Homework 3 – Exercise 2

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

August 14, 2004

Problem Description

Implement a two-dimensional finite difference algorithm on a square domain of dimension $n \times n = N$ for a 5-point stencil using MPI. Measure performance on one to four parallel computers and use performance models to explain your results. Plot execution time as a function of the number of points (N) and as a function of the number of processors (P) for N up to 100×100 and P up to 4 processors and for 10^4 time steps. See the example "A simple Jacobi iteration" for assistance.

Problem Solution

Computation Time

$$n \times n \text{ grid} \;\Rightarrow\; T_{\text{comp}} = t_c n^2$$

2-D Decomposition

$$4n \text{ channels, } 8\frac{n}{\sqrt{P}} \text{ messages} \;\Rightarrow\; T_{\text{comm}} = t_s 4P + t_w 8n\sqrt{P}$$

$$T_{\text{2-D}} = \frac{T_{\text{comp}} + T_{\text{comm}}}{P} = \frac{t_c n^2 + t_s 4P + t_w 8n\sqrt{P}}{P}$$

$$\Rightarrow\; T_P(N) = S\left(\frac{t_c N}{P} + t_s 4 + t_w 8\sqrt{\frac{N}{P}}\right), \text{ where } S \text{ is the number of time steps}$$

Measured Values

The following values were obtained from running two simple programs on the network at the Department of Computer Science (see Appendix for used programs).

$t_c = 0.0000000709$ sec

$t_s = 0.0000890850$ sec

$t_w = 0.0000024400$ sec

Computation Models

To make problem size N dividable for process sizes $1, \ldots, 4$ without leaving a remainder, grid dimensions 12×12, 24×24, 48×48, and 96×96 are used throughout all computations.

$$T_1(N) = S\left(t_c N\right) = 10^4 \left(t_c \begin{bmatrix} 144 \\ 576 \\ 2304 \\ 9216 \end{bmatrix} \right) = \begin{bmatrix} 0.1021 \\ 0.4084 \\ 1.6335 \\ 6.5341 \end{bmatrix} \text{ sec}$$

$$T_2(N) = 10^4 \left(\frac{t_c N}{2} + t_s 4 + t_w 8 \sqrt{\frac{N}{2}} \right) = \begin{bmatrix} 5.2708 \\ 7.0802 \\ 11.0055 \\ 20.0811 \end{bmatrix} \text{ sec}$$

$$T_3(N) = 10^4 \left(\frac{t_c N}{3} + t_s 4 + t_w 8 \sqrt{\frac{N}{3}} \right) = \begin{bmatrix} 4.9498 \\ 6.4043 \\ 9.5175 \\ 16.5605 \end{bmatrix} \text{ sec}$$

$$T_4(N) = 10^4 \left(\frac{t_c N}{4} + t_s 4 + t_w 8 \sqrt{\frac{N}{4}} \right) = \begin{bmatrix} 4.7601 \\ 6.0079 \\ 8.6566 \\ 14.5665 \end{bmatrix} \text{ sec}$$

Results

Program Output

The following results were obtained from running the finite difference program and the two simple programs on the network at the Department of Computer Science (see Appendix for used programs).

```
assapan:~/hpsc> mpirun.mpich -np 1 hw3
Size    Steps   Difference      Time (sec)

 144    10000   0.000000        0.102032
 576    10000   0.000000        0.357568
2304    10000   0.000000        1.662068
9216    10000   0.008967        7.237296

assapan:~/hpsc> mpirun.mpich -np 2 hw3
Size    Steps   Difference      Time (sec)

 144    10000   0.000000        2.942824
 576    10000   0.000000        3.468608
2304    10000   0.000000        4.998781
9216    10000   0.008967        9.497605
```

```
assapan:~/hpsc> mpirun.mpich -np 3 hw3
Size    Steps   Difference      Time (sec)

 144    10000   0.000000        5.037264
 576    10000   0.000000        5.453506
2304    10000   0.000000        7.445599
9216    10000   0.008967        11.164356

assapan:~/hpsc> mpirun.mpich -np 4 hw3
Size    Steps   Difference      Time (sec)

 144    10000   0.000000        5.234124
 576    10000   0.000000        5.178204
2304    10000   0.000000        6.296845
9216    10000   0.008967        9.096694

assapan:~/hpsc> mpirun.mpich -np 1 tc
tc = 0.0000000709 sec

assapan:~/hpsc> mpirun.mpich -np 2 tstw
ts = 0.0000890850 sec
tw = 0.0000024400 sec
```

Output Visualization

The following four figures that visualize the above results were obtained by *Mathematica*. Figure 1 shows the modelled time required to determine the finite difference, as a function of N, for running the program on $1, \ldots, 4$ processors. In Figure 2 below, the corresponding measured time can be seen, which differs not only in the four slopes, but also in the amount of actually consumed execution time. Compared to the modelled time, it points out that the measured time of using four processors is much lower at the beginning for small sizes of N, whereas for large sizes of N it is still below the time of using three processors. As a general observation, the four curves of the modelled time clearly diverge from each other, whereas the three curves representing $1, \ldots, 3$ processors of the measured time appear to converge to some exterior point. Figure 3 shows the modelled time, as a function of P, for problem sizes 144, 576, 2304, and 9216, which is very similar to Figure 4 representing the corresponding measured time. Compared to the modelled time, which predicts a descent of execution time for all four sizes of N already at the use of 3 processors, the measured time reveals that in practise this descent does not occur before involving at least 4 processors. This effect comes out even more essentially the smaller the problem size, as depicted by the bottom curve that keeps raising in the rightmost third.

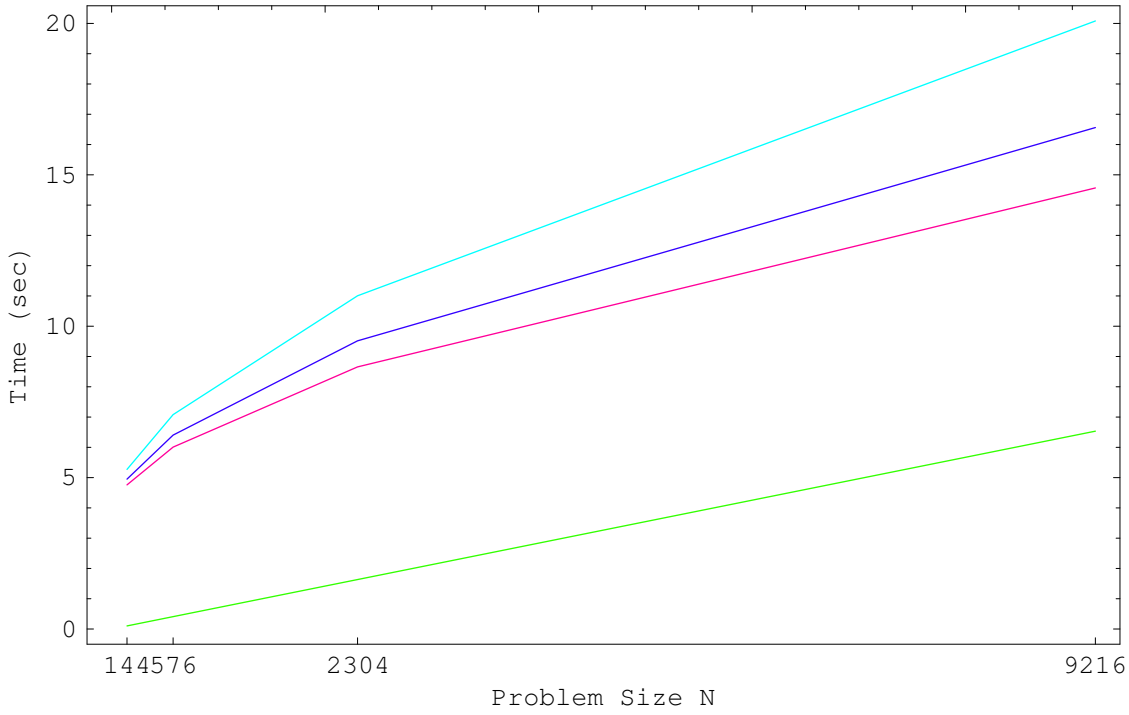

Figure 1: Model data for P_1 (green), P_2 (cyan), P_3 (blue), and P_4 (red).

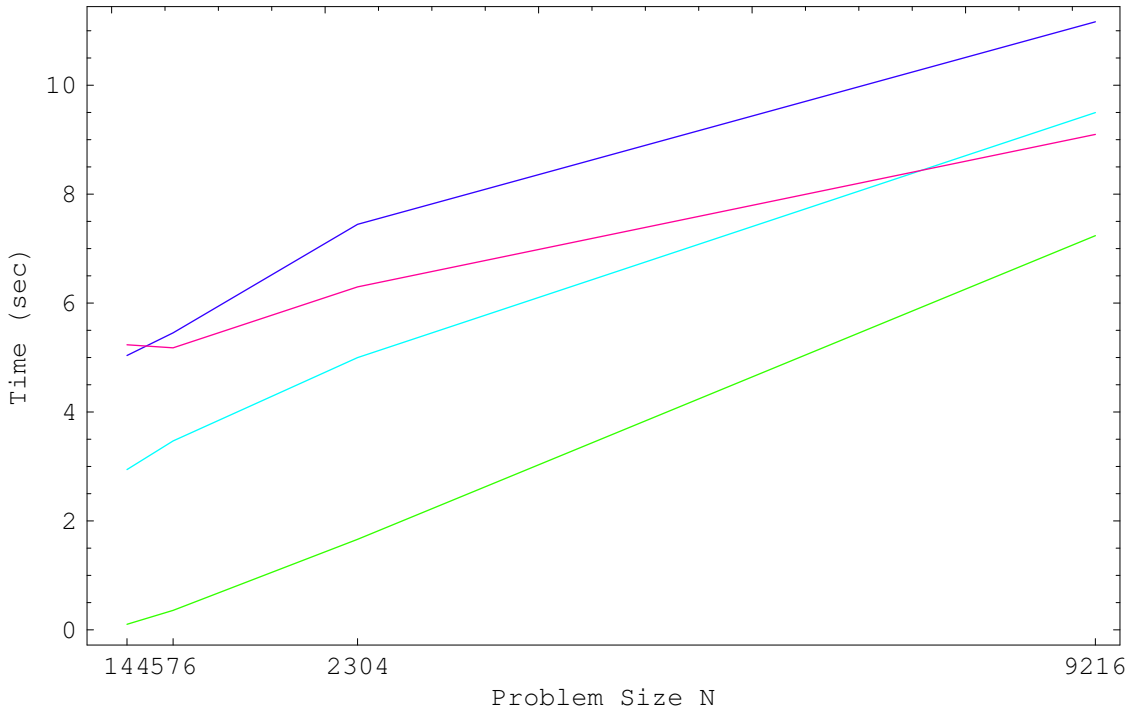

Figure 2: Real data for P_1 (green), P_2 (cyan), P_3 (blue), and P_4 (red).

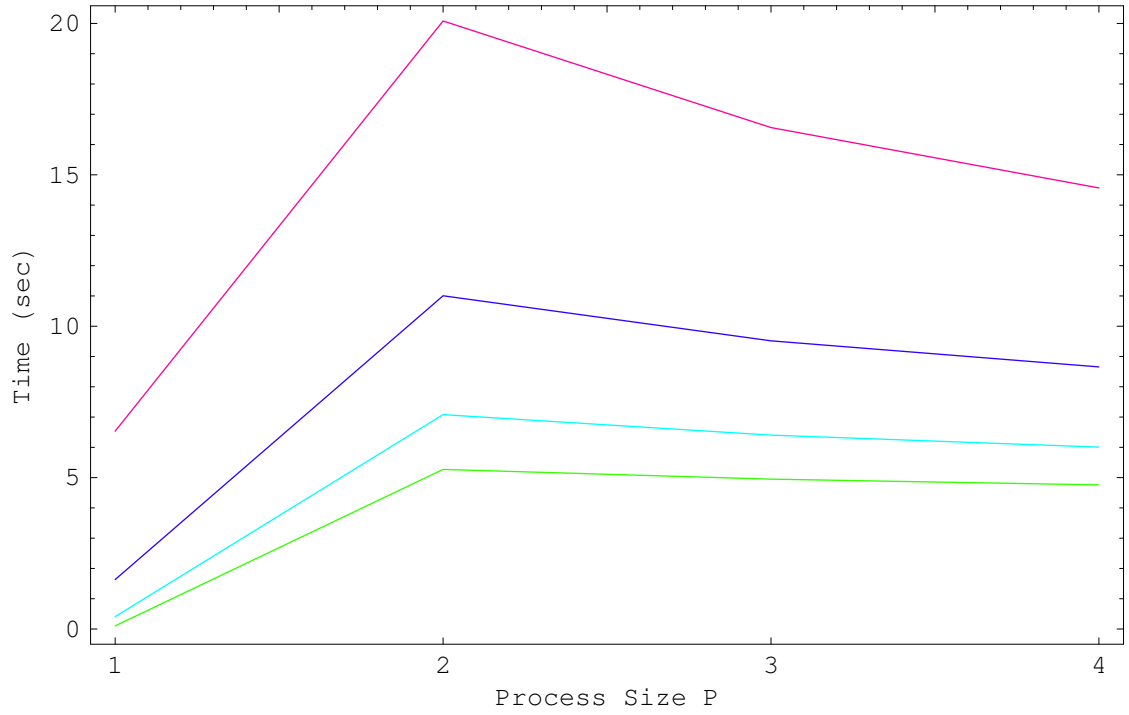

Figure 3: Model data for N_{144} (green), N_{576} (cyan), N_{2304} (blue), and N_{9216} (red).

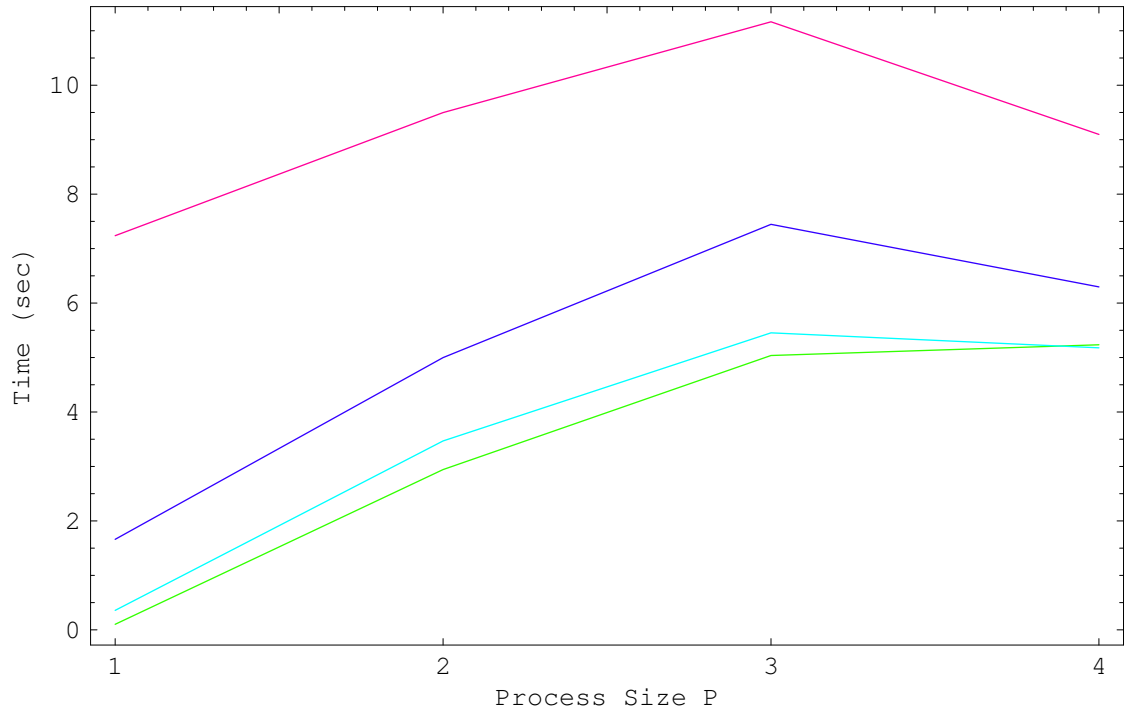

Figure 4: Real data for N_{144} (green), N_{576} (cyan), N_{2304} (blue), and N_{9216} (red).

Appendix

Program to determine the Finite Difference

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

#define MAX_N     96
#define INITIAL   100.0
#define MAX_STEPS 10000

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


int main( int argc, char **argv )
{
    char       *p, data_name[256];
    int        size, rank, steps, i_max, i_bot, i_top, i, j, n;
    double     **x, **g_x, norm, g_norm, time;
    FILE       *data_file;
    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 < 1 || size > 4 )
    {
        printf( "Number of processes must be in 1...4!\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    i_max = MAX_N / size + 2;

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

    for( i = 0; i < i_max; ++i )
    {
        if( (x[i] = (double *) malloc( MAX_N * sizeof( double ) )) == NULL ||
```

```
      (g_x[i] = (double *) malloc( MAX_N * sizeof( double ) )) == NULL )
    {
        printf( "Buffer allocation at process %d failed!\n", rank );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
}

if( rank == 0 )
{
    strcpy( data_name, argv[0] );

    if( (p = strrchr( data_name, SEPARATOR )) == NULL )
    {
        p = data_name;
    }
    else
    {
        ++p;
    }

    sprintf( p, "time_p%d.dat", size );

    if( (data_file = fopen( data_name, "wt" )) == NULL )
    {
        printf( "Cannot open file %s!\n", data_name );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    printf( "Size\tSteps\tDifference\tTime (sec)\n\n" );
}

for( n = 12; n <= MAX_N; n <<= 1 )
{
    steps = 0;
    i_max = n / size;
    i_bot = 1 + (rank == 0);
    i_top = i_max - (rank == size - 1);

    for( i = 1; i <= i_max; ++i )
    {
        for( j = 0; j < n; ++j )
        {
            x[i][j] = INITIAL;
        }
    }

    for( j = 0; j < n; ++j )
    {
        x[i_bot-1][j] = -1.0;
        x[i_top+1][j] = -1.0;
    }

    time = MPI_Wtime( );

    do
    {
```

```
        if( rank < size - 1 )
        {
            MPI_Send( x[i_max], n, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD );
        }

        if( rank > 0 )
        {
            MPI_Recv( x[0], n, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &status );
            MPI_Send( x[1], n, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD );
        }

        if( rank < size - 1 )
        {
            MPI_Recv( x[i_max+1], n, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD, &status );
        }

        norm = 0.0;

        for( i = i_bot; i <= i_top; ++i )
        {
            for( j = 1; j < n - 1; ++j )
            {
                g_x[i][j] = (x[i][j+1] + x[i][j-1] + x[i+1][j] + x[i-1][j]) / 4.0;
                norm += (g_x[i][j] - x[i][j]) * (g_x[i][j] - x[i][j]);
            }
        }

        for( i = i_bot; i <= i_top; ++i )
        {
            for( j = 1; j < n - 1; ++j )
            {
                x[i][j] = g_x[i][j];
            }
        }

        MPI_Allreduce( &norm, &g_norm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );
        g_norm = sqrt( g_norm );
    }
    while( ++steps < MAX_STEPS );

    time = MPI_Wtime( ) - time;

    if( rank == 0 )
    {
        printf( "%4d\t%5d\t%lf\t%lf\n", n * n, steps, g_norm, time );
        fprintf( data_file, "%d,%lf\n", n * n, time );
    }
}

if( rank == 0 )
{
    fclose( data_file );
}

for( i = 0; i < i_max; ++i )
{
```

```
        free( x[i] );
        free( g_x[i] );
    }

    free( x );
    free( g_x );
    MPI_Finalize( );
    return( 0 );
}
```

Program to determine t_c

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

#define MAX_STEPS 10000

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


int main( int argc, char **argv )
{
    char       *p, data_name[256];
    int        size, count;
    double     time, tc;
    FILE       *data_file;

    strcpy( data_name, argv[0] );

    if( (p = strrchr( data_name, SEPARATOR )) == NULL )
    {
         p = data_name;
    }
    else
    {
        ++p;
    }

    sprintf( p, "time_p1.dat" );

    if( (data_file = fopen( data_name, "rt" )) == NULL )
    {
        printf( "Cannot open file %s!\n", data_name );
        return( 1 );
    }

    count = 0;
    tc = 0.0;

    while( !feof( data_file ) )
    {
        fscanf( data_file, "%d,%lf\n", &size, &time );
```

```
        tc += time / (double) size;
        ++count;
    }

    printf( "tc = %1.10lf sec\n", tc / (double)(count * MAX_STEPS) );
    fclose( data_file );
    return( 0 );
}
```

Program to determine t_s and t_w

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

#define MAX_UPDATES 100


int main( int argc, char **argv )
{
    int        size, rank, k, n;
    double     data, time, ts, tw;
    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 );
    }

    ts = 1.7e+308;
    tw = 1.7e+308;

    for( n = 0; n < MAX_UPDATES; ++n )
    {
        if( rank == 0 )
        {
            time = MPI_Wtime( );

            for( k = 0; k < MAX_UPDATES; ++k )
            {
                MPI_Send( MPI_BOTTOM, 0, MPI_INT, 1, k, MPI_COMM_WORLD );
                MPI_Recv( MPI_BOTTOM, 0, MPI_INT, 1, k, MPI_COMM_WORLD, &status );
            }

            time = (MPI_Wtime( ) - time) / (double) MAX_UPDATES;
```

```
            if( time < ts )
            {
                ts = time;
            }

            time = MPI_Wtime( );

            for( k = 0; k < MAX_UPDATES; ++k )
            {
                MPI_Send( &data, 1, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
                MPI_Recv( &data, 1, MPI_DOUBLE, 1, k, MPI_COMM_WORLD, &status );
            }

            time = (MPI_Wtime( ) - time) / (double) MAX_UPDATES;

            if( time < tw )
            {
                tw = time;
            }
        }
        else
        if( rank == 1 )
        {
            for( k = 0; k < MAX_UPDATES; ++k )
            {
                MPI_Recv( MPI_BOTTOM, 0, MPI_INT, 0, k, MPI_COMM_WORLD, &status );
                MPI_Send( MPI_BOTTOM, 0, MPI_INT, 0, k, MPI_COMM_WORLD );
            }

            for( k = 0; k < MAX_UPDATES; ++k )
            {
                MPI_Recv( &data, 1, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status );
                MPI_Send( &data, 1, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
            }
        }
    }

    if( rank == 0 )
    {
        printf( "ts = %1.10lf sec\ntw = %1.10lf sec\n", ts * 0.5, (tw - ts) * 0.5 );
    }

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