ME 489 Homework 05
Parallelization of 2-Dimensional Wave Equation with MPI

Enes Çağlar Korkmazgöz 2370567 - Hasan Kaan Özen 2378586

January 7, 2024

Abstract

This report presents a code using a message passing interface in 2-dimensional wave equation solver parallelize. Strong scaling analysis and speedups are presented and discussed. The results show that as the code is parallelized, the run-time reduces significantly up to a certain point. This limit is depicted in the strong-scaling analysis, where the domain size is kept constant whilst the number of processes increases. Moreover, different grid sizes for resolution analysis are presented, and a mesh-independence study is done. It is shown that domain discretization for each process affects the error, yielding dependency on the mesh size.

Contents

1 Introduction

Parallel computing helps reduce the total run-time in a C-written program. This yields reducing the computational costs and saving money to the user especially when the problem size is large enough and not feasible for a serial solver. In such a parallelization procedure, one may use OpenMP or MPI to utilize different processes in the solution algorithm; however, it was shown in Homework 04 [1] that MPI is more advantageous over OpenMP in terms of time reduction capability. So, in this report, the steps of implementing an MPI in an explicit in-time, 2-dimensional wave Equation 1 are presented, followed by results of total run times and scaling analysis. The findings are then discussed and interpreted.

$$\frac{\partial^2 q}{\partial t^2} = c^2 \left(\frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} \right), \quad 0 \leq x \leq 1, \quad 0 \leq y \leq 1 \tag{1}$$

2 Understanding The Solver

The solver is an explicit in-time solver that initializes the field variable according to initial and boundary conditions. These initial and boundary conditions are obtained from the exact solution provided in the code. Then, for the discretization method, central differencing is applied to solve the field variable at the next time step. The central differencing method for the discretization of time and space is presented in Equation 2, Equation 3, and Equation 4, respectively.

$$\frac{\partial^2 q}{\partial t^2} \approx \frac{q(x, y, t + \Delta t) - 2q(x, y, t) + q(x, y, t - \Delta t)}{\Delta t^2} \tag{2}$$

$$\frac{\partial^2 q}{\partial x^2} \approx \frac{q(x + \Delta x, y, t) - 2q(x, y, t) + q(x - \Delta x, y, t)}{\Delta x^2} \tag{3}$$

$$\frac{\partial^2 q}{\partial y^2} \approx \frac{q(x, y + \Delta y, t) - 2q(x, y, t) + q(x, y - \Delta y, t)}{\Delta y^2} \tag{4}$$

In the discretized version of the time and the domain terms, it may be seen that there are three different time values involved in the equations; these are the previous, current, and next time steps of the field variable q. Section 3 presents the procedure for applying these conditions.

3 Implementation Details

3.1 Domain Decomposition

For the parallelization of a discrete equation, a domain decomposition is a mandatory step to assign every processor a domain to compute in. For the case of this report, the global

domain is divided to equal parts on the vertical axis. The resulting domain size for each processor is presented in Equation 5, where NX is the global number of horizontal nodes, NY is the global number of vertical nodes. For this report, the division on the right-hand side is assumed to be an exact division. The implementation is presented below.

$$N_{size} = NX * \frac{NY}{N_{processors}} \tag{5}$$

```
int nx = NX;       // local number of nodes in x direction
int ny = NY/size;      // local number of nodes in y direction
```

3.2 Uniform Spacing

The uniform space between each node has to be calculated to compute the coordinates of each node. The calculation is presented in Equations 6 and 7. In these equations, "hx" and "hy" refer to uniform spacings in the horizontal and vertical axes, and size refers to the number of processors assigned. In Equation 7, the global vertical size, "ymax-ymin" is not divided by the global number of vertical nodes because although the exact division assumption is made for the calculation of the local number of vertical nodes, using "ny*size" corrects the spacing calculation for non-exact divisions. The implementation is presented below.

$$hx = \frac{x_{max} - x_{min}}{NX - 1} \tag{6}$$

$$hx = \frac{y_{max} - y_{min}}{(ny * size) - 1} \tag{7}$$

```
// find uniform spacing in x and y directions
// Correct uniform y spacing for the local number of nodes as it may be round value.
double hx = (xmax - xmin)/(NX-1.0);
double hy = (ymax - ymin)/((ny*size)-1.0);
```

3.3 Coordinate Calculation

The coordinates of the nodes have to be calculated so that the exact solution and the boundary conditions can be calculated for each node since they depend on position. The horizontal and vertical position calculations for an arbitrary node (i,j) are presented in Equations 8 and 9. It should be noted that the vertical position calculation contains a multiplication with "rank" on the right-hand side. This implementation is necessary for parallelization since we decomposed the domain on the vertical axis and the vertical position of every rank is the

sum of the vertical size of the domains of previous ranks. The implementation is presented below.

$$x[i + j * nx] = x_{min} + i * hx \tag{8}$$

$$y[i + j * nx] = y_{min} + j * hx + (rank * hy * ny) \tag{9}$$

```
// Compute coordinates of the nodes
// Compute y coordinates according to ranks
for(int j=0; j < ny; ++j){
  for(int i=0; i < nx;++i){
    // Every processors vertical size is (rank*hy*ny). This is the MPI correction for
    ↪ vertical position.
    double xn = xmin + i*hx;
    double yn = ymin + (j*hy) + (rank*hy*ny);
    x[i+j*nx] = xn;
    y[i+j*nx] = yn;}}
```

3.4 Memory Allocation and Calculating Initial Solution

There are three different arrays to allocate memory for the solution of the wave equation. These are the solution at the next time step, "qn", the solution at the initial time step, "q0", and the solution at the previous time step, "q1". For a serial code, the required memory for these arrays are the domain size multiplied by the size of a "double" storage. However, for an MPI code, extra memory allocation for the communications between processors is a necessity. Consequently, the communication structure has to be formed in order to allocate correct memory to the arrays.

3.4.1 MPI Communication Structure

Since the problem works on a decomposed domain in the vertical axis, a row of shadow nodes at the size of "nx", the number of horizontal nodes, has to be added to the bottom and top rows of a domain. These rows will store values either from the next rank or the previous rank depending on the row so that central differencing can be calculated correctly. The communication structure is presented in Figure 1.

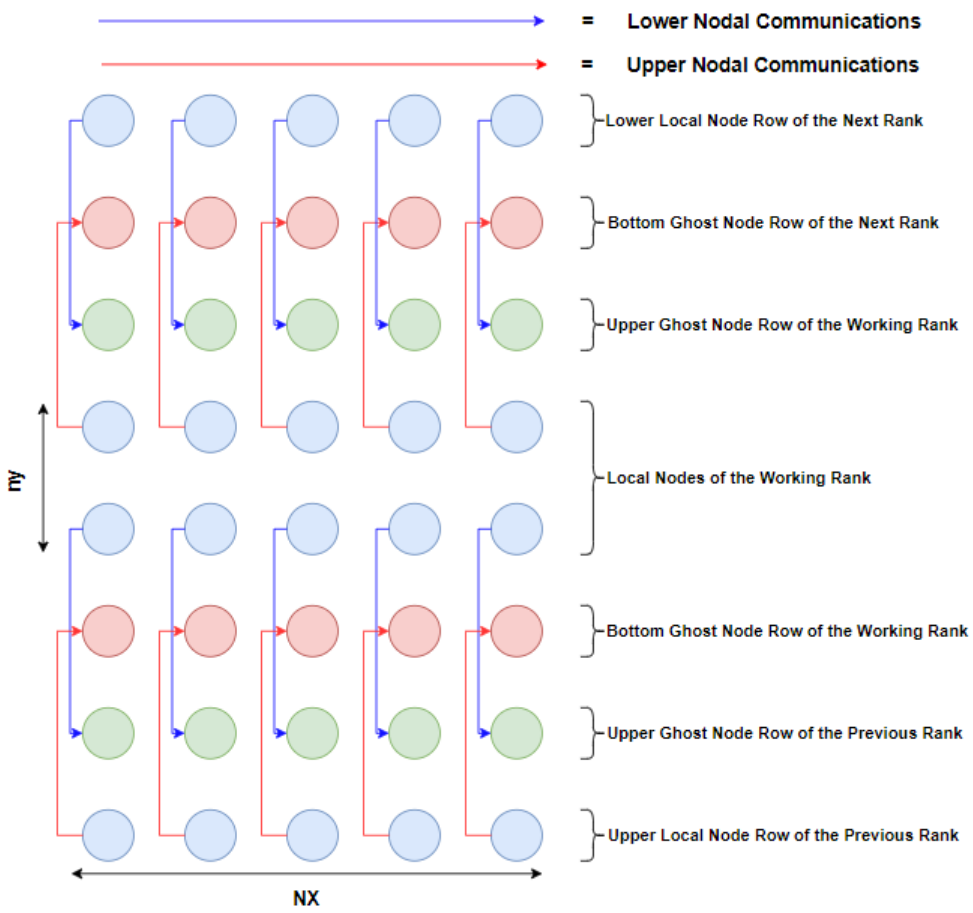

Figure 1: MPI Communication Structure

With this communication structure, every rank will work on a field variable array with an extra size of "2*nx" which is directly equal to two extra rows added to the original local domain. The memory allocation implementation is presented below.

```
// ALLOCATE MEMORY for SOLUTION and its HISTORY
// Create ghost nodes at the lower and upper y axes for MPI communications.
// Solution at time (t+dt)
qn = ( double * ) malloc ( ((nx*ny)+(2*nx)) * sizeof ( double ) );
// Solution at time (t)
q0 = ( double * ) malloc ( ((nx*ny)+(2*nx)) * sizeof ( double ) );
// Solution at time t-dt
q1 = ( double * ) malloc ( ((nx*ny)+(2*nx)) * sizeof ( double ) );
```

3.4.2 Initial Solution

The initial solution has to be calculated for every node except for the boundary nodes so that the necessary history points are acquired for central differencing. The implementation is presented below with the functions that are used. It should be noted that the field variable arrays have an "nx" added to the destination of the value. The reasoning behind it is to avoid any computation for the ghost nodes of a domain which are located in the first and last "nx" sized destination.

```
  // USE EXACT SOLUTION TO FILL HISTORY
    for(int j=0; j < ny; ++j){
      for(int i=0; i < nx;++i){
        const double xn = x[i+j*nx];
        const double yn = y[i+j*nx];
        // Exact solutions at history tstart and tstart+dt
        q0[i+ (j*nx) + nx] = exactSoln(c, xn, yn, tstart + dt);
        q1[i+ (j*nx) + nx] = exactSoln(c, xn, yn, tstart);
    }
  }
```

```
double exactSoln( double c, double x, double y, double t){
  const double pi = 3.141592653589793;
  double value = sin( 2.0*pi*( x - c*t));
  return value;
```

3.5 Boundary Condition

The boundary condition for the problem of this report is directly equal to the exact solution function provided in the previous section. For a global domain size of "NX*NY", the boundaries of this domain will be the following:

- Left Boundary: i = 0
- Right Boundary: i = NX-1
- Bottom Boundary: j = 0
- Top Boundary: j = NY-1

Since we have a decomposed domain in the vertical, the boundary condition function will need corrections such that only the ranks that contain the bottom and top boundaries will have boundary condition calculations for. There is no correction required for the left and

right boundaries since all domains contain boundary nodes in these boundaries. The implementation detail is presented below.

```
    // Apply Boundary Conditions i.e. at i, j = 0, i,j = nx-1, ny-1
    applyBC(q0, x, y, c, time, nx, ny, rank, size);
```

```
void applyBC(double *data,  double *x, double *y, double c, double time, int nx, int ny,
↪  int rank, int size){

  // Apply Boundary Conditions
  double xn, yn;
  //Modfiy Boundary Condition loops according to domain decomposition and ranks.
  for(int j=0; j<ny;++j){ // left right boundaries i.e. i=0 and i=nx-1
    xn = x[0 + j*nx];
    yn = y[0 + j*nx];
    data[j*nx + nx] = exactSoln(c, xn, yn, time);

    xn = x[nx-1 + j*nx];
    yn = y[nx-1 + j*nx];
    data[nx-1 + j*nx + nx] = exactSoln(c, xn, yn, time);
  }

  for(int i=0; i< nx; ++i){ // top and  bottom boundaries i.e. j=0 and j=ny-1
    xn = x[i];
    yn = y[i];
    //Check if rank == 0 since it contains the lower boundary.
    if(rank==0){
      data[i + nx] = exactSoln(c, xn, yn, time);
    }
    xn = x[i+ (ny-1)*nx];
    yn = y[i+ (ny-1)*nx];
    //Check if rank == size -1 since it contains the upper boundary.
    if(rank==(size-1)){
      data[i + nx + (ny-1)*nx] = exactSoln(c, xn, yn, time);
    }
  }
}
```

3.6 MPI Communication Implementation

With the communication structure provided in Figure 1 covered, the implementation details can now be provided. For this implementation, the MPI functions, "MPI_Send" and "MPI_Recv" are used. These functions point to the destination of the first point in the upper or lower ghost/local row where the information will either be received at or sent by,

respectively. Afterward, "nx" amount of elements will be sent or received such that it covers the information on the entire row of subject. The key point here is not using a loop to send information from point to point, ultimately saving total run time. The implementation detail is presented below.

```
    //Set up Communications
    if (rank > 0) {
        // Send the bottom row of the current domain to the upper ghost row of the
        ↪ previous rank
        MPI_Send(&q0[nx], nx, MPI_DOUBLE, rank - 1, downcomms, MPI_COMM_WORLD);

        // Receive the upper row of the previous domain at the lower ghost row of the
        ↪ working rank
        MPI_Recv(&q0[0], nx, MPI_DOUBLE, rank - 1, uppercomms, MPI_COMM_WORLD, &status);
    }
    if (rank < size - 1) {
        // Send the upper row of the current domain to the lower ghost row of the next
        ↪ rank
        MPI_Send(&q0[nx + (nx * (ny - 1))], nx, MPI_DOUBLE, rank + 1, uppercomms,
        ↪ MPI_COMM_WORLD);

        // Receive the bottom row of the next domain at the upper ghost row of the
        ↪ working rank
        MPI_Recv(&q0[nx + (nx * ny)], nx, MPI_DOUBLE, rank + 1, downcomms,
        ↪ MPI_COMM_WORLD, &status);
    }
```

3.7 Central Differencing

With all the necessary implementations covered, the program continues with the central differencing stage. Because of the uniform spacing in time and space, the Equations 2, 3,4 reduce to Equation 10. With this equation in hand, the neighboring nodes have to be calculated for an arbitrary node " **[i + j*nx + nx]** ". The nx addition is necessary to avoid the bottom ghost row of a domain, which is not a local node and will only take part as the southern neighbor, " **[i + j*nx + nx - j*nx]** " and for the upper ghost row, " **[i + j*nx + nx + j*nx]** ". The implementation details are presented below. It should be noted that the boundary nodes are avoided with if conditions such that central differencing will not be applied to these nodes since they depend on the in-time boundary condition.

$$q_{i,j}^{n+1} = 2q_{i,j}^{n} - q_{i,j}^{n-1} + \alpha_x(q_{i+1,j}^{n} - 2q_{i,j}^{n} + q_{i-1,j}^{n}) + \alpha_y(q_{i,j+1}^{n} - 2q_{i,j}^{n} + q_{i,j-1}^{n}) \tag{10}$$

where $\alpha_x = c^2\Delta t^2/\Delta x^2$ and $\alpha_y = c^2\Delta t^2/\Delta y^2$.

```
    // Update solution using second order central differencing in time and space
    for(int j=0; j < ny; ++j){
      for(int i=1; i < nx-1;++i){// exclude left and right boundaries
        const int n0   = i + j*nx + nx;
        const int nim1 = i - 1 + j*nx + nx; // node i-1,j
        const int nip1 = i + 1 + j*nx + nx; // node i+1,j
        const int njm1 = i + (j-1)*nx + nx; // node i, j-1
        const int njp1 = i + (j+1)*nx + nx; // node i, j+1
        // update solution
        //Check if lower boundary
        if(rank == 0){
          if (n0 >= 2*nx){
            qn[n0] = 2.0*q0[n0] - q1[n0] + alphax2*(q0[nip1]- 2.0*q0[n0] + q0[nim1])
                                         + alphay2*(q0[njp1] -2.0*q0[n0] + q0[njm1]);
          }
        }
        //Check if upper boundary
        else if(rank == size-1){
          if (n0 < (nx + nx*(ny-1))){
            qn[n0] = 2.0*q0[n0] - q1[n0] + alphax2*(q0[nip1]- 2.0*q0[n0] + q0[nim1])
                                         + alphay2*(q0[njp1] -2.0*q0[n0] + q0[njm1]);
          }
        }
        else{
            qn[n0] = 2.0*q0[n0] - q1[n0] + alphax2*(q0[nip1]- 2.0*q0[n0] + q0[nim1])
                                         + alphay2*(q0[njp1] -2.0*q0[n0] + q0[njm1]);
        }
      }
    }
```

After finding the solution at the next time step, a new loop is introduced to update the solution arrays. The implementation is presented below. Boundaries are avoided in this section aswell.

```
    // Update history q1 = q0; q0 = qn, except the boundaries
    for(int j=0; j < ny; ++j){
      for(int i=1; i < nx-1;++i){
        int n0   = i + j*nx + nx;
        //Check if lower boundary
        if(rank == 0){
          if (n0 >= 2*nx){
            q1[n0] = q0[n0];
            q0[n0] = qn[n0];
            }
        }
        //Check if upper boundary
        else if(rank == size-1){
          if (n0 < (nx + nx*(ny-1))){
```

```
            q1[n0] = q0[n0];
            q0[n0] = qn[n0];
            }
        }
        else{
          q1[n0] = q0[n0];
          q0[n0] = qn[n0];
        }
      }
    }
```

3.8 Calculating Infinity Norm

The infinity norm is an indicator that shows the maximum amount of difference between the simulation result and the exact solution. In this program, the infinity norm is calculated using the provided definition for every node in a loop, and updating the infinity norm if it is found higher than the previous node. Afterward, since every rank calculates it's infinity norm, the MPI function, MPI_Reduce is used to read the infinity norms of all ranks and write the maximum of them to a variable named global infinity norm. The implementation details are presented below.

```
  // Compute Linf norm of error at tend
    double linf = 0.0;
    for(int j=0; j < ny; ++j){
      for(int i=0; i < nx;++i){
         double xn = x[i+ j*nx];
         double yn = y[i+ j*nx];
         // solution and the exact one
         double qn = q0[i + j*nx + nx];
         double qe = exactSoln(c, xn, yn, time);
         linf  = fabs(qn-qe)>linf ? fabs(qn -qe):linf;
          if (rank==1)
          {
            //printf("    Infinity norm of the error: %.4e %.8e at node %d  \n", qn,
            ↪  time, i+j*nx);
          }
       }
    }
    double global_linf = 0.0;
    MPI_Reduce(&linf, &global_linf, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

3.9 Printing the Results

The code outputs the infinity norm of error, the total run-time as Wall Clock time, and the total number of processors used. The implementation is presented below.

```
1  if(rank==0){
2    printf ( "  Wall clock elapsed seconds = %f for %d number of processors.\n",
     ↪  global_time,size );
3    printf("    Infinity norm of the error: %.4e %.8e \n", global_linf, time);
```

The code can be compiled and executed with the following steps:

- **Compile:** mpicc -o wmpi wave2d_mpi.c -lm
- **Run:** mpirun -np (number of processors) ./wmpi input.dat

4 Discussion and Results

4.1 Performance Measurements

The performance measurements are carried out with both strong-scaling analyses and speedup analyses. These results are done by using different resolutions (namely, 400x400, 200x200, and 100x100 grids) and different number of processes. The hardware used to run the program is depicted in Table 1.

Table 1: Processor Specs

Specification	**Details**
Product	Intel i7-12650H
Total Cores	10
Performance Cores	6
Efficient Cores	4
Total Threads	16
Base Frequency	1 GHz
Max Turbo Frequency	4.7 GHz
Cache	24 MB

4.1.1 Strong Scaling Study

A strong scaling analysis is required to measure the efficiency of increasing the number of processes' effect on the solution time. This is done simply by measuring the wall clock time of different sets of process numbers and then comparing their speeds in a plot while the domain size is kept constant. Our analyses are carried out in three different domain sizes (100x100, 200x200, 400x400), and their results are presented in Figure 2, Figure 3, and Figure 4, respectively.

It may be seen from the strong scaling analyses that increasing the number of processes increases the performance by reducing the wall clock time; however, after a *saturation point*, the increase in the number of processes does not yield better performance. This is due to the fact that as the number of processes increases in a program, the overall messaging procedure starts to dominate the wall clock time. Also, the wall clock time for a greater domain size still seems to decrease with a very small slope. This immediately shows that as the problem size is greater, an increase in the number of processes is required.

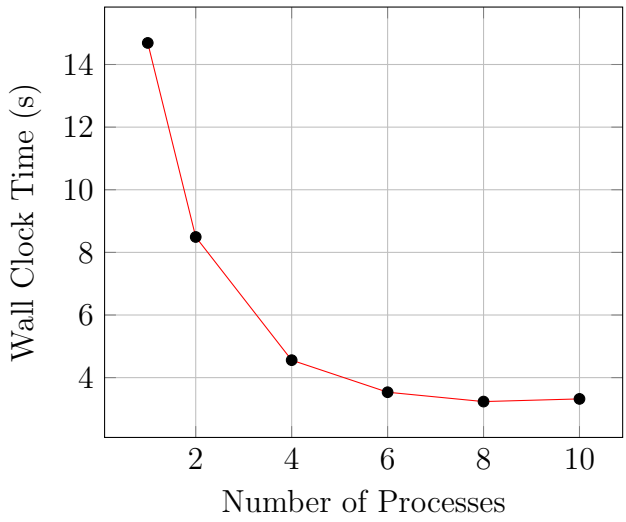

Figure 2: Strong Scaling Analysis for the fixed domain : 100 x 100.

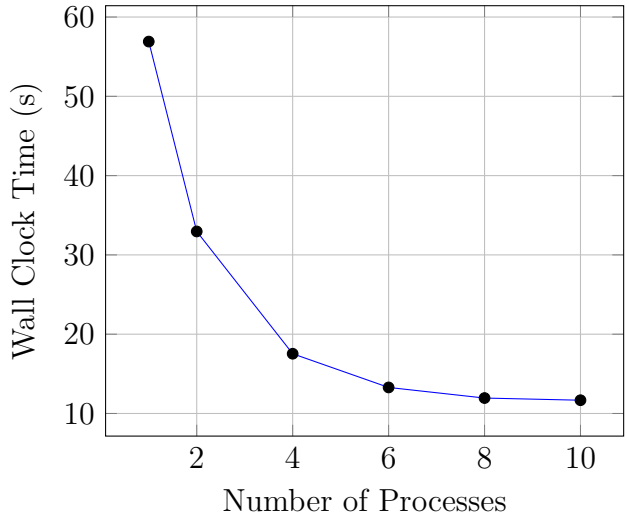

Figure 3: Strong Scaling Analysis for the fixed domain: 200 x 200.

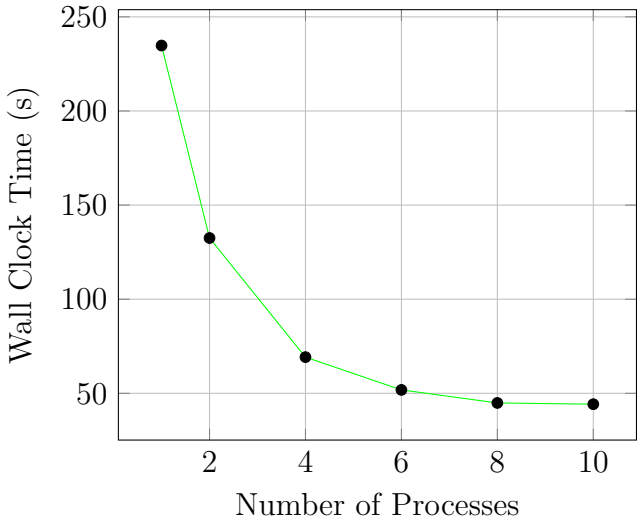

Figure 4: Strong Scaling Analysis for the fixed domain: 400 x 400.

4.1.2 Strong Scaling Speedup Study

A strong scaling speedup study measures the parallelization performance by showing at what fold the number of processes increased the performance of a program. It is the sole result of time taken for a serial process divided by the parallel method with a given number of processes. Again, for the fixed domain sizes (100x100, 200x200, 400x400), the strong scaling speedup analyses are done, and the results are depicted in Figure 5, Figure 6, and Figure 7. In Figure 5, it seems that the speedup ratio increases up to 8 processes; however, after that point, a break-even occurs, and the speedup decreases. As the domain size is 100x100 and relatively small, the messaging between processes takes more time than the speedup supplied by these processes. As a result, the total run-time increases.

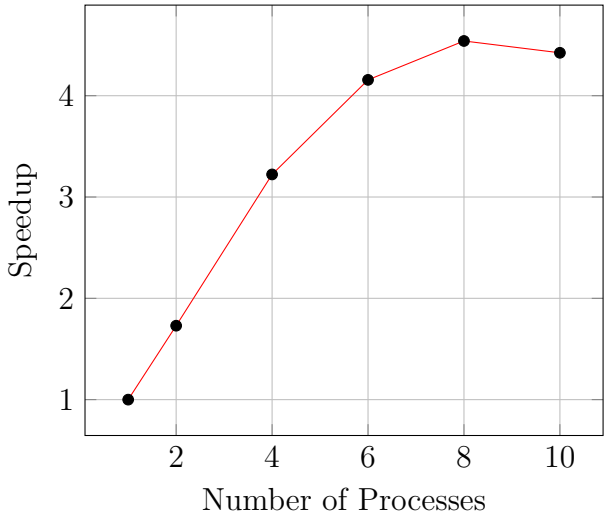

Figure 5: Speedup Analysis for the fixed domain: 100 x 100.

Figure 6 and Figure 7 show the strong scaling speedup analyses for domains 200x200 and 400x400. One may conclude from these plots that increasing process number increases the speedup; however, there is a bottleneck for both domains around process size 9. Although for the case of the 100x100 domain, the speedup decreased after that point, for both 200x200 and 400x400, the speedup continuously increased with a small slope.

Moreover, one may see the speed increase in the 400x400 domain is greater than the 200x200 domain after the four processes implementation. The final result shows that for ten process implementations, the speedup crosses 5-fold for the 400x400 domain, whereas for the 200x200 domain, it stays below the 5-fold value. This can be noted as parallelization is more beneficial for high domain sizes.

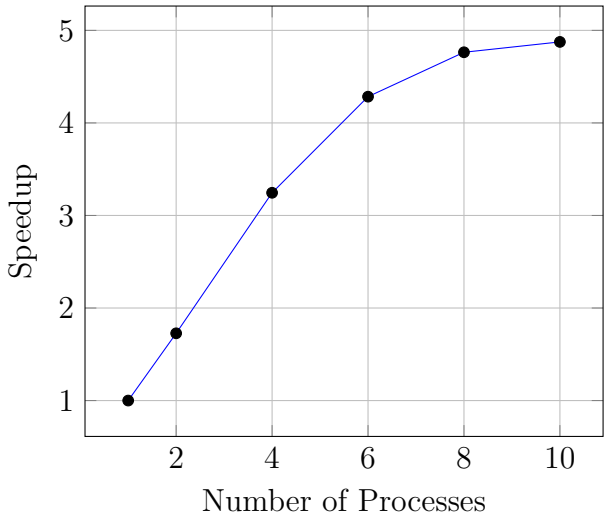

Figure 6: Speedup Analysis for the fixed domain: 200 x 200.

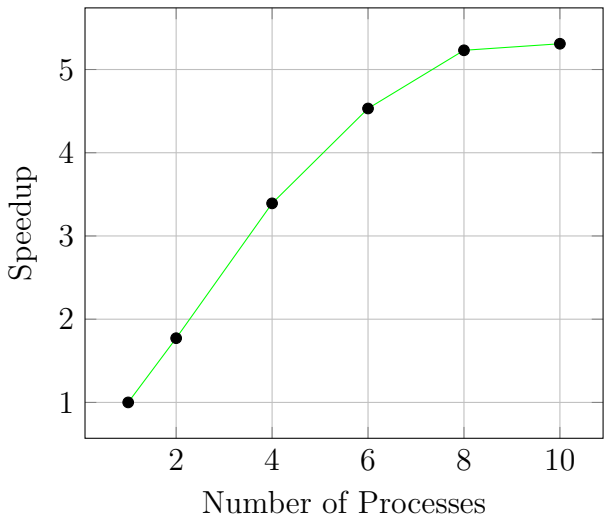

Figure 7: Speedup Analysis for the fixed domain: 400 x 400.

4.1.3 Tabulated Data

There are three tabulated data tables for the resolutions mentioned previously. They depict the communication world size, the run-time of each program, speedup ratios, and the infinite norm of the error.

There is an interesting fact about these data. As described in Section 3 the domain is divided in the y-direction into several processes available. This is done assuming that the y-length of the problem is divisible by the number of processes, i.e., the communication world size. However, when we used six processes, which cannot divide the domains of sizes 100x100, 200x200, and 400x400, the infinite norm of the error is increased at that instant. So, one may also be cautious about this division operation across processes.

We have also constructed a 300x300 domain divisible by the six processes to show this. The results are well-aligned with the output of the serial processor now. This can be seen in the Table 5.

Table 2: Performance for Problem Size 100x100

Processes	Run-time	Speedup	Infinite Norm of Error
10	3.32069	4.423478253	9.08e-4
8	3.2353	4.540228109	9.0811e-4
6	3.5342	4.156244695	9.0811e-4
4	4.5578	3.222826802	9.0869e-4
2	8.4918	1.729786382	9.0869e-4
1	14.689	1	9.0869e-4

Table 3: Performance for Problem Size 200x200

Processes	Run-time	Speedup	Infinite Norm of Error
10	11.6718	4.875426241	3.239e-4
8	11.94614	4.763463345	3.239e-4
6	13.2841	4.283692535	3.2513e-4
4	17.5352	3.245186824	3.239e-4
2	32.957	1.726643809	3.239e-4
1	56.905	1	3.239e-4

Table 4: Performance for Problem Size 400x400

Processes	Run-time	Speedup	Infinite Norm of Error
10	44.216443	5.309495384	2.1767e-4
8	44.881	5.230877209	2.1767e-4
6	51.807	4.531569093	2.1766e-4
4	69.217	3.391753471	2.1767e-4
2	132.51	1.771692702	2.1767e-4
1	234.767	1	2.1767e-4

Table 5: Error for Problem Size 300x300

Processes	Infinite Norm
6	2.4363e-04
1	2.4363e-04

5 Conclusion

In this project, we have implemented MPI to parallelize a 2-dimensional wave equation solver, which is explicit in time. The solver is first run in a serial processor, and then the implementation results of the MPI for the parallel solver are depicted in tables and figures. This implementation is time-consuming, especially compared to the OpenMP applications; however, the results show that this approach effectively reduces the run-time whilst preserving the results' accuracy.

The application is based on the fact that MPI's distributed memory structure. Successful implementation of MPI then requires *shadowrows* approach, as the domain is divided into sub-domains by splitting it in the y-direction. This is done by sending and receiving information on the node of interest (boundary/shadow nodes) to the next and previous ranks, respectively. This approach also requires a thorough calculation of the world size and the domain size, as it is assumed that the domain is divisible by the number of processes available.

Measuring the performance of the different domains and process sizes is the report's main goal. So, a strong-scaling study is carried out to compare the results. Not only wall clock times but also speedups are compared, and it is concluded that as the domain size increases, the parallelization is more beneficial to the user, and as the processor number is increased, after a certain instant, the costs take over the benefit from the parallelization as the message passing between processes takes longer times to compute.

Furthermore, saving energy is increasingly important in today's high-performance computing (HPC) systems. As you add more processes, it directly impacts how much power is used. Therefore, energy consumption is a key consideration when comparing MPI and OpenMP for decision-making.

In conclusion, while MPI boasts superior scalability and flexibility in distributed environments, choosing a number of processes is another consideration as we do not have unlimited resources, and also unlimited resources (processes availability) do not always result in higher performances. Users should be informed of the task's specific requirements, including the problem's scale, the available hardware, and the desired balance between computational performance and energy efficiency to choose the number of processes.

6 References

[1] Korkmazgoz, Ozen, ME489 HW4 (2023).
[2] Lecture notes of Ali Karakuş for ME489 (2023).