Parallel Computing
ECSE 420

Roman Andoni - 260585085
Yordan Neshev - 260587938

SIMULATION OF A SET OF FINAL ELEMENTS USING MPI

Lab Experiment Report

Prof. Zeljko Zilic
McGill Faculty of Engineering

November 8th, 2016

Abstract:

In this experiment, it was required to write a drum simulation algorithm that would mimic a perturbation on a square surface of a drum. Specifically, it was required to implement the drum surface use a using a two-dimensional grid of finite elements. The finite element model allows to simulate a complex physical object as a group of simple object that behave and interact with each other according to physical laws and properties of the object itself. For the simulation, it was required to introduce a perturbation to the simulated surface. The perturbation then would propagate to the neighboring parts of the surface up till the edges and the corners of the surface. Given the simulation model, the perturbation followed specific path from the center of the surface till the edges and the corners, as it will be described later in this report. In order to achieve this simulation, it was required to setup an algorithm that would interpret the surface as a 2 dimensional NxN grid with an element (node) at each position in the grid. Because of the nature of the process, the vertical changes of the surface could be updated in parallel since each node, or a cluster of nodes can be computed using a separate process and then the result could be passed on to the next node or a cluster. The penalization was done using MPI and following a specific update order.

Implementation (Physical model)

As mentioned above, the drum surface is this experiment is viewed as an object broken down in several finite elements (tensile material elements in our case) that interact with each other and behave according to physical law. For our algorithm, we represent the surface using a two-dimensional grid of finite elements. A 4x4 surface representation can be seen in the figure below:

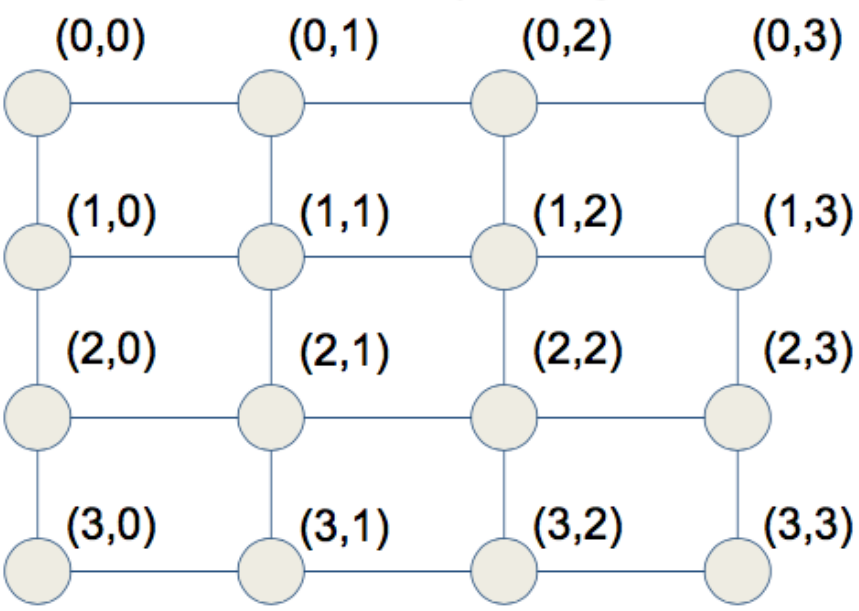

Fig.1 (An example of a 4x4 mesh setup)[1]

As it can be observed, each node is interconnected in the mesh. This is necessary to interchange the state of each node throughout the mesh following a specific sequence and defined set of equations described below.

The grid is responding to the impulse applied to the center. At each time step, or iteration, the following sequence is followed: First, the vertical displacement of the nodes in the interior of the mesh is updated. In the mesh above, specifically nodes (1,1), (1,2), (2,1) and (2,2) are considered to be the interior elements. Following, the edge elements are updated. For example, (1,0) and (2,0) in the mesh above are considered to be a set that represents the left most edge. Finally, the corner elements are updated. Because the perturbation is observed during a period of time and travels along the surface with time, the perturbation propagation (vertical displacement updates of each node) can be observed at several consecutive iterations. Hence, the update iteration at each iteration can be thus followed:

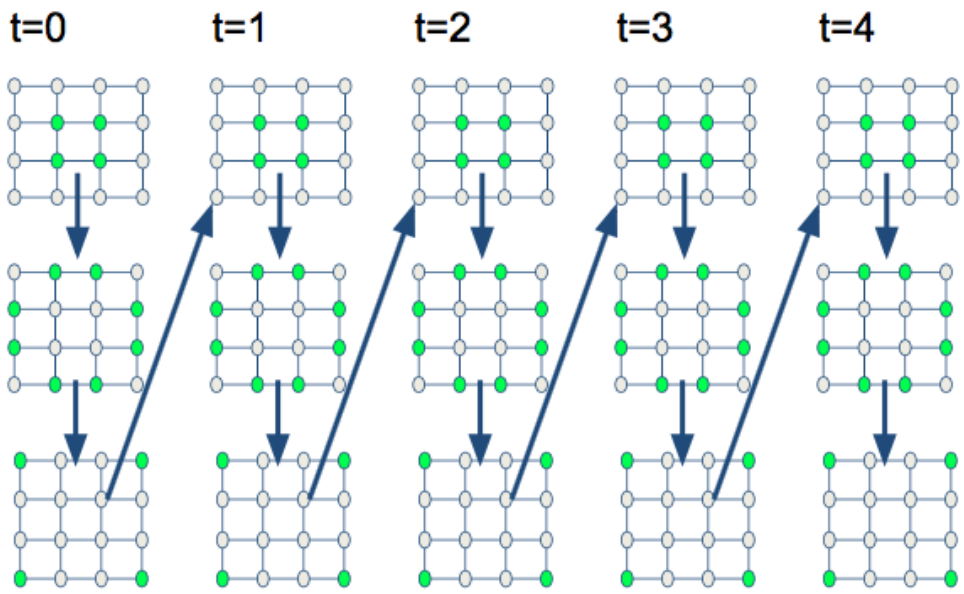

Fig.2 (Elements update sequence at consecutive iterations)[1]

Implementation (Mathematical setup)

At each iteration demonstrated above, the node vertical displacement updates must follow a specific set of equations that represents the displacement updates of each part of the surface. At first, the center elements are updated using the following equation:

$$u(i,j) = \frac{\rho(u1(i-1,j) + u1(i+1,j) + u1(i,j-1) + u1(i,j+1) - 4u1(i,j)) + 2u1(i,j) - (1-\eta)u2(i,j)}{1+\eta}$$

Fig.3 (Center nodes vertical displacement update formula)[1]

The elements that *u(i,j),* the vertical displacement of the node at current time period, depends on are the following:

- *u1(i-1,j),*the displacement of the left neighbor of the node at the previous iteration
- *u1(i+1,j)* is the displacement of the right neighbor of the node at the previous iteration
- *u1(i,j+1)* is the displacement of the upper neighbor of the node at the previous iteration
- *u1(i1,j-1)* is the displacement of the lower neighbor of the node at the previous iteration
- *u2(i,j)* is the displacement of the same node two iterations prior to the current one
- *ρ, η:* physical property of the surface, respectively, 0.5 and 0.0002 as provided

Given that the center elements update depends on their neighbors, they must be able to obtain their neighbors value. The process of sharing the node displacements will be described further in the algorithm section below.

Secondly, the edge elements are updated using the following relation:

$$u(0,i) = Gu(1,i)$$

$$u(N-1,i) = Gu(N-2,i)$$

$$u(i,0) = Gu(i,1)$$

$$u(i,N-1) = Gu(i,N-2)$$

$$1 < i < N-1$$

Fig.4 (Edge nodes vertical displacement update relation)[1]

As it can be observed, nodes described above are situated on the the upper, lower, left and right edges, where i and j indices are limited between 1 and N, the grid size. As well, each of the node in the edge depends only and only on the current displacement of the adjacent node in the center. This is important for node displacement sharing as the only values that edge nodes will need to obtain are the values of their adjacent center neighbors. Is it also to be noted that the edge node updates must happen only and only after the center nodes were updated.

Finally, the corner updates happen accordingly to the following relation:

$$u(0,0) = Gu(1,0)$$

$$u(N-1,0) = Gu(N-2,0)$$

$$u(0,N-1) = Gu(0,N-2)$$

$$u(N-1,N-1) = Gu(N-1,N-2)$$

Fig.5 (Edge nodes vertical displacement update relation)[1]

Again**,** one can observe that the upper corner values, *u(0,i)* and *u(N-1,0)* depend on adjacent element from the upper edge. As well, the left and right lower corners, depend from adjacent elements from left and right edge respectively. Just like the edge updates, the corner vertical displacement updates depend on their neighbors current updated displacement. Thus, the corner displacement updates happen strictly following the edge update event.

Implementation (Algorithm setup)

The algorithm we setup follows the update formulas above at each iteration. First, we setup a two dimensional array representing the grid. Each element on the grid has an array with values representing the three iterations recorded (u2,u1,u) where u2 is the vertical displacement at t-2, u1 is the vertical displacement at t-1 and u is the current value that is updated.

After the data structure is setup, we call the MPI instantiation with specific number of processes. Depending on the grid size, 4 or 512 as required, the grid is then iterated and each part of the grid is assigned to specific process based on how many nodes a process will contain and the process rank. For an instance, if the grid size is 4 (4x4 array) and we call the algorithm with 16 MPI processes, each process is assigned exactly one element from the grid. Otherwise in the case of 512x512 grid, the work is divided depending on how many processes we provide (1,2,4,8,16 or 32). Thus, in case of a 4x4 grid, each rank is assigned 1 node and is responsible to get and send values to all of its neighbors. On the other hand, a larger grid (512x512), will result each rank having a cluster of the nodes, where each node needs to be updated according to the rules described above. For an instance, if we call the algorithm with 512x512 grid with 32

processes, each process will have (512*512/32) = 8192 nodes to service. We also make sure to keep track of first and last (offset) rows in each process depending on the process rank. The work division for each MPI process is illustrated in the diagrams below:

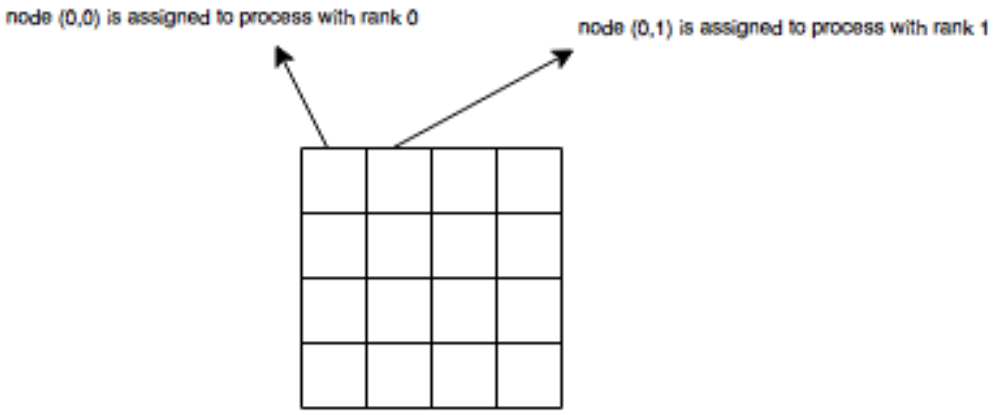

Fig.6 (Case of 4x4 grid work division between 16 processes)

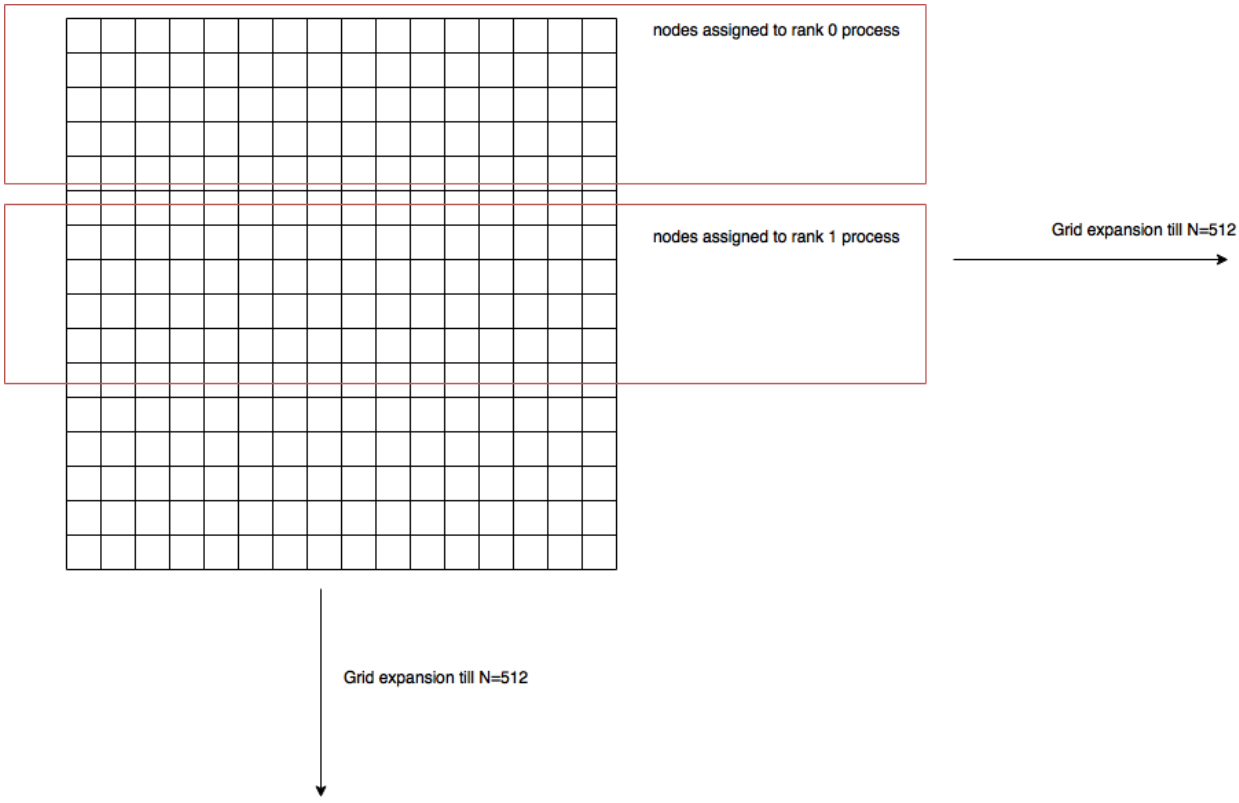

Fig.7 (Case of 512x512 grid work division between n processes)

Furthermore, the domain division for the 512x512 case required us to maintain additional upper and lower ghost rows into which the processes would receive the neighboring values. The number of these rows and their position depended on the rank of the process. For an instance, the rank 0 process could only send its last row (0+offset index) and receive into its lower ghost row (offset + 1). The processes between 0 and n-1 could sent their first and last rows but receive into their first_row -1 index row and last_row + 1 index row. The image below illustrates the necessary ghost rows required to exchange the MPI data:

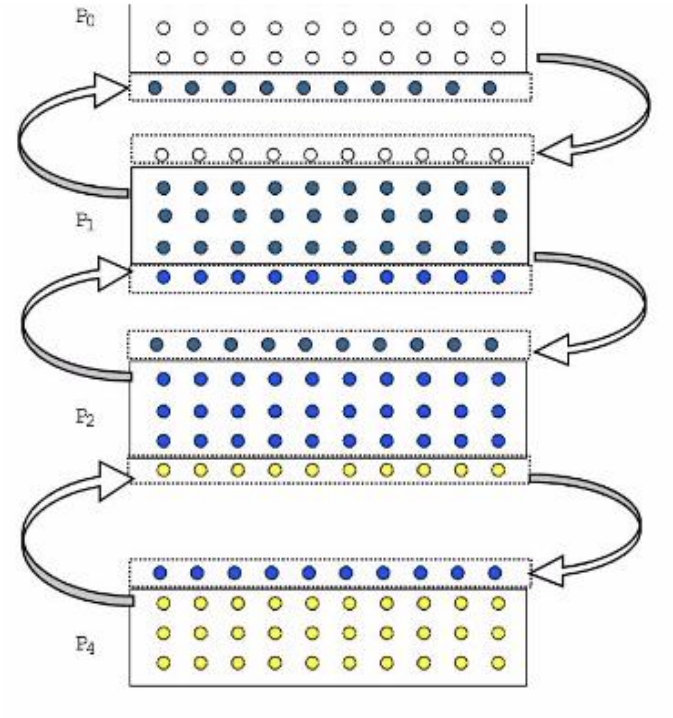

Fig.8 (Case of 512x512 necessary temporary ghost rows for MPI send/recv)[1]

To node, since the grid was 512x512 and the number of processes was 1,2,4,8,16 and 32, the domain was perfectly divisible and all processes were assigned equal amount of work. Once the work is divided for each process, both algorithms, for 4x4 and 512x512 then proceed to a loop that iterates over the specified amount of repetitions. At every iteration, we call functions responsible for node data exchange as well as the update for center, side and corner areas. The order of calls also makes sure to maintain the upgrade strict ordering (center update first, followed by edge update and finalizing with corner updates.

During an iteration, several calls are made to various helper methods that is containing the logic to identify what nodes need to be currently upgraded, which are neighbors of these nodes and what values must be transmitted/received based on one of the three conditions and the given work division setup (either the node is contained within the current process, or the neighbor node is outside of the current process scope, thus data needs to be transmitted/received via MPI send and receive methods). Furthermore, the sending and receiving is also restricted to type of nodes.

Separate helper functions have been also implemented to identify if a given node is an edge, a center or a corner node and at which position this node is situated. This allows us to introduce several optimization in order to determine to which nodes the send must be done uniquely. Several extra data exchanges can be thus avoided. Based on these properties, the exchange of data through MPI would also happen strictly according to the current upgrade parameter (center, edge or corner). For an instance, for a 4x4 case, if the function currently services the center node upgrade, the function first receives u1's from neighboring center and edge nodes. Then, it sends u1's of all center nodes it its neighbors, but not to the edges or corners, since the edges depend on center node's current value and not the previous value. The newly updated value will be sent afterwards, when the upgrade process will happened (after calling the appropriate equation described above in the mathematical setup). In consequence, the edges wait for their turn at next iterating function call. Edges then will receive all the new values only from the adjacent nodes at the center and only the values that they need for the computation. The receive will happen only and only the current center node values have been upgraded. Same happens for the corners. The upper corners depend upper edge adjacent current node values and the lower right and left corners depend on left and right adjacent edge current node values. The nodes wait for values that have been updated by the edges. Hence, the edges first send their u1's to center nodes, then receive value from the adjacent center nodes (after their upgrade). Then, they upgrade their own values and finally send their current u's to corners. The corners, on their side, get the current edge u's and upgrade their value themselves. Again, following the same analogy, the corners get updated only and only after the edge upgrade has been finalized.

Finally, the upgrade process also depends on whether the neighbor nodes are within the current processes scope or outside. For example, for the 4x4 case, if the neighbor process is outside, the upgrade function queries an array filled with neighbor values. (Neighbors are received by the send/receive function described above with their coordinate tag and value as key). In such setup, each node will have an array assigned and filled with neighboring nodes. Each time we receive a value from these nodes, we will update the array and then update current node values using the received values from the array. If the neighbor node that we access is not on the neighbor array, it means that the current process did not receive the value from that neighbor, hence the neighbor is in the current process scope. Therefore, the node can directly access its neighbors value since the grid has been divided in the beginning and each node in the grid has been assigned an array of [u2,u1,u], which updates at each time step.

The whole process for 4x4 grid can be summarized below:

- Setup the grid, place perturbation at (n/2,n/2) i.e: place a 1 in u1 at (n/2,n/2)
- Call MPI, divide work for each process
- For each specified iteration (t1,t2,t3,t4..) do the following:
    - Do center upgrade:

        - For each node that belongs to center, send u1's to neighbors if they are outside of the current process. (Omit the neighbors at the edges).
        - Receive u1's from all the neighbors that are outside the scope, including the edges. Call the update function.
            - In the update function, get all neighbor values first. If neighbor is in current process's grid, get u1 value directly. Else, fetch the received neighbor array and get the value there. Call appropriate upgrade formula. Send the new value to the edges if they are outside of the current scope (otherwise the edges will be able to access the new value directly)
    - Do edge upgrade:
        - For each node that belongs to edge, send u1's to adjacent center neighbors if they are outside of the current process.
        - Receive the NEW value from center nodes and call the update function.
            - In the update function, get all NEW center values first. If the center neighbor is in current process's grid, access u1 value directly. Else, fetch the received neighbor array and get the value there. Call the appropriate upgrade formula based on nodes position. Send the NEW edge values to the appropriate corners.
    - Call corner upgrade:
        - For each node that belongs to corner, receive the NEW value from edges and upgrade current value according to the appropriate formulas.

REPEAT the process for next time iteration.

For 512x512 grid case, the MPI exchanges and upgrade happen slightly differently. As mentioned before, the work is divided, in this case, equally among the processes. Each process, with specific rank, will be receiving and sending rows specifically to the process rank. For an instance, the rank 0 process will need to send its last row only and receive the first row of the rank 1 process. All processes between rank 1 and n-1 where n is the size of the MPI world, will be sending and receiving rows for their first and last rows in the cluster and finally the process with rank n-1 will only send its first and get the last row of the process with rank n-1. The algorithm also calls the upgrade of nodes in specific order (center, edges and then corners). The send and receive MPI function calls, however, need to happen only once during the iteration data exchange must happen only once for the 512x512. This optimization was introduced since it is necessary to exchange data between the processes uniquely for all center processes. The 4x4 case, where every node with its rank is assigned one node, we need to perform more send and receives since now every node has a neighbor. For an instance, the edges will have to exchange the data with all adjacent center nodes since they are all in separate processes. However, thanks to the nature of the domain separation in the 512x512 grid, edges can directly access the newly updated values from the adjacent center nodes. More over, the edge nodes do not need the values

of the neighbor processes since they uniquely depend on adjacent center nodes (left center nodes in the right edge and right center nodes in the left edge. Similar explanation is valid for upper and lower edges. Finally, the corners also do not need any MPI send or receives since they will be always in the same rank as the edge nodes that they depend on. A short example is provided in the diagram below:

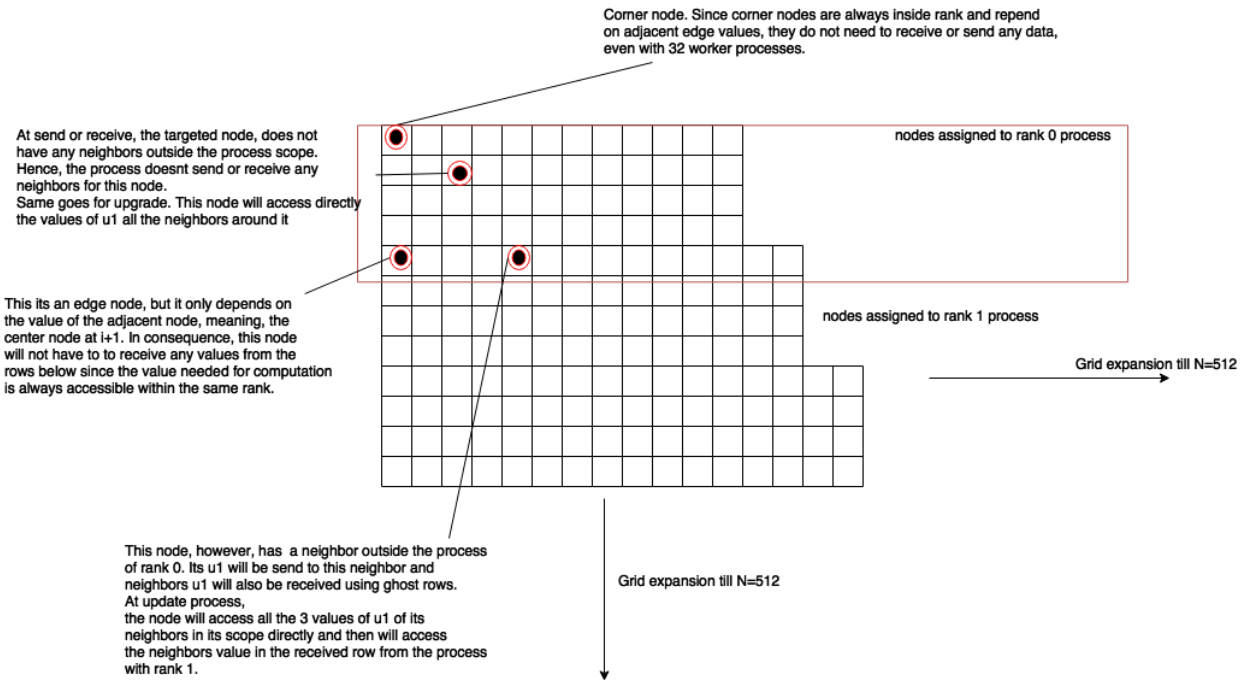

Fig.9 (Case of 512x512 send/receive conditions depending on node position)

Tests

The 4x4 grid algorithm was tested strictly with 16 processes uniquely while 512x512 grid was tested with 1,2,4,8,16 and 32 processes. The 4x4 grid algorithm was found discovered to have roughly similar run times across each of the 16 processes, linearly related to the number of iterations. For an instance, several runs were produced with 1 iteration, 10 iterations and 100 iterations. For 1 iteration case, the average run time for each process was observed to be 1.54 ms. For the case of 10 iterations, it was 11.65 ms and finally for 100 iterations it was 111.5ms. The 512x512 grid however, was tested with 1,2,4,8,16 and 32 MPI processes. The average run time for each process in case of 512x512 grid was also consistent across all processes since they had roughly the amount of work to execute. The following table summarizes the average times for processes (1,2,4,8,16 and 32) at 1,10,100 and 2000 iterations

	1 proc	2 proc	4 proc	8 proc	16 proc	32 proc
1 iter	**11.89ms**	**9.68ms**	**8.75**	**11.61ms**	**10.27**	**13.97ms**
10 iter	**98.8ms**	**52.2ms**	**40.21**	**41.55ms**	**57.08ms**	**129.64ms**
100 iter	**1021.22 ms**	**520.34ms**	**367.82**	**407.14ms**	**677.8ms**	**927.20ms**
2000 iter	**21614.13ms**	**11065.77ms**	**7880.09ms**	**8375.27ms**	**11545.45ms**	**18890.13ms**

Fig.10 (Case of run times for different sizes of MPI groups at different iterations)

The table above represents running time per process. As it can be observed, after the number of processes increased beyond 4, for all iterations, running time per process started to increase. This is to be expected since the tests were conducted on a machine with a quad core CPU (Intel Core i7). This proves the point that adding more processes in our MPI pool will not make the running time faster as every core of the CPU will have to context switch between the groups of MPI processes that try to gain its processing time. Therefore, every process will be allocated the CPU processing resource, as every process will have to share CPU time with other processes that have been assigned the same core.

Conclusion

To conclude, this experiment was conducted in order to better comprehend interposes communication and data sharing through MPI and demonstrate that adding more threads or processes for parallelization does not always result in a large increase of speedup. The 4x4 and 512x512 grid algorithms both utilize MPI and demonstrate domain two different domain decompositions, along with several optimizations to avoid unnecessary data transfers. It was found that due to the architecture of the machine under test, the speedup and runtimes depend on how many cores the machine has and how many processes we allocate to each core. In our case, 4 processes were optimal for every amount of iteration due to the fact that our machine had 4 cores. Once the number of MPI worker processes was increased, the CPU increased its context switching, thus increasing the overall run time and making the overall speedup decline as number of processes increased.

Sources

1. Lugosh, Loren. ECSE420,Finite Elements Notes, PDF, McGill, 2016