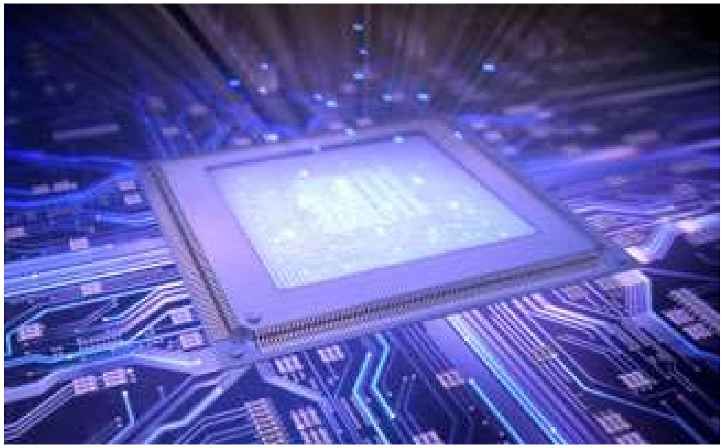
NYU
New York University

Parallel Computing

MPI - III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adopted and slightly modified from:
- Gerassimos Barlas
- Peter S. Pacheco



```
int MPI_Reduce(
      void*          input_data_p   /* in  */,
      void*          output_data_p  /* out */,
      int            count          /* in  */,
      MPI_Datatype   datatype       /* in  */,
      MPI_Op         operator       /* in  */,
      int            dest_process   /* in  */,
      MPI_Comm       comm           /* in  */);
```

```
int MPI_Allreduce(
      void*          input_data_p   /* in  */,
      void*          output_data_p  /* out */,
      int            count          /* in  */,
      MPI_Datatype   datatype       /* in  */,
      MPI_Op         operator       /* in  */,
      MPI_Comm       comm           /* in  */);
```

```
int MPI_Bcast(
      void*          data_p         /* in/out */,
      int            count          /* in     */,
      MPI_Datatype   datatype       /* in     */,
      int            source_proc    /* in     */,
      MPI_Comm       comm           /* in     */);
```

Collective

point-to-point

```
int MPI_Send(

      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```

Data distributions

$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0+y_0, x_1+y_1, \ldots, x_{n-1}+y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Sequential version

Different partitions of a 12-component vector among 3 processes

Process	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

- **Block:** Assign blocks of consecutive components to each process.
- **Cyclic:** Assign components in a round robin fashion.
- **Block-cyclic:** Use a cyclic distribution of blocks of components.

Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double   local_x[]  /* in   */,
      double   local_y[]  /* in   */,
      double   local_z[]  /* out  */,
      int      local_n    /* in   */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

How will you distribute parts of x[] and y[] to processes?

Scatter

- Read an entire vector on process 0
- MPI_Scatter sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            src_proc     /* in   */,
      MPI_Comm       comm         /* in   */);
```

data items going to each process

Important:

- All arguments are important for the source process (process 0 in our example)
- For all other processes, only recv_buf_p, recv_count, recv_type, src_proc, and comm are important

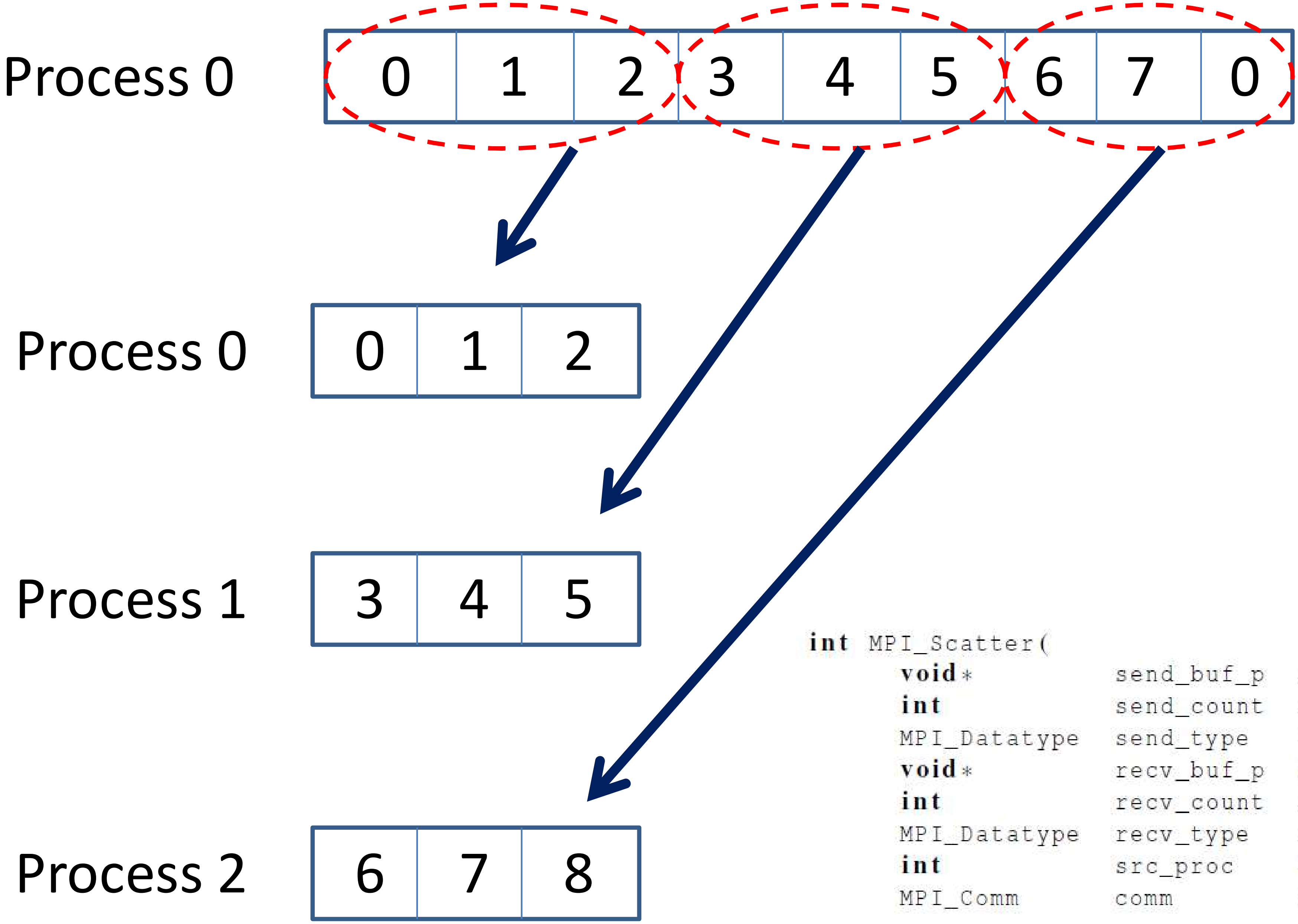
Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

process 0 itself
also receives data.

```
int MPI_Scatter(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            src_proc     /* in   */,
      MPI_Comm       comm         /* in   */);
```

- send_buf_p
  - is not used except by the sender.
  - However, it must be defined or NULL on others to make the code correct.
  - Must have at least communicator size * send_count elements
- All processes must call MPI_Scatter, not only the sender.
- send_count the number of data items sent to each process.
- recv_buf_p must have at least send_count elements
- MPI_Scatter uses block distribution




```
int MPI_Scatter(
        void*          send_buf_p   /* in  */,
        int            send_count   /* in  */,
        MPI_Datatype   send_type    /* in  */,
        void*          recv_buf_p   /* out */,
        int            recv_count   /* in  */,
        MPI_Datatype   recv_type    /* in  */,
        int            src_proc     /* in  */,
        MPI_Comm       comm         /* in  */);
```

Gather

- MPI_Gather collects all of the components of the vector onto process dest process, ordered in rank order.

```
int MPI_Gather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */
      MPI_Datatype   recv_type    /* in  */,
      int            dest_proc    /* in  */,
      MPI_Comm       comm         /* in  */);
```

number of elements in send_buf_p

number of elements for any single receive

Important:

- All arguments are important for the destination process.
- For all other processes, only send_buf_p, send_count, send_type, dest_proc, and comm are important

Print a distributed vector (1)

```
void Print_vector(
        double    local_b[]  /* in */,
        int       local_n    /* in */,
        int       n          /* in */,
        char      title[]    /* in */,
        int       my_rank    /* in */,
        MPI_Comm  comm       /* in */) {

    double* b = NULL;
    int i;
```

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Print_vector */
```

Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Matrix-vector multiplication

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;

   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Pseudo-code Serial Version

C style arrays

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Serial matrix-vector multiplication

```
void Mat_vect_mult(
        double  A[]  /* in  */,
        double  x[]  /* in  */,
        double  y[]  /* out */,
        int     m    /* in  */,
        int     n    /* in  */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}  /* Mat_vect_mult */
```

Let's assume x[] is distributed among the different processes

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
      double     local_A[]  /* in   */,
      double     local_x[]  /* in   */,
      double     local_y[]  /* out  */,
      int        local_m    /* in   */,
      int        n          /* in   */,
      int        local_n    /* in   */,
      MPI_Comm   comm       /* in   */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

Assuming x was
Scattered among the processes.

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

Keep in mind …

- In distributed memory systems, communication is more expensive than computation.
- Distributing a fixed amount of data among several messages is more expensive than sending a single big message.

Derived datatypes

- Used to represent any collection of data items
- If a function that sends data knows this information about the collection of data items, it can collect the items from memory before they are sent.
- A function that receives data can distribute the items into their correct destinations in memory when they're received.

Derived datatypes

- A sequence of basic MPI data types together with a displacement for each of the data types.

Address in memory where the variables are stored
a and b are double; n is int

Variable	Address
a	24
b	40
n	48

{(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)}

displacement from the beginning of the type
(We assume we start with a.)

MPI_Type create_struct

- Builds a derived datatype that consists of individual elements that have different basic types.

```
int MPI_Type_create_struct(
      int              count                      /* in  */,
      int              array_of_blocklengths[]    /* in  */,
      MPI_Aint         array_of_displacements[]   /* in  */,
      MPI_Datatype     array_of_types[]           /* in  */,
      MPI_Datatype*    new_type_p                 /* out */);
```

Number of elements in the type

an integer type that is big enough to store an address on the system.

From the address of item 0

```
int MPI_Get_address(
      void*        location_p  /* in  */,
      MPI_Aint*    address_p   /* out */);
```

Before you start using your new data type

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

When you are finished with your new type

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

This frees any additional storage used.

Example (1)

```
void Build_mpi_type(
      double*        a_p            /* in  */,
      double*        b_p            /* in  */,
      int*           n_p            /* in  */,
      MPI_Datatype*  input_mpi_t_p  /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};
```

Example (2)

```
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[1] = b_addr-a_addr;
    array_of_displacements[2] = n_addr-a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
          array_of_displacements, array_of_types,
          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */
```

Example (3)

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

The receiving end can use the received complex data item as if it is a structure.

MEASURING TIME IN MPI

We have seen in the past …

- *time* in Linux
- *clock()* inside your code
- Does MPI offer anything else?

Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);
```

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

Elapsed time for the calling process

Important

- MPI_Wtime() returns wall clock time.
  - So, includes any idle time.
- clock() returns CPU time.

How to Sync Processes?

MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

Let's see how we can analyze the performance of an MPI program

The matrix-vector multiplication

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish − local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Conclusions

- Reducing messages sent is a good performance strategy!
  - Collective vs point-to-point
- Distributing a fixed amount of data among several messages is more expensive than sending a single big message.