High Performance Technical Computing

Augustin Reille

2 November 2017

Abstract

A one-space dimensional problem is considered, to examine the application of distributed memory parallel programming techniques for the numerical solution of the heat conduction equation.
The effect of changing time and space steps are investigated, and so is the runtime between serial and parallel programs using MPI, resolving the same problem.

Contents

List of Figures

List of Tables

Listings

Nomenclature

T_i^n	Temperature at space i and time n
D	Diffusivity of the wall
L	Total length of the wall
T	Total time of the study
Δx	Space step
Δt	Time step
δx	partial derivative according to space
δt	partial derivative according to time
n_{space}	number of iterations over space
n_{pes}	number of processors

1 Introduction

A PDE (Partial Differential Equation) is an equation which includes derivatives of an unknown function, with at least 2 variables. PDEs can describe the behavior of lot of engineering phenomena [1], that's why it is important to known how to resolve them computationally. One way to resolve PDEs is to discretize the problem into a mesh. The bigger is the mesh, the more accurate the results are.
As runtime depends on the study size, a good way to provide a good accuracy for results and keeping a correct runtime is to parallelize the programs and the algorithms used.
In this study, we will use MPI [2], a well-known communication protocol for programming in parallel, to see how the accuracy and the runtime are evolving between a serial and a parallel implementation of a program resolving a one-dimensional PDE.

2 Methods

Our study results are to be stored in matrixes. Several schemes are used :

- FTCS simple explicit (forward time, central space, explicit)
- Laasonen simple implicit (forward time, central space, implicit)
- Crank-Nicholson (Trapezoidal)

For all of the used schemes, the initialization of the result matrix is run the same way. First the study grid must be chosen. The initials conditions are:

- T_{int} of 100°F
- T_{sur} of 300°F
- $D = 0.1 ft^2/hr$

For each methods two sets of step size are to be used:

1. $\Delta x = 0.5$ and $\Delta t = 0.1$
2. $\Delta x = 0.005$ and $\Delta t = 0.001$

so that the result matrix could be initialized. We set:

$$n_{space} = \frac{L}{\Delta x} + 1 \tag{1}$$

$$n_{time} = \frac{T}{\Delta t} \tag{2}$$

n_{space} is the number of iterations over the grid, according to the coordinate x. n_{time} is the number of iterations over the grid, according to the time t.

$$\begin{array}{ccccc c}
i=0 & i=1 & \cdots & n_{space}-1 & n_{space} & \\
\left(\begin{matrix} T_{ext} \\ T_{ext} \\ \vdots \\ T_{ext} \end{matrix}\right. & \begin{matrix} T_{int} \\ 0 \\ \vdots \\ 0 \end{matrix} & \begin{matrix} \cdots \\ \cdots \\ \ddots \\ \cdots \end{matrix} & \begin{matrix} T_{int} \\ 0 \\ \vdots \\ 0 \end{matrix} & \left.\begin{matrix} T_{ext} \\ T_{ext} \\ \vdots \\ T_{ext} \end{matrix}\right) & \begin{matrix} j=0 \\ j=1 \\ \vdots \\ n_{time} \end{matrix}
\end{array}$$

This is how the matrix is initialized, according to initial conditions.

2.1 FTCS simple explicit

In order to perform the parallelization, we assume that the number of space steps can be divided by the number of processors. The number of space steps divided by the number of processors defines the new space step for each processor:

$$n'_{space} = \frac{n_{space}}{n_{pes}} \tag{3}$$

We have the equation :

$$T_i^n = T_i^{n-1} + r(T_{i+1}^{n-1} - 2T_i^{n-1} + T_{i-1}^{n-1}) \tag{4}$$

with

$$r = \frac{D\delta t}{\delta x^2}$$

. The stencil for FTCS explicit scheme is the following:

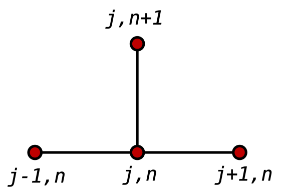

Figure 1: Stencil for FTCS explicit scheme

That means that to calculate a temperature for a time step and a space step, we need some previous values in time : the value in space, which we can access easily, and the previous and next values in space. That was the main problem for parallelizing this problem. The way we managed it is the following.
For each processor, a vector of size $[n'_{space}+2]$ is created. The size is incremented by 2 because each vector will contain 2 "ghost" values (will be explained later).
The first vector will be our boundary values vector, so we fill this vector with T_{int}.
For the first and last processor respectively, the first and last value of the vector are replaced by T_{ext}.
For example, for $n_{space} = 12$ and $n_{pes} = 4$, at $t = 0h$ we have :

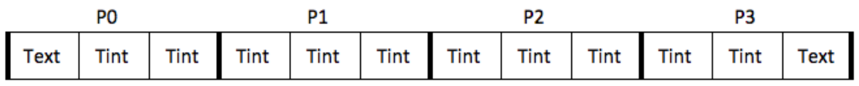

Figure 2: Separation of values per processors at t=0h

At each time step, each processor perform 4 MPI actions:

- it sends its first value to the previous processor (MPI_Send())

- it receives the first value of the next processor (MPI_Receive())
- it sends its last value to the next processor (MPI_Send())
- it receives the first value of the previous processor (MPI_Receive())

N-B : In the previous explanation, ghost values are not considered : they are only used to store the values received from neighbour processors.
In Figure 3, we can see which values are exchanged at the boundaries, in order to calculate the next boundary value. This pattern is due to the stencil : to continue with the example of the Figure 3, in order to calculate the value of $h[1]$ at time step $t + \Delta t$, the processor needs the $h[0]$ value, which comes from the previous processor.

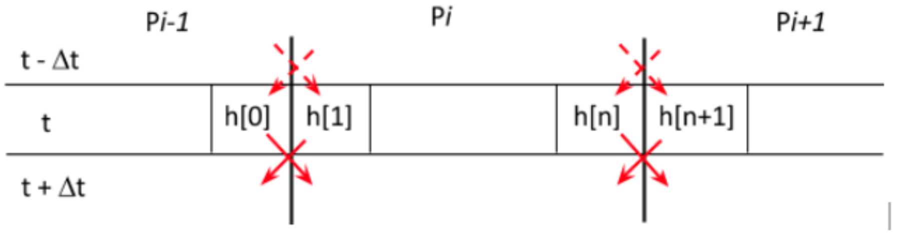

Figure 3: Exchanges between processors at each time step

Calculated values are gathered (MPI_Gather()) in the final result matrix, which is stored and printed in root processor.

2.2 Implicit schemes

For both implicit schemes, the vector solution is given by a matrix equation. Thomas algorithm is used to resolve the system for each space step.

2.2.1 Laasonen

We have :

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = D\frac{T_{j+1}^{n+1} - 2T_j^{n+1} + T_{j-1}^{n+1}}{\Delta x^2} \tag{5}$$

The stencil for Laasonen implicit scheme is the following:

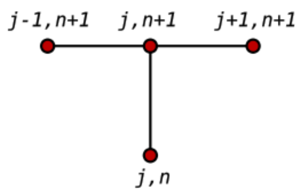

Figure 4: Stencil for Laasonen implicit scheme

Written in a matrical way, the solution at space step $n+1$ is given by :

$$\begin{bmatrix} 1+2C & -C & 0 & \cdots & 0 \\ -C & 1+2C & -C & \ddots & \vdots \\ 0 & -C & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 1+2C & -C \\ 0 & \cdots & 0 & -C & 1+2C \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_i \\ \vdots \\ T_{ntime} \end{bmatrix}_{n+1} = \begin{bmatrix} T_1 + CT_0 \\ T_2 \\ \vdots \\ T_i \\ \vdots \\ T_{ntime} + CT_f \end{bmatrix}_n \tag{6}$$

with

$$C = \frac{D\Delta t}{\Delta x^2}$$

The first vector at space n is our boundaries conditions vector, so it is known. In this scheme we must be aware that the right-term vector is actually bigger than the other, of 2 values, which corresponds to the exterior temperatures, which stays the same all the time.
In serial, we apply Thomas algorithm for each space step, at the reduced vector, and add the $n+1$ vector to our result matrix. The new right term vector is the one found with the Thomas algorithm.
There was several ways to parallelize this study. The one I chose is to parallelize Thomas algorithm. The algorithm I used is described in [5] and in [6]. The main idea is to make a recursive procedure with a LU decomposition. We must note that this algorithm will work with a matrix with differents tridiagonal values, and only for a number of processors which is a power of two.

$$\begin{bmatrix} a_1 & c_1 & 0 & \cdots & 0 \\ b_2 & a_2 & c_2 & \ddots & \vdots \\ 0 & b_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & a_{N-1} & c_{N-1} \\ 0 & \cdots & 0 & b_N & a_N \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_2 & 1 & & 0 & \\ & l_3 & \ddots & & \\ & 0 & \ddots & 1 & \\ & & & l_N & 1 \end{bmatrix} \begin{bmatrix} d_1 & u_1 & & & \\ & d_2 & u_2 & 0 & \\ & & d_3 & u_3 & \\ & 0 & & \ddots & \ddots \\ & & & & d_N \end{bmatrix} \tag{7}$$

Then we have the relations between coefficients :

$$a_1 = d_1 \tag{8}$$

$$c_j = u_k \tag{9}$$

$$a_k = d_k + l_k u_{k-1} \tag{10}$$

$$b_k = l_k d_{k-1} \tag{11}$$

where $j = 1, ..., N$ and $k = 2, ..., N$. By putting 9 and 11 into 10, we have the recursive relationship to find d_j:

$$d_j = \frac{a_j d_{j-1} - b_j c_{j-1}}{d_{j-1}} \tag{12}$$

Then we create 2x2 matrices in order to process the parallelisation recursively (for $j = 1, ..., N$):

$$R_0 = \begin{bmatrix} a_0 & 0 \\ 1 & 0 \end{bmatrix} \tag{13}$$

and

$$R_j = \begin{bmatrix} a_j & -b_j c_{j-1} \\ 1 & 0 \end{bmatrix} \tag{14}$$

and by setting $S_j = R_j R_{j-1} ... R_0$, we have

$$d_j = \frac{\begin{pmatrix} 1 \\ 0 \end{pmatrix}^t S_j \begin{pmatrix} 1 \\ 1 \end{pmatrix}}{\begin{pmatrix} 0 \\ 1 \end{pmatrix}^t S_j \begin{pmatrix} 1 \\ 1 \end{pmatrix}} \tag{15}$$

The algorithm is the following:

1. On each processor, matrix R_k is created. k is the number of rows which have to be treated by the process.
2. On each processor, matrix S_j is created.
3. Calculate local d_k.
4. Send local d_k to next processor.
5. Each process calculate the local l_k
6. Distribute d_j and l_j values on all processors.
7. On each processor, a local forward and backward substitution are performed to obtain the solution

This algorithm is performed on each time step, and the found value is inserted in the result matrix. Previous time vector is used for actual time space.

2.2.2 Crank-Nicholson

We have :

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = D\frac{1}{2}(\frac{T_{j+1}^{n+1} - 2T_j^{n+1} + T_{j-1}^{n+1}}{\Delta x^2} + \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2}) \tag{16}$$

The stencil for Crank trapezoidal scheme is the following:

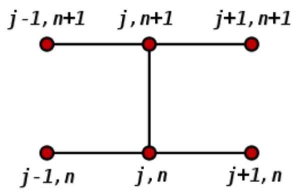

Figure 5: Stencil for Crank trapezoidal scheme

Written in a matrical way, the solution at space step $n+1$ is given by :

$$\begin{bmatrix} 1+C & -C/2 & 0 & \cdots & 0 \\ -C/2 & 1+C & -C/2 & \ddots & \vdots \\ 0 & -C/2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 1+C & -C/2 \\ 0 & \cdots & 0 & -C/2 & 1+C \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_i \\ \vdots \\ T_{ntime} \end{bmatrix}_{n+1} \tag{17}$$

$$= \begin{bmatrix} 1-C & C/2 & 0 & \cdots & 0 \\ C/2 & 1-C & C/2 & \ddots & \vdots \\ 0 & C/2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 1-C & C/2 \\ 0 & \cdots & 0 & C/2 & 1-C \end{bmatrix} \begin{bmatrix} T_1 + CT_0 \\ T_2 \\ \vdots \\ T_i \\ \vdots \\ T_{ntime} + CT_f \end{bmatrix}_{n}$$

with

$$C = \frac{D\Delta t}{\Delta x^2}$$

The method and the algorithm used are the same as Laasonen scheme, except that the vector which is found with Thomas algorithm is multiplied by the right-term matrix before the next time step.

2.3 Analytical solution

The analytical solution of the problem considered will be used to compare our results and to calculate the errors. The result of the analytical solution for a time t and a position x is given by :

$$T = T_{ext} + 2(T_{int} - T_{ext}) \sum_{m=1}^{m=\infty} e^{-D(m\pi/L)^2 t} \frac{1-(-1)^m}{m\pi} sin(\frac{m\pi x}{L}) \tag{18}$$

2.4 Speed-up calculation

To investigate the performance of our programs, we used MPI functions :

- MPI_Wtime : returns an elapsed time on the calling processor.[2]
- MPI_Wtick : returns the resolution of MPI_Wtime.[2]

The actual speed-up is calculated as following:

$$speedup = finaltime - starttime \tag{19}$$

where $finaltime$ is a double containing the value of MPI_Wtime at the end of the program, and *starttime* is a double containing the value of MPI_Wtime at the beginning of the program.

3 Results

3.1 Step size effect

We must note that due to the step sizes, first calculations could not be done in parallel.

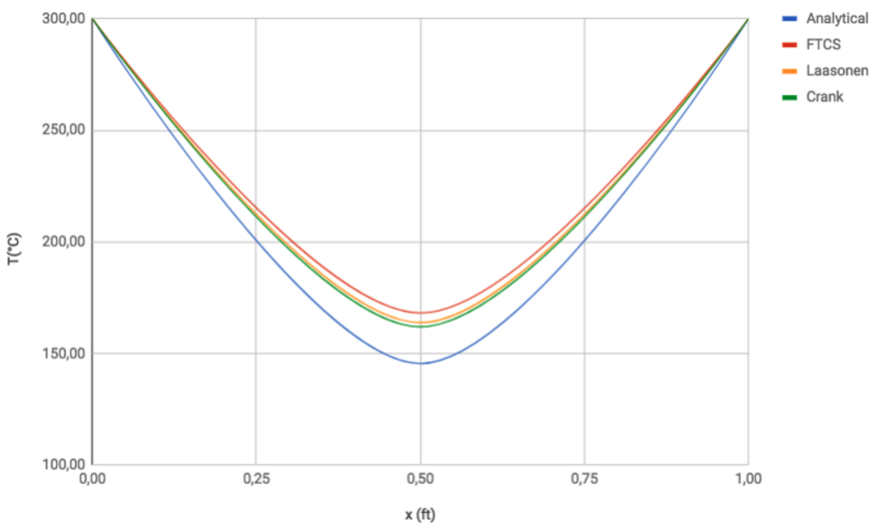

Figure 6: Results with differents serial schemes at $t = 0,5h$; $\Delta x = 0,5$; $\Delta t = 0,1$

We can see quite a big difference between the analytical solution and the other schemes. The errors had been calculated to evaluate the accuracy of each method.

Table 1: Errors for different serial schemes at $t = 0,5h$; $\Delta x = 0,5$; $\Delta t = 0,1$

FTCS	Laasonen	Crank-Nicholson
3,04%	2,46%	2,21%

We observe that for these step sizes, at t = 0,5h, Crank-Nicholson method is the most accurate, and Forward Time Central Space is the less accurate. Let see the differences when we decrease the step sizes.

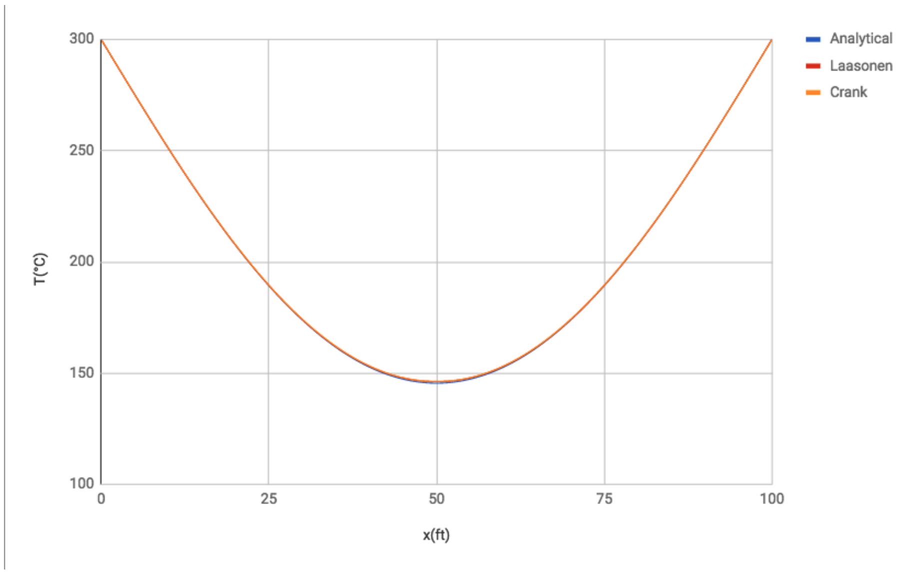

Figure 7: Results with differents parallelized schemes at $t = 0,5h$; $\Delta x = 0,005$; $\Delta t = 0,001$

With these time and space steps, the stability criterion for FCTS method was not verified. The values are not displayed because of their lack of interest. We can see a very small difference between the analytical solution and the other schemes. The errors had been calculated to evaluate the accuracy of each method.

Table 2: Errors for different parallelized schemes at $t = 0,5h$; $\Delta x = 0,005$; $\Delta t = 0,001$

	Laasonen	Crank-Nicholson
Serial	0,01%	0,04%
Parallel	0,10%	0,15%

We observe that by reducing a lot the step size, accuracy of the results is very good. We also observe that serial results are quite the same as the analytical results. We will discuss this later.

3.2 Speed-up investigation

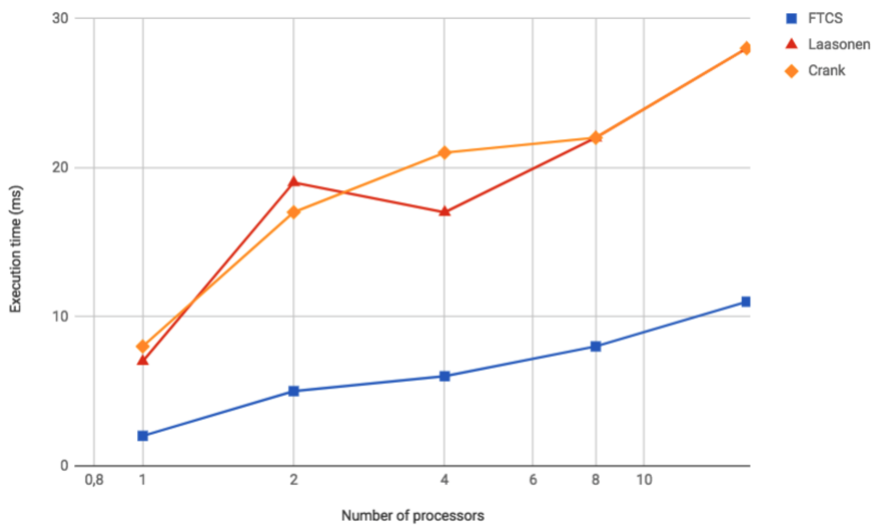

Figure 8: Speed-up results for each method

As we see in Fig. 6, the speed-up time increase as we increase the number of processors. The use of an external library is not quite usefull in our case.

4 Discussion

4.1 Analytical solution

First thing that should be considered is that the analytical solution should not be considered as the perfect exact solution, because of several factors.
First, the analytical solution is calculated by the computer, so there may be some numerical error.
Secondly, we have to consider the calculation of the sum inside equation (18). Indeed, as m should be infinite, in theory, the biggest it is is the most accurate. However, increasing computational calculation may alterate the results. I choosed a compromise where values of the analytical solution at $t = 0$ were as close to initial conditions as possible, with an m not too big, to reduce run time and numerical error.

4.2 Step size study

With a big step size, we can see that serial results are quite accurate (less than 5%). However, by reducing the time step and increasing the size of the grid, we observe better accuracy (less than 1%) for serial results.
So we can say that by increasing the size of the grid, we improve the results of the serial programs.

However, we observe bigger errors for parallel programs than for serial programs, for a small step size.
That can be due to the fact that our problem is not complex enough, and putting it into parallel causes some more errors than in the serial program, which manages the problem well enough. We must consider the fact that our algorithms may not be the most efficients for the considered problem.

4.3 Performance study

We have observed that the parallelization of our serial programs was leading to a loss of runtime speed. But the purpose of parallel programs is to reduce the speed-up time.[4] One possible cause of this loss of speed could be, as seen before, the fact that our problem is not complex enough to be put in parallel. Another cause of this loss can also be the fact that our algorithms may not be the most efficients for the considered problem.

5 Future work

Our future work would be based on two major axes :

- parallel algorithms improvement
- study complexification

6 Conclusion

For physical problems, PDE can be resolved using discretisation schemes. A good way to increase the accuracy of the results is to increase the size of the study, by reducing the time and space steps. But as the accuracy is increasing, so are the runtime and the computational costs. As these problems can be quite complex, to reduce computing time and to provide accurate results, parallelization can be used.
It may not be interesting to perform a parallelization for every problem : indeed, in some cases, especially for small programs, the parallelization of a program may increase the runtime and the error. But for huge and complex programs, the parallelization allows to save runtime and ressources.

7 References

References

[1] Wazwaz, Abdul-Majid. *Partial differential equations.* CRC Press, 2002.

[2] Open MPI v2.0.4 documentation.
`https://www.open-mpi.org/doc/v2.0/`

[3] Luciano Rezzolla *Numerical Methods for the Solution of Partial Differential Equations*, 2011

[4] Gottlieb, Allan; Almasi, George S. (1989). *Highly parallel computing.* Redwood City, Calif.: Benjamin/Cummings.

[5] George Karniadakis *Parallel Scientific Computing in C++ and MPI.* 1994.

[6] E.F. Van de Velde. *Concurrent Scientific Computing.* 1994.

8 Appendices

8.1 Code sample

Listing 1: Analytical solution code

```c
#include <math.h>
#include <stdio.h>

#define PI 3.141592

double calculateSum(int max_m, double D, double L, double t, double x)
{
    double sum = 0.0;
    for (int m = 1; m < max_m; m++)
    {
        double term_1 = -D * (m * PI / L) * (m * PI / L) * t;
        double term_2 = (1 - pow(-1, m)) / (m * PI);
        double term_3 = m * PI * x / L;
        sum += exp(term_1) * term_2 * sin(term_3);
    }
    return sum;
}

int main(int argc, char *argv[])
{
    // Initialization
    double D = 0.1;
    double dx = 0.005;
    double dt = 0.001;
    double L = 1.0;
    double T = 1.0;
    double r = D * dt / pow(dx, 2);
    int Text = 300;
    int Tint = 100;
    int tag;
    int ntime = T / dt;
    int nspace = L / dx + 1;

    printf("\n");
    printf("nspace␣:␣%d,␣ntime␣:␣%d\n", nspace, ntime);

    double analytical[ntime][nspace];

    for (int n = 0; n < ntime; n++)
    {
        for (int i = 0; i < nspace; i++)
        {
            double x = i * dx;
            double t = n * dt;
            double sum = 0.0;

            sum = calculateSum(1000, D, L, t, x);

            analytical[n][i] = Text + 2 * (Tint - Text) * sum;
        }
    }

    printf("\n\n");
    printf("Results␣matrix␣:␣\n\n");
    for (int i = 0; i < ntime; i++)
    {
        if (i == ntime / 2)
        for (int j = 0; j < nspace; j++)
        {
                printf("%3.2f,␣", analytical[i][j]);
        }
        // printf("\n");
    }
    return 0;
}
```

Listing 2: FTCS code

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

// TRY TO DO IT WRITING DIRECTLY IN A FILE

int main(int argc, char *argv[])
{
    // MPI things
```

```c
int npes, myrank;
double starttime, finaltime, precision;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

precision = MPI_Wtick();
starttime = MPI_Wtime();

// Initialization
double D = 0.1;
double dx = 0.005;
double dt = 0.001;
double L = 1.0;
double T = 1.0;
double r = D * dt / pow(dx, 2);
int Text = 300;
int Tint = 100;
int tag;
int ntime = T / dt;
int nspace = L / dx + 1;
int root;
if (myrank == 0)
    root = 1;
else
    root = 0;

if (root)
{
    printf("\n");
    printf("nspace␣:␣%d,␣ntime␣:␣%d\n", nspace, ntime);
}
// SECURITY : CHECKS THE STABILITY CRITERION VALUE
if (root)
{
    printf("\n");
    printf("stability␣criterion␣r␣=␣%f\n", r);
}
if (0.5 <= r)
{
    if (root)
    {
        printf("\n");
        printf("Exit␣:␣r␣>=␣0.5␣\n");
        printf("Try␣to␣change␣   x ␣and␣   t \n");
    }
    // exit(-1);
}
else
{
    if (root)
    {
        printf("\n");
        printf("Continuing␣:␣stability␣criterion␣ok.\n");
    }
}

double results[ntime][nspace];

// Initialize grid
for (int n = 0; n < ntime; n++)
    for (int i = 0; i < nspace; i++)
        results[n][i] = 0.0;
for (int i = 0; i < nspace; i++)
    results[0][i] = Tint;
for (int n = 0; n < ntime; n++)
{
    results[n][0] = Text;
    results[n][nspace - 1] = Text;
}
results[1][0] = Text;
results[1][nspace - 1] = Text;

if (npes > nspace)
{
    if (root)
    {
        printf("\n");
        printf("Too␣much␣processors␣for␣considered␣problem.\n");
        exit(-1);
    }
}
else if (npes == 1 || npes == nspace)
{
    if (root)
        for (int n = 1; n < ntime; n++)
            for (int i = 1; i < nspace - 1; i++)
                results[n][i] = results[n - 1][i] + r * (results[n - 1][i + 1] - 2 * results[n - 1][i] + results[n - 1][i - 1]);
    // Prints solution matrix
    if (root)
    {
        printf("\n");
        for (int i = 0; i < ntime; i++)
        {
            printf("rank␣:␣%d,␣line␣%d␣:␣", myrank, i + 1);
            for (int j = 0; j < nspace; j++)
            {
                printf("%3.2f␣", results[i][j]);
            }
            printf("\n");
        }
    }
}
else
{
    // case 1 (good problem division)
    if (nspace % npes == 0)
    {
        int newspace = nspace / npes;
        double p_results[ntime + 1][newspace + 2];
        if (root)
            printf("newspace␣:␣%d\n", newspace);
        // INITIALISATION : FILL MATRIX WITH 0 AND BOUNDARY CONDITIONS
        for (int i = 0; i <= ntime; i++)
        {
            for (int j = 0; j <= newspace; j++)
            {
                if (i == 0)
                    p_results[i][j] = Tint;
                else
                    p_results[i][j] = 0.0;
            }

            if (root)
                p_results[i][0] = Text;
            if (myrank == npes - 1)
                p_results[i][newspace - 1] = Text;

            // for (int i = 0; i < ntime; i++)
            // {
            //     printf("rank : %d, line %d : ", myrank, i + 1);
            //     for (int j = 0; j < newspace; j++)
            //     {
            //         printf("%3.2f ", p_results[i][j]);
            //     }
            //     printf("\n");
            // }
        }

        // FILL UP LINE 1 to NTIME
        for (int n = 1; n <= ntime; n++)
        {
            if (myrank > 0)
            {
                tag = 1;
                MPI_Send(&p_results[n - 1][1], 1, MPI_DOUBLE, myrank - 1, tag, MPI_COMM_WORLD);
            }
            if (myrank < npes - 1)
            {
                tag = 1;
                MPI_Recv(&p_results[n - 1][newspace + 1], 1, MPI_DOUBLE, myrank + 1, tag, MPI_COMM_WORLD, &status);
            }
            if (myrank < npes - 1)
            {
                tag = 2;
                MPI_Send(&p_results[n - 1][newspace], 1, MPI_DOUBLE, myrank + 1, tag, MPI_COMM_WORLD);
            }
            if (myrank > 0)
            {
                tag = 2;
                MPI_Recv(&p_results[n - 1][0], 1, MPI_DOUBLE, myrank - 1, tag, MPI_COMM_WORLD, &status);
            }
            for (int i = 1; i <= newspace; i++)
            {
                p_results[n][i] = p_results[n - 1][i] + r * (p_results[n - 1][i + 1] - 2 * p_results[n - 1][i] + p_results[n - 1][i - 1]);
            }

            // MPI_Gather(p_results[n], newspace, MPI_DOUBLE, results[n], newspace, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            if (root)
                p_results[n][0] = Text;
            if (myrank == npes - 1)
                p_results[n][newspace - 1] = Text;

            // MPI_Gather(p_results[n], newspace, MPI_DOUBLE, results[n], newspace, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
        // ALL RESULTS ARE GATHERED ON PROCESSOR 0
        for (int i = 0; i < ntime; i++)
        {
            MPI_Gather(p_results[i], newspace, MPI_DOUBLE, results[i], newspace, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
        // OUTPUT: PRINTS PARTICULAR RESULTS
```

```
            // for (int i = 0; i < ntime; i++)
            // {
            //     printf("rank : %d, line %d : ", myrank, i + 1);
            //     for (int j = 0; j < newspace; j++)
            //     {
            //         printf("%3.2f ", p_results[i][j]);
            //     }
            //     printf("\n");
            // }
        }
        else
        {
            if (root)
            {
                printf("\n");
                printf("Wrong␣number␣of␣processors␣(%d)␣for␣the␣problem␣size␣(%d)\n", npes, nspace);
                printf("Please␣try␣a␣number␣of␣processors␣which␣is␣able␣to␣divide␣the␣problem␣size.\n");
                exit(-1);
            }
        }
    }
    if (root)
    {
        printf("\n\n");
        printf("Results␣matrix␣:␣\n\n");
        for (int i = 0; i < ntime; i++)
        {
            // printf("rank : %d, line %d : ", myrank, i + 1);
            if (i == ntime / 2)
            {
                for (int j = 0; j < nspace; j++)
                {
                    printf("%3.2f,␣", results[i][j]);
                }
            }
            printf("\n");
        }
    }
    finaltime = MPI_Wtime();
    if(root)
    {
        printf("\n\n");
        printf("The␣execution␣time␣was␣%fs␣with␣a␣precision␣of␣%f.\n", finaltime-starttime, precision);

    }

    MPI_Finalize();
}
```

Listing 3: Laasonen code

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

// LAPACK ROUTINES
extern void dgttrf_(int *n, double *dl, double *d, double *du, double *du2, int *ipiv, int *info);
extern void dgttrs_(char *trans, int *n, int *nrhs, double *dl, double *d, double *du, double *du2, int *ipiv, double *b, int *ldb, int *info);

void thomasAlg(const double *a, const double *b, double *c, double *d, double *x, unsigned int n)
{
    int i;

    if (b[0] == 0)
    {
        printf("\n");
        printf("Warning␣:␣division␣by␣0.␣Aborted.\n");
        return;
    }
    else
    {
        c[0] = c[0] / b[0];
        d[0] = d[0] / b[0];
    }

    double val;

    for (i = 1; i < n; i++)
    {
        if ((b[i] - c[i - 1] * a[i]) == 0)
        {
            printf("\n");
            printf("Warning␣:␣division␣by␣0.␣Aborted.\n");
            exit(-1);
        }
        else
        {
            val = 1.0 / (b[i] - c[i - 1] * a[i]);
            c[i] = c[i] * val;
            d[i] = (d[i] - a[i] * d[i - 1]) * val;
        }
    }
    x[n - 1] = d[n - 1];
    for (i = n - 2; i > -1; i--)
        x[i] = d[i] - c[i] * x[i + 1];
}

void thomasAlg_parallel(int myrank, int npes, int N, double *b, double *a, double *c, double *x, double *q)
{
    int i, j, k, i_global;
    int newspace, p_offset;
    double S[2][2], T[2][2], s1tmp, s2tmp;
    MPI_Status status;
    double *l = (double *)malloc(N * sizeof(double));
    double *d = (double *)malloc(N * sizeof(double));
    double *y = (double *)malloc(N * sizeof(double));
    // double l, d;
    for (i = 0; i < N; i++)
        l[i] = d[i] = y[i] = 0.0;
    // l = d = 0.0;
    // Do it with malloc
    S[0][0] = S[1][1] = 1.0;
    S[1][0] = S[0][1] = 0.0;
    newspace = (int)floor(N / npes);
    p_offset = myrank * newspace;

    if (myrank == 0)
    {
        s1tmp = a[p_offset] * S[0][0];
        S[1][0] = S[0][0];
        S[1][1] = S[0][1];
        S[0][1] = a[p_offset] * S[0][1];
        S[0][0] = s1tmp;
        for (i = 1; i < newspace; i++)
        {
            s1tmp = a[i + p_offset] * S[0][0] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][0];
            s2tmp = a[i + p_offset] * S[0][1] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1tmp;
        }
        S[0][1] = s2tmp;
    }
    else
    {
        for (i = 0; i < newspace; i++)
        {
            s1tmp = a[i + p_offset] * S[0][0] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][0];
            s2tmp = a[i + p_offset] * S[0][1] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1tmp;
            S[0][1] = s2tmp;
        }
    }

    for (i = 0; i <= log2(npes); i++)
    {
        if (myrank + pow(2, i) < npes)
            MPI_Send(S, 4, MPI_DOUBLE, (int)(myrank + pow(2, i)), 0, MPI_COMM_WORLD);
        if (myrank - pow(2, i) >= 0)
        {
            MPI_Recv(T, 4, MPI_DOUBLE, (int)(myrank - pow(2, i)), 0, MPI_COMM_WORLD, &status);
            s1tmp = S[0][0] * T[0][0] + S[0][1] * T[1][0];
            S[0][1] = S[0][0] * T[0][1] + S[0][1] * T[1][1];
            S[0][0] = s1tmp;
            s1tmp = S[1][0] * T[0][0] + S[1][1] * T[1][0];
            S[1][1] = S[1][0] * T[0][1] + S[1][1] * T[1][1];
            S[1][0] = s1tmp;
        }
    }

    d[p_offset + newspace - 1] = (S[0][0] + S[0][1]) / (S[1][0] + S[1][1]);
    if (myrank == 0)
    {
        MPI_Send(&d[p_offset + newspace - 1], 1, MPI_DOUBLE,
                 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(&d[p_offset - 1], 1, MPI_DOUBLE, myrank - 1, 0, MPI_COMM_WORLD, &status);
        if (myrank != npes - 1)
            MPI_Send(&d[p_offset + newspace - 1], 1, MPI_DOUBLE,
                     myrank + 1, 0, MPI_COMM_WORLD);
    }

    if (myrank == 0)
    {
        l[0] = 0;
        d[0] = a[0];
        for (i = 1; i < newspace - 1; i++)
        {
            l[p_offset + i] = b[p_offset + i - 1] / d[p_offset + i - 1];
            d[p_offset + i] = a[p_offset + i] - l[p_offset + i] * c[p_offset + i - 1];
        }
        l[p_offset + newspace - 1] = b[p_offset + newspace - 2] /
                                     d[p_offset + newspace - 2];
    }
    else
    {
        for (i = 0; i < newspace - 1; i++)
        {
            l[p_offset + i] = b[p_offset + i - 1] /
                              d[p_offset + i - 1];
            d[p_offset + i] = a[p_offset + i] -
                              l[p_offset + i] * c[p_offset + i - 1];
        }
        l[p_offset + newspace - 1] = b[p_offset + newspace - 2] /
                                     d[p_offset + newspace - 2];
    }
    if (myrank > 0)
        d[p_offset - 1] = 0;
    double *tmp = (double *)malloc(N * sizeof(double));
    for (i = 0; i < N; i++)
        tmp[i] = d[i];
    MPI_Allreduce(tmp, d, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    for (i = 0; i < N; i++)
        tmp[i] = l[i];
    MPI_Allreduce(tmp, l, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(tmp);
    if (myrank == 0)
    {
        y[0] = q[0];
        for (i = 1; i < N; i++)
            y[i] = q[i] - l[i] * y[i - 1];
        x[N - 1] = y[N - 1] / d[N - 1];
        for (i = N - 2; i >= 0; i--)
            x[i] = (y[i] - c[i] * x[i + 1]) / d[i];
    }

    free(l);
    free(y);
    free(d);
    return;
}

void lapack_resolve(int *N, double *b, double *a, double *c, double *x, double *q)
{
    int *ipiv;
    int info, ldb;
    char trans = 'N';
    int val = 1;
    dgttrf_(N, a, b, c, x, ipiv, &info);
    dgttrs_(&trans, N, &val, a, b, c, x, ipiv, q, &ldb, &info);
}

int main(int argc, char *argv[])
{
    // MPI THINGS
    int npes, myrank;
    double starttime, finaltime, precision;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Request request;

    precision = MPI_Wtick();
    starttime = MPI_Wtime();

    // Initialization
    double D = 0.1;
    double dx = 0.005;
    double dt = 0.001;
    double L = 1.0;
    double T = 1.0;
    double r = D * dt / pow(dx, 2);
    int Text = 300;
    int Tint = 100;
    int tag;
    int ntime = T / dt;
    int nspace = L / dx + 1;
    int root;
    if (myrank == 0)
        root = 1;
    else
        root = 0;

    if (root)
    {
```

```
    printf("\n");
    printf("nspace␣:␣%d,␣ntime␣:␣%d\n", nspace, ntime);
}

double results[ntime][nspace];
for (int n = 0; n < ntime; n++)
    for (int i = 0; i < nspace; i++)
        results[n][i] = 0.0;
// Initialisation with boudary conditions
for (int i = 0; i < nspace; i++)
{
    results[0][i] = Tint;
}
for (int n = 0; n < ntime; n++)
{
    results[n][0] = Text;
    results[n][nspace - 1] = Text;
}

if (npes > nspace)
{
    if (root)
    {
        printf("\n");
        printf("Too␣much␣processors␣for␣considered␣problem.\n");
        exit(-1);
    }
}
else if (npes == 1 || npes == nspace)
{
    double a[nspace - 2];
    double b[nspace - 2];
    double c[nspace - 2];
    double c_copy[nspace - 2];
    double x[nspace - 2];
    double d[nspace];
    double d_reduced[nspace - 2];
    // Creating matrix C and vectors.
    for (int i = 0; i < nspace - 2; i++)
    {
        a[i] = -r;
        c[i] = -r;
        b[i] = 1 + 2 * r;
        x[i] = 0.0;
    }
    a[0] = 0.0;
    c[nspace - 3] = 0.0;

    for (int n = 1; n < ntime; n++)
    {
        // COPY THE PREVIOUS  T   LINE IN d
        for (int i = 0; i < nspace; i++)
            d[i] = results[n - 1][i];
        // CHANGES THE BOUNDARIES VALUES
        d[1] += r * d[0];
        d[nspace - 2] += r * d[nspace - 1];
        for (int i = 0; i < nspace - 2; i++)
        {
            // COPY THE REDUCED VECTOR IN d_reduced
            d_reduced[i] = d[i + 1];
            // COPY GOOD VALUE OF C
            c_copy[i] = c[i];
        }
        // RESOLVES a[i]x[i-1]+b[i]x[i]+c[i]x[i+1] = d[i] // WARNING : C AND X ARE MODIFIED
        thomasAlg(a, b, c_copy, d_reduced, x, nspace - 2);

        // COPY THE RESULT x INTO RESULTS MATRIX
        for (int i = 0; i < nspace - 2; i++)
            results[n][i + 1] = x[i];
        results[n][0] = Text;
        results[n][nspace - 1] = Text;
    }
    if (root)
    {
        printf("\n\n");
        printf("Results␣matrix␣:␣\n\n");
        for (int i = 0; i < ntime; i++)
        {
            if (i == ntime / 2)
            {
                for (int j = 0; j < nspace; j++)
                {
                    printf("%3.2f,␣", results[i][j]);
                }
            }
            // printf("\n");
        }
    }
}
else
{
        // if (nspace % npes == 0)
        // {

        double a[nspace - 2];
        double b[nspace - 2];
        double c[nspace - 2];
        double c_copy[nspace - 2];
        double x[nspace - 2];
        double d[nspace];
        double d_reduced[nspace - 2];
        int offset;
        // Creating matrix C and vectors.
        for (int i = 0; i < nspace - 2; i++)
        {
            a[i] = -r;
            c[i] = -r;
            b[i] = 1 + 2 * r;
            x[i] = 0.0;
        }

        for (int n = 1; n < ntime; n++)
        {
            // COPY THE PREVIOUS  T   LINE IN d
            for (int i = 0; i < nspace; i++)
                d[i] = results[n - 1][i];
            // CHANGES THE BOUNDARIES VALUES
            d[1] += r * d[0];
            d[nspace - 2] += r * d[nspace - 1];
            for (int i = 0; i < nspace - 2; i++)
            {
                // COPY THE REDUCED VECTOR IN d_reduced
                d_reduced[i] = d[i + 1];
                // COPY GOOD VALUE OF C
                c_copy[i] = c[i];
            }
            // RESOLVES a[i]x[i-1]+b[i]x[i]+c[i]x[i+1] = d[i] // WARNING : C AND X ARE MODIFIED
            thomasAlg_parallel(myrank, npes, nspace - 2, a, b, c_copy, x, d_reduced);
            // int lapack_n = nspace - 2;
            // lapack_resolve(&lapack_n, b, a, c_copy, x, d_reduced);

            // COPY THE RESULT x INTO RESULTS MATRIX
            for (int i = 0; i < nspace - 2; i++)
                results[n][i + 1] = x[i];
            results[n][0] = Text;
            results[n][nspace - 1] = Text;
        }
        if (root)
        {
            printf("\n\n");
            printf("Results␣matrix␣ahem␣:␣\n\n");
            for (int i = 0; i < ntime; i++)
            {
                if (i == ntime / 2)
                {
                    for (int j = 0; j < nspace; j++)
                    {
                        printf("%3.2f␣", results[i][j]);
                    }
                }
                // printf("\n");
            }
        }
    }
    finaltime = MPI_Wtime();
    if (root)
    {
        printf("\n\n");
        printf("The␣execution␣time␣was␣%fs␣with␣a␣precision␣of␣%f.\n", finaltime - starttime, precision);
    }
    // if (root)
    // {
    //     printf("\n");
    //     printf("-----TODO------\n");
    // }
    // }
    MPI_Finalize();
}
```

Listing 4: Crank-Nicholson code

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

void thomasAlg(const double *a, const double *b, double *c, double *d, double *x, unsigned int n)
{
    int i;
    if (b[0] == 0)
    {
        printf("\n");
        printf("Warning␣:␣division␣by␣0.␣Aborted.\n");
        return;
    }
    else
    {
        c[0] = c[0] / b[0];
        d[0] = d[0] / b[0];
    }

    double val;

    for (i = 1; i < n; i++)
    {
        if ((b[i] - c[i - 1] * a[i]) == 0)
        {
            printf("\n");
            printf("Warning␣:␣division␣by␣0.␣Aborted.\n");
            exit(-1);
        }
        else
        {
            val = 1.0 / (b[i] - c[i - 1] * a[i]);
            c[i] = c[i] * val;
            d[i] = (d[i] - a[i] * d[i - 1]) * val;
        }
    }
    x[n - 1] = d[n - 1];
    for (i = n - 2; i > -1; i--)
        x[i] = d[i] - c[i] * x[i + 1];
}

void thomasAlg_parallel(int myrank, int npes, int N, double *b, double *a, double *c, double *x, double *q)
{
    int i, j, k, i_global;
    int newspace, p_offset;
    double S[2][2], T[2][2], s1tmp, s2tmp;
    MPI_Status status;
    double *l = (double *)malloc(N * sizeof(double));
    double *d = (double *)malloc(N * sizeof(double));
    double *y = (double *)malloc(N * sizeof(double));
    for (i = 0; i < N; i++)
        l[i] = d[i] = y[i] = 0.0;
    S[0][0] = S[1][1] = 1.0;
    S[1][0] = S[0][1] = 0.0;
    newspace = (int)floor(N / npes);
    p_offset = myrank * newspace;

    // Form local products of R_k matrices
    if (myrank == 0)
    {
        s1tmp = a[p_offset] * S[0][0];
        S[1][0] = S[0][0];
        S[1][1] = S[0][1];
        S[0][1] = a[p_offset] * S[0][1];
        S[0][0] = s1tmp;
        for (i = 1; i < newspace; i++)
        {
            s1tmp = a[i + p_offset] * S[0][0] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][0];
            s2tmp = a[i + p_offset] * S[0][1] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1tmp;
        }
        S[0][1] = s2tmp;
    }
    else
    {
        for (i = 0; i < newspace; i++)
        {
            s1tmp = a[i + p_offset] * S[0][0] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][0];
            s2tmp = a[i + p_offset] * S[0][1] - b[i + p_offset - 1] * c[i + p_offset - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1tmp;
            S[0][1] = s2tmp;
        }
    }
    // Full-recursive doubling algorithm for distribution
    for (i = 0; i <= log2(npes); i++)
    {
        if (myrank + pow(2, i) < npes)
            MPI_Send(S, 4, MPI_DOUBLE, (int)(myrank + pow(2, i)), 0, MPI_COMM_WORLD);
        if (myrank - pow(2, i) >= 0)
        {
            MPI_Recv(T, 4, MPI_DOUBLE, (int)(myrank - pow(2, i)), 0, MPI_COMM_WORLD, &status);
            s1tmp = S[0][0] * T[0][0] + S[0][1] * T[1][0];
            S[0][1] = S[0][0] * T[0][1] + S[0][1] * T[1][1];
            S[0][0] = s1tmp;
            s1tmp = S[1][0] * T[0][0] + S[1][1] * T[1][0];
            S[1][1] = S[1][0] * T[0][1] + S[1][1] * T[1][1];
            S[1][0] = s1tmp;
        }
    }
    //Calculate last d_k first so that it can be distributed, //and then do the distribution.
    d[p_offset + newspace - 1] = (S[0][0] + S[0][1]) / (S[1][0] + S[1][1]);
    if (myrank == 0)
    {
        MPI_Send(&d[p_offset + newspace - 1], 1, MPI_DOUBLE,
                 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(&d[p_offset - 1], 1, MPI_DOUBLE, myrank - 1, 0, MPI_COMM_WORLD, &status);
        if (myrank != npes - 1)
            MPI_Send(&d[p_offset + newspace - 1], 1, MPI_DOUBLE,
                     myrank + 1, 0, MPI_COMM_WORLD);
    }
    // Compute in parallel the local values of d_k and l_k
    if (myrank == 0)
    {
        l[0] = 0;
        d[0] = a[0];
        for (i = 1; i < newspace - 1; i++)
        {
            l[p_offset + i] = b[p_offset + i - 1] / d[p_offset + i - 1];
            d[p_offset + i] = a[p_offset + i] - l[p_offset + i] * c[p_offset + i - 1];
        }
        l[p_offset + newspace - 1] = b[p_offset + newspace - 2] /
                                     d[p_offset + newspace - 2];
    }
    else
    {
        for (i = 0; i < newspace - 1; i++)
        {
            l[p_offset + i] = b[p_offset + i - 1] /
                              d[p_offset + i - 1];
            d[p_offset + i] = a[p_offset + i] -
                              l[p_offset + i] * c[p_offset + i - 1];
        }
        l[p_offset + newspace - 1] = b[p_offset + newspace - 2] /
                                     d[p_offset + newspace - 2];
    }
    /****************************************************************/
    if (myrank > 0)
        d[p_offset - 1] = 0;
    // Distribute d_k and l_k to all processes
    double *tmp = (double *)malloc(N * sizeof(double));
    for (i = 0; i < N; i++)
        tmp[i] = d[i];
    MPI_Allreduce(tmp, d, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    for (i = 0; i < N; i++)
        tmp[i] = l[i];
    MPI_Allreduce(tmp, l, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    free(tmp);
    if (myrank == 0)
    {
        /* Forward Substitution [L][y] = [q] */ y[0] = q[0];
        for (i = 1; i < N; i++)
            y[i] = q[i] - l[i] * y[i - 1];
        /* Backward Substitution [U][x] = [y] */ x[N - 1] = y[N - 1] / d[N - 1];
        for (i = N - 2; i >= 0; i--)
            x[i] = (y[i] - c[i] * x[i + 1]) / d[i];
    }
    free(l);
    free(y);
    free(d);
    return;
}
int main(int argc, char *argv[])
{
    // MPI THINGS
    int npes, myrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Request request;

    // Initialization
    double D = 0.1;
    double dx = 0.005;
    double dt = 0.001;
    double L = 1.0;
    double T = 1.0;
    double r = D * dt / pow(dx, 2);
    int Text = 300;
    int Tint = 100;
    int tag;
    int ntime = T / dt;
    int nspace = L / dx + 1;
    int root;
    if (myrank == 0)
        root = 1;
    else
        root = 0;

    if (root)
    {
        printf("\n");
        printf("nspace␣:␣%d,␣ntime␣:␣%d\n", nspace, ntime);
    }

    double results[ntime][nspace];
    for (int n = 0; n < ntime; n++)
        for (int i = 0; i < nspace; i++)
            results[n][i] = 0.0;
    // Initialisation with boudary conditions
    for (int i = 0; i < nspace; i++)
    {
        results[0][i] = Tint;
    }
    for (int n = 0; n < ntime; n++)
    {
        results[n][0] = Text;
        results[n][nspace - 1] = Text;
    }

    if (npes > nspace)
    {
        if (root)
        {
            printf("\n");
            printf("Too␣much␣processors␣for␣considered␣problem.\n");
            exit(-1);
        }
    }
    else if (npes == 1 || npes == nspace)
    {
        double a[nspace - 2];
        double b[nspace - 2];
        double c[nspace - 2];
        double e[nspace - 2];
        double f[nspace - 2];
        double g[nspace - 2];
        double c_copy[nspace - 2];
        double x[nspace - 2];
        double x_new[nspace - 2];
        double d[nspace];
        double d_reduced[nspace - 2];

        // Creating matrices A and E and vectors.
        for (int i = 0; i < nspace - 2; i++)
        {
            a[i] = -r / 2;
            c[i] = -r / 2;
            b[i] = 1 + r;
            e[i] = r / 2;
            g[i] = r / 2;
            f[i] = 1 - r;
            x[i] = 0.0;
        }

        for (int n = 1; n < ntime; n++)
        {
            // COPY THE PREVIOUS  T   LINE IN d
            for (int i = 0; i < nspace; i++)
                d[i] = results[n - 1][i];
            // CHANGES THE BOUNDARIES VALUES
            d[1] += r * d[0];
            d[nspace - 2] += r * d[nspace - 1];
            for (int i = 0; i < nspace - 2; i++)
            {
                // COPY THE REDUCED VECTOR IN d_reduced
                d_reduced[i] = d[i + 1];
                // COPY GOOD VALUE OF C
                c_copy[i] = c[i];
            }

            // CALCULATING NEW VECTOR X

            // x_new = E * d_reduced;
            x_new[0] = f[0] * d_reduced[0] + g[0] * d_reduced[1];
            for (int i = 1; i < nspace - 3; i++)
            {
                x_new[i] = e[0] * d_reduced[i - 1] + f[0] * d_reduced[i] + g[0] * d_reduced[i + 1];
            }
            x_new[nspace - 3] = e[0] * d_reduced[nspace - 4] + f[0] * d_reduced[nspace - 3];

            // RESOLVES a[i]x[i-1]+b[i]x[i]+c[i]x[i+1] = d[i] // WARNING : C AND X ARE MODIFIEd
            thomasAlg(a, b, c_copy, x_new, x, nspace - 2);
            // COPY THE RESULT x INTO RESULTS MATRIX
            for (int i = 0; i < nspace - 2; i++)
                results[n][i + 1] = x[i];
            results[n][0] = Text;
            results[n][nspace - 1] = Text;
        }
        if (root)
        {
            printf("\n\n");
            printf("Results␣matrix␣:␣\n\n");
            for (int i = 0; i < ntime; i++)
            {
                if (i == ntime / 2)
                {
                    for (int j = 0; j < nspace; j++)
                    {
                        printf("%3.2f,␣", results[i][j]);
                    }
                }
                // printf("\n");
            }
        }
    }
    else
    {
        // if (nspace % npes == 0)
        // {

        double a[nspace - 2];
        double b[nspace - 2];
        double c[nspace - 2];
        double e[nspace - 2];
        double f[nspace - 2];
        double g[nspace - 2];
        double c_copy[nspace - 2];
        double x[nspace - 2];
        double x_new[nspace - 2];
        double d[nspace];
        double d_reduced[nspace - 2];
        // Creating matrix C and vectors.
        // Creating matrices A and E and vectors.
        for (int i = 0; i < nspace - 2; i++)
        {
            a[i] = -r / 2;
            c[i] = -r / 2;
            b[i] = 1 + r;
            e[i] = r / 2;
            g[i] = r / 2;
            f[i] = 1 - r;
            x[i] = 0.0;
        }

        for (int n = 1; n < ntime; n++)
        {
            // COPY THE PREVIOUS  T   LINE IN d
            for (int i = 0; i < nspace; i++)
                d[i] = results[n - 1][i];
            // CHANGES THE BOUNDARIES VALUES
            d[1] += r * d[0];
            d[nspace - 2] += r * d[nspace - 1];
            for (int i = 0; i < nspace - 2; i++)
            {
                // COPY THE REDUCED VECTOR IN d_reduced
                d_reduced[i] = d[i + 1];
                // COPY GOOD VALUE OF C
                c_copy[i] = c[i];
            }

            // CALCULATING NEW VECTOR X

            // x_new = E * d_reduced;
            x_new[0] = f[0] * d_reduced[0] + g[0] * d_reduced[1];
            for (int i = 1; i < nspace - 3; i++)
            {
                x_new[i] = e[0] * d_reduced[i - 1] + f[0] * d_reduced[i] + g[0] * d_reduced[i + 1];
            }
            x_new[nspace - 3] = e[0] * d_reduced[nspace - 4] + f[0] * d_reduced[nspace - 3];

            // RESOLVES a[i]x[i-1]+b[i]x[i]+c[i]x[i+1] = d[i] // WARNING : C AND X ARE MODIFIED
            thomasAlg_parallel(myrank, npes, nspace - 2, a, b, c_copy, x, x_new);

            // COPY THE RESULT x INTO RESULTS MATRIX
            for (int i = 0; i < nspace - 2; i++)
            {
                results[n][i + 1] = x[i];
            }
            results[n][0] = Text;
            results[n][nspace - 1] = Text;
        }
        if (root)
```

```
        {
            printf("\n\n");
            printf("Results␣matrix␣:␣\n\n");
            for (int i = 0; i < ntime; i++)
            {
                if (i == nspace / 2)
                {
                    for (int j = 0; j < nspace; j++)
                    {
                        printf("%3.2f␣", results[i][j]);
                    }
                }
                // printf("\n");
            }
        }
    }

    // if (root)
    // {
    //     printf("\n");
    //     printf("-----TODO------\n");
    // }
    // }
    MPI_Finalize();
}
```