Homework 2 – Exercise MPI_exa

Rainer Karl Trummer
rtrummer@cosy.sbg.ac.at

August 8, 2004

Problem Description

Write a program to measure the time it takes to send $1, 2, 4, \ldots, 1$ Mega C-doubles from one processor to another using `MPI_Send` and `MPI_Recv`. Use more trials to average out variations and overhead in `MPI_Wtime`.
Print the size, time, and rate in MB/sec for each test. Describe how the program works.

Problem Solution

The used program represents a modified version of the provided example benchmark program. Most of the modifications were applied to optimize program execution and reduce some time-consuming overhead.

Program Description

The program starts with the usual MPI initialization followed by some checks required for proper execution. Afterwards, the message buffer is allocated, adapted to the largest message exchanged during the test. Since multiple calls to `malloc()` can highly reduce every program's performance, this modification of replacing individual buffer allocations with an initial one helps to improve overall performance. Further improvements are achieved by storing all measurement results in the buffer during the test phase, rather than immediately printing them out. This also allows an exchange of the test's loop order, so that the complete procedure of sending and receiving $1, 2, 4, \ldots, 1$ Mega C-doubles can be repeated several times, rather than performing several repetitions for each individual message size. In this sense, the outer loop performs the number of desired test repetitions, whereas the inner loop iterates over the number of C-doubles to be exchanged. Before this nested loop is executed, the specific buffer locations used to hold the measurement results are initialized to the largest possible C-double value. Within the nested loop, an `if`-statement is used to determine which of the two processes is executing the current code. In case of process 0, a message is sent to process 1 via `MPI_Send` followed by an `MPI_Recv` call to complete the message's round trip, whose duration time is measured by `MPI_Wtime`. In case of process 1, which merely needs to close the round trip loop, `MPI_Recv` is called first followed by `MPI_Send`. Upon completion of the nested loop, all results are printed out in tabular form and are written to files for further visualization with *Mathematica*.

Results

Program Output

The following tabular results were obtained from running the program on the network at the Department of Computer Science (see Appendix for used program).

```
alpaka:~/hpsc> mpirun.mpich -np 2 hw2
```

Data (Byte)	Time (sec)	Rate (MB/sec)
8	0.000109	0.587156
16	0.000101	1.267327
32	0.000105	2.438095
64	0.000114	4.491228
128	0.000130	7.876923
256	0.000163	12.564417
512	0.000227	18.044053
1024	0.000356	22.978962
2048	0.000513	31.968780
4096	0.000693	47.284271
8192	0.001051	62.326201
16384	0.001737	75.458837
32768	0.003163	82.865181
65536	0.006029	86.968234
131072	0.011974	87.567414
262144	0.023517	89.177896
524288	0.046427	90.341913
1048576	0.092418	90.768610
2097152	0.183855	91.252433
4194304	0.367782	91.234568
8388608	0.736510	91.117445

Output Visualization

The following four figures that visualize the above results were obtained by *Mathematica*. Figure 1 shows the required time to send $1, 2, 4, \ldots, 1$ Mega C-doubles from one processor to another. It points out that this transfer time grows linearly with the amount of sent data over the entire tested range. Since the message size doubles during the test with each new message, i.e., grows exponentially, an equally spaced representation of these values, so that they correspond to the exponent of the 2^x data size, clearly reflects this circumstance, as illustrated in Figure 2. In contrast, the transfer rate grows logarithmically with the amount of sent data, as can be seen in Figure 3. It points out that a nearly constant rate is achieved as soon as the message size exceeds 524288 Byte. Representing these values equally spaced corresponding to the 2^x data size, as shown in Figure 4, reveals the true slope of the transfer rate, which grows exponentially for message sizes below 2^{10} Byte.

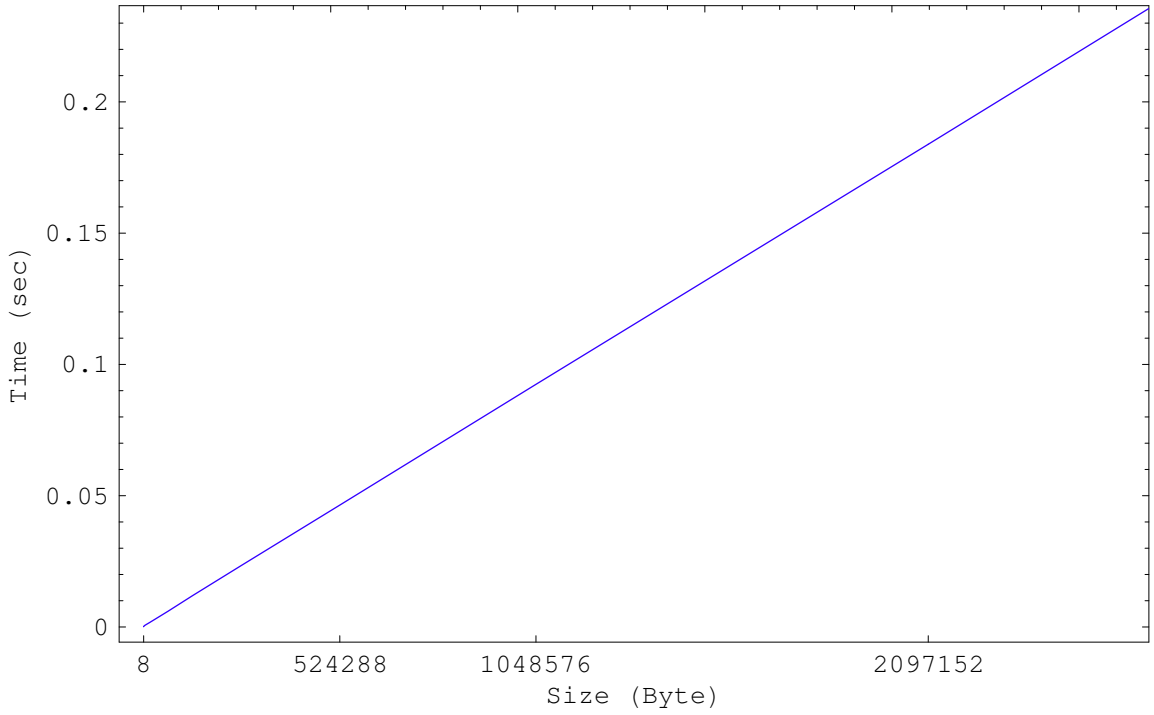

Figure 1: Transfer time for exchanging $1, 2, 4, \ldots, 1$ Mega C-doubles.

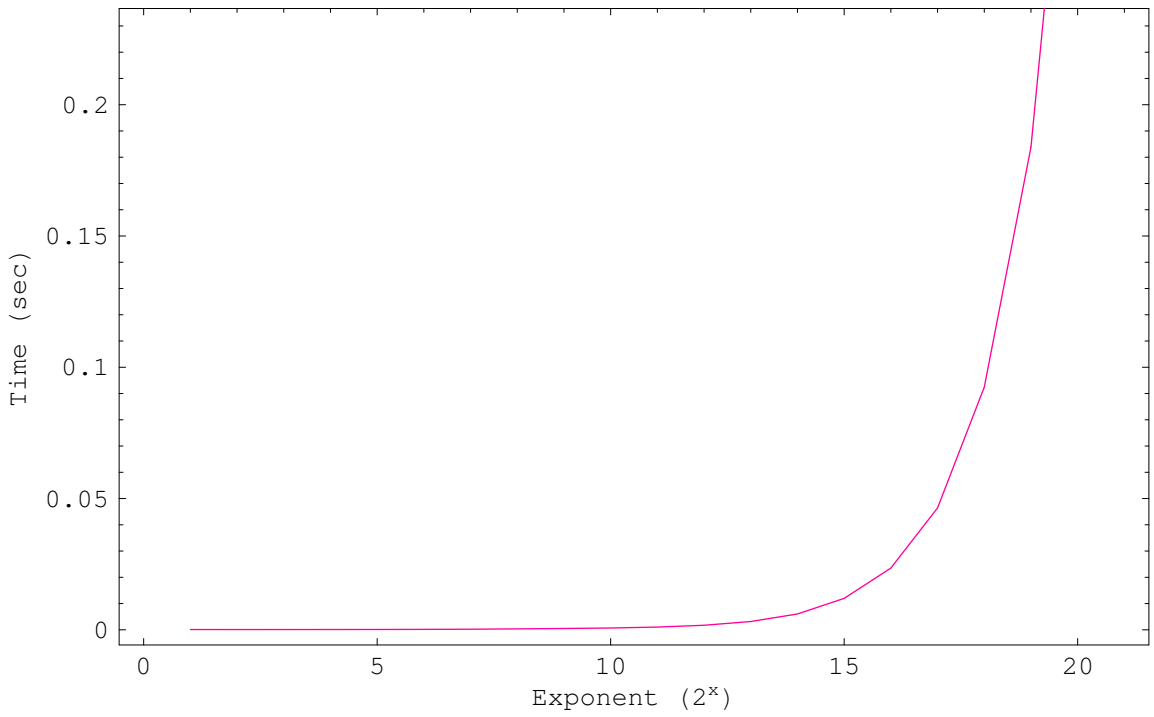

Figure 2: Equally spaced transfer time corresponding to 2^x data size.

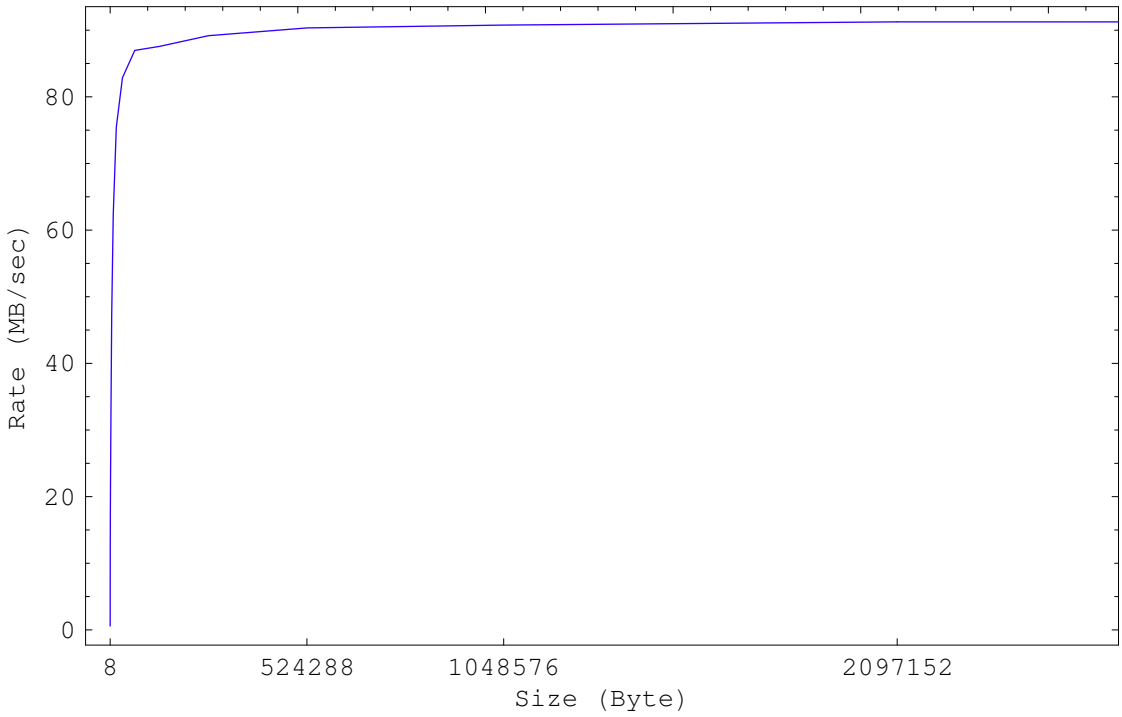

Figure 3: Transfer rate for exchanging $1, 2, 4, \ldots, 1$ Mega C-doubles.

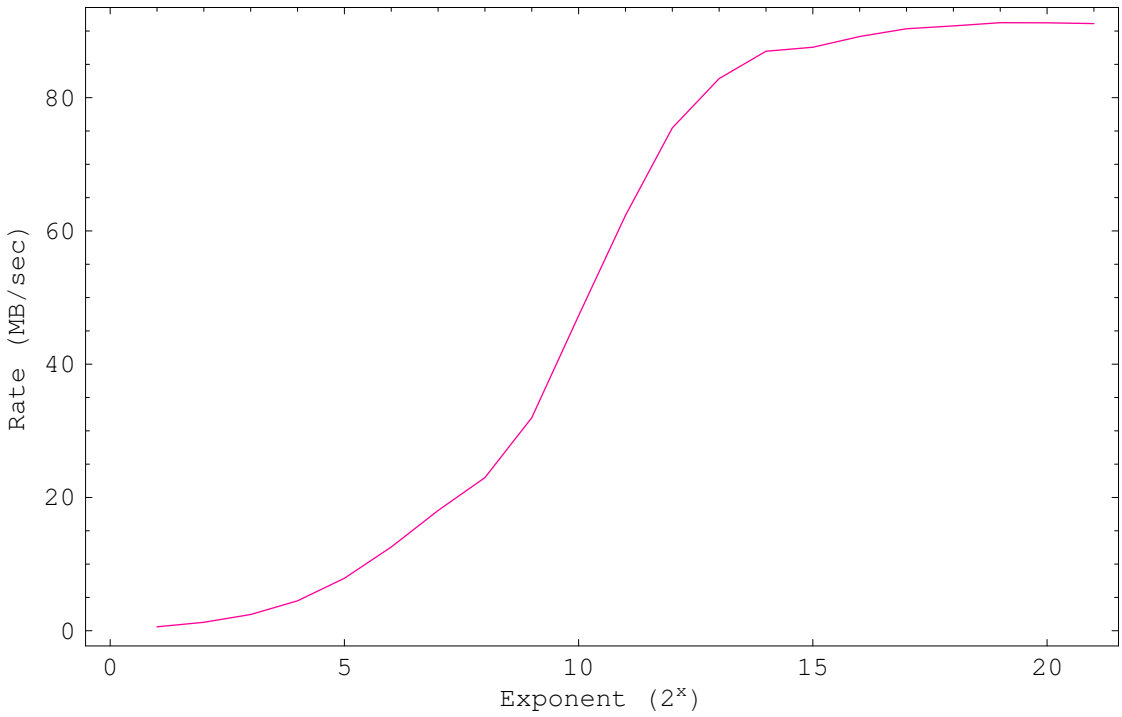

Figure 4: Equally spaced transfer rate corresponding to 2^x data size.

Appendix

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define MAX_DOUBLES 1048576
#define MAX_UPDATES 10

#ifdef _WIN32
  #define SEPARATOR '\\'
#else
  #define SEPARATOR '/'
#endif


int main( int argc, char **argv )
{
    int        size, rank, data, k, n;
    double     *buf, time, rate;
    MPI_Status status;

    if( MPI_Init( &argc, &argv ) != MPI_SUCCESS )
    {
        printf( "MPI initialization failed!\n" );
        return( 1 );
    }

    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if( size != 2 )
    {
        printf( "Number of processes must be 2!\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    if( (buf = (double *) malloc( MAX_DOUBLES * sizeof( double ) )) == NULL )
    {
        printf( "Buffer allocation at process %d failed!\n", rank );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    if( rank == 0 )
    {
        for( n = 1; n <= MAX_DOUBLES; n <<= 1 )
        {
            buf[n-1] = 1.7e+308;
        }
    }

    for( k = 0; k < MAX_UPDATES; ++k )
    {
        for( n = 1; n <= MAX_DOUBLES; n <<= 1 )
        {
```

```
        if( rank == 0 )
        {
            MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, MAX_UPDATES,
                          MPI_BOTTOM, 0, MPI_INT, 1, MAX_UPDATES,
                          MPI_COMM_WORLD, &status );

            time = MPI_Wtime( );
            MPI_Send( buf, n, MPI_DOUBLE, 1, n, MPI_COMM_WORLD );
            MPI_Recv( buf, n, MPI_DOUBLE, 1, n, MPI_COMM_WORLD, &status );
            time = MPI_Wtime( ) - time;

            if( time < buf[n-1] )
            {
                buf[n-1] = time;
            }
        }
        else
        if( rank == 1 )
        {
            MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, MAX_UPDATES,
                          MPI_BOTTOM, 0, MPI_INT, 0, MAX_UPDATES,
                          MPI_COMM_WORLD, &status );

            MPI_Recv( buf, n, MPI_DOUBLE, 0, n, MPI_COMM_WORLD, &status );
            MPI_Send( buf, n, MPI_DOUBLE, 0, n, MPI_COMM_WORLD );
        }
    }
}

if( rank == 0 )
{
    char *p, time_path[256], rate_path[256];
    char *time_name = "time.dat", *rate_name = "rate.dat";
    FILE *time_file, *rate_file;

    strcpy( time_path, argv[0] );
    strcpy( rate_path, argv[0] );

    if( (p = strrchr( time_path, SEPARATOR )) != NULL )
    {
        strcpy( p + 1, time_name );
        time_name = time_path;
    }

    if( (p = strrchr( rate_path, SEPARATOR )) != NULL )
    {
        strcpy( p + 1, rate_name );
        rate_name = rate_path;
    }

    if( (time_file = fopen( time_name, "wt" )) == NULL )
    {
        printf( "Cannot open file %s!\n", time_name );
    }
    else
    if( (rate_file = fopen( rate_name, "wt" )) == NULL )
```

```
        {
            printf( "Cannot open file %s!\n", rate_name );
            fclose( time_file );
        }
        else
        {
            printf( "Data (Byte)\tTime (sec)\tRate (MB/sec)\n\n" );

            for( n = 1; n <= MAX_DOUBLES; n <<= 1 )
            {
                time = buf[n-1] / 2.0;
                data = n * sizeof( double );
                rate = (double) data * 8.0e-6 / time;
                fprintf( time_file, "%d,%lf\n", data, time );
                fprintf( rate_file, "%d,%lf\n", data, rate );
                printf( "%10d\t%10.6lf\t%10.6lf\n", data, time, rate );
            }

            fclose( time_file );
            fclose( rate_file );
        }
    }

    free( buf );
    MPI_Finalize( );
    return( 0 );
}
```