Лекция 5
Параллельное численное интегрирование

Курносов Михаил Георгиевич

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

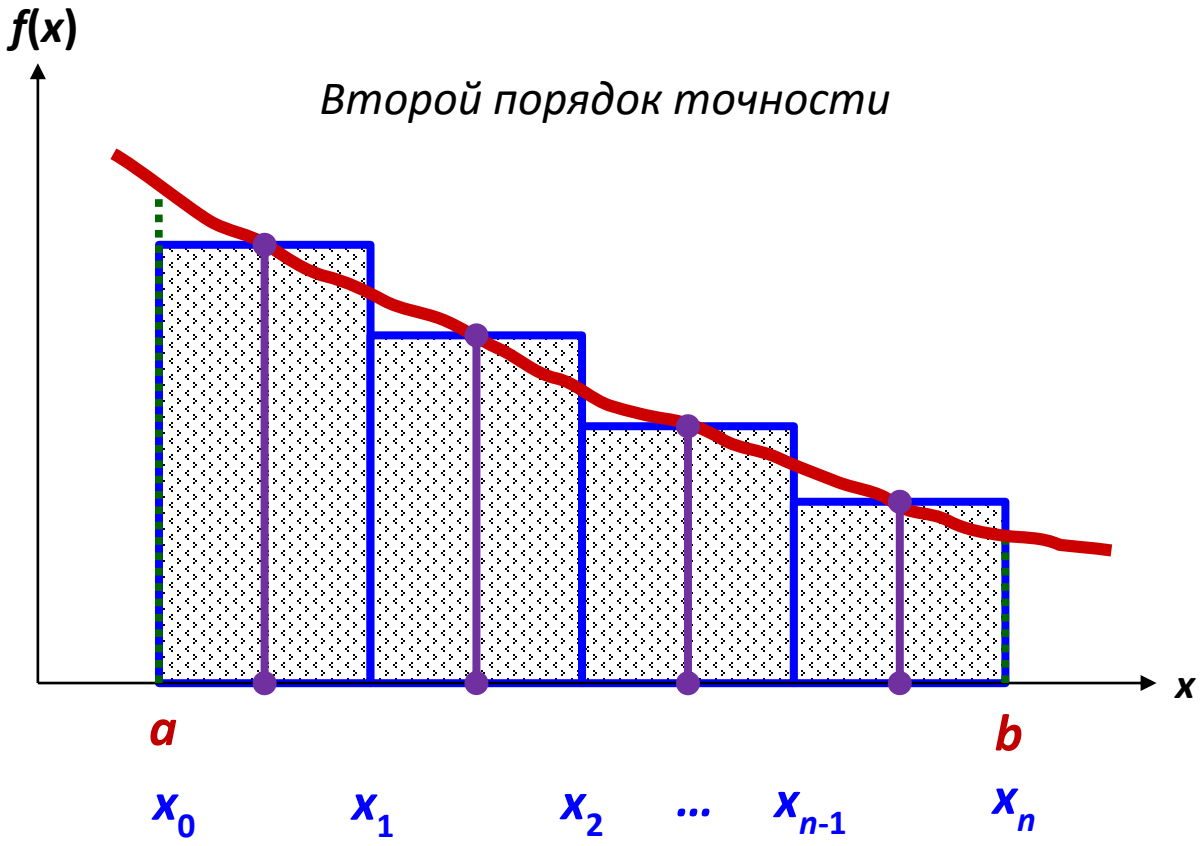
Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Осенний семестр, 2017

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)

Формула средних прямоугольников (midpoint rule)

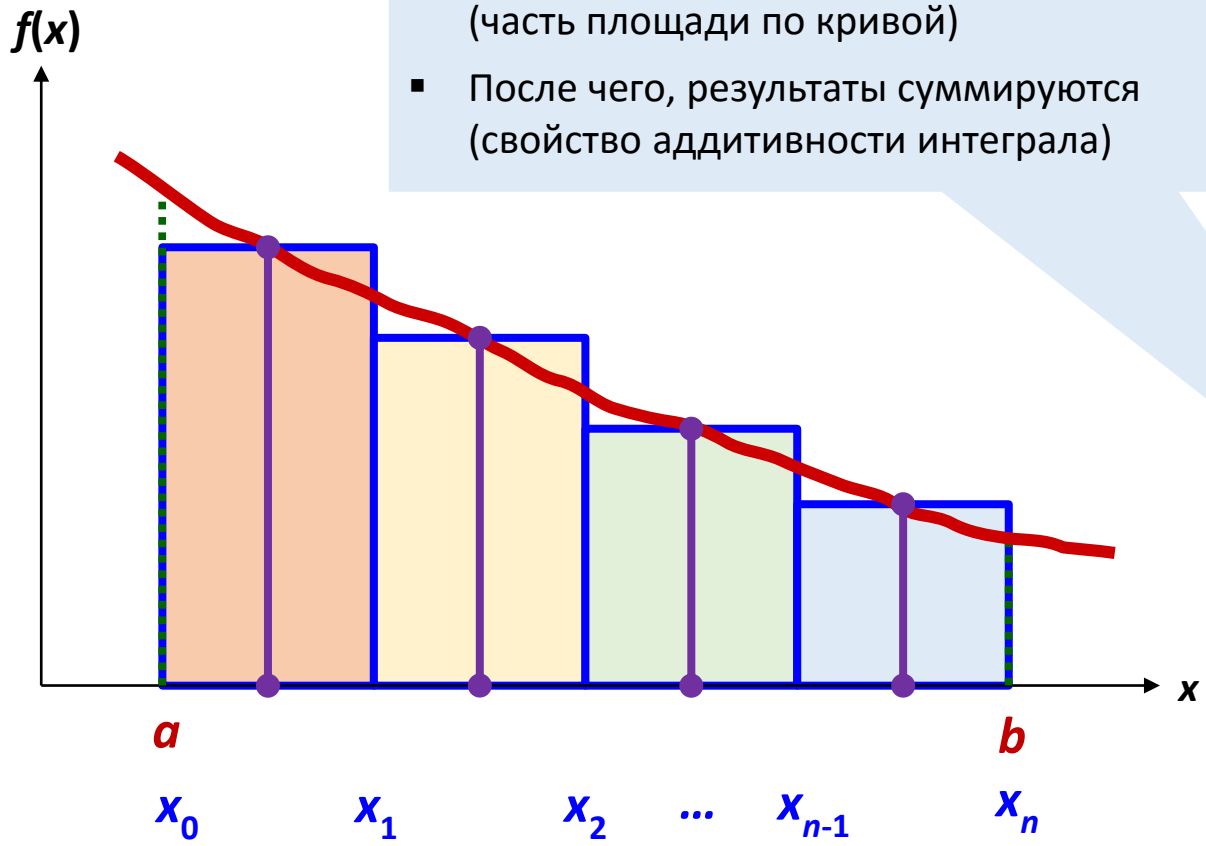


$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{n} f\left(x_{i-1}+\frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i-\frac{h}{2}\right), \qquad h=\frac{b-a}{n}$$

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



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

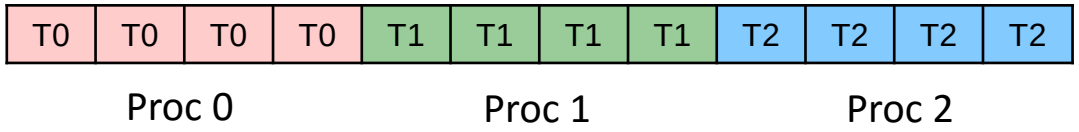
```
double func(double x)
{
    return exp(-x * x);
}

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)

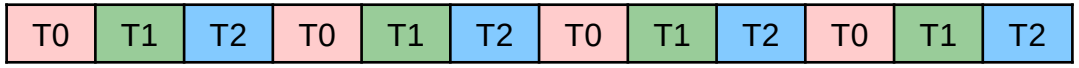
1. Итерации цикла *for* распределяются между процессами
2. Каждый поток вычисляет часть суммы (площади)
3. Суммирование результатов потоков (во всех или одном процессе)

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, ...

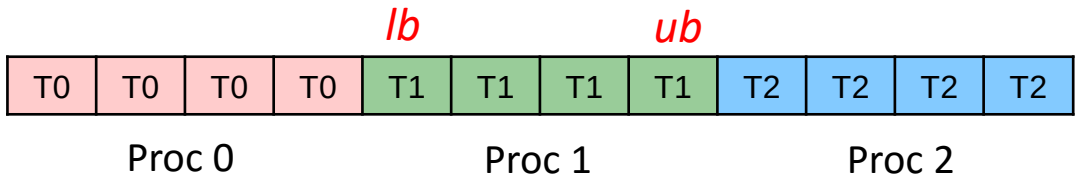
```
double func(double x)
{
    return exp(-x * x);
}

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)
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int points_per_proc = n / commsize;
    int lb = rank * points_per_proc;
    int ub = (rank == commsize - 1) ? (n - 1) : (lb + points_per_proc - 1);

    double sum = 0.0;
    double h = (b - a) / n;
    for (int i = lb; i <= ub; i++)
        sum += func(a + h * (i + 0.5));
```

Шаг 1. Вычисление частичной суммы каждым процессом

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



Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

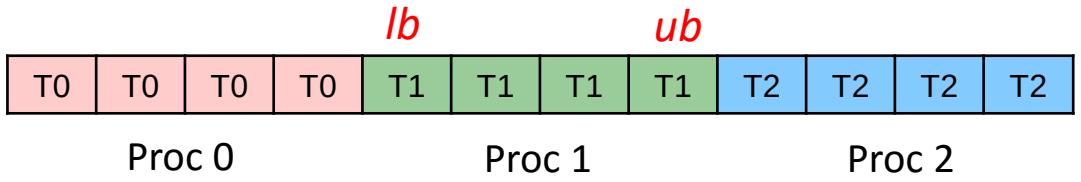
Шаг 2. Суммирование (редукция)

```
    /* Продолжение ... */

    double gsum = 0.0;
    MPI_Reduce(&sum, &gsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        gsum *= h;
        printf("Result Pi: %.12f; error %.12f\n", gsum * gsum, fabs(gsum - sqrt(PI)));
    }

    MPI_Finalize();
    return 0;
}
```

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)

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

int main(int argc, char **argv)
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int points_per_proc = n / commsize;
    int lb = rank * points_per_proc;
    int ub = (rank == commsize - 1) ? (n - 1) : (lb + points_per_proc - 1);

    double sum = 0.0;
    double h = (b - a) / n;
    for (int i = lb; i <= ub; i++)
        sum += func(a + h * (i + 0.5));
```

Разбиение на *p* смежных непрерывных частей



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;
}
```

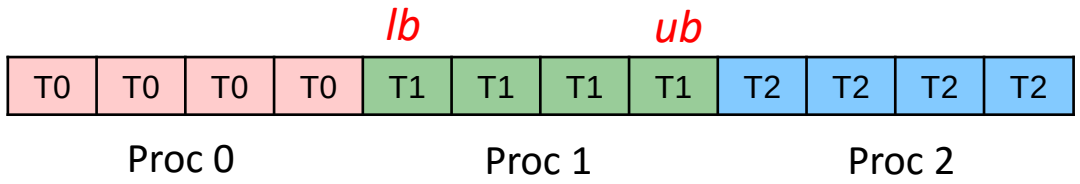
I. Распараллелить внутренний цикл?

- ❑ 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)

```
int main(int argc, char **argv)
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int n = n0, k;
    double sq[2], delta = 1;
    for (k = 0; delta > eps; n *= 2, k ^= 1) {

        int points_per_proc = n / commsize;
        int lb = rank * points_per_proc;
        int ub = (rank == commsize - 1) ? (n - 1) : (lb + points_per_proc - 1);
        double h = (b - a) / n;

        double s = 0.0;
        for (int i = lb; i <= ub; i++)
            s += func(a + h * (i + 0.5));
```

Разбиение на *p* смежных непрерывных частей



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

```
        /* Продолжение… */

        MPI_Allreduce(&s, &sq[k], 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        sq[k] *= h;
        if (n > n0)
            delta = fabs(sq[k] - sq[k ^ 1]) / 3.0;
    } /* for */

    if (rank == 0) {
        printf("Result Pi: %.12f; Runge rule: EPS %e, n %d\n", sq[k] * sq[k], eps, n / 2);
    }

    MPI_Finalize();
    return 0;
}
```

Альтернатива

1. Глобальная сумма формируется в процессе 0 (MPI_Reduce)
2. Процесс 0 вычисляет погрешность и передает всем флаг (MPI_Bcast) – завершения работы или продолжение счета

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

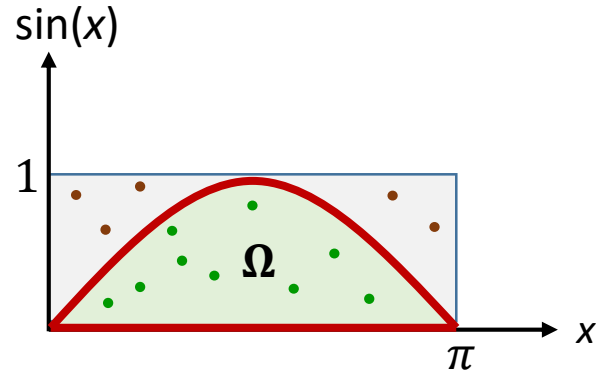
$$I \approx \frac{V}{n'} \sum_{i=1}^{n'} f(x_i, y_i)$$

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

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); }
```

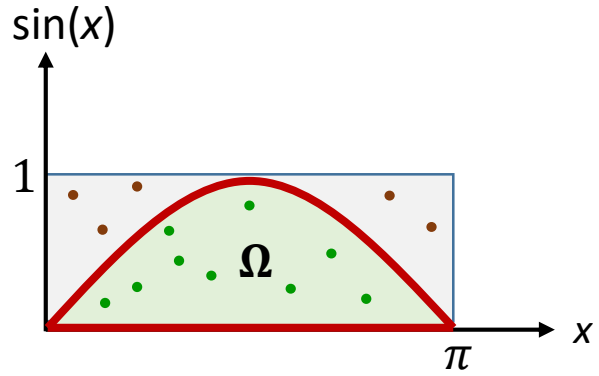


Параллельное вычисление двойного интеграла методом Монте-Карло

```
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* процессов генерирует и бросает *n* / *p* точек
2. Глобальная сумма формируется в корневом процессе



Параллельное вычисление двойного интеграла методом Монте-Карло

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, commsize;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    int in = 0;
    double s = 0;
    for (int i = rank; i < n; i += commsize) {
        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);
        }
    }
```

Каждый из *p* процессов генерирует и бросает *n* / *p* точек

Параллельное вычисление двойного интеграла методом Монте-Карло

```
    /* Продолжение … */

    int gin = 0;
    MPI_Reduce(&in, &gin, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    double gsum = 0.0;
    MPI_Reduce(&s, &gsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double v = PI * gin / n;
        double res = v * gsum / gin;
        printf("Result: %.12f, n %d\n", res, n);
    }

    MPI_Finalize();
    return 0;
}
```

Формируем суммы числа попаданий точек в область и значений функции в них

Параллельное вычисление двойного интеграла методом Монте-Карло

```
    /* Продолжение … */

    int gin = 0;
    MPI_Reduce(&in, &gin, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    double gsum = 0.0;
    MPI_Reduce(&s, &gsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double v = PI * gin / n;
        double res = v * gsum / gin;
        printf("Result: %.12f, n %d\n", res, n);
    }

    MPI_Finalize();
    return 0;
}
```

Формируем суммы числа попаданий точек в область и значений функции в них

Ошибка!

Все процессы вычислили одинаковые суммы **s**

```
$ mpiexec -np 6 ./integrate
Numerical integration by Monte Carlo method: n = 10000000
proc 5: s = 566379.693371
proc 1: s = 566379.693371
proc 0: s = 566379.693371
proc 4: s = 566379.693371
proc 2: s = 566379.693371
proc 3: s = 566379.693371
Result: 1.067600570303, n 10000000
```

Параллельное вычисление двойного интеграла методом Монте-Карло

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, commsize;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    int in = 0;
    double s = 0;
    for (int i = rank; i < n; i += commsize) {
        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);
        }
    }
```

- Все процессы проинициализировали генераторы псевдо-случайных чисел значением по умолчанию (см. man 3 rand)
- Во всех процессах генерируется одна и та же последовательность псевдо-случайных чисел – генерируются одни и те же точки

Параллельное вычисление двойного интеграла методом Монте-Карло

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, commsize;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

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

Инициализируем генератор целочисленным значением, уникальным для каждого процесса – его номером

```
$ mpiexec -np 6 ./integrate
Numerical integration by Monte Carlo method: n = 10000000
proc 1: s = 566379.693371
proc 5: s = 566172.989379
proc 0: s = 566379.693371
proc 3: s = 565738.049105
proc 2: s = 565649.159179
proc 4: s = 565971.946261
Result: 1.066976452219, n 10000000
```