CSE260 Programming Assignment 3 Report

Aronya Baksy, Syed Mujtaba Jafri

November 29, 2024

Introduction

In this assignment, we implement an approximate simulation of a 2-dimensional wave equation using the method of finite differences over a 5-point stencil using MPI. The continuous wave equation is given as:

$$\frac{\partial^2 u}{\partial t^2} - \alpha^2 \left(\frac{\partial^2 u}{\partial i^2} + \frac{\partial^2 u}{\partial j^2}\right) = 0 \tag{1}$$

Here u is the field variable being measured (e.g. amplitude, wave energy etc.) and α is a constant representing the *propagation speed* of the wave. We approximate this second-order partial differential equation using the method of *finite differences*, and solve the below equation for $u(t+d, i, j)$:

$$\frac{u(t+d) + u(t-d) - 2u(t)}{d^2} - \alpha^2 \frac{u(i-h,j) + u(i+h,j) + u(i,j-h) + u(i,j+h) - 4u(i,j)}{h^2} = 0 \tag{2}$$

Setting $d = h = 1$ and solving the above for $u(t+d)$ yields the simulation that we implement in code. We repeat this procedure for a fixed number of iterations (this is called a 5-point stencil as Equation 2 makes it clear that $u(t+d)$ is dependent only on its 4 neighbors to the north, south, east and west in space).

This 2-dimensional field is split into tiles, with one processor computing the wave equation at a given timestamp t for its specified tile. Since the processors need to exchange the boundary values of their respective tiles with each other, this exchange of boundary cells (a.k.a *ghost cells*) is achieved using MPI. Our implementation is tested on the San Diego Supercomputer Center's *Expanse* cluster which has 728 compute nodes having 128 processor cores each.

We use the following symbols through the following sections of this report:

1. The processing cores are arranged in a rectangular grid, with p_x cores along the x dimension and p_y cores along the y dimension.
2. The global size of the problem is $N \times N$. This program is only tested on square grid sizes, but extending to non-square problems should be trivial.
3. Each processing core handles a rectangular sub-tile of the global problem. We define the size of this portion as $m \times n$. Not that m and n may not be equal to each other, and that different processors in the grid may handle sub-problems of different sizes, as N may not be an exact multiple of p_x and p_y (this case is described below).

1a) Program Overview

The program utilizes the following C++ classes in its operation:

- `ArrBuff`: This contains buffers for the current timestamp t, as well as corresponding buffers for timestamps $t-1$ and $t+1$. A fourth buffer holds the value of α. Each buffer is of size $(m+2) \times (n+2)$ with the 2 additional rows and columns being added to hold the *ghost cells*. In addition to providing storage for these buffers, the `ArrBuff` class and its parent `Buffers` class offer the following utilities:
    1. Translating co-ordinates from the global problem space to the local co-ordinate space of a single processor
    2. Checking if a given set of global co-ordinates lie within the bounds of the current processing element's buffer.
    3. Accessor methods to get the value or address of any point in any of the buffers.

- `Compute`: This class runs the simulation over the specified number of iterations, reading information about the initial stimuli from a JSON configuration file, and setting the initial conditions, and then running the simulation of the wave equation for the specified number of iterations. Methods to compute the field variable u are present in this class
- `Stats`: This class accumulates the values of the required statistics (in our case the maximum value in the grid, and the sum of squares of the grid elements) from all the processors and reduces them to a single value in the root process (i.e. the process with MPI Rank 0) using the appropriate reduction function. This class can be configured to print these summary statistics periodically, as well as at the end of the simulation.

Global Structure of Program

The program follows the following overall structure of Figure 1. Note that we split the ghost-cell exchange into two parts: the first part simply sends the boundary cells in all four directions, and receives the corresponding ghost cells from neighboring processors. The second part waits for all the ghost cell exchanges to finish, and then unpacks the columns into the actual ghost cells in our buffer. In between performing the actual exchange and waiting for the results of the send/receive operations, we perform the computation for the **interior cells** of the problem, since they are not dependent on ghost cell values (this is done in the `calcU` function. Once the ghost cells are received and unpacked, we perform the computation for the edge cells in the `calcEdgeU` function.

```
//Stimulus loop
iter = 0;
while (!sList.empty() && iter < cb.niters)
{
    // generate stimulus from sList
#ifdef _MPI_
    // send and recv ghost cells
#endif
    calcU(u);
#ifdef _MPI_
    // Waitall(); unpack incoming ghost cells into grid
#endif
    calcEdgeU(u, kappa);
    u.AdvBuffers();
    iter++;
}
// Finishing remaining iterations
for(; iter< cb.nIter; iter++)
{
    #ifdef _MPI_
        // send and recv ghost cells
    #endif
        calcU(u);
    #ifdef _MPI_
        // Waitall(); unpack incoming ghost cells into grid
    #endif
        calcEdgeU(u, kappa);
        u.AdvBuffers();
        iter++;
}
```

Figure 1: Overall Code Structure for solving 2D Wave Equation

Splitting the Global problem efficiently

We lay out the following condition: the values of m and n for a processor in the grid cannot exceed the value of m and n for any other processor by more than 1. The following snippet in the `Buffers` class enforces this condition:

p_x and p_y are the number of processors in the grid of processing elements along the x and y directions respectively. Along both dimensions, we simply compute the remainder of the division between the global dimension and the processor grid dimension, and split this remainder evenly between the processing elements. This computation is done in the functions `getExtraRow()` and `getExtraCol()`. The splitting condition implies that the global problem will be split between the processors as evenly as possible.

```
// cb.m,25n is global problem size, cb.px,py is processor geometry
n = cb.n / cb.px + (getExtraCol() ? 1 : 0);
m = cb.m / cb.py + (getExtraRow() ? 1 : 0);

bool getExtraRow(int myRow, int m, int py)
{
   return((m % py) > (myRow));
}

bool getExtraCol(int myRank, int n, int px)
{
    int myCol = myRank % px;
    return((n % px) > myCol);
}
```

Figure 2: Splitting global grid between processors evenly

Ghost Cell Exchanges

Since the 5-point stencil requires information about all 4 neighbors of a given point, we must exchange the boundaries of each grid with the neighboring processing elements as well for them to be able to compute the stencil. This exchange requires that each buffer contain two extra rows and columns which are referred to as the *ghost cells*. Exchanging ghost cells with a northern or southern neighboring processor is fairly simple on a row-major order machine, as explained in Figure 3.

```
if (!topGlobalEdge)  // Replace with botGlobalEdge for southward exchange
{
    // CORNER_SIZE=1
    // Receive value for ghost cell from PE to the north
    MPI_Irecv(u.cur(0, CORNER_SIZE), u.N, MPI_DOUBLE,
    myRank - cb.px, SOUTH, MPI_COMM_WORLD, rcvRqst + messageCounter);

    // Send top boundary of grid to the PE north
    MPI_Isend(u.cur(CORNER_SIZE, CORNER_SIZE), u.N, MPI_DOUBLE,
    myRank - cb.px, NORTH, MPI_COMM_WORLD, sndRqst);

    // Increment counter so that we can use MPI_Waitall
    messageCounter += 1;
}
```

Figure 3: Northward Ghost Cell Exchanges

We simply provide the MPI send and receive calls with the start address of the row to send, as well as the number of elements to send (n) and a "tag"(`NORTH`/`SOUTH`) that separates messages being sent northwards and southwards. Note the use of `IRecv` and `ISend` with "I" indicating that we use **asynchronous** sends and receives, which necessitates waiting for the exchanges to finish before we proceed with the computation.

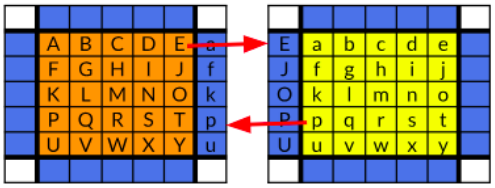
Ghost cell exchanges along a column boundary are more complicated as they involve sending columns of data. We have the option of using the `MPI_Type_vector` routine with a custom stride length to create a custom datatype to store the column data (since the machine is a row-major order machine, we need to do strided access to access a single column). This technique is described in [1]. However, we choose not to use any custom MPI vector data types for this purpose, instead choosing to allocate buffers for the eastward and westward exchange as shown in Listing 4. This choice is made primarily to enhance the readability of our code,

We first pack the values in the column to be sent out into the buffer `out_W` before using the `MPI_ISend` and `MPI_IRecv` calls as before.

The use of **asynchronous communication** means that we also require some waiting mechanism, which is described in Listing 5.

We use the `MPI_Waitall` function to wait for all the ghost cell exchanges to complete. It takes in the array of type `MPI_Request` named **rcvRqst** and blocks until all the requests in the array complete (recvStatus is an output parameter used by the Waitall function).

Two adjacent PEs - ghost cells not shown



Two adjacent PEs - with ghost cells shown in blue;
red arrows show an example ghost cell transmission.

```
double* in_W = new double[u.M];
double* out_W = new double[u.M];

if (!leftGlobalEdge) // Replace with rightGlobalEdge for eastward exchange
{
    // Copy elements from left column of grid into out_W
    for(i = 1; i < u.gridM - 1; ++i){
        out_W[i-1] = u.curV(i, 1);
    }

    // Receive values into buffer in_W
    MPI_Irecv(in_W, u.M, MPI_DOUBLE, myRank - 1, EAST, \
        MPI_COMM_WORLD, rcvRqst + messageCounter);

    // Send out_W
    MPI_Isend(out_W, u.M, MPI_DOUBLE, myRank - 1, WEST, \
        MPI_COMM_WORLD, sndRqst + 2);

    messageCounter += 1;
}
```

Figure 4: Westward Ghost Cell Exchange

```
// Wait for all ghost cell exchanges to complete
MPI_Waitall(messageCounter, rcvRqst, recvStatus.begin());

if (!leftGlobalEdge)
{
    // Populate values from in_W
    for(i = 1; i <= u.M; ++i){
        *(u.cur(i, 0)) = in_W[i-1];
    }
}
// repeat above for eastward exchanges (rightGlobalEdge)
```

Figure 5: Waiting and Unpacking Mechanism for Ghost Cell Exchange

Handling Global Edges in the problem space

In the snippets above, we have made use of four Boolean variables named `leftGlobalEdge`, `rightGlobalEdge`, `topGlobalEdge`, and `botGlobalEdge`. This is computed in the following fashion:

```
/// my_pi, my_pj: position of the process in the x-by-y grid of processes
// (we distribute the processes in row major order)
my_pi = myRank / cb.px;
my_pj = myRank % cb.px;
// Set top and bottom global edges based on my_pi
if (my_pi == 0)
    topGlobalEdge = true;
if (my_pi == cb.py - 1)
    botGlobalEdge = true;

// Set left and right global edges based on my_pj
if (my_pj == 0)
    leftGlobalEdge = true;
if (my_pj == cb.px - 1)
    rightGlobalEdge = true
```

Figure 6: Detecting Cells on global edges of the problem grid

The snippet in Listing 6 shows how these values are computed. We calculate the current position in the processor grid using the MPI **rank** assigned to the current processor which can be obtained using the `MPI_Comm_rank` routine, and use the co-ordinates within the grid of processors to determine if the current processor is on a global edge.

Generating summary statistics

The summary statistics that we compute are the maximum value of the grid, as well as the sum of the squares of each element in the grid (also referred to as the L2 Norm). We use the `MPI_Reduce` function to get the local maximum and L2 Norm values, and accumulate them in the MPI Rank 0 process (a.k.a. the **root process**). The local values are computed by methods in the `Stats` class.

```
MPI_Reduce(&maxV, &globalMaxV, 1, MPI_DOUBLE,
    MPI_MAX, 0, MPI_COMM_WORLD); // max

MPI_Reduce(&sumSq, &globalSumSq, 1, MPI_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD); // L2 Norm
```

Figure 7: Collecting summary statistics from all processors

Vectorization of the Compute Kernel

Here, we refer to the kernel as the core of the program which handles the actual computation. This kernel can be visualized as 2 nested loops which sweep over every cell in the local grid and compute the values at each point using the 5-point stencil. We **vectorize** this kernel using x86 SSE2 vector intrinsics [2]. The vectorized method computes the value of U_{ij} and $U_{i,j+1}$ (marked in orange) in a single loop iteration by loading the values marked in red, green, violet and blue in the left, and using a combination of vectorized add, multiply and subtract operations for the computation.

$$\begin{array}{cccccc}
\vdots & \vdots & \vdots & \vdots & & \\
\cdots & U_{i-1,j-1} & U_{i-1,j} & U_{i-1,j+1} & U_{i-1,j+2}\cdots & \\
\cdots & U_{i-1,j-1} & U_{ij} & U_{i,j+1} & U_{i,j+2}\cdots & \text{and} \\
\cdots & U_{i+1,j-1} & U_{i+1,j} & U_{i+1,j+1} & U_{i+1,j+2}\cdots & \\
 & \vdots & \vdots & \vdots & \vdots &
\end{array}
\qquad
\begin{array}{ccccc}
\vdots & \vdots & \vdots & \vdots & \\
\cdots & U_{i-1,j-1} & U_{i-1,j} & U_{i-1,j+1} & U_{i-1,j+2}\cdots \\
\cdots & U_{i-1,j-1} & U_{ij} & U_{i,j+1} & U_{i,j+2}\cdots \\
\cdots & U_{i+1,j-1} & U_{i+1,j} & U_{i+1,j+1} & U_{i+1,j+2}\cdots \\
 & \vdots & \vdots & \vdots & \vdots
\end{array}$$

1b) Development Process

We iteratively work through the development of this program in the following steps:

- Implement basic functionality such as the utility functions that translate global co-ordinates to local, as well as MPI calls to collect summary statistics. At this point the code is not fully functional since we have not implemented ghost cell exchanges.

- Implement ghost cell communication along the **rows**. This restricts the processor geometry to a single column ($p \times 1$ geometry). This is simpler than the case of column-wise ghost exchange since there is no need for buffering the output and input columns.
- We then implement ghost cell communication along the **columns**. This involves buffering the output columns to send to the neighbors, as well as unpacking the input values received into the ghost cells. Now the program works on arbitrary process geometries $p_x \times p_y$.
- **Reorder** the compute kernel and the MPI communication calls to compute the edges and the interior cells of the grid separately. We do this so that we can process the interior cells of the grid while waiting for the ghost cell values to arrive from the neighboring cells. Once the ghost cell values are communicated, we work on calculating the edge values of the grid.
- **Vectorize** the compute kernel using x64 intrinsics. We restrict our implementation to a vector length of 2, computing two points in a row with a single iteration as described above.
- Test process geometries on Expanse and check for presence of strong and weak scaling.

1c) Optimization Process

Our primary optimizations are two-fold:

- We split the computation of the entire grid into *interior cells* and *edge cells*. While we wait for the ghost celll values to arrive from the neighbouring processors, we can compute the interior cells (since they are not dependent on ghost cell values), and compute the edges once the ghost cell values arrive
- We vectorize the computational kernel using the process described in Section 1b. This vectorization makes use of x64 SSE intrinsics like `_mmu_load_pd` to load two adjacent doubles, and then compute the stencil using intrinsics like `_mmu_add_pd`, `_mmu_sub_pd` and `_mmu_mul_pd` that operate on a 128-bit vector.

2a) Computational Performance vs Workload

From Equation 2, we see that calculating a single point needs 9 floating point operations. Hence, we convert the figure for Mpts/sec to Mflops/sec by multiplying by 9. Dividing this number by 1000 gives us the value in GFlops/sec.

2b) Single-Core Performance

We compare the performance of the MPI code running on a single processor, to that of the starter code which does not contain any MPI calls. We obtain the following results as shown in Table 1.

Cores	Type	Mpts/sec	GFlops/sec
2 (2x1)	MPI Enabled	76.590	0.689
1	Single CPU	33.313	0.2998

Table 1: Speedup between multicore MPI and non-MPI versions

We thus calculate the absolute speedup as:

$$\text{Absolute Speedup} = \frac{\text{Perf. of MPI Code}}{\text{Perf. of Serial Code}} = \frac{76.590}{33.313} = 2.299$$

2c) Strong Scaling study on small number of cores

The intent of this experiment is to demonstrate that our implementation shows **strong scaling** (i.e. an increase in performance as more processors are used, keeping the problem size constant) for number of processors less than or equal to 16. We fix the problem size to be 1000×1000 for this experiment. We also report the overhead of communication. This is implemented using a command-line argument to our program (`-k`) which disables all ghost cell exchanges (as well as the unpacking stage for column buffers), but **does not** disable the communication required to collect the maximum and L2 norm values from the processors, since this is a necessary part of our program that avoids numerical instability (i.e. avoids `nan` values as results).

We see that there is a very small overhead of communication, since the number of processors is very small which results in minimal communication (since a single node in Expanse has 128 cores, all the experiments here

Cores	Geometry	Mpts/sec	GFlops/sec
1	1x1	45.101	0.406
2	2x1	76.590	0.689
4	4x1	157.146	1.414
8	4x2	305.018	2.745
16	8x2	616.903	5.552

Table 2: Performance for $N = 1000$ up to 16 cores

Cores	Geometry	Standard (Mpts/s)	noComm (Mpts/s)
1	1x1	45.101	45.101
2	2x1	76.590	89.154
4	4x1	157.146	180.310
8	4x2	305.018	340.657
16	8x2	616.903	663.790

Table 3: Comm. Overhead for $N = 1000$ up to 16 cores

ideally run on a single node, and hence all MPI communication is within a single socket, which is far cheaper than communication across sockets or across entire nodes).

We calculate the absolute speedup using the formula: speedup $= \frac{perf_P}{perf_1}$ where $perf_1$ is the single core performance calculated as 33.313 Mpts/s. We then graph the speedup obtained as a function of the number of processors in the log-log plot in Figure 8 (the black line is a line of best fit plotted to visualize the linear relationship).

Cores	Perf. (Mpts/s)	Speedup	Efficiency
1	45.101000	1.353856	1.353856
2	76.590000	2.299102	1.149551
4	157.146000	4.717258	1.179314
8	305.018000	9.156125	1.144516
16	616.903000	18.518386	1.157399

Table 4: Absolute Speedup and Efficiency for N=1000

The almost linear growth seen in Figure 8 demonstrates strong scaling of our implementation.

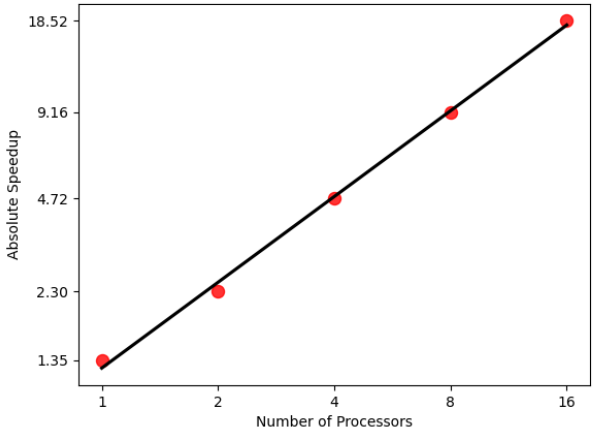


Figure 8: Speedup for $N = 1000$ on 1 to 16 processors

2d) Strong Scaling study on medium number of cores

Cores	Geometry	Mpts/sec	GFlops/sec
16	8x2	591.804	5.326
32	8x4	1148.765	10.339
64	8x8	2739.726	24.658
128	16x8	4839.685	43.557

Table 5: Performance for $N = 2000$ up to 128 cores

Cores	Geometry	Standard (Mpts/s)	noComm (Mpts/s)
16	8x2	591.804	648.246
32	8x4	1148.765	1242.429
64	8x8	2739.726	2775.850
128	16x8	4839.685	4932.182

Table 6: Comm. Overhead for $N = 2000$ up to 128 cores

We repeat a similar calculation to obtain the absolute speedup and efficiency. The results are graphed in a log-log plot in Figure 9.

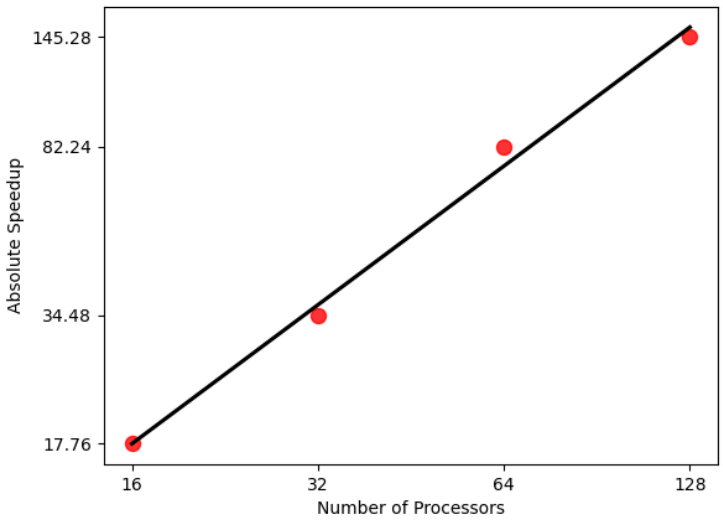


Figure 9: Speedup for $N = 2000$ on 16 to 128 processors

We continue to see a linear increase in performance which shows us that our implementation continues to showcase strong scaling behavior.

2e) Strong Scaling study on large number of cores

Cores	Geometry	Mpts/sec	GFlops/sec
128	16x8	4875.076	43.876
192	32x6	7774.538	69.971
256	32x8	9353.079	84.177
384	64x6	13377.926	120.401

Table 7: Performance for $N = 4000$ up to 384 cores

Cores	Geometry	Standard (Mpts/s)	noComm (Mpts/s)
128	16x8	4875.076	4949.985
192	32x6	7774.538	7809.958
256	32x8	9353.079	9764.036
384	64x6	13377.926	14427.412

Table 8: Comm. Overhead for $N = 4000$ up to 384 cores

We repeat a similar calculation to obtain the absolute speedup and efficiency. The results are graphed in a log-log plot in Figure 10.

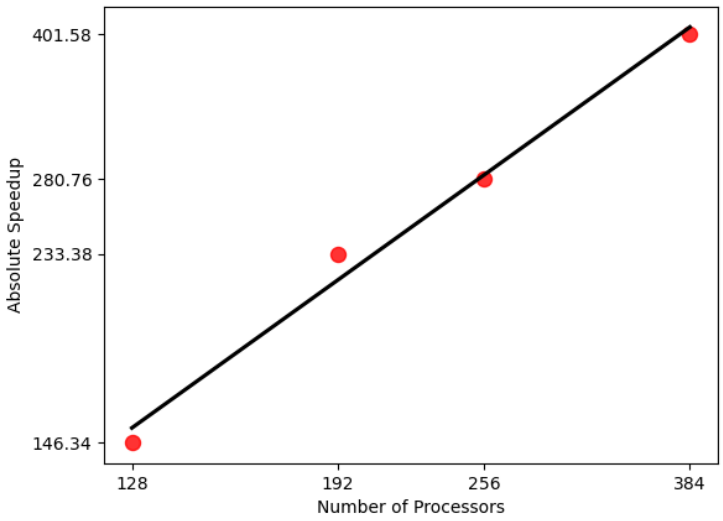


Figure 10: Speedup for $N = 4000$ on 128 to 384 processors

We continue to see a linear increase in performance which shows us that our implementation continues to showcase strong scaling behavior.

2f) Differences in communication overhead

We now plot the communication overhead for number of processors ranging from 2 to 384. We see that for number of processors less than 128, there is very minimal communication overhead. This is because in Expanse, a single compute node contains 128 processors, and hence for less than 128 cores, all communication between cores would be within a single socket or at worst, a single node of the cluster. The graph clearly shows that once we cross 128 cores, the communication overhead starts to increase as the standard version's performance suffers compared to the noComm version with communication turned off. This is because for > 128 nodes, we need to rely on the datacenter network to communicate between cores, which adds additional overhead compared to communicating within a single socket/node.

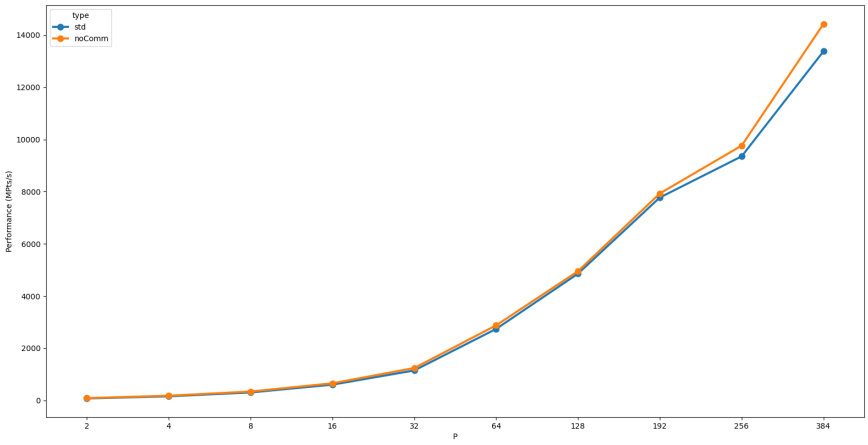


Figure 11: MPI Communication overhead for 2D Wave Simulation

3) Geometry Study

The intent of the following experiments is to demonstrate the ideal process geometry. We must choose a processor geometry that balances computational workload with the overhead of communication. The communication overhead undertaken by a single sub-grid of the problem of size $m \times n$ is proportional to its perimeter (for cells on the interior of the global problem). We define the computational intensity q as:

$$\begin{aligned} Q &= \frac{n_{FLOPS}}{n_{DATA}} \\ &= \frac{mn}{2(m+n)} \end{aligned}$$

where n_{DATA} is the number of data movement operations conducted, which is proportional to the *perimeter* of the local grid. Clearly, maximizing Q involves minimizing the perimeter. For a given number of processors, the size of the region processed by each processor remains constant, hence the constraint for the minimization problem becomes $mn = k$ (where k is some constant). Solving this minimization problem yields:

$$m = n = \sqrt{k}$$

which seems to lead us in the direction of a **square** grid and a **square** process geometry to get the highest computational intensity. We now see the results of our geometry study confirm this or not.

3a) Geometry study on small number of processors

The goal of this experiment is to build an idea of the processor geometry that delivers the best performance in terms of number of points processed per second. We fix the number of processors used as 32, and the size of the problem as 1000×1000.

Cores	Geometry	Mpts/sec	GFlops/sec
32	1x32	954.198	8.587
32	2x16	957.854	8.620
32	4x8	1029.336	9.264
32	8x4	1080.497	9.724
32	16x2	1086.957	9.782
32	32x1	1053.741	9.484

Table 9: Performance and processor geometry for $N = 1000$ using 32 cores

We see from the results in Table 9 that there is a noticeable increase in performance as we transition from a long and wide geometry (1x32) to a tall and thin geometry. This is primarily because of the difference in communication cost when exchanging ghost cells between row boundaries (northward or southward) and column boundaries (eastward or westward), since a column exchange requires additional overhead in terms of copying the output data as well as unpacking the received inputs. In a tall and thin processor geometry, the number of eastward/westward ghost cell exchanges are minimized, hence resulting in better performance. We see this validated as we progress with the remainder of the geometry study.

3b) Geometry study on medium number of processors

We repeat the same experiment as above, with a larger problem size (2000×20000) and more processing cores (128).

Cores	Geometry	Mpts/sec	GFlops/sec
128	4x32	4616.272	41.546
128	8x16	4796.163	43.165
128	16x8	4839.685	43.557
128	32x4	4781.829	43.036
128	64x2	4550.626	40.956

Table 10: Performance and processor geometry for $N = 2000$ using 128 cores

We see from the results in Table 10 that the top-performing geometries are 8x16, 16x8 and 32x4. The results also clearly show the preference for taller and thinner processor geometries, due to the reasons outlined above. Another interesting trend which is clear from this set of results is that the preference for taller and thinner geometries diminishes beyond a point, as a geometry that is too tall (i.e. has too many cores along the y dimension) will have more communication overhead compared to a shorter, more balanced geometry. The factor which prevents us from favoring geometries that are too tall or too thin is caching. In a grid that is too tall and thin, we will have each processor managing a very wide portion of the problem (i.e. we will have $m >> n$). This implies that each row will be longer, hence more cache misses will occur as we move along the row. Hence we want a process geometry that balances between the communication overhead and the cache misses.

3c) Geometry study on large number of processors

We repeat the same experiment as above, this time with an even larger problem size (4000×4000) and 256 processor cores.

Cores	Geometry	Mpts/sec	GFlops/sec
256	8x32	8339.124	75.052
256	16x16	8507.621	76.568
256	32x8	9353.079	84.177
256	64x4	8992.132	80.929
256	128x2	8804.109	79.237

Table 11: Performance and processor geometry for $N = 4000$ using 256 cores

The most surprising result from the above is the fact that the square geometry does not offer the best performance, even though it is within 10% of the best performing geometry. This is because at such a large geometry, the communication overhead dominates the computation.

We see a similar preference for tall and thin process geometries, and we also clearly see the effects of process geometries becoming too tall and too thin. We see the similar pattern of favouring taller and thinner process geometries, but not too tall and thin which makes us lose the benefits of cache locality.

4) Scaling Behaviour and Analysis

In Figures 8, 9 and 10, we see a linear increase in performance as we increase the number of processors while keeping the problem size constant. We can conclude that this program demonstrates strong scaling, since the conditions are satisfied.

In order to demonstrate weak scaling, we demonstrate the performance of our code on the following test cases. The test cases are chosen so as to keep the work per processor constant, hence the problem size scales at the same rate as the number of processors.

Cores	Problem Size (N)	Performance (Mpts/s)
1000	64	2797.203
2000	128	4839.685
4000	256	9353.079

Table 12: Weak Scaling - Performance scaling with processor size and problem size

From Table 12 it is clear that our code shows weak scaling for the range of problem size and number of processor sizes that we have shown. The linear behaviour of the weak scaling is in agreement with Gustafson's Law, which states that the speedup grows linearly with more processors added, for a well-designed parallel program.

5) Sources of error

The following are potential sources of error in our program:

- **Synchronization Bugs**: If the `MPI_Waitall()` call exits before the ghost cell communication is complete, then the ghost cell values that are unpacked will be incomplete, resulting in incorrect answers
- **Network Errors**: Due to improper error checking or errors at the physical layer of the interconnect between nodes, ghost cell values may not be properly communicated
- **Compute Kernel Errors**: We fixed a bug in our implementation wherein, because of incorrect loop start and end indices, we were computing the edges of the local subproblem twice, instead of once (the edges are only to be calculated in the `calcEdgeU()` routine).

6) Potential Future Work

Some ideas for potentially improving the performance of our computation are as follows:

1. Test with larger vector widths: For now, our compute kernel uses a vector length of 2 doubles (i.e. 128 bits). The effect of increasing the vector width on the computation performance would be an interesting experiment to carry out.
2. Use OpenMP to parallelize the code. This seems like a powerful choice to increase the single core performance of the code since parallel threads can compute different portions of the local grid.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN: 0262527391.

[2] Intel Technology. *Intel(R) Intrinsics Guide*. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Intel Technology, June 2024.