3.5 Discussion

Beside the above row-block decomposition algorithm, we venture to give a parallel algorithm, say the column decomposition algorithm, as follows:

Let us make a partition of the matrix A as p blocks of sub-matrices. Each block consists of n_k columns, i.e., $A = [B_1, B_2, \cdots, B_p]$, where B_k is a $n \times n_k$ matrix. We divide the vector x into its sub-vectors, i.e., $x = [x^1, x^2, \cdots, x^p]'$, where x^k is a vector of length n_k. Then in the parallel algorithm, the processor k simply performs a sub-problem, a matrix-vector product $B_k\, x^k = z^k$. Thus, similar to the previous row block decomposition, in each processor, we can generate locally the matrix B_k and the vector x^k. The resulting vector of length n, z^k, is calculated locally. In fact, the vector y which we need to compute, is a summation of p vectors of length n, i.e. $y = z^1 + z^2 + \cdots + z^p$. Again, a parallel algorithm for adding these vectors arises because the addition in different rows can be done independently. We notice that, before the second parallel algorithm is done, the vectors $z^1, z^2, \cdots, z^p$ should be stored in the root as either a matrix or a vector of length np (for example, $r_{(i-1)p+j} = z_i^j,\ j = 1, 2, \cdots, p,\ i = 1, 2, \cdots, n$).

Using the similar way of estimating the complexity of the previous algorithm, for the current algorithm we consider two sequential parts:

- For the first part (local matrix-vector product), the computation time on processor k is approximated by $2n\, n_k$.

- For the second part (vector-vector addition), we can use the so-called 'fan-in' algorithm (we refer to [1] for more detail discussion). In the processor k, we need to calculate n_k elements. Each of them is a summation of p values. The computation time of such the calculation using 'fan-in' algorithm is approximated by $\log_2 p$. Therefore, the total computation time for this second part is $n_k \log_2 p$.

If the communication time can be omitted, the total execution time for the current algorithm should be $n_k\,(2n + log_2 p)$, which is larger than that of previous section. If p is small compared to n, this value is comparable to what has been shown the previous section.

In summary, we have studied the row-block decomposition parallel algorithm used for calculating the matrix-vector product, in both theoretical and numerical points of view. As illustrated in the above figure, this is a quite satisfactory as it is popularly in used. In addition, we also gave the parallel algorithm which is a promising way of calculating a matrix-vector product.

4 The vibrating string

4.1 The model

The wave equation, which is of the form

$$\frac{\partial^2 \psi}{\partial t^2} = c^2 \frac{\partial^2 \psi}{\partial x^2}, \tag{6}$$

comes from many real and interesting applications, such as the vibrations of a uniform string, the sound waves in gases or liquids, and optical waves. As an example, we study an isolated guitar string of length 1, with two fixed endpoints. We refer to [2] for more details of the construction of the problem.

Briefly, we need to solve equation (6) to find out the vibration amplitude $\psi(x,t)$ of the string at position x ($0 \leq x \leq 1$) and time t ($t \geq 0$). Since the string is fixed on both endpoints, mathematically we have

$$\psi(0,t) = \psi(1,t) = 0. \tag{7}$$

We suppose that the vibration amplitude is initialized as

$$\psi(x,0) = f(x), \tag{8}$$

where $f(x)$ can be any smooth function that satisfies the boundary conditions (7). For that purpose, we chose $f(x) = \sin(2k\pi x), k \in \mathbb{Z}^+$.

Analytically, using Fourier transformation (see [2]), our solution is of the form

$$\psi(x,t) = \sum_{m=1}^{\infty} B_m \sin(m\pi x)\cos(m\pi ct), \tag{9}$$

where the coefficient B_m is determined by the initial condition as

$$B_m = 2\int_0^1 f(x)\sin(m\pi x)dx. \tag{10}$$

For our particular initial value $f(x) = \sin(2k\pi x)$ we have $B_m = 0, m \neq 2k$ and $B_{2k} = 1$. Therefore, the final solution is $\psi(x,t) = \sin(2k\pi x)\cos(2k\pi ct)$. Thus,

$$||\psi(x,t)|| \leq ||\sin(2k\pi x)|| = ||f(x)|| = ||\psi(x,0)||. \tag{11}$$

4.2 Numerical approach

In order to find the numerical solution of our problem (6), we use an uniform discretization for the spatial domain as well as the temporal domain and approximate the differential operators by the so-called finite difference expressions (see [2]). That is, the spatial domain is discretized as $x_i = i\Delta x, i = 0, 1, \cdots, N$, and $\Delta x = 1/N$, where N is number of grid points. The temporal domain is decomposed as $t_j = j\Delta t$, where Δt is a predetermined time step. The spatial and temporal second derivatives (at current position x_i and time t_j) are respectively approximated by

$$\begin{aligned} \frac{\partial^2\psi(x_i,t_j)}{\partial t^2} &\approx \frac{\psi(x_i,t_{j-1}) - 2\psi(x_i,t_j) + \psi(x_i,t_{j+1})}{\Delta t^2}, \\ \frac{\partial^2\psi(x_i,t_j)}{\partial x^2} &\approx \frac{\psi(x_{i-1},t_j) - 2\psi(x_i,t_j) + \psi(x_{i+1},t_j)}{\Delta x^2}, \end{aligned} \tag{12}$$

where $\psi(x_0,\cdot) = \psi(x_N,\cdot) = 0, \psi(\cdot,t_0) = f(\cdot)$. By substituting the expression (12) into (6), one can get the solution at the next point in time $\psi(x_i,t_{j+1})$, using the previous calculated values, as

$$\psi(x_i,t_{j+1}) = 2\psi(x_i,t_j) - \psi(x_i,t_{j-1}) + \tau^2\left(\psi(x_{i-1},t_j) - 2\psi(x_i,t_j) + \psi(x_{i+1},t_j)\right), \tag{13}$$

where $\tau = \frac{c\Delta t}{\Delta x}$. In other words, to update the solution at a particular point in the space we need only few local grid points (itself and 4 neighbors) of the current time level. The scheme is therefore *well parallelized*.

Remark : Our discretization scheme is *stable* (i.e., a small local error does not lead to a large cumulative error) if $0 \leq \tau \leq 1$ (see [2]). The time step is therefore restricted by $\Delta t \leq \Delta x/c$.

4.3 Parallel algorithm

Since we consider the initial value problem, the spatial decomposition is used. That is, each processor gets a part of the string and simulates the time-behavior of this specific part. The algorithm is as follows: We suppose that the processors are organized as a ring/line topology. Then, each processor has its own left and right neighbors, say `left` and `right`. In the beginning, a part of starting values is given to each processor. Then, at each time-integration step, the processor exchanges the boundary points (the most left and the most right points) with the left and right neighbors to performs the update (13), while the internal points are fully parallelized.

Remark The temporal decomposition is not used here because the data elements in processor k are only calculated if data elements in processor $k-1$ are available (i.e., in fact we have no parallelization).

4.4 Implementation

Being the first step, each processor picks up the starting values, such as the environment (`MPI_Init`) and its rank in the topology (`MPI_Comm_size` and `MPI_Comm_rank`). Using the routine `parseargs`, the root receives all necessary parameters, e.g. number of grid points N, the time interval, the time step etc. into variable `parms` and passes those parameters to all processors. The `parseargs` function is used in the following manner

```
if(parseargs(argc, argv)){
   MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   exit(EXIT_FAILURE);
}
```

Here `MPI_Abort` is used (not `MPI_Finalized`) since the passing-parameters program turns out to be unsuccessful. There are two ways of passing the parameters, namely `TrfArgs1` and `TrfArg2`. The former bcasts the parameters in a package (using the routine `MPI_Type_struct`). The latter bcasts one parameter by one parameter. Therefore, `TrfArg2` should be the *cleaner and faster* method because the package is considered as a single datatype and the communication time is smaller. For example, for a set of parameters, it turns out that the execution time using `TrfArgs1` is 0.08 seconds, while it is only 0.04 seconds for the program using `TrfArgs2`.

Now, to create a ring/line topology `ring`, we use the routine `MPI_Cart_create`. The coordinate and the rank ID of the current process, which are respectively denoted by

coords and rank, are found by using the routines MPI_Cart_get and MPI_Cart_rank. Then, the left (of coordinate coords-1) and the right(of coordinate coords+1) processors, whose rank IDs are left and right, can be easily obtained

```
MPI_Cart_shift (ring, 0, 1, &left, &right);
```

Then, similar to the data decomposition mentioned in Section 3, each processor gets a proportional amount of data points npoints, where the position of the first data point in the global set is denoted by offset. Since only a part of global string needs to be updated in one processor, we need to store only the parts of solution at previous time levels $\psi(\cdot, t_{j-1})$ and $\psi(\cdot, t_j)$, as well as at next time level $\psi(\cdot, t_{j+1})$. These vectors (of the length npoints) are denoted by oldval, curval and newval respectively. At the beginning, oldval and curval are both initialized as $f(x)$, which is a SIN function. For the test purpose, we also implement a PLUCKED function as an initial value (see Initialize).

Now, we are able to have the computation processed. The program requires niters iterations to reach the end-time point (i.e. parms.stime). In a single iteration (see routine compute), three steps need to be done: exchanging the boundary points; updating the solution vector; and preparing for the next iteration.

- To exchange the boundary points, we denote the most left point in vector curval (which is curval[0]) by l_neighbor and the most right point (which is curval[npoints-1]) by r_neighbor. Then, using MPI_Sendrecv, we send r_neighbor to process on right while receiving into l_point from process on left. On the other hand, we send l_neighbor to process on left while receiving into r_point from process on right. l_point and r_point will be used to calculate newval.

```
/* Take the most left and most right points */

  l_neighbor[0]=*(curval+0);
  r_neighbor[0]=*(curval+npoints-1);

  /* Send r_neighbor to process on right while receiving into l_point
     from process on left. Then, send the l_neighbor to process on left,
     while receiving into r_point from the process on right.          */

  if (SendAndRecv){
      MPI_Sendrecv(&r_neighbor[0], 1, MPI_DOUBLE, right, rank,
                   &l_point[0], 1, MPI_DOUBLE, left, left, ring, &status);
      MPI_Sendrecv(&l_neighbor[0], 1, MPI_DOUBLE, left, rank,
                   &r_point[0], 1, MPI_DOUBLE, right, right, ring, &status);
  }else{
      MPI_Send(&r_neighbor[0], 1, MPI_DOUBLE, right, rank, ring);
      MPI_Recv(&l_point[0], 1, MPI_DOUBLE, left, left, ring, &status);
      MPI_Send(&l_neighbor[0], 1, MPI_DOUBLE, left, rank, ring);
      MPI_Recv(&r_point[0], 1, MPI_DOUBLE, right, right, ring, &status);
  }
```

We note that one can use `MPI_Send` and `MPI_Recv` separately to exchange the boundary points. However, to *avoid deadlock* `MPI_Sendrecv` is preferred since sending and receiving happen simultaneously.

- To calculate `newval`, we need to consider carefully whether or not the data point is in the boundary of the domain. If so, the boundary condition should be used. Otherwise, we update the array according to (13), with notice of using `l_point` and `r_point`. The code is as follows:

```
for (i=0; i <npoints; i++){
  if  ((offset+i==GlbLeft)||(offset+i==GlbRight)){
    *(newval+i) = 0.0;  /*Fix the two end points of string*/
  }else{
    l = (i==0) ? l_point[0] : *(curval+i-1);
    m = *(curval+i);
    r = (i==npoints-1) ? r_point[0] : *(curval+i+1);
    d = *(oldval+i);
    *(newval+i) = 2.0*m - d + tau*tau*(l-2.0*m+r);
  }
}
```

- Finally, replace `oldval` by `curval` and `curval` by `newval`.

```
for (i=0; i < npoints; i++) {
    *(oldval+i) = *(curval+i);
    *(curval+i) = *(newval+i);
}
```

Complexity As always, it is good practice to analyze the parallel program in term of its complexity. Assuming that the time integration takes most of the execution time. It is reasonable to just consider a single time integration (once of calling `compute()`).

Using the same notations as in Section 3.3, however, we use "the number of grid points" N instead of using "the problem size" n. The required time to compute one grid point is $8\tau_{calc}$. Therefore,

$$T_1 = 8N\tau_{calc}, \tag{14}$$

$$T_p = \frac{T_1}{p} + T_{comm}$$

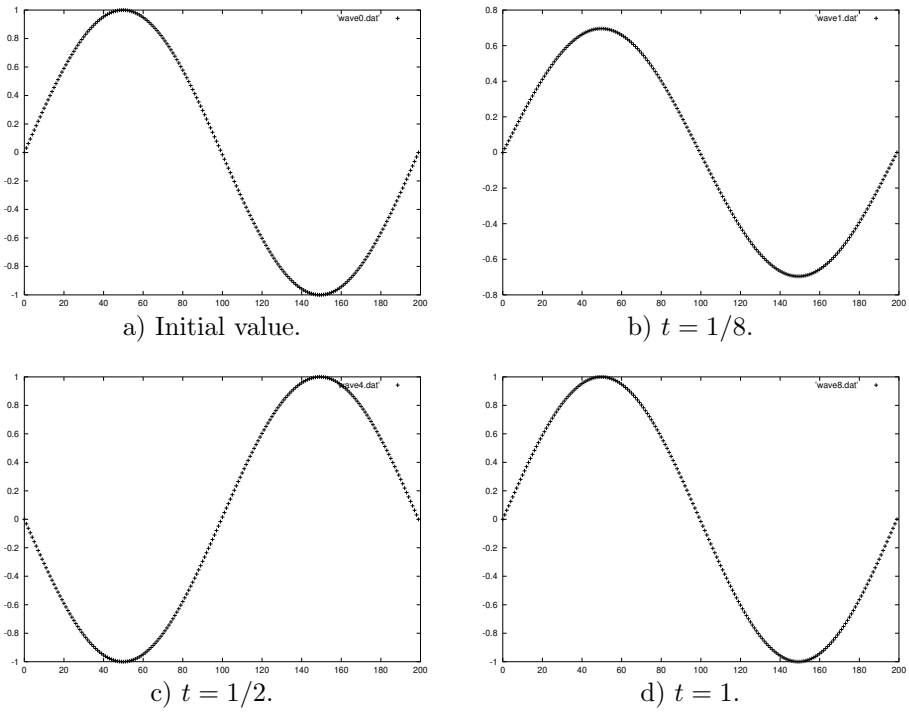
$$= \frac{8N\tau_{calc}}{p} + 2(\tau_{setup} + \tau_{exchange}), \tag{15}$$

$$S_p = \frac{T_1}{T_p} = \frac{1}{\frac{1}{p} + \frac{1}{4N}\frac{\tau_{setup}}{\tau_{calc}} + \frac{1}{4N}\frac{\tau_{exchange}}{\tau_{calc}}}, \tag{16}$$

$$E_p = \frac{T_1}{T_p} = \frac{1}{1 + \frac{p}{4N}\frac{\tau_{setup}}{\tau_{calc}} + \frac{p}{4N}\frac{\tau_{exchange}}{\tau_{calc}}}. \tag{17}$$

4.5 Results and discussion

To check the correctness of the program, let $c = 1$ and $k = 1$, our exact solution is then of the form $\psi(x,t) = \sin(2\pi x)\cos(2\pi t)$, which scales the initial value by a factor $\cos(2\pi t) < 1$. At $t = 1/8$, $t = 1/2$ and $t = 1$, the factors are $\sqrt{2}/2 = 0.7071$, -1 and 1 respectively. This is also numerically observed as shown in Figure 2.



a) Initial value. b) $t = 1/8$.

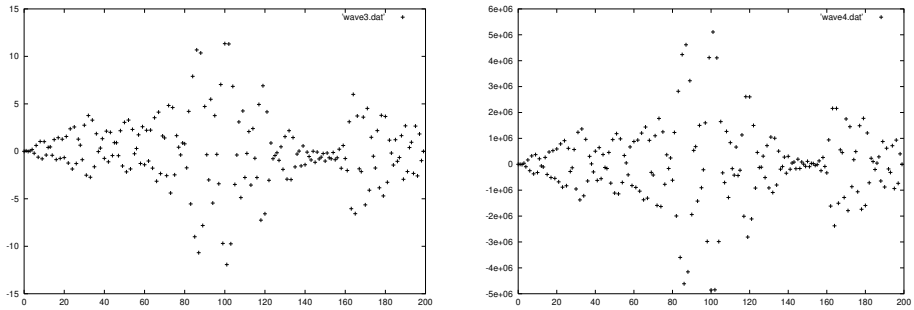
c) $t = 1/2$. d) $t = 1$.

Figure 2: The vibration amplitude at the particular points in time. $\Delta t = 0.005$.

Moreover, as mentioned in the remark in section 4.2, our numerical scheme is only stable if $\Delta t \leq \Delta x$. We have tested the program with two different values of Δt while $\Delta x = 1/199$ is unchanged. Indeed, it is shown that our method is stable with $\Delta t = 0.005$ (Figure 2) while it is unstable with $\Delta t = 0.01$ (Figure 3). From Figure 3a, we observe that the amplitude is large ($\simeq 15$), after a very short time interval, and keep increasing ($\simeq 10^6$ in Figure 3b). The solution is no longer bounded by the initial value (as mentioned in Section 4.2).

Having the analysis given in the section 4.4, we now focus on the performance of the algorithm. Again, to have an explanation of these results, we shall have a look at the formulas presenting the complexity. From (15), the execution time (in seconds) is proportional to the ratio of N and p. That is why any curve in Figure 4a is monotonously increasing as N increases, while it is monotonously decreasing as P increases (Figure 4d).

From Figures 4b and 4e we observe that the speedup increases if N or p increases.



a) $t = 0.15$. b) $t = 0.20$.

Figure 3: Unstable behavior (solution is plotted point by point). $\Delta t = 0.01$

This is supported by the theoretical expression (16). However, the increment of p does not means that we then also get a high efficiency, because the larger p the larger communication time, which might be cover the computation time. In fact, the efficiency depends on the ratio of p and N, as you can see from formula (17). Thus, each N has its own optimal value p^*. From Figure 4c, $p^* = 2$ and $p^* = 4$ seem to be a good choice for our problem, because the efficiency are high ($\simeq 0.78$) and maintain as N increases. For larger p ($p = 8$ and $p = 16$) we have not the such high efficiency, because the communication time covers the computation time. To prove that, inside the program we implemented a small technique to compute the exchanged time. It turns out that the exchange time is really large (about 2/3 of execution time, or even more), especially for large p. That is why in Figure 4f the efficiency curves quickly decrease and are small as p increases. Our conclusion is that we should use at most 4 processors for this particular application.

5 The Time Dependent Diffusion Equation

5.1 The model

Let us consider the two-dimensional time dependent diffusion equation

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right), \tag{18}$$

where the concentration c is a function of spatial variables x and y and time t. Without loss of generality, we assume that $0 \leq x \leq 1$ and $0 \leq y \leq 1$. For the boundary conditions, we assume that

$$c(x, y = 1, t) = 1, \quad c(x, y = 0, t) = 0, \tag{19}$$

$$c(x = 0, y, t) = c(x = 1, y, t). \tag{20}$$

For the initial condition, we take

$$c(x, y, t = 0) = 0 \quad \text{for } 0 \leq x \leq 1, 0 \leq y < 1. \tag{21}$$

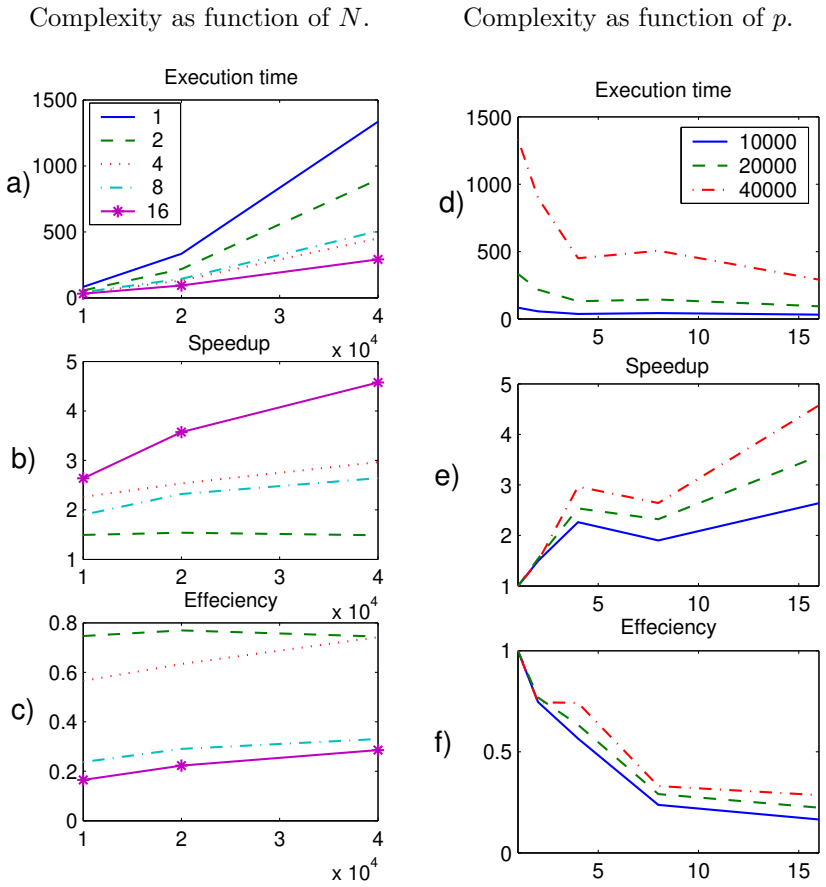


Figure 4: Complexity as function of N and p.

For this particular set of boundary and initial condition, our solution *does not depends on* x and

$$\lim_{t\to\infty} c(y,t) = y. \tag{22}$$

5.2 Numerical approach

In order to find the numerical solution of the above problem, we use explicit discretization in time and central discretization for the second order derivative operators in the space (see [5] for more details). That is, the solution at position $x_l = l\Delta x$, $y_m = m\Delta y$ and $t_{n+1} = (n+1)\Delta t$ is calculated through the following formula

$$c_{l,m}^{n+1} = c_{l,m}^{n} + \frac{D\Delta t}{\Delta x^2}\left(c_{l-1,m}^{n} + c_{l+1,m}^{n} + c_{l,m-1}^{n} + c_{l,m+1}^{n} - 4c_{l,m}^{n}\right), \tag{23}$$

where for simplicity we used the uniform discretizations and assuming $\Delta x = \Delta y$. Using this scheme, to update the concentration at the next time step on a certain grid point (l, m), we only need information about the concentration, on previous time step, on that grid point and on the nearest neighbors $l \pm 1, m \pm 1$. *The scheme is therefore very suitable for parallel computing.*

Remark Being the explicit discretization, our scheme is conditional stable. It is shown that (see [4, 5]) the scheme is stable if

$$D\frac{\Delta t}{\Delta x^2} \leq \frac{1}{4}. \tag{24}$$

5.3 The parallel algorithm and the complexity

In this section, we consider two methods to decompose the spatial domain. These are a trip wise and a block wise decomposition. By "trip wise" we mean that each processor receives a block of $N_k \times N$ data elements ($\sum_{k=1}^{p} N_k = N$) and simulates the time-behavior of this specific part. "Block wise" decomposition means that each processor receives $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$ data elements and simulates the time-behavior of this part.

Similar to the algorithm given in Section 4.3, if we suppose that the processors are organized in a certain topology, e.g. ring/line topology, each processor has its own up and down neighbors in case of "trip wise" and together with left and right neighbors in case of "block wise" decomposition. Having a part of starting values, each processor joins in parallel processing. So, the processor need to exchange the boundary points with its neighboring processors. The internal points are fully parallelized.

To have an insight of these decompositions, we shall have a look at their complexities. Using all assumptions and notations given in Section 4.4, we firstly consider the trip wise decomposition. In each time integration, the calculation is performed on a grid of $N \times N$ points. So,

$$T_1 = N^2 \tau_{calc}. \tag{25}$$

Using trip wise decomposition, each process exchanges $2N$ boundary points with the two neighbors. The execution time, the speedup and the efficiency are then

$$\begin{aligned} T_p &= \frac{N^2}{p}\tau_{calc} + T_{comm} \\ &= \frac{N^2}{p}\tau_{calc} + 2(\tau_{setup} + N\tau_{exchange}), && (26) \\ S_p &= \frac{1}{\frac{1}{p} + \frac{2}{N^2}\frac{\tau_{setup}}{\tau_{calc}} + \frac{2}{N}\frac{\tau_{exchange}}{\tau_{calc}}}, && (27) \\ E_p &= \frac{1}{1 + \frac{2p}{N^2}\frac{\tau_{setup}}{\tau_{calc}} + \frac{2p}{N}\frac{\tau_{exchange}}{\tau_{calc}}}. && (28) \end{aligned}$$

Now, consider the block wise decomposition. The difference from the above method is that the current process exchanges $N/\sqrt{p}$ boundary points with each of 4 neighbors.

Therefore, the communication time is slightly different, that is,

$$\begin{aligned} T_p &= \frac{N^2}{p}\tau_{calc} + T_{comm} \\ &= \frac{N^2}{p}\tau_{calc} + 4(\tau_{setup} + \frac{N}{\sqrt{p}}\tau_{exchange}), \end{aligned} \tag{29}$$

$$S_p = \frac{1}{\frac{1}{p} + \frac{4}{N^2}\frac{\tau_{setup}}{\tau_{calc}} + \frac{4}{N\sqrt{p}}\frac{\tau_{exchange}}{\tau_{calc}}}, \tag{30}$$

$$E_p = \frac{1}{1 + \frac{4p}{N^2}\frac{\tau_{setup}}{\tau_{calc}} + \frac{4\sqrt{p}}{N}\frac{\tau_{exchange}}{\tau_{calc}}}. \tag{31}$$

In term of the communication time, the block wise decomposition is more efficient than the other when p is large (i.e. $4N/\sqrt{p} < 2N$). However implementing the block wise decomposition method requires a bit more complicated. For that reason, we take the trip wise decomposition and implement in C using MPI library.

5.4 Implementation

In this section we discuss in details the algorithm using MPI specification.

As always, the environment is initialized using routines `MPI_Init`, `MPI_Comm_size` and `MPI_Comm_rank`. To create the virtual topology we use `MPI_Cart_create`. The coordinate as well as the rank ID of the current processor are found by `MPI_Cart_get` and `MPI_Cart_rank`. Because of the periodic boundary condition in the x-direction, we use a decomposition in the y-direction and use the ring topology. In this way, each processor is responsible for a part of vertical axis (along the whole horizontal axis). For convenient with the algorithm given in Section 4.4, we use again `left` and `right` to denote the rank ID of the upper processor and the lower processor. These rank IDs are easily found by `MPI_Cart_shift` routine. The data decomposition, i.e. `npoints` and `offset`, are similar to what we had in the previous sections.

In the beginning, the local solution vector `curval`, which is dynamically allocated, is initialized according to (19) and (21). Then, in each iteration we perform one time integration `compute()`. To update the (local) solution vector, the current processor exchanges two arrays of length N with `left` and with `right`. Again, this is done by using `MPI_Sendrecv` call. The updating process is accompanied by considering the boundary condition in the y-direction and the periodic boundary condition in the x-direction. The full code is as follows:

```
void compute(void) {
    MPI_Status  status;
    int         i,j;
    double      l_point[N], r_point[N], l_neighbor[N], r_neighbor[N];
    double      l, r, u, d, m;

    /* Take the first and the last rows in current processor */
    for (j=0; j<N; j++){
        l_neighbor[j]=*(curval+index(0,j));
        r_neighbor[j]=*(curval+index(npoints-1,j));
```

```
    }
    /* Send r_neighbor to process on right while receiving into l_point
       from process on left. Then, send the l_neighbor to process on left,
       while receiving into r_point from the process on right.          */
    MPI_Sendrecv(&r_neighbor[0], N, MPI_DOUBLE, right, rank,
                 &l_point[0],N, MPI_DOUBLE, left, left, ring, &status);
    MPI_Sendrecv(&l_neighbor[0], N, MPI_DOUBLE, left, rank,
                 &r_point[0],N, MPI_DOUBLE, right, right, ring, &status);
    for (j=0; j<N; j++){
        for (i=0; i <npoints; i++){
            if (offset+i == GlbRight){
                *(newval+index(i,j)) = 1.0; /*Boundary condition in y-direction */
            }else{
                if (offset+i == GlbLeft){
                    *(newval+index(i,j)) = 0.0;
                }else{
                    u = (i==0) ? l_point[j] : *(curval+ index(i-1,j));
                    m = *(curval+index(i,j));
                    d = (i==npoints-1) ? r_point[j] : *(curval+index(i+1,j));
                    l = (j==0)  ? *(curval+index(i,GlbRight-1)) : *(curval+index(i,j-1));
                    /* (because of periodicity in x-direction)  */
                    r = (j==N-1) ? *(curval+index(i,GlbLeft+1)) : *(curval+index(i,j+1));
                    *(newval+index(i,j)) = *(curval+index(i,j)) +
                                           D*dt/(dx*dx)*(l+r+u+d - 4.0*m);
                }
            }
        }
    }
    /* Update the arrays */
    for (j=0; j<N; j++){
        for (i=0; i <npoints; i++){
            *(curval+index(i,j)) = *(newval+index(i,j));
        }
    }
}
```

To analyze the data later, the output is written to a file after every `ITV` iterations. It is shown that for our particular initial value, our solution does not depend on x. Therefore, one can take a column from the concentration field, in the current process. Then, that vector, which is denoted by `subvector`, is collected into the root using `MPI_Gather`. The output is printed to a file there. Although this I/O operator will also induce overhead, it has not the negative effect on the efficiency of our parallel program. In fact, we have tested the program with and without that I/O operator. It turns out that the efficiency does not change, more or less. For example, for $\Delta x = 0.01$, if we do not use I/O operator then $E_2 = 0.38, E_4 = 0.34$, while if we use I/O operator then $E_2 = 0.39, E_4 = 0.33$.