Лекция 3
Стандарт MPI
Коллективные обмены

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

Коллективные обмены (Collective communications)

Трансляционный обмен (One-to-all)
- MPI_Bcast
- MPI_Scatter
- MPI_Scatterv

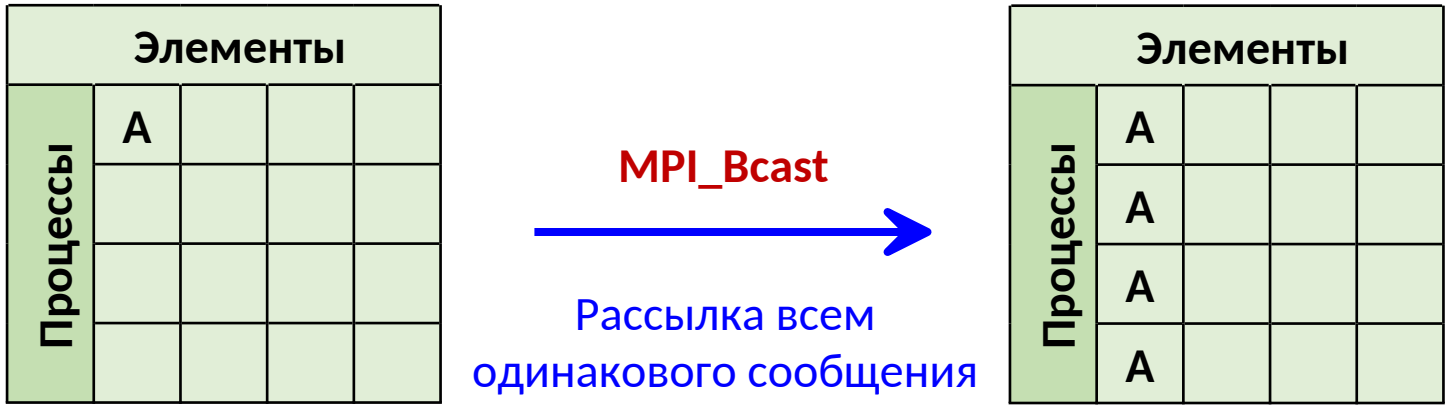
Коллекторный обмен (All-to-one)
- MPI_Gather
- MPI_Gatherv
- MPI_Reduce

Трансляционно-циклический обмен (All-to-all)
- MPI_Allgather
- MPI_Allgatherv
- MPI_Alltoall
- MPI_Alltoallv
- MPI_Allreduce
- MPI_Reduce_scatter

- Участвуют все процессы коммуникатора
- Коллективная функция должна быть вызвана каждым процессом коммуникатора
- Коллективные и двусторонние обмены в рамках одного коммуникатора используют различные контексты

MPI_Bcast

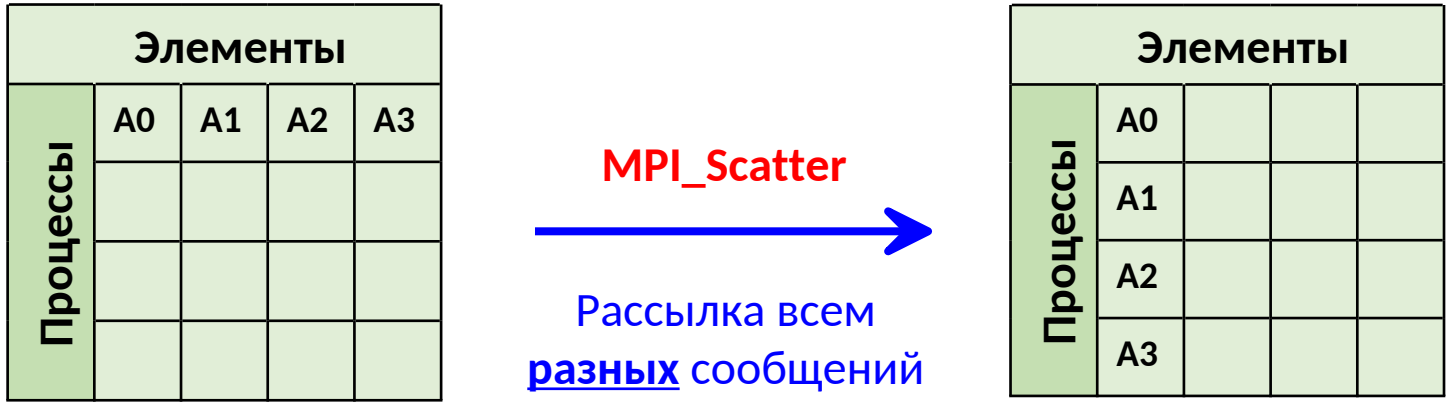
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm)
```



- **MPI_Bcast** – рассылка всем процессам сообщения buf
- Если номер процесса совпадает с root, то он отправитель, иначе – приемник

MPI_Scatter

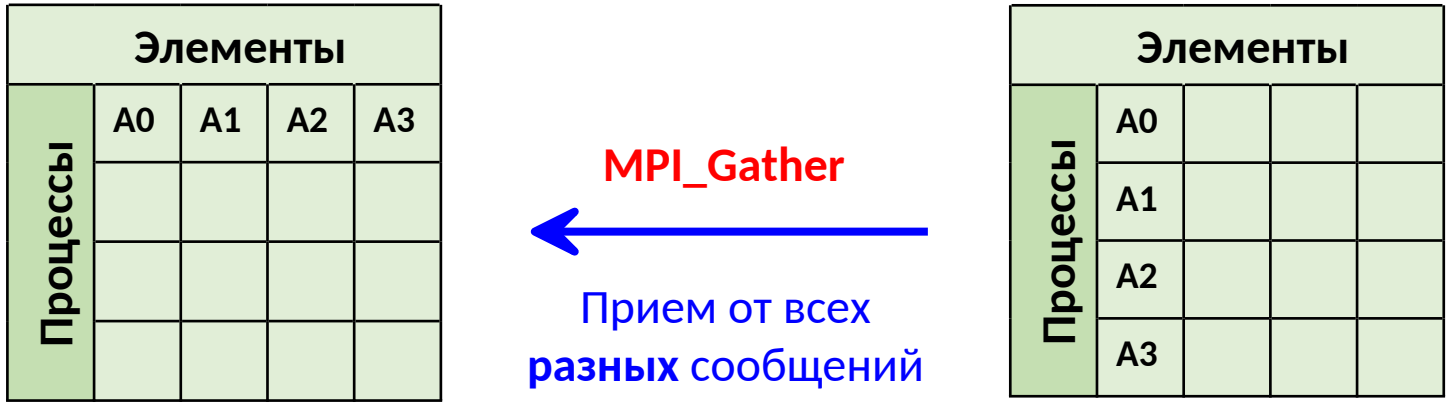
```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcnt,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```



- Размер **sendbuf** = sizeof(sendtype) * sendcnt * commsize
- Размер **recvbuf** = sizeof(sendtype) * recvcnt

MPI_Gather

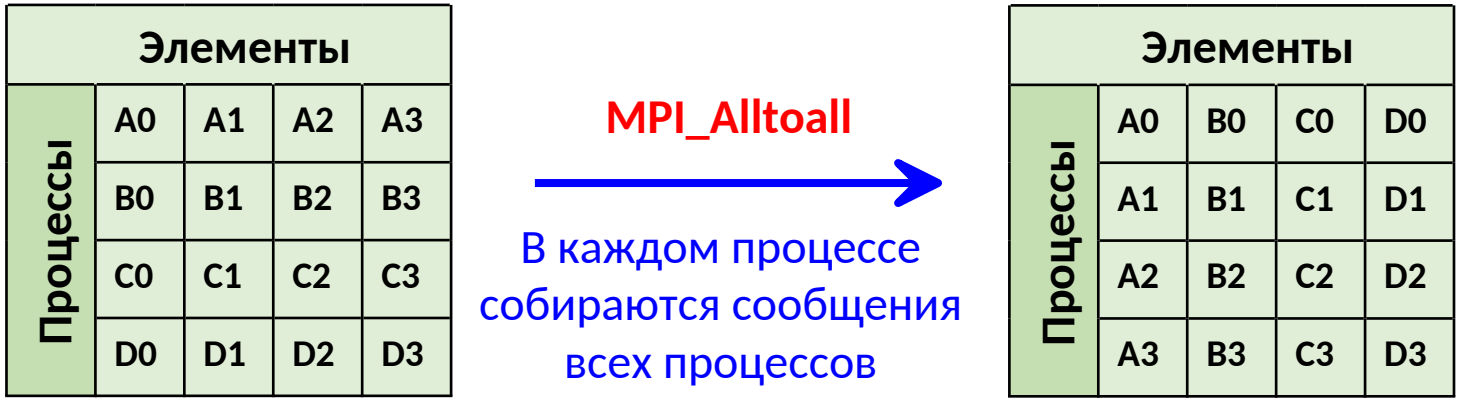
```
int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```



- Размер **sendbuf**: sizeof(sendtype) * sendcnt
- Размер **recvbuf**: sizeof(sendtype) * sendcnt * commsize

MPI_Alltoall

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 MPI_Comm comm)
```



- Размер sendbuf: sizeof(sendtype) * sendcount * commsize
- Размер recvbuf: sizeof(recvtype) * recvcount * commsize

All-to-all

```
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
                   MPI_Datatype sendtype,
                   void *recvbuf, int *recvcounts,
                   int *displs,
                   MPI_Datatype recvtype,
                   MPI_Comm comm)

int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

- Размер sendbuf: sizeof(datatype) * count
- Размер recvbuf: sizeof(datatype) * count

Операции MPI_Reduce

- MPI_MAX
- MPI_MIN
- MPI_MAXLOC
- MPI_MINLOC
- MPI_SUM
- MPI_PROD
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_BAND
- MPI_BOR
- MPI_BXOR

```
int MPI_Op_create(MPI_User_function *function,
                  int commute, MPI_Op *op)
```

- Операция пользователя должна быть ассоциативной
  A * (B * C) = (A * B) * C
- Если commute = 1, то операция коммутативная
  A * B = B * A

Барьерная синхронизация

```
int MPI_Barrier(MPI_Comm comm)
```

- Блокирует работу процессов коммуникатора, вызвавших данную функцию, до тех пор, пока все процессы не выполнят эту процедуру

Неблокирующие коллективные операции (MPI 3.0)

MPI 3.0

- **Неблокирующий коллективный обмен (Non-blocking collective communication)** – коллективная операция, выход из которой осуществляется не дожидаясь завершения операций обменов
- Пользователю возвращается дескриптор запроса (request), который он может использовать для проверки состояния операции
- Цель – обеспечить возможность совмещения вычислений и обменов информацией

```
int MPI_Ibcast(void *buf, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm,
               MPI_Request *request)
```

Неблокирующие коллективные операции (MPI 3.0)

```
MPI_Request req;

MPI_Ibcast(buf, count, MPI_INT, 0, MPI_COMM_WORLD, &req);
while (!flag) {

    // Вычисления...

    // Проверяем состояние операции
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Подсчет количества простых чисел (serial version)

```c
int is_prime_number(int n)
{
    int limit = sqrt(n) + 1;
    for (int i = 2; i <= limit; i++) {
        if (n % i == 0)
            return 0;
    }
    return (n > 1) ? 1 : 0;
}

int count_prime_numbers(int a, int b)
{
    int nprimes = 0;

    if (a <= 2) {
        nprimes = 1;      /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)       /* Shift 'a' to odd number */
        a++;

    /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
    for (int i = a; i <= b; i += 2) {
        if (is_prime_number(i))
            nprimes++;
    }
    return nprimes;
}
```

Определят, является ли число *n* простым *O*(sqrt(*n*))

Подсчитывает количество простых чисел в интервале [*a*, *b*]

a = 16 nprimes = 5 b = 35

16	**17**	18	**19**	20	21	22	**23**	24	25	26	27	28	**29**	30	**31**	32	33	34	35

Подсчет количества простых чисел (MPI version)

```
int is_prime_number(int n)
{
    int limit = sqrt(n) + 1;
    for (int i = 2; i <= limit; i++) {
        if (n % i == 0)
            return 0;
    }
    return (n > 1) ? 1 : 0;
}

int count_prime_numbers(int a, int b)
{
    int nprimes = 0;

    if (a <= 2) {
        nprimes = 1;      /* Count '2' as a prime number */
        a = 2;
    }
    if (a % 2 == 0)       /* Shift 'a' to odd number */
        a++;

    /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
    for (int i = a; i <= b; i += 2) {
        if (is_prime_number(i))
            nprimes++;
    }
    return nprimes;
}
```

Распределим итерации цикла между процессами MPI-программы

Подсчет количества простых чисел (MPI)

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int a = (argc > 1) ? atoi(argv[1]) : 1;
    int b = (argc > 2) ? atoi(argv[2]) : 1000000;

    MPI_Init(&argc, &argv);

    double t = MPI_Wtime();
    int n = count_prime_numbers_par(a, b);
    t = MPI_Wtime() - t;
    printf("Process %d/%d time: %.6f\n", get_comm_rank(), get_comm_size(), t);

    double tmax = 0;
    MPI_Reduce(&t, &tmax, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (get_comm_rank() == 0) {
        printf("Prime numbers on [%d, %d]: %d\n", a, b, n);
        printf("Execution time (sec): %.6f\n", tmax);
    }

    MPI_Finalize();
    return 0;
}
```

Подсчет количества простых чисел (MPI)

```
int count_prime_numbers_par(int a, int b)
{
    int nprimes = 0;
    int lb, ub;
    get_chunk(a, b, get_comm_size(), get_comm_rank(), &lb, &ub);

    /* Count '2' as a prime number */
    if (lb <= 2) {
        nprimes = 1;
        lb = 2;
    }

    /* Shift 'a' to odd number */
    if (lb % 2 == 0)
        lb++;

    /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
    for (int i = lb; i <= ub; i += 2) {
        if (is_prime_number(i))
            nprimes++;
    }

    int nprimes_global;
    MPI_Reduce(&nprimes, &nprimes_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    return nprimes_global;
}
```

Распределим итерации цикла между процессами MPI-программы
отрезок процесса – [lb, ub]

Подсчет количества простых чисел (MPI)

```
void get_chunk(int a, int b, int commsize, int rank, int *lb, int *ub)
{
    /*
     * This algorithm is based on OpenMP 4.0 spec (Sec. 2.7.1, default schedule for loops)
     * For a team of commsize processes and a sequence of n items, let ceil(n ? commsize) be the integer q
     * that satisfies n = commsize * q - r, with 0 <= r < commsize.
     * Assign q iterations to the first commsize - r processes, and q - 1 iterations to the remaining r processes.
     */
    int n = b - a + 1;
    int q = n / commsize;
    if (n % commsize)
        q++;
    int r = commsize * q - n;

    /* Compute chunk size for the process */
    int chunk = q;
    if (rank >= commsize - r)
        chunk = q - 1;

    /* Determine start item for the process */
    *lb = a;
    if (rank > 0) {
        /* Count sum of previous chunks */
        if (rank <= commsize - r)
            *lb += q * rank;
        else
            *lb += q * (commsize - r) + (q - 1) * (rank - (commsize - r));
    }
    *ub = *lb + chunk - 1;
}
```

Отрезок [a, b] разбивается на commsize частей

Подсчет количества простых чисел (MPI)

```
# 1 процесс (serial), кластер Jet
Prime numbers on [1, 10000000]: 664579
Execution time (sec.): 7.250906

# 8 процессов, кластер Jet
Process 0/8 time: 1.267749
Process 1/8 time: 0.641178
Process 2/8 time: 0.918755
Process 3/8 time: 0.915586
Process 4/8 time: 1.267748
Process 5/8 time: 1.104172
Process 6/8 time: 1.268977
Process 7/8 time: 1.266601
Prime numbers on [1, 10000000]: 664579
Execution time (sec): 1.268977
```

Ускорение в 5.7 раз

Процессы загружены вычислениями неравномерно (*load imbalance*)

Подсчет количества простых чисел (MPI)

```
int count_prime_numbers_par(int a, int b)
{
    int nprimes = 0;
    int lb, ub;
    get_chunk(a, b, get_comm_size(), get_comm_rank(), &lb, &ub);

    /* Count '2' as a prime number */
    if (lb <= 2) {
        nprimes = 1;
        lb = 2;
    }

    /* Shift 'a' to odd number */
    if (lb % 2 == 0)
        lb++;

    /* Loop over odd numbers: a, a + 2, a + 4, ... , b */
    for (int i = lb; i <= ub; i += 2) {
        if (is_prime_number(i))
            nprimes++;
    }

    int nprimes_global;
    MPI_Reduce(&nprimes, &nprimes_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    return nprimes_global;
}
```

Проблема 1

Неравномерная загрузка процессов

- Process 0: 0, 1, 2, 3
- Process 1: 4, 5, 6, 7
- Process 3: 8, 9, 10, 11

Подсчет количества простых чисел (MPI v2)

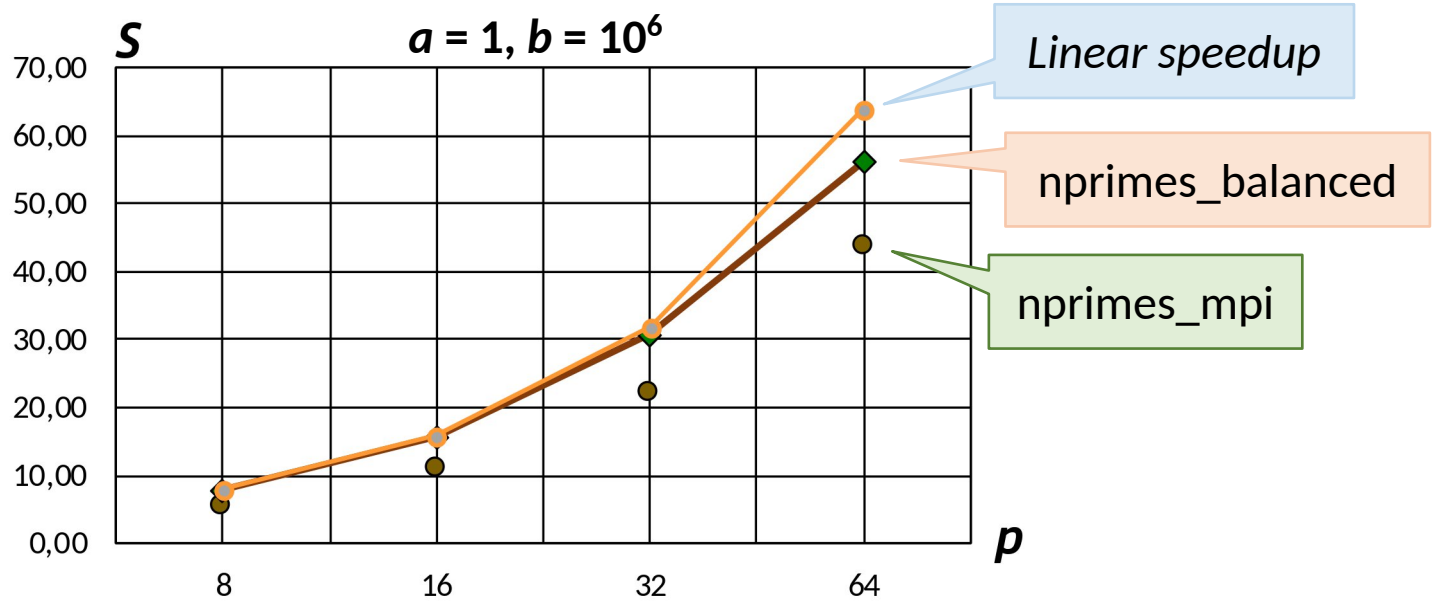
```
int roundup_to_odd(int a) { return (a % 2 == 0) ? a + 1 : a; }
int next_nth_odd(int a, int n) { /* assert: a % 2 != 0 */ return a + 2 * n; }

int count_prime_numbers_par(int a, int b)
{
    int nprimes = 0;

    /* Count '2' as a prime number */
    int commsize = get_comm_size();
    int rank = get_comm_rank();

    if (a <= 2) {
        a = 2;
        if (rank == 0)
            nprimes = 1;
    }

    a = roundup_to_odd(a);
    for (int i = next_nth_odd(a, rank); i <= b; i = next_nth_odd(i, commsize)) {
        /* i is odd number */
        if (is_prime_number(i))
            nprimes++;
    }
    int nprimes_global = 0;
    MPI_Reduce(&nprimes, &nprimes_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    return nprimes_global;
}
```

Циклическое (round-robin) распределение итераций по процессам

Экспериментальный анализ масштабируемости

- **Анализ строгой масштабируемости** (strong scaling) – при фиксированном размере входных данных выполняем измерения при различном числе процессов

1	***P* = 8**		***P* = 32**		…	***P* = 64**	
T_1	T_8	$S_8 = T_1 / T_8$	T_{32}	$S_{32} = T_1 / T_{32}$			$S_{64} = T_1 / T_{64}$



Вычисление числа π

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

    int n = 1000000000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    double h = 1.0 / (double)n;
    double sum = 0.0;
    for (int i = rank + 1; i <= n; i += commsize) {
        double x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    double pi_local = h * sum;

    double pi = 0.0;
    MPI_Reduce(&pi_local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) printf("PI is approximately %.16f\n", pi);
    MPI_Finalize();
    return 0;
}
```

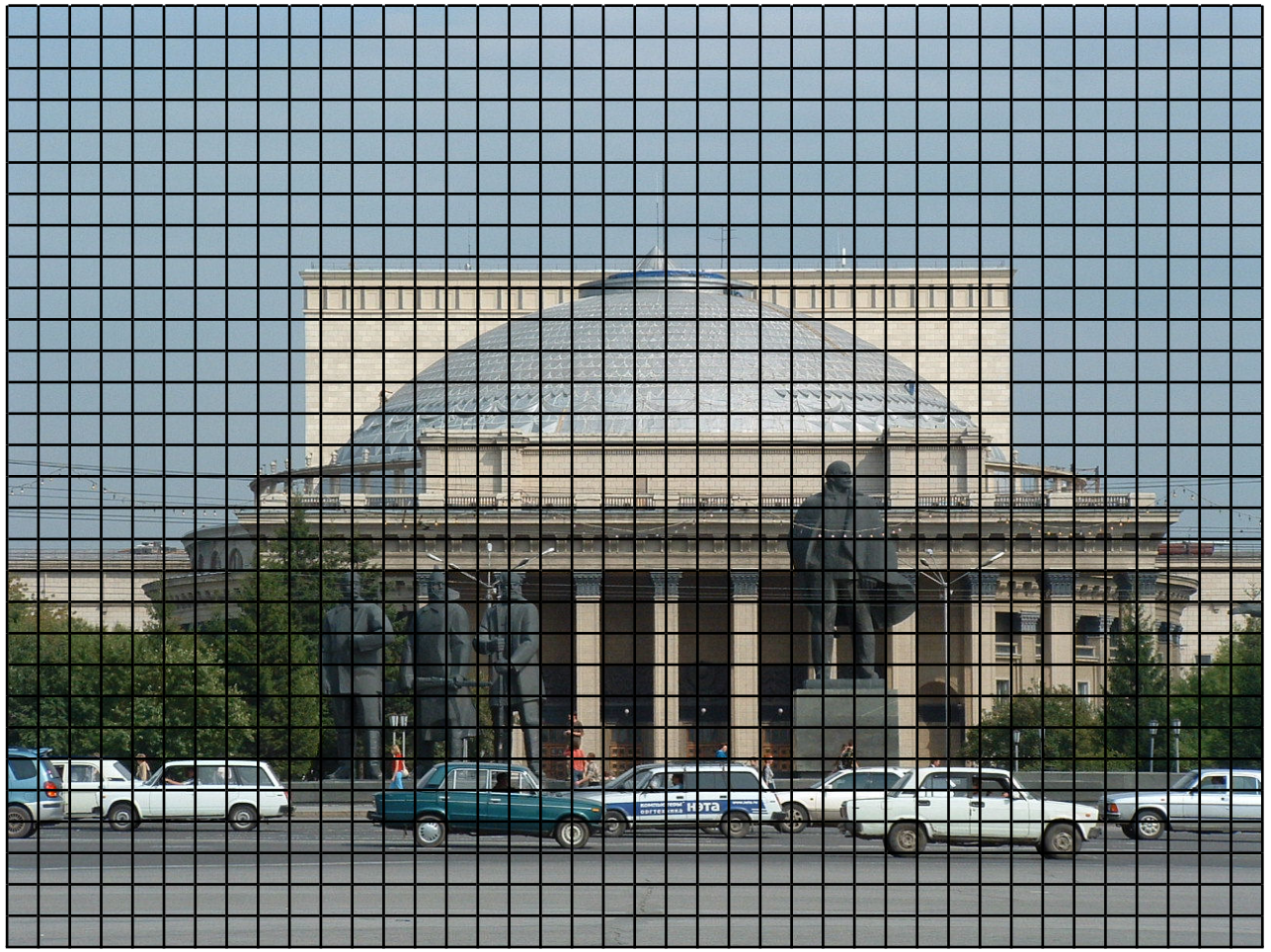
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

$$\pi \approx h \sum_{i=1}^{n} \frac{4}{1+(h(i-0.5))^2} \qquad h = \frac{1}{n}$$

Итерации циклически (round-robin) распределены межу процессами

Обработка изображения (contrast)

npixels = width * height;
npixels_per_process = npixels / commsize;



Обработка изображения (contrast)

```
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 width, height, npixels, npixels_per_process;
    uint8_t *pixels = NULL;
    if (rank == 0) {
        width = 15360;  // 15360 x 8640: 16K Digital Cinema (UHDTV) ~ 127 MiB
        height = 8640;
        npixels = width * height;
        pixels = xmalloc(sizeof(*pixels) * npixels);
        for (int i = 0; i < npixels; i++)
            pixels[i] = rand() % 255;
    }

    MPI_Bcast(&npixels, 1, MPI_INT, 0, MPI_COMM_WORLD);  // Send size of image
    npixels_per_process = npixels / commsize;
    uint8_t *rbuf = xmalloc(sizeof(*rbuf) * npixels_per_process);
    // Send a part of image to each process
    MPI_Scatter(pixels, npixels_per_process, MPI_UINT8_T, rbuf, npixels_per_process,
                MPI_UINT8_T, 0, MPI_COMM_WORLD);
```

Обработка изображения (contrast, 2)

```
int sum_local = 0;
for (int i = 0; i < npixels_per_process; i++)
    sum_local += rbuf[i] * rbuf[i];

/* Calculate global sum of the squares */
int sum = 0;
// MPI_Reduce(&sum_local, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Allreduce(&sum_local, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

double rms;
// if (rank == 0)
rms = sqrt((double)sum / (double)npixels);

//MPI_Bcast(&rms, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Обработка изображения (contrast, 3)

```
    /* Contrast operation on subimage */
    for (int i = 0; i < npixels_per_process; i++) {
        int pixel = 2 * rbuf[i] - rms;
        if (pixel < 0)
            rbuf[i] = 0;
        else if (pixel > 255)
            rbuf[i] = 255;
        else
            rbuf[i] = pixel;
    }
    MPI_Gather(rbuf, npixels_per_process, MPI_UINT8_T, pixels,
               npixels_per_process, MPI_UINT8_T, 0, MPI_COMM_WORLD);
    if (rank == 0)
        // Save image...

    free(rbuf);
    if (rank == 0)
        free(pixels);
    MPI_Finalize();
}
```