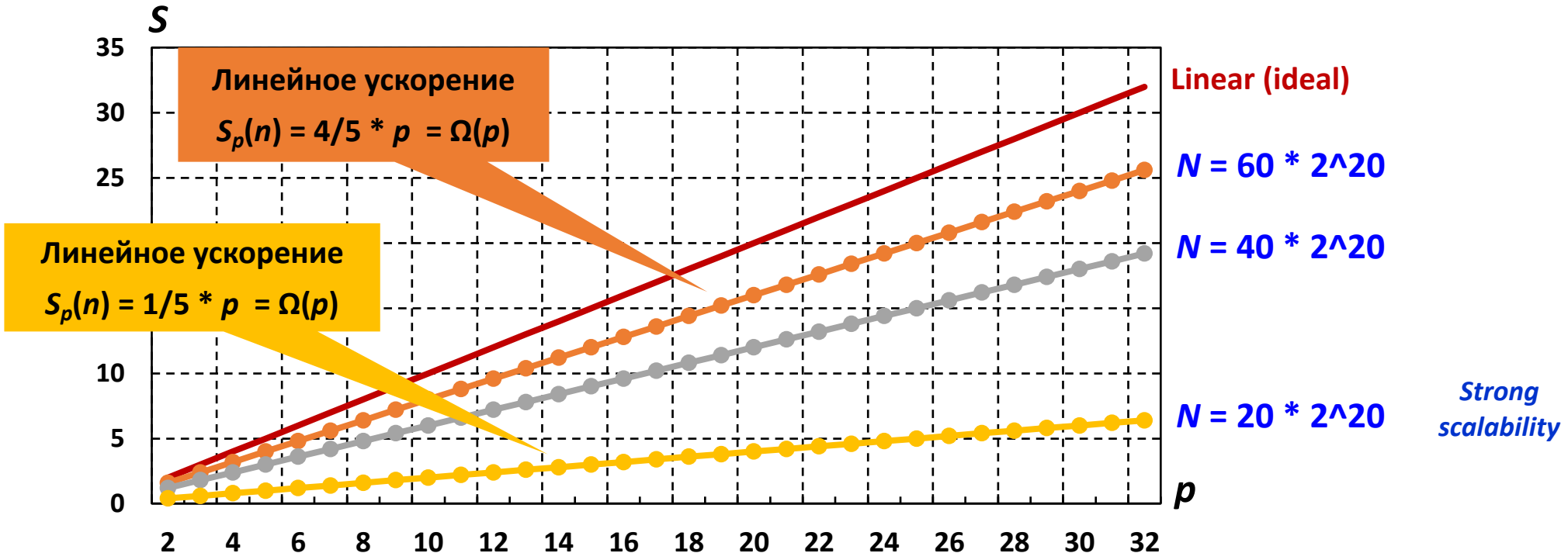
Коэффициент ускорения (Speedup)



Зависимость коэффициента ускорения *S*
параллельного алгоритма X от количества *p* процессоров

- Ускорение программы может расти с увеличением размера входных данных
- Время вычислений превосходит накладные расходы на взаимодействия потоков (управление потоками, синхронизацию, обмен сообщениями, …)

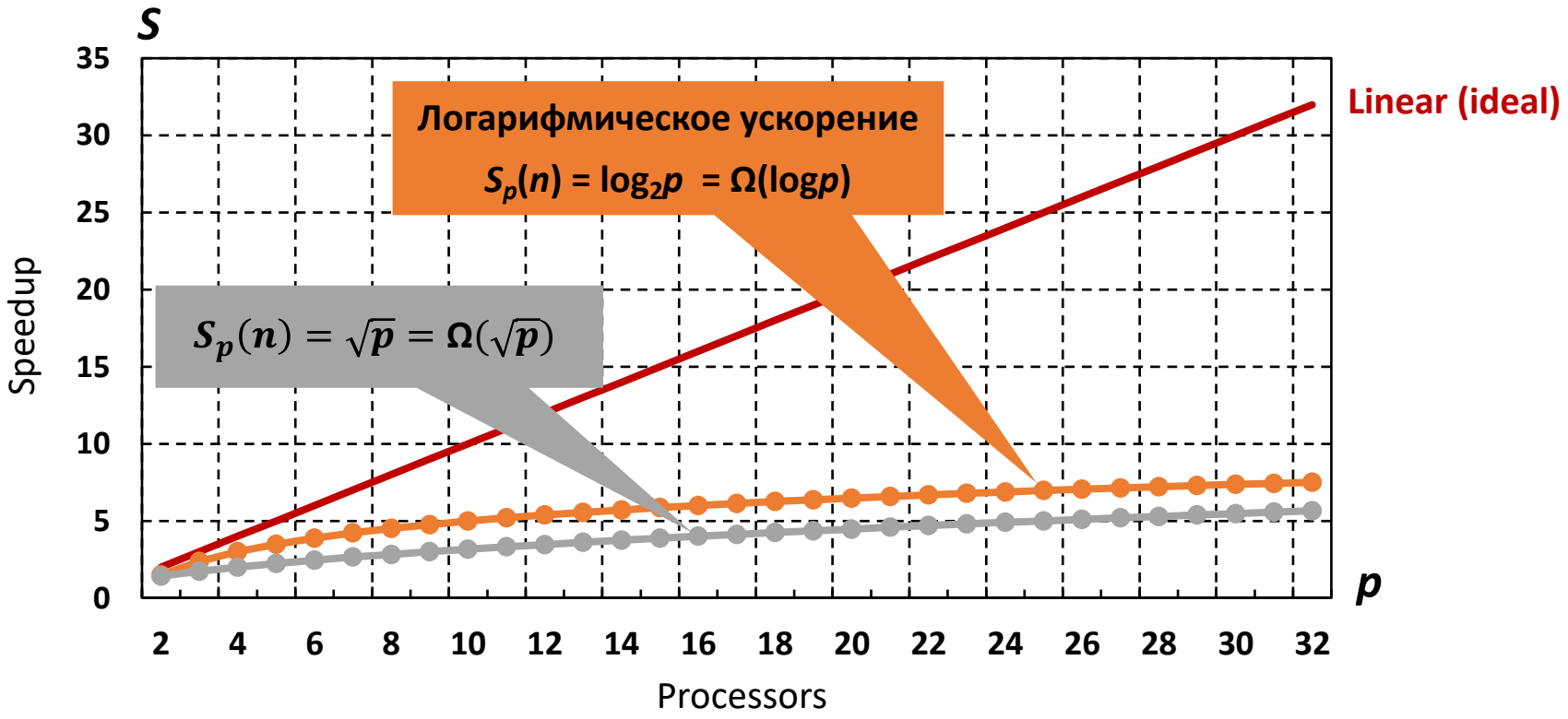
Коэффициент ускорения (Speedup)



Зависимость коэффициента ускорения *S*
параллельного алгоритма X от количества *p* процессоров

- Ускорение программы может расти с увеличением размера входных данных
- Время вычислений превосходит накладные расходы на взаимодействия потоков (управление потоками, синхронизацию, обмен сообщениями, …)

Коэффициент ускорения (Speedup)



Зависимость коэффициента ускорения *S*
параллельных алгоритмов Y и Z от количества *p* процессоров

Суперлинейное ускорение (superlinear speedup)

- Параллельная программа может характеризоваться **суперлинейным ускорением** (superlinear speedup) – коэффициент ускорения $S_p(n)$ принимает значение больше p

$$\boldsymbol{S_p(n) > p}$$

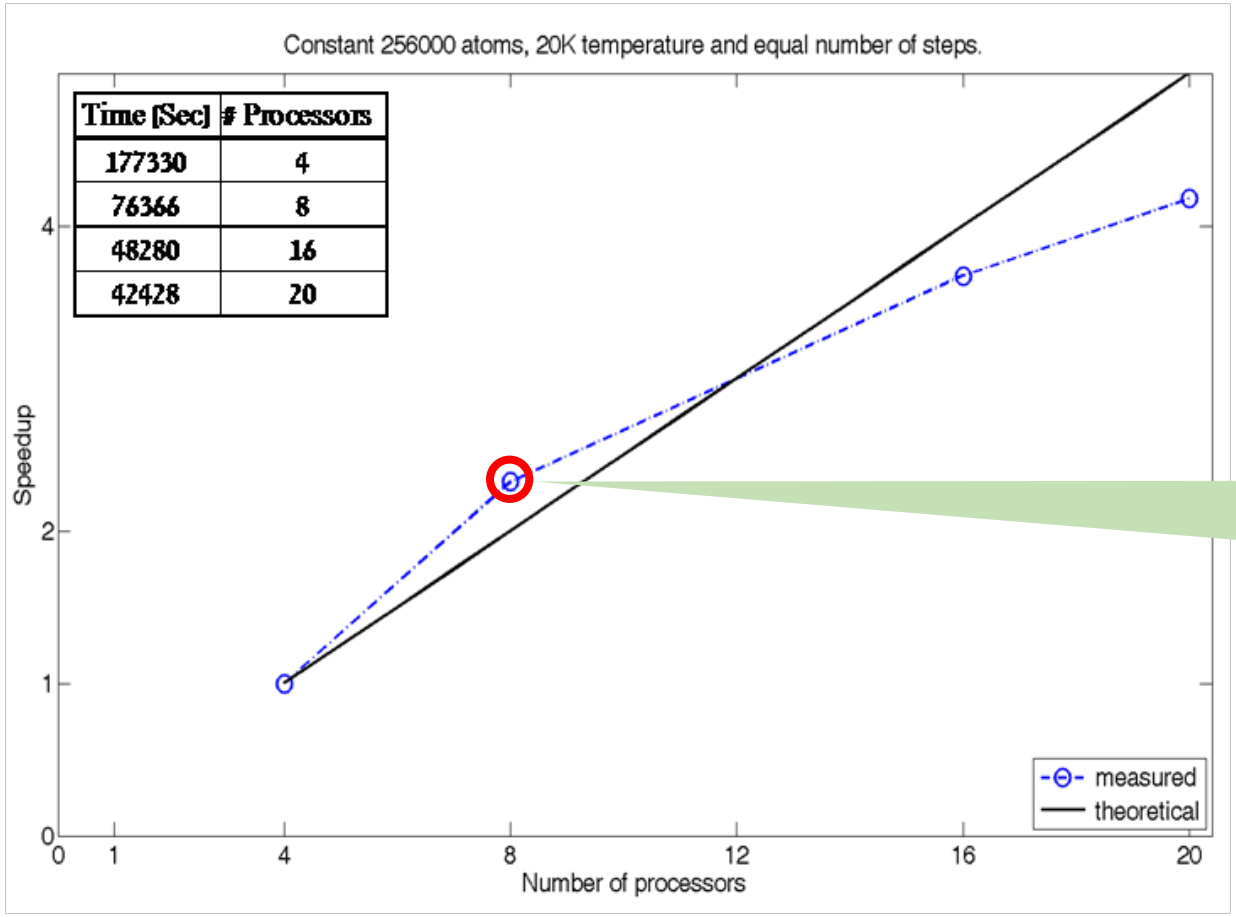
- Причина: иерархическая организация памяти:
  Cache – RAM – Local disk (HDD/SSD) – Network storage

- Последовательная программ выполняется на одном процессоре
  и обрабатывает данные размера n

- Параллельная программа имеет p потоков на p процессорах, каждый поток работает со своей частью данных, большая часть которых может попасть в кеш-память,
  в результате в каждом потоке сокращается время доступа к данным

- Тот же самый эффект можно наблюдать имя два уровня иерархической памяти:
  диск – память

Суперлинейное ускорение (superlinear speedup)



Superlinear speedup

$$S_8 = \frac{T_4}{T_8} = 2.32$$

Parallel Molecular Dynamic Simulation

MPI, Spatial decomposition; Cluster nodes: 2 x AMD Opteron Dual Core; InfiniBand network

http://phycomp.technion.ac.il/~pavelba/Comp_Phys/Project/Project.html

Равномерность распределения вычислений

- **По какому показателю оценивать равномерность времени выполнения потоков/процессов параллельной программы?**
- Известно время выполнения потоков t_0, t_1, …, t_p
- Коэффициент *V* вариации

$$\boldsymbol{V = \frac{\sigma[t_i]}{\mu[t_i]}}$$

- Отношение min/max

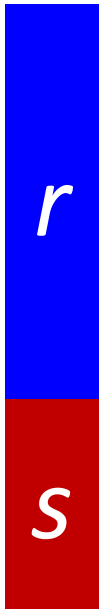
$$\boldsymbol{M = \frac{\min\{t_i\}}{\max\{t_i\}}}$$

- Jain’s fairness index

$$\boldsymbol{f = \frac{\left(\sum_{i=0}^{p-1} t_i\right)^2}{n \sum_{i=0}^{p-1} t_i^2} \in [0, 1]}$$

Закон Дж. Амдала (Amdahl's law)

- Пусть имеется последовательная программа с временем выполнения $T(n)$
- Обозначим:
  - $r \in [0, 1]$ – часть программы, которая может быть распараллелена (perfectly parallelized)
  - $s = 1 - r$ – часть программы, которая не может быть распараллелена (purely sequential)



- Время выполнения параллельной программы на p процессорах (время каждого потока) складывается из последовательной части s и параллельной r:

$$T_p(n) = T(n)s + \frac{T(n)}{p}r$$

- Вычислим значение коэффициент ускорения (по определению)

$$S_p(n) = \frac{T(n)}{T_p(n)} = \frac{T(n)}{T(n)s + \frac{T(n)}{p}r} = \frac{1}{s + \frac{r}{p}} = \frac{1}{(1 - r) + \frac{r}{p}}$$

- Полученная формула по значениям r и s позволяет оценить максимальное ускорение

Закон Дж. Амдала (Amdahl's law)

- Пусть имеется последовательная программа с временем выполнения $T(n)$
- Обозначим:
  - $\boldsymbol{r} \in [\mathbf{0}, \mathbf{1}]$ – часть программы, которая может быть распараллелена (perfectly parallelized)
  - $\boldsymbol{s} = \mathbf{1} - \boldsymbol{r}$ – часть программы, которая не может быть распараллелена (purely sequential)
- **Закон Дж. Амдала (Gene Amdahl, 1967) [1]:**

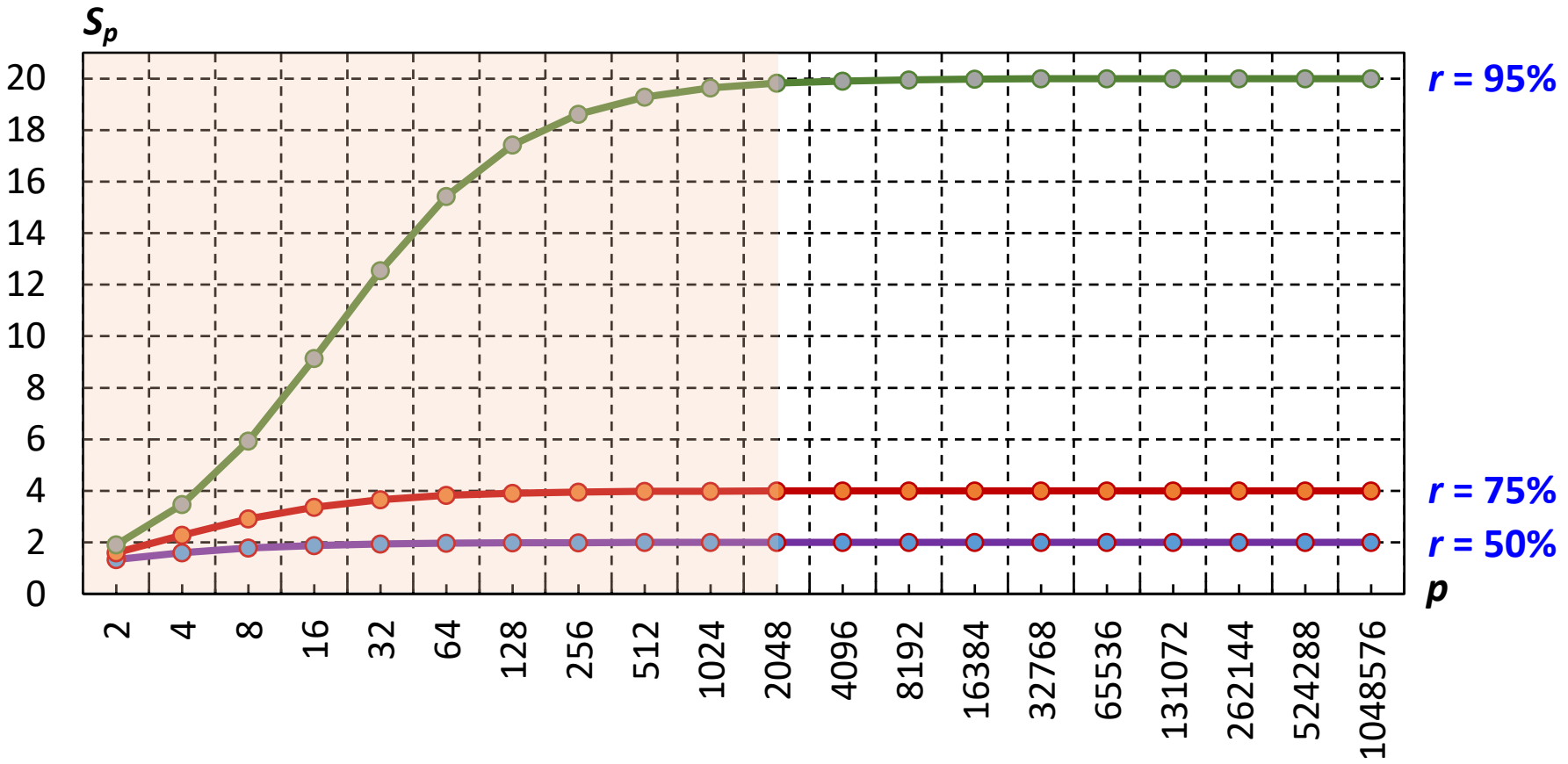
  Максимальное ускорение S_p программы на p процессорах равняется

$$\boldsymbol{S_p} = \frac{\mathbf{1}}{(\mathbf{1} - \boldsymbol{r}) + \frac{\boldsymbol{r}}{\boldsymbol{p}}}$$

$$\boldsymbol{S_\infty} = \lim_{p \to \infty} S_p = \lim_{p \to \infty} \frac{1}{(1 - r) + \frac{r}{p}} = \frac{1}{1 - r} = \frac{\mathbf{1}}{\boldsymbol{s}}$$

❑ Amdahl Gene. **Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities** // AFIPS Conference Proceedings, 1967, pp. 483-485, http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

Закон Дж. Амдала (Amdahl's law)



Зависимость коэффициента S_p ускорения параллельной программы от количества p процессоров

Допущения закона Дж. Амдала (Amdahl's law)

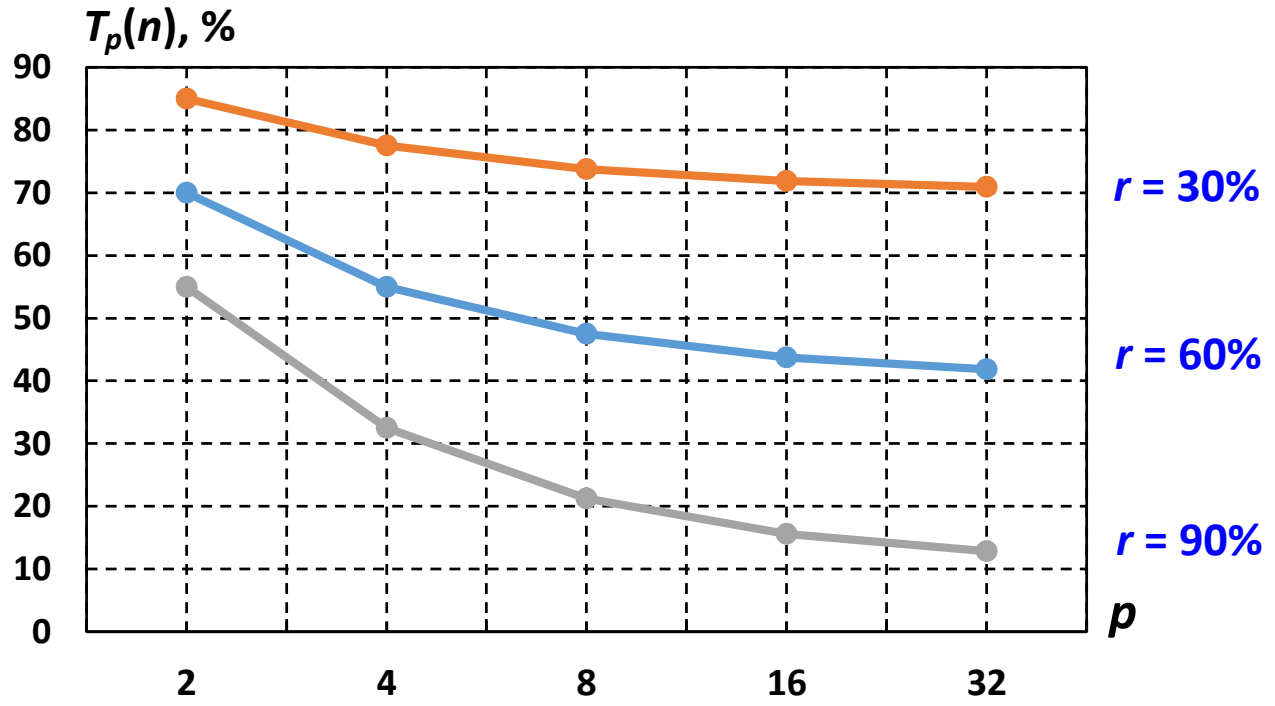
- **Последовательный алгоритм является наиболее оптимальным способом решения задачи**
- Возможны ситуации когда параллельная программа (алгоритм) эффективнее решает задачу (может эффективнее использовать кеш-память, конвейер, SIMD-инструкции, …)
- **Время выполнения параллельной программы оценивается через время выполнения последовательной**, однако потоки параллельной программы могут выполнятся эффективнее

$$T_p(n) = T(n)s + \frac{T(n)}{p}r, \quad \text{на практике возможна ситуация } \frac{T(n)}{p} > T_p(n)$$

- **Ускорение $S_p(n)$ оценивается для фиксированного размера n данных при любых значениях p**
- В реальности при увеличении числа используемых процессоров размер n входных данных также увеличивают, так как может быть доступно больше памяти

Закон Дж. Амдала (Amdahl's law)

- На что потратить ресурсы – на увеличение доли *r* параллельной части в программе или увеличение числа процессоров, на которых запускается программа?



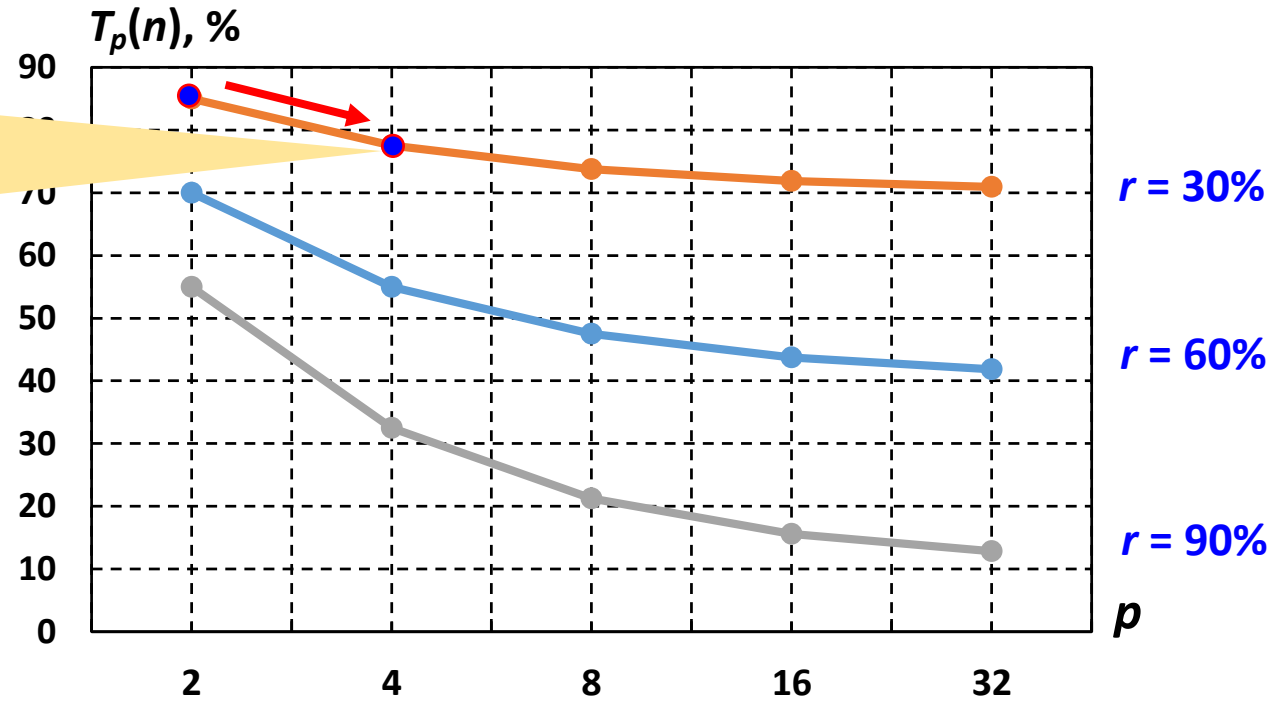
Зависимость времени $T_p(n)$ выполнения параллельной программы от количества *p* процессоров и доли *r* распараллеленного кода (время в % от времени $T_1(n)$)

Закон Дж. Амдала (Amdahl's law)

- На что потратить ресурсы – на увеличение доли *r* параллельной части в программе или увеличение числа процессоров, на которых запускается программа?

Увеличили число процессоров с 2-х до 4-х (программу не меняли)

Время выполнения сократилось с 85% до 77,5%



Зависимость времени $T_p(n)$ выполнения параллельной программы от количества *p* процессоров и доли *r* распараллеленного кода (время в % от времени $T_1(n)$)

Закон Дж. Амдала (Amdahl's law)

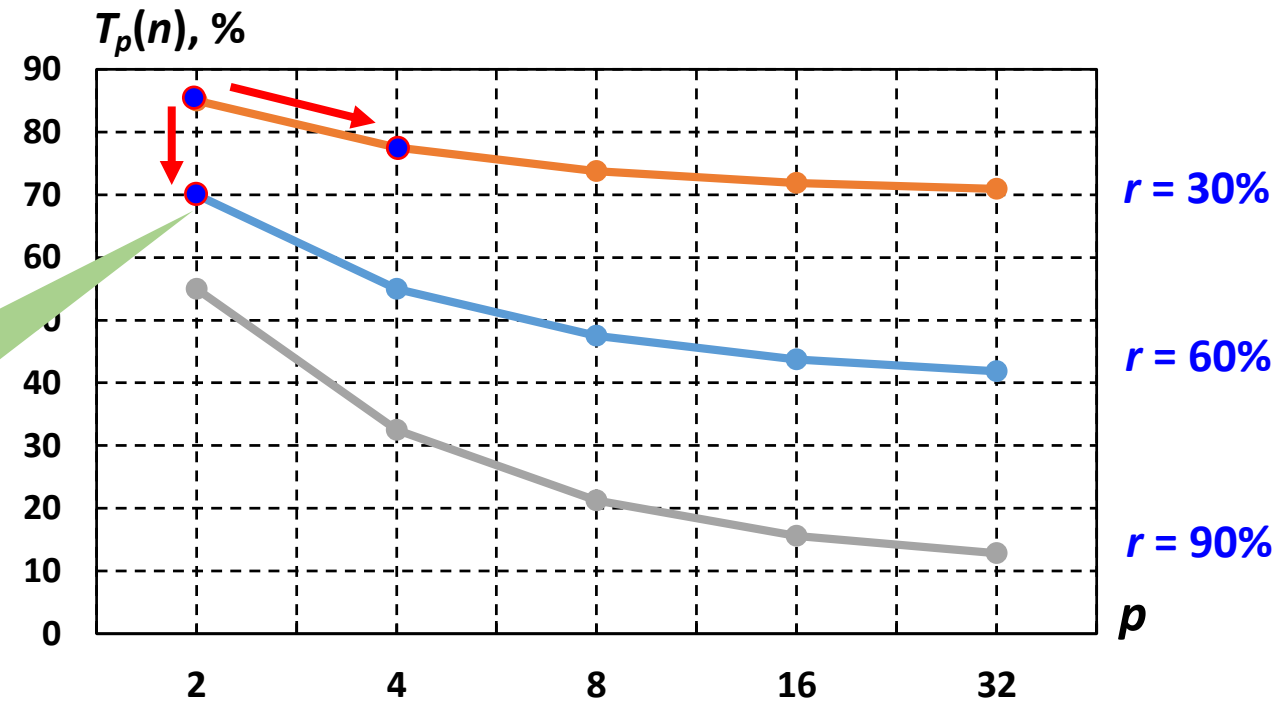
- **На что потратить ресурсы – на увеличение доли *r* параллельной части в программе или увеличение числа процессоров, на которых запускается программа?**

Увеличили число процессоров с 2-х до 4-х (программу не меняли)

Время выполнения сократилось с 85% до 77,5%

Увеличим в 2 раза долю параллельного кода

Время выполнения сократилось с 85% до 70%



Зависимость времени $T_p(n)$ выполнения параллельной программы от количества *p* процессоров и доли *r* распараллеленного кода (время в % от времени $T_1(n)$)

Закон Густафсона-Барсиса

- Пусть имеется последовательная программа с временем выполнения $T(n)$
- Обозначим $s \in [0, 1]$ – часть параллельной программы, которая выполняется последовательно (purely sequential)
- **Закон Густафсона-Барсиса (Gustafson–Barsis' law) [1]:**

Масштабируемое ускорение S_p программы на p процессорах равняется

$$S_p = p - s(p - 1)$$

- **Обоснование:** пусть a – время последовательной части, b – время параллельной части

$$T_p(n) = a + b, \qquad T(n) = a + pb$$

$$s = a/(a + b), \qquad S_p(n) = s + p(1 - s) = p - s(p - 1)$$

- **Время выполнения последовательной программы выражается через время выполнения параллельной**

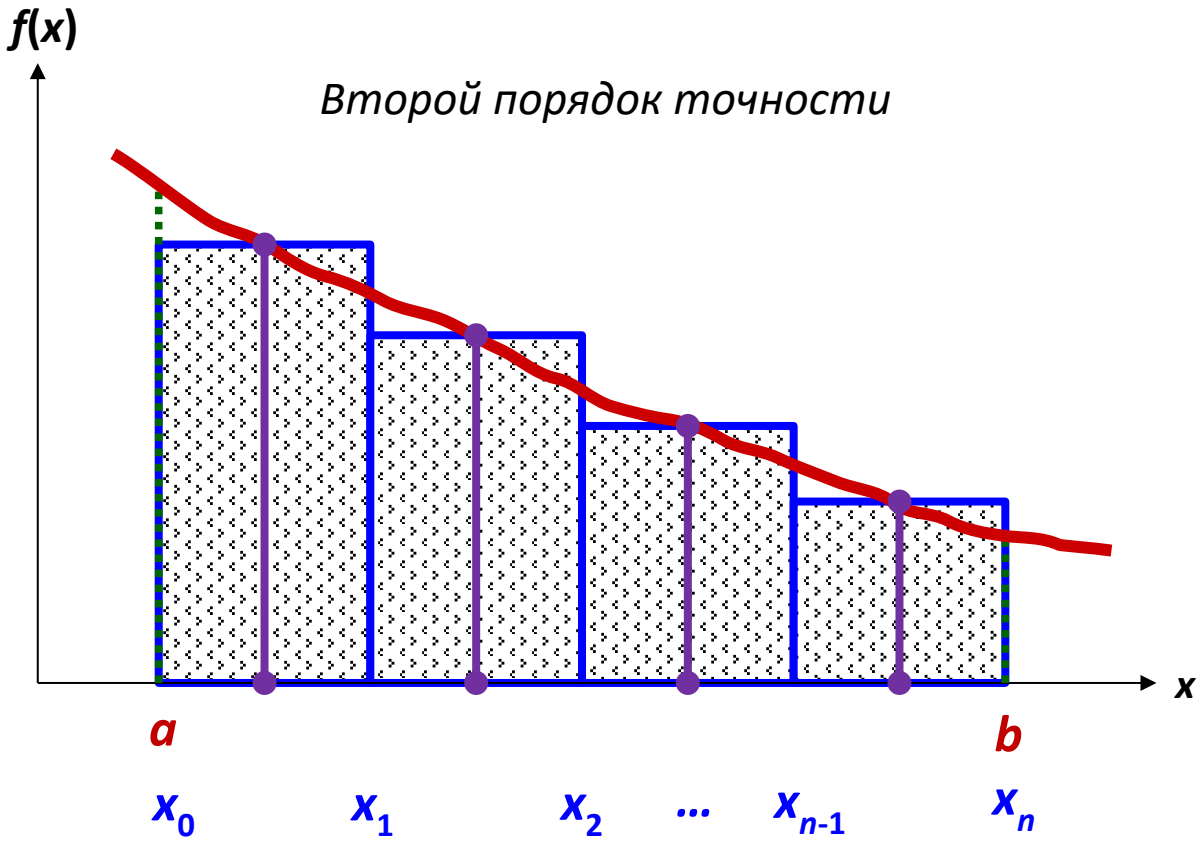
❑ **Reevaluating Amdahl's Law**, John L. Gustafson, Communications of the ACM 31(5), 1988. pp. 532-533 // http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html

Численное интегрирование (numerical integration)

Численное интегрирование (numerical integration)

- **Численное интегрирование (numerical integration)** – вычисление значения определенного интеграла
- Применяется, когда:
  - подынтегральная функция не задана аналитически. Например, представлена в виде таблицы значений в узлах расчётной сетки
  - аналитическое представление подынтегральной функции известно, но её первообразная не выражается через аналитические функции
  - вид первообразной может быть настолько сложен, что быстрее вычислить значение интеграла численным методом

Формула средних прямоугольников (midpoint rule)

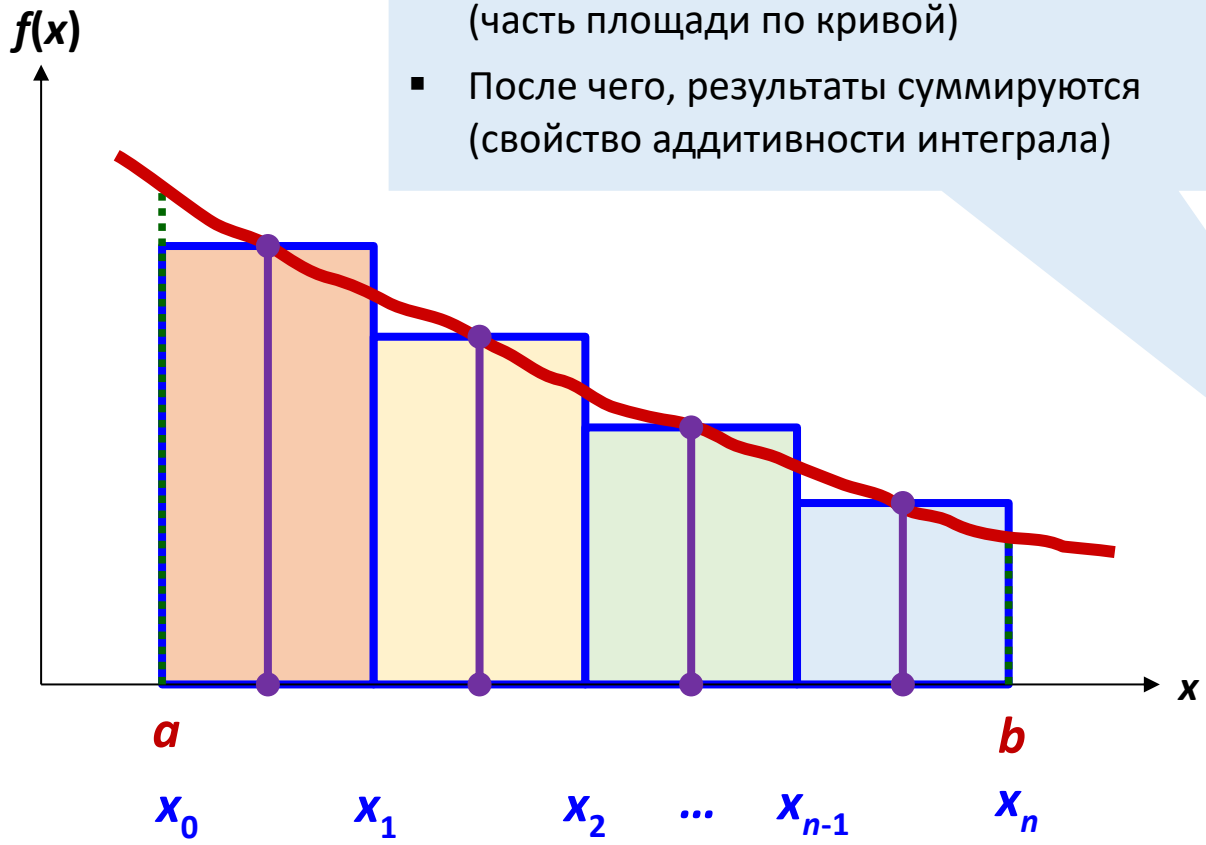


$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{n} f\left(x_{i-1}+\frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i-\frac{h}{2}\right), \qquad h=\frac{b-a}{n}$$

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

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

- Каждый поток вычисляет частичную сумму (часть площади по кривой)
- После чего, результаты суммируются (свойство аддитивности интеграла)



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

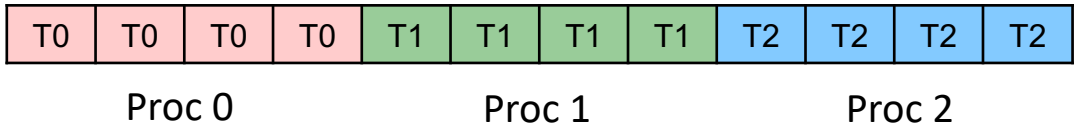
```
double func(double x)
{
    return exp(-x * x);
}

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

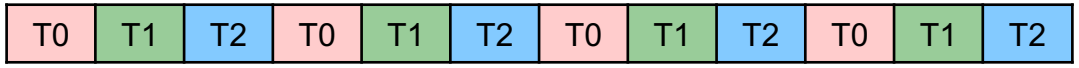
1. Итерации цикла *for* распределяются между потоками
2. Каждый поток вычисляет часть суммы (площади)
3. Суммирование результатов потоков (во всех или одном процессе)

Варианты распределения итераций (точек) между процессами:

1) Разбиение на *p* смежных непрерывных частей



2) Циклическое распределение итераций по потокам

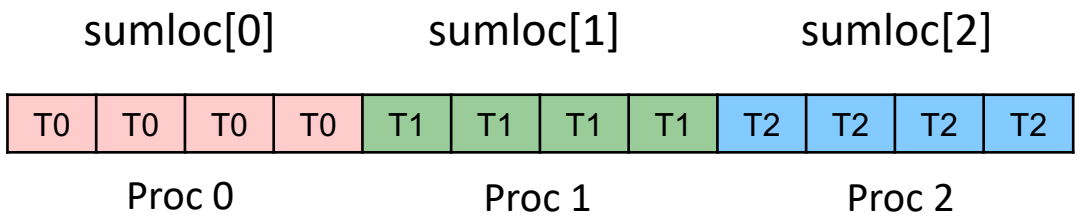


Proc 0, Proc 1, Proc 2, Proc 0, Proc 1, Proc 2, ...

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

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

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

int main(int argc, char **argv)
{
    double t = omp_get_wtime();
    const double a = -4.0;
    const double b = 4.0;
    const int n = 10000000;
    printf("Numerical integration: [%f, %f], n = %d\n", a, b, n);

    double h = (b - a) / n;
    double s = 0.0;
    double sumloc[omp_get_max_threads()];
```

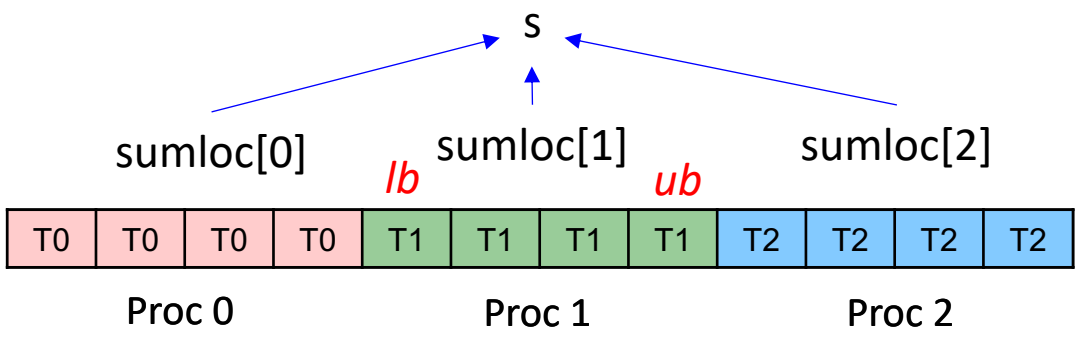


Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

```
    // продолжение…

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int tid = omp_get_thread_num();
        int points_per_thread = n / nthreads;
        int lo = tid * points_per_thread;
        int hi = (tid == nthreads - 1) ? n - 1 : lo + points_per_thread;
        sumloc[tid] = 0.0;
        for (int i = lo; i <= hi; i++)
            sumloc[tid] += func(a + h * (i + 0.5));

        #pragma omp atomic
        s += sumloc[tid];
    }
    s *= h;

    printf("Result Pi: %.12f\n", s * s);
    t = omp_get_wtime() - t;
    printf("Elapsed time (sec.): %.12f\n", t);
    return 0;
}
```

Шаг 1. Вычисление частичной суммы каждым потоком

Распараллеливание цикла – разбиение пространства итераций на *p* смежных непрерывных частей



Атомарные операции
#pragma omp atomic

```
#pragma omp atomic
x = x binop expr;
```

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

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

- **Intel 64 locked atomic operation** (BTS, XADD, CMPXCHG, ADD, …)
  - ❑ Integer operand – one atomic operation
  - ❑ Floating point operand – loop with CAS

Атомарные операции
#pragma omp atomic

2.13.6 `atomic` Construct

Summary

The **`atomic`** construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Syntax

In the following syntax, *atomic-clause* is a clause that indicates the semantics for which atomicity is enforced and is one of the following:

`read`

`write`

`update`

`capture`

C / C++

The syntax of the **`atomic`** construct takes one of the following forms:

`#pragma omp atomic` *[***`seq_cst`***[,]] atomic-clause [[,]***`seq_cst`***] new-line*
expression-stmt

or

`#pragma omp atomic` *[***`seq_cst`***] new-line*
expression-stmt

or

`#pragma omp atomic` *[***`seq_cst`***[,]]* **`capture`** *[[,]***`seq_cst`***] new-line*
structured-block

where *expression-stmt* is an expression statement with one of the following forms:

- If *atomic-clause* is **`read`**:
  v `=` *x*`;`
- If *atomic-clause* is **`write`**:
  x `=` *expr*`;`

C/C++ (cont.)

- If *atomic-clause* is **`update`** or not present:
  x`++;`
  x`--;`
  `++`*x*`;`
  `--`*x*`;`
  x *binop*`=` *expr*`;`
  x `=` *x* *binop* *expr*`;`
  x `=` *expr* *binop* *x*`;`
- If *atomic-clause* is **`capture`**:
  v `=` *x*`++;`
  v `=` *x*`--;`
  v `=` `++`*x*`;`
  v `=` `--`*x*`;`
  v `=` *x* *binop*`=` *expr*`;`
  v `=` *x* `=` *x* *binop* *expr*`;`
  v `=` *x* `=` *expr* *binop* *x*`;`

  and where *structured-block* is a structured block with one of the following forms:

  `{`*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*`;}`

  In the preceding expressions:

- *x* and *v* (as applicable) are both *l-value* expressions with scalar type.
- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.
- Neither of *v* and *expr* (as applicable) may access the storage location designated by *x*.

Атомарные операции
#pragma omp atomic

```
int counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += i;
}
```

```
lock addl %ecx, (%rsi)
```

```
double counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += static_cast<double>(i);
}
```

```
.L8:  movq          %rax, %rdx
.L4:  movq          %rdx, 8(%rsp)
      movq          %rdx, %rax
      movsd 8(%rsp), %xmm1
      addsd %xmm0, %xmm1
      movq          %xmm1, %rsi
      lock cmpxchgq %rsi, (%rcx)
      cmpq          %rax, %rdx
      jne .L8
```

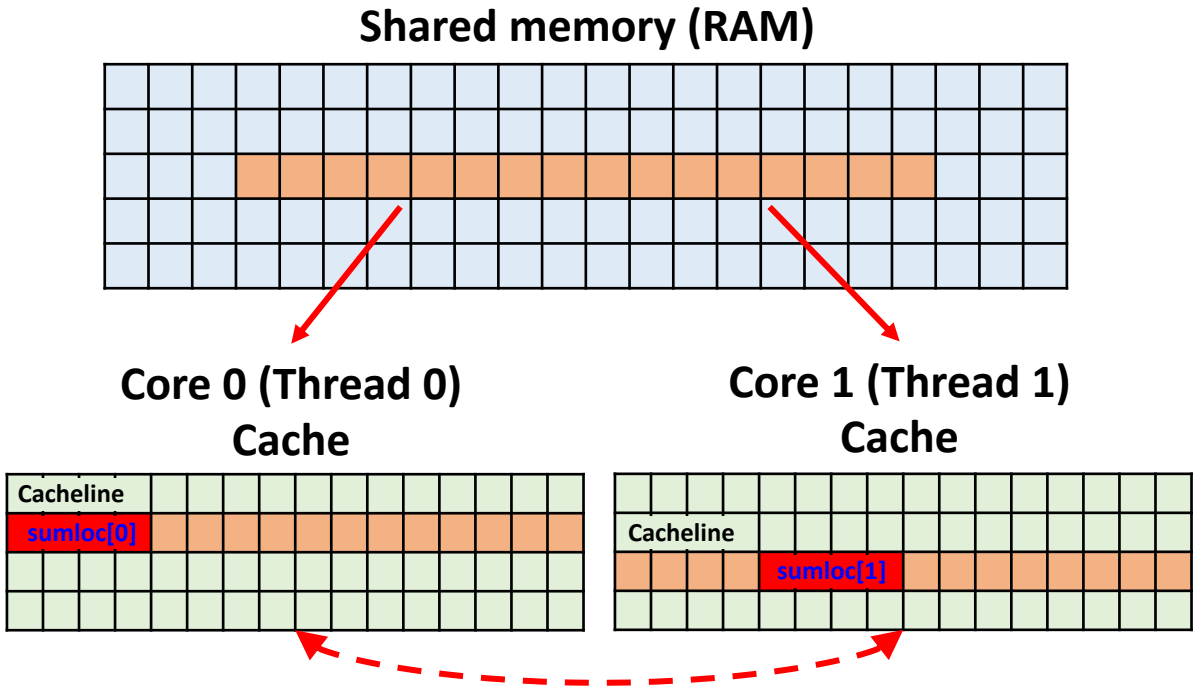
Loop!

Цикл выполняется пока ячейка успешно не обновится

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

```
// продолжение…

#pragma omp parallel
{
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int points_per_thread = n / nthreads;
    int lo = tid * points_per_thread;
    int hi = (tid == nthreads - 1) ?
             n - 1 : lo + points_per_thread;
    sumloc[tid] = 0.0;
    for (int i = lo; i <= hi; i++)
        sumloc[tid] += func(a + h * (i + 0.5));

    #pragma omp atomic
    s += sumloc[tid];
}
s *
pri
t =
printf(
return 0;
}
```

Ложное разделение данных (false sharing)



MESI (Intel MESIF) cache coherency protocol

- Несколько разделяемых переменных попали в одну строку кеш-памяти процессорных ядер
- Две строки в кеш-памяти процессорных ядер постоянно обновляются актуальными данными из памяти – кеширование “отключается”

❑ **Avoiding and Identifying False Sharing Among Threads** // http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Устранение ложного разделения данных

```
struct thread_data {
    double sum;
    uint8_t padding[64 - sizeof(double)];
};

int main(int argc, char **argv)
{
    // ...
    double s = 0.0;
    struct thread_data sumloc[omp_get_max_threads()];
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int tid = omp_get_thread_num();
        int points_per_thread = n / nthreads;
        int lo = tid * points_per_thread;
        int hi = (tid == nthreads - 1) ? n - 1 : lo + points_per_thread;
        sumloc[tid].sum = 0.0;
        for (int i = lo; i <= hi; i++)
            sumloc[tid].sum += func(a + h * (i + 0.5));

        #pragma omp atomic
        s += sumloc[tid].sum;
    }
    s *= h;
    // ...
}
```

Один элемент массива sumloc[] занимает одну строку кеш-памяти – 64 байта (Intel 64)

Параллельный алгоритм интегрирования методом средних прямоугольников: циклическое распределение

```
double h = (b - a) / n;
double s = 0.0;

#pragma omp parallel
{
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();

    double sloc = 0.0;
    for (int i = tid; i < n; i += nthreads)
        sloc += func(a + h * (i + 0.5));

    #pragma omp atomic
    s += sloc;
}
s *= h;
```

Циклическая схема распределения итераций (round-robin)

0	1	2	0	1	2	0	1	2	0	1	2

Параллельный алгоритм интегрирования методом средних прямоугольников
#pragma omp for

```
int main(int argc, char **argv)
{
    // ...
    double h = (b - a) / n;
    double s = 0.0;

    #pragma omp parallel
    {
        double sloc = 0.0;
        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            sloc += func(a + h * (i + 0.5));
        #pragma omp atomic
        s += sloc;
    }
    s *= h;
    // ...
    return 0;
}
```

Итерации распределяются по потокам блоками смежных итераций

#pragma omp for

The syntax of the loop construct is as follows:

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

where clause is one of the following:

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

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.6 on page 53).

C / C++

Апостериорная оценка погрешности по правилу Рунге

Правило Рунге

1. Интеграл вычисляется по выбранной квадратурной формуле (прямоугольников, трапеций, Симпсона) при числе шагов n и при числе шагов $2n$
2. Погрешность вычисления значения интеграла при числе шагов $2n$ определяется по *формуле Рунге*:

$$\Delta_{2n} \approx \frac{|L_{2n} - L_n|}{2^p - 1},$$

где p – порядок точности метода (для метода средних прямоугольников $p = 2$)

Интеграл вычисляется для последовательных значений числа шагов $n = n_0,\ 2n_0,\ 4n_0,\ 8n_0,\ 16n_0, \ldots$ пока не будет достигнута заданная точность:

$$\Delta_{2n} < \varepsilon$$

Интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

```
const double eps = 1E-6;
const int n0 = 100;

int main()
{
    int n = n0, k;
    double sq[2], delta = 1;
    for (k = 0; delta > eps; n *= 2, k ^= 1) {
        double h = (b - a) / n;
        double s = 0.0;
        for (int i = 0; i < n; i++)
            s += func(a + h * (i + 0.5));
        sq[k] = s * h;
        if (n > n0)
            delta = fabs(sq[k] - sq[k ^ 1]) / 3.0;
    }
    printf("Result Pi: %.12f; Runge rule: EPS %e, n %d\n", sq[k] * sq[k], eps, n / 2);

    return 0;
}
```

- sq[0] – сумма для числа шагов *n*
- sq[1] – сумма для числа шагов 2*n*

- 0 ^ 1 = 1
- 1 ^ 1 = 0

Параллельное интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

```
const double eps = 1E-6;
const int n0 = 100;

int main()
{
    int n = n0, k;
    double sq[2], delta = 1;
    for (k = 0; delta > eps; n *= 2, k ^= 1) {
        double h = (b - a) / n;
        double s = 0.0;
        for (int i = 0; i < n; i++)
            s += func(a + h * (i + 0.5));
        sq[k] = s * h;
        if (n > n0)
            delta = fabs(sq[k] - sq[k ^ 1]) / 3.0;
    }
    printf("Result Pi: %.12f; Runge rule: EPS %e, n %d\n", sq[k] * sq[k], eps, n / 2);

    return 0;
}
```

I. Распараллелить внешний цикл?

- 2 процесса: один вычисляет интеграл для n шагов, второй для $2n$ шагов
- p процессов: каждый процесс вычисляет интеграл при заданном числе шагов: n, $2n$, $4n$, …., $2^{P-1}n$

  Избыточные вычисления – требуемая точность может быть достигнута при $kn < pn$

Параллельное интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

```
const double eps = 1E-6;
const int n0 = 100;

int main()
{
    int n = n0, k;
    double sq[2], delta = 1;
    for (k = 0; delta > eps; n *= 2, k ^= 1) {
        double h = (b - a) / n;
        double s = 0.0;
        for (int i = 0; i < n; i++)
            s += func(a + h * (i + 0.5));
        sq[k] = s * h;
        if (n > n0)
            delta = fabs(sq[k] - sq[k ^ 1]) / 3.0;
    }
    printf("Result Pi: %.12f; Runge rule: EPS %e, n %d\n", sq[k] * sq[k], eps, n / 2);

    return 0;
}
```

I. Распараллелить внутренний цикл?

- ❑ Каждый процесс вычисляет частичную сумму
- ❑ Выполняется глобальное суммирование
- ❑ Процессы вычисляют погрешность (delta) и проверяют условия завершения вычислений (delta < eps)

Параллельное интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

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

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

int main(int argc, char **argv)
{
    double t = omp_get_wtime();
    const double eps = 1E-6;
    const double a = -4.0;
    const double b = 4.0;
    const int n0 = 100000000;
    printf("Numerical integration: [%f, %f], n0 = %d, EPS = %f\n", a, b, n0, eps);
```

Параллельное интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

```
// продолжение main()
double sq[2];
#pragma omp parallel
{
    int n = n0, k;
    double delta = 1;
    for (k = 0; delta > eps; n *= 2, k ^= 1) {
        double h = (b - a) / n;
        double s = 0.0;
        sq[k] = 0;
        // Ждем пока все потоки закончат обнуление sq[k]
        #pragma omp barrier

        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            s += func(a + h * (i + 0.5));

        #pragma omp atomic
        sq[k] += s * h;
```

Параллельное интегрирование методом средних прямоугольников с заданной точностью (правило Рунге)

```
            // Ждем пока все потоки обновят sq[k]
            #pragma omp barrier
            if (n > n0)
                delta = fabs(sq[k] - sq[k ^ 1]) / 3.0;
            #if 0
            printf("n=%d i=%d sq=%.12f delta=%.12f\n", n, k, sq[k], delta);
            #endif
        }

        #pragma omp master
        printf("Result Pi: %.12f; Runge rule: EPS %e, n %d\n", sq[k] * sq[k], eps, n / 2);
    }
    t = omp_get_wtime() - t;
    printf("Elapsed time (sec.): %.6f\n", t);
    return 0;
}
```

Интегрирование методом Монте-Карло

Интегрирование методом Монте-Карло (ММК, Monte Carlo method)

- **Применяется для интегралов большой размерности**
  - Например для одномерной функции достаточно разбиения на 10 отрезков и вычисление 10 значений функции (см. метод прямоугольников)
  - Если функция *n*-мерная (задачи теории струн и т.д.), то по каждой размерности разбиваем на 10 отрезков, следовательно потребуется 10^n вычислений значения функции

- **Суть метода МК (для одномерного случая)**

  1. Бросаем *n* точек, равномерно распределённых на [*a*, *b*], для каждой точки u_i вычисляем $f(u_i)$
  2. Затем вычисляем выборочное среднее

$$\frac{1}{n}\sum_{i=1}^{n} f(u_i)$$

  3. Получаем оценку интеграла

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=1}^{n} f(u_i)$$

Точность оценки зависит только от количества точек *n*

Вычисление кратных интегралов

- Вычислить двойной интеграл методом Монте-Карло

$$I = \iint\limits_{\Omega} 3y^2 \sin^2(x)\, dxdy, \qquad \Omega = \{x \in [0, \pi], y \in [0, \sin(x)]\}$$

- Выберем n псевдо-случайных точек (x_i, y_i), равномерно распределенных в области Ω
- Из общего числа n точек n' попали в область Ω, остальные $n - n'$ оказались вне области
- При значительном числе n интеграл приближенно равен

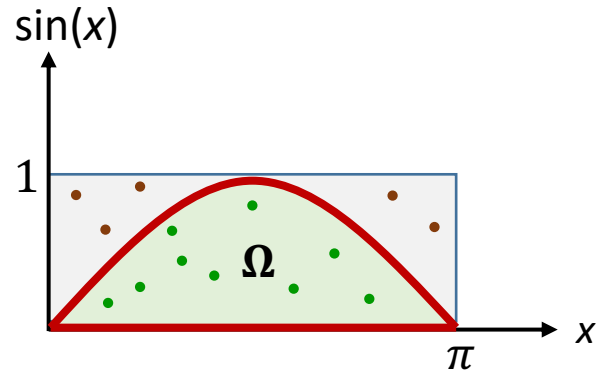
$$I \approx \frac{V}{n'} \sum_{i=1}^{n'} f(x_i, y_i)$$

$$f(x, y) = 3y^2 \sin^2(x), \qquad V = \int\limits_0^{\pi} \sin(x)\, dx = 2$$

Вычисление двойного интеграла методом Монте-Карло

```
const double PI = 3.14159265358979323846;
const int n = 10000000;

int main(int argc, char **argv)
{
    int in = 0;
    double s = 0;

    for (int i = 0; i < n; i++) {
        double x = getrand() * PI;  /* x in [0, pi] */
        double y = getrand();       /* y in [0, sin(x)] */
        if (y <= sin(x)) {
            in++;
            s += func(x, y);
        }
    }
    double v = PI * in / n;
    double res = v * s / in;

    printf("Result: %.12f, n %d\n", res, n);
    return 0;
}
```

```
/* returns pseudo-random number in the [0, 1] */
double getrand()
{ return (double)rand() / RAND_MAX; }

double func(double x, double y)
{ return 3 * pow(y, 2) * pow(sin(x), 2); }
```

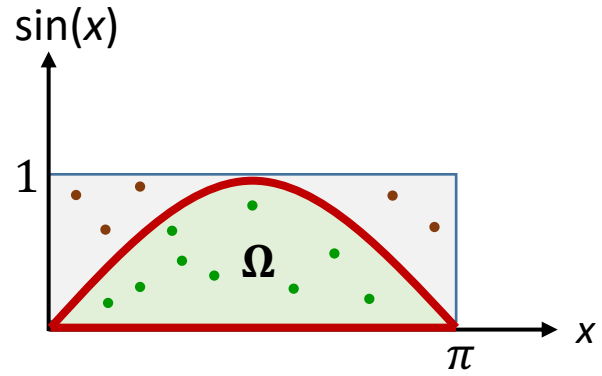


Параллельное вычисление двойного интеграла методом Монте-Карло

```
const double PI = 3.14159265358979323846;
const int n = 10000000;

int main(int argc, char **argv)
{
    int in = 0;
    double s = 0;

    for (int i = 0; i < n; i++) {
        double x = getrand() * PI;  /* x in [0, pi] */
        double y = getrand();       /* y in [0, sin(x)] */
        if (y <= sin(x)) {
            in++;
            s += func(x, y);
        }
    }
    double v = PI * in / n;
    double res = v * s / in;

    printf("Result: %.12f, n %d\n", res, n);
    return 0;
}
```

Схема распараллеливания

1. Каждый из *p* потоков генерирует и бросает *n* / *p* точек
2. Глобальная сумма формируется в мастер-потоке



Параллельное вычисление двойного интеграла методом Монте-Карло

```
#define _POSIX_C_SOURCE 1
#include <stdlib.h>

double getrand(unsigned int *seed)
{
    return (double)rand_r(seed) / RAND_MAX;
}

int main(int argc, char **argv)
{
    const int n = 10000000;
    printf("Numerical integration by Monte Carlo method: n = %d\n", n);

    int in = 0;
    double s = 0;
    #pragma omp parallel
    {
        double s_loc = 0;
        int in_loc = 0;
        unsigned int seed = omp_get_thread_num();
```

```
$ man 3 rand
...
The  function rand() is not reentrant, since it uses
hidden state that is modified on each call.  This might
just be the seed value to be used by the next call, or
it might be something more elaborate.  In order  to  get
reproducible  behavior in a threaded application, this
state must be made explicit; this can be done using the
reentrant function rand_r().
...
```

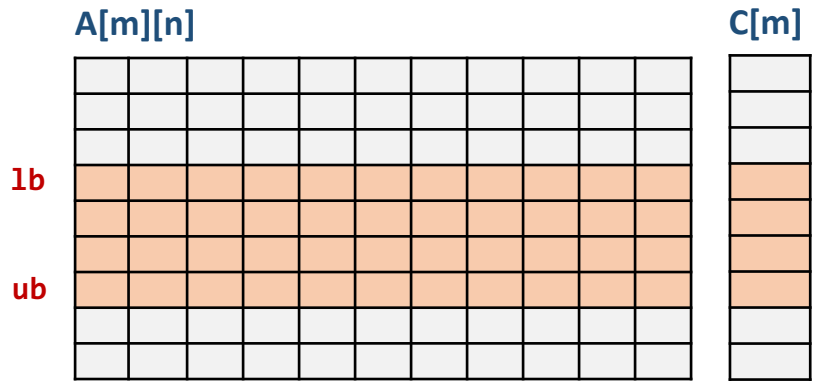
Параллельное вычисление двойного интеграла методом Монте-Карло

```
        #pragma omp for nowait
        for (int i = 0; i < n; i++) {
            double x = getrand(&seed) * PI; /* x in [0, pi] */
            double y = getrand(&seed);      /* y in [0, sin(x)] */
            if (y <= sin(x)) {
                in_loc++;
                s_loc += func(x, y);
            }
        }
        #pragma omp atomic
        s += s_loc;
        #pragma omp atomic
        in += in_loc;
    }
    double v = PI * in / n;
    double res = v * s / in;
    printf("Result: %.12f, n %d\n", res, n);
    return 0;
}
```

Модификация DGEMV с использованием #pragma omp for

DGEMV: параллельная версия

```
/* matrix_vector_product_omp: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        c[i] = 0.0;
        for (int j = 0; j < n; j++)
            c[i] += a[i * n + j] * b[j];
    }
}
```



Литература

- Эндрюс Г**. Основы многопоточного, параллельного и распределенного программирования**. – М.: Вильямс, 2003.
- Хорошевский В.Г. **Архитектура вычислительных систем**. – М.: МГТУ им. Н.Э. Баумана, 2008. – 520 с.
- Корнеев В.В. **Вычислительные системы**. – М.: Гелиос АРВ, 2004. – 512 с.
- Степаненко С.А. **Мультипроцессорные среды суперЭВМ. Масштабирование эффективности**. – М.: ФИЗМАТЛИТ, 2016. – 312 с.