A Massively Parallel Explicit Grid-Solver for Non-Regular 2D Domains (April 2019)

Vineet Negi, *Dept. of Mechanical, Aerospace, and Nuclear Engineering, Rensselaer Polytechnic Institute.*

***Abstract*—Many Numerical methods, like Finite Difference, Finite Volume, Lattice Boltzmann Method etc., use a gridded domain. Here, we developed an explicit solver for gridded domains of arbitrary shape. We verified the accuracy of the code by comparing the numerical solution from a heat conduction problem to its analytical solution. We used several parallel performance metrics and demonstrated high parallel scalability of the code.**

***Index Terms*— Grid Solver, Arbitrary Domains, Finite Difference, Parallel Computation.**

I. Introduction

THERE has been an ever growing need to seek solution of large scale problems in science and engineering, like weather simulation, drug discovery research, seismic modeling etc.. To simulate physics of these problems various types of numerical methods are used which relay on some form of discretization of the problem domain. Methods like Finite Difference, Finite Volume, Lattice Boltzmann Method etc. involve gridded domain. The domain can be of any arbitrary shape and can have either uniform structured grid or non-uniform grid, e.g. Multigrid Methods[1].

The solution of these problems may involve large length scales of the domain as well as large time scales. The advent of massively parallel supercomputers has opened a scope for solution of these large problems provided the numerical method can be parallelized.

The numerical solution methods can largely be divided into two classes – Implicit and Explicit. Implicit methods involve solving the grid point solution simultaneously for the whole grid through matrix inversion. Whereas, Explicit methods involve find the present grid point solution for each grid point individually based on the previous solution values of the grid point as well as its neighbors. The advantage of Explicit method is that it does not involve expensive matrix inversion or linear equation solution.

Finite Difference method has been used to solve heart equations in various problem scenarios. Use of central or Backward difference formulation for representing the heat equation results in an implicit scheme which requires linear equation solution [2]. On the other hand, use of Forward difference formulation of derivatives in the heat equation gives an explicit grid point update rule [2]. The grid required for the Finite Difference method has 4 adjacent points – top, bottom, left, and right to every point.

In case of explicit methods in a gridded domain, Domain decomposition is a well-used method for parallelization[3]–[5]. It essentially involves dividing an arbitrary domain into many subdomains and solving each subdomain separately. However, some inter- subdomain communication overhead is incurred to convey the data across the subdomains' boundary. The time incrementation within all subdomains can be same or different. Domain decomposition has also been used for meshed domains in finite element analysis [6].

Lattice Boltzmann Method is a computational fluid dynamics method used to solve Navier-Stokes equation [7]. It works on a grid having 8 neighbors to every grid point-top, bottom, left, right, and diagonal. The update rule is explicit in nature and thus the method can be parallelized using the domain decomposition method [8]–[11].

In this work, we develop a parallel grid solver framework which can accept structured grids of any nodal connectivity, i.e. number of adjacent nodes, and arbitrary shape. This is enabled by storing the gridded domain as a Graph rather than as a multi-dimensional array. The solver utilizes both MPI and Pthreads to obtain optimal scalability. A heat conduction finite difference problem is used to validate the code. Various performance metrics are used to study the scalability, effect of Pthreads, and effect of MPI communicator topology.

II. Theoretical Background

In this work, we use the non-regular 2D domain grid solver to model the heat conduction with and without Advection. Specifically, the grid solver is used to solve Time Explicit Finite Difference (FD) equations.

Heat conduction equation in solids, also known as Fourier-Biot equation, is a parabolic partial differential equation. In 2D, it is as described below.

$$\frac{\partial u}{\partial t} = \frac{k}{\rho c_p}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \frac{q_v}{\rho c_p} \quad (1)$$

where, u denotes the temperature at a point, x and y are the spatial coordinates, t is the temporal coordinate, k is the heat conductivity of the material, q_v is the heat generation per unit volume, ρ is the density of the material, and c_p is the heat capacity of the material. The quantity $k/\rho\, c_p$ is called as the thermal diffusivity of a material.

Advection is the transport of a material by a bulk motion, as opposed to diffusion. Heat conduction in the presence of advection in described by eqn (2).

$$\frac{\partial u}{\partial t} + V_x\frac{\partial u}{\partial x} + V_y\frac{\partial u}{\partial y} = \frac{k}{\rho c_p}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \frac{q_v}{\rho c_p} \quad (2)$$

where, V_x and V_y are the components of the predefined bulk velocity field in the material. Note that setting the bulk velocity to 0 reduces the Advection-Heat Conduction equation, eqn (2), to Fourier Biot equation, eqn (1).

Advection-Heat Conduction equation can be used to study heat conduction in a fluid with laminar flow. Fourier Biot equation is mathematically similar to diffusion equation in which material diffusivity replaces the thermal diffusivity, as in eqn (1). Conversely, eqn (2) can be used to model diffusion in a laminar fluid flow, e.g. diffusion of ink in a slow moving water stream. Another application of the Advection-Heat Conduction equation is the modeling of temperature distribution around a moving heat source, e.g. a welding torch, where a fictitious material velocity, equal and opposite of the heat source velocity, is assumed.

III. Numerical Method

We employ Finite Difference method to solve the Heat conduction with Advection, eqn 2, and without it, eqn 1. The Finite Difference (FD) method is based on discretizing the partial differential equations using difference method.

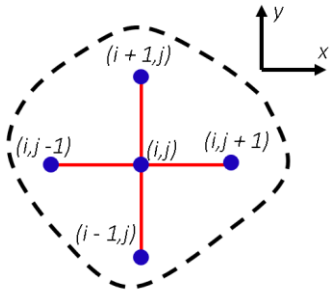


Fig. 1. A schematic showing the grid for the 2D Finite Difference Method

In the present case, the Difference equation corresponding to eqn 2 results in the following update rule –

$$u_{i,j}^{n+1} = \left(1 - \frac{2k\,\Delta t}{\rho c_p \Delta x^2} - \frac{2k\,\Delta t}{\rho c_p \Delta y^2}\right)u_{i,j}^{n} + \frac{k\,\Delta t}{\rho c_p \Delta x^2}\left(u_{i,j+1}^{n} + u_{i,j-1}^{n}\right) + \frac{k\,\Delta t}{\rho c_p \Delta y^2}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n}\right) - \frac{V_x\,\Delta t}{\Delta x}\left(u_{i,j+1}^{n} - u_{i,j}^{n}\right) - \frac{V_y\,\Delta t}{\Delta y}\left(u_{i+1,j}^{n} - u_{i,j}^{n}\right) + \left(\frac{(q_v)_{i,j}^{n}}{\rho c_p}\right) \quad (3)$$

where, i, j are the indices of the grid as shown in fig 1, n is the iteration number. The FD form of eqn 1 can be obtained from eqn 2 by setting velocities as zero.

Eqn 3 provides us the required grid connectivity number, 4, and the update rule. From eqn 3 it can be seen at each node must be able to store previous as well as present solution value. To this end, odd iteration solution values and even iteration solution values are stores separately and one becomes the present and the other previous depending on the evenness of the iteration number.

IV. Code Description

A. Constants and Macros

In this work, *int_t* defines any integer datatype and *real_t* defines any real datatype. *int_t* is taken to be of primitive type int (4 byte) and *real_t* is taken to be of primitive type double (8 byte).

NODAL_Z called as nodal coordination or connectivity number and it defines the number of adjacent nodes to any node in the grid. For the FD method's grid, the connectivity is 4.

NUM_THREADS is the number of Pthreads for each MPI process.

B. Datatypes

There are four important datatypes in this work – *node*, *process*, *physics*, and *solution*.

node – This datatype defines a vertex or a node in the grid along with all its attributes. Each node has following attributes: *id* storing the global index of the nodes, *nodetype* to store whether a the node is fixed (boundary node) or free, *loc*[2] to stores the x and y coordinates of the node, *disp*[2] to stores the scalar displacement for even, at 0 index, and odd, at 1 index, iterations, *forces*[2] to store the nodal force (heat generation) for even and odd index iterations, *field*[4] to stores a 2D vector field (e.g. Velocity field) for even and odd iterations, *iter* to stores the current iteration number, *rank_status* to store nature of the node with respect to the parallel setup, and a list of adjacent nodes' memory locations in **neighbors*[*NODAL_Z*].

```
struct node
{
   int_t id;
   int_t iter;
   int_t nodetype;
   real_t loc[2];
   real_t disp[2];
   real_t forces[2];
   real_t field[4];
   int_t rank_status;
   struct node *neighbors[NODAL_Z];
};
```

Note that with respect to parallelization, any node on a process can be either owned by the process or is a remote copy of a node on some other process. Nodes which are owned by a process and their remote copies exist on other processes are called as ghosted nodes. Whereas, the remote copies of a ghosted node are themselves called as ghost nodes. So, each process has ghosted nodes, ghost nodes, and nodes which are not shared.

process – This datatype defines an adjacent process to any process. It holds all adjacent process communication related information. Specifically, it holds the number of the ghosted nodes (num_send_nodes) and ghost nodes (num_recv_nodes) with an adjacent process. Further, it holds the send (*send_buf) and receive buffers (*recv_buf) for the communication. Finally, it stores the pointers to the locations of the ghost nodes (**recv_locs) and ghosted nodes (**send_locs). The declaration is as shown below:

```
struct process
{
   int_t rank;
   int_t num_recv_nodes;
   int_t num_send_nodes;
   real_t *recv_buf;
   real_t *send_buf;
   struct node **recv_locs;
   struct node **send_locs;
};
```

Note that for the case of FD method, the communication between adjacent processes is symmetrical and therefore the number of send nodes is same as the number of received nodes. Also, the ghost nodes referenced by **recv_locs array are sorted in the ascending order of their global node id. Same is also true for the ghosted node ids referenced by **send_locs array. This is essential to make sure that the order of ghosted nodes in the **send_locs of the sending rank would be same as the order of the ghost nodes in **recv_locs of the receiving rank. This enables direct transfer of MPI exchange data from the buffers to their actual locations without any need for rearrangement.

Besides the above two datatypes, we also define *physics* and *solution* as datatypes to stores all relevant information about the corresponding aspects.

C. Data Structures

Here we describe important data structures used in this code. *nodes* is an array of datatype *node* with size equal of *num_nodes*, which is the number of nodes owned by a process. It stores the nodes owned by a process in the ascending order of their global ids. *ghost_nodes* is an array of datatype *node* with size equal to *num_ghost_nodes*. This array stores all the ghost nodes at a process in increasing order of their global node ids. Note that each node in the nodes array stores the pointers to the adjacent nodes which may lie either in *nodes* array or *ghost_nodes* array.

Another important data structure is the array of adjacent processes for any process, called as *adj_processes*. The array of size *num_adj_processes* is of the datatype *process*. *phy*, an instance of datatype *physics*, stores the physics info and *sol*, an instance of datatype *solution*, stores all the solution related information like Δt, Δx, Δy etc.

Besides, these important datatypes there are other data structures for input processing, Pthreads definition etc.

D. Input Data & Preprocessing

The input data for this code can be divided into two stages – raw input data and partitioned input data. The raw input data specifies the problem domain, i.e. the grid, and its attributes like type of the node (fixed or free), location of node, initial value of the scalar solution variable's at a node, value of the nodal forces (heat generation in this case), and the value of the vector nodal fields (velocity field in the case). This information is organized into three ASCII files – *xadj*, *adjncy*, and *nodal_data*.

The raw input data considers the grid overlaying an 2D problem domain of arbitrary shape such that the nodes are numbered sequentially from the bottom left node (index 0) to top right (index number of nodes – 1) with left-right being defined along the x direction and top-bottom along the y direction and the direction of traversal is left-right.

The format of raw input files is a header line specifying the number of lines of data to be read followed by the data lines. *xadj* and *adjncy* specify the grid as graph in the compressed storage format (CSR), as described in []. Briefly, *xadj* is an array of length number of nodes + 1 where *xadj[i]*, *xadj[i+1]* denotes the starting index and end index (not including the end index) on the *adjncy* array for each global node id *i*. The *adjncy* array stores the global node ids of neighbors. So, the neighboring node ids of the *i-th* node would be the node ids lying between *xadj[i]* and *xadj[i+1]* (not including *xadj[i+1]*) indices in *adjncy*. All the nodal data corresponding to the *ith* node id can be obtained from the *i-th* row in the *nodal_data* array. Note that raw input data does not specify any partitioning and it needs to be preprocessed to create partitioned input data to be fed into the grid-solver code. This is done using METIS_PartGraphKway function from the METIS graph partitioning code []. The partitioned grid is stores into 5 binary files – *g_parts*, *g_xadj*, *g_adjncy*, *g_global_vert_ids*, and *g_nodal_data*.

g_parts is an array of size equal to the number of processes or partitions + 1. For *i-th* partition, the global node ids lying between indices *g_parts[i]* and *g_parts[i+1]* (not including *g_parts[i+1]*) in the *g_global_vert_ids* array are members of the partition subdomain. Their nodal data can be

obtained likewise from *g_nodal_data* array. The values between the indices *g_parts[i]* and *g_parts[i+1]* (not including *g_parts[i+1]*) in the *g_xadj* array points to the *g_adjncy* array as described for the raw input data before. The partitioned grid inputs are directly used as inputs for the grid-solver. Care must be taken to make sure that their binary format's endianness is compatible with the machine on which the grid-solver is to be run. The code for conversion of raw input data to partitioned input data, *mesh_partition.c*, has options to set the endianness of the binary partitioned input data.

E. Algorithms and Functions

1) Grid setup at a process

The partitioned grid input data is read at rank 0 using *read_data()* function to setup *g_parts*, *g_xadj*, *g_adjncy*, *g_global_vert_ids*, and *g_nodal_data* arrays. Rank 0 process then determines the data to be send to each process, as described previously in section IV (D), and sends it using *send_data()* function, which uses non-blocking *MPI_Isend()*.

Ranks other than 0, call *receive_data()* function, utilizing *MPI_Irecv()*, to receive the grid data corresponding to their grid subdomain. The end result of this communication is the setting up of *l_xadj*, *l_adjncy*, *l_global_vert_ids*, and *l_nodal_data* arrays (where the prefix *l* implies the local subdomain of a process). Note that there is no *l_parts* array as any partition has no sub-partitions. *l_xadj* points to *l_adjncy*, *l_global_vert_ids*, *l_nodal_data* arrays as described for *g_xadj* array. Besides these *l_** arrays, a global node id to partition or rank id map, called as *g_n2r_map*, in the form of an array is also distributed to all ranks from rank 0. In this array, the id of the partition or rank of *i-th* node is *g_n2r_map[i]*. Another data passed on to each MPI process from rank 0 is the total number of nodes present in all the MPI processes with ids less than the process being informed. This is termed as *node_offset* and it is useful for MPI_IO operation as will be discussed in section IV (F).

At the end of data distribution, each process or rank processes its local subdomain grid data and global node id to rank id map. Each rank then calls *grid_initialize()* and *init_ghost_nodes()* function to create the *node*s, *ghost_nodes*, *adj_processes*, *phy*, and *sol* data structures as described in section IV(C). Post the creation of the desired data structures, the initial raw data arrays, *l_** and *g_n2r_map*, are deleted from the memory as they are no longer required. The main solution algorithm only requires *node*s, *ghost_nodes*, *adj_processes*, *phy*, and *sol* data structures.

2) Inter-process MPI data exchange

Inter-process data exchange during the solution takes place through four functions – *request_ghost_nodes()*, *update_ghost_nodes()*, *send_ghosted_nodes()*, and *copy_ghosted_nodes()*.

request_ghost_nodes() goes over all the adjacent processes in *adj_processes* array are posts non-blocking *MPI_Irecv()* to their respective *recv_buf* arrays.

```
int_t i;
for (i = 0; i < num_adj_processes; i++)
{
     MPI_Irecv (adj_processes[i].recv_buf,
adj_processes[i].num_recv_nodes,
     mpi_real_t, adj_processes[i].rank, 0, comm,
&recv_requests[i]);
 }
```

update_ghost_nodes() function copies the received data (displacements) from the recv_buf for each adjacent process in *adj_processes* to their ghost nodes locations in ***recv_locs* pointer array once the communication is finished.

```
   int_t i, j, idx;
   idx = it%2; //determine if the current iteration is odd or
even

   for (i = 0; i <  num_adj_processes; i++)
   {
       MPI_Wait(&recv_requests[i],
MPI_STATUS_IGNORE); // wait for the Irecv to finish

        for (j = 0; j < adj_processes[i].num_recv_nodes; j++)
        {
            adj_processes[i].recv_locs[j]->iter = it; //update
the iteration counter

            adj_processes[i].recv_locs[j]->disp[idx] =
adj_processes[i].recv_buf[j]; // copy from recv buffer
        }
    }
```

send_ghosted_nodes() post the non-blocking send requests involving send_buf in a manner similar to *request_ghost_nodes()* function.

Similarly, *copy_ghosted_nodes()* copies the displacement data from the ghosted node locs in the ***send_locs* pointer array to *send_buf*.

3) Solution algorithm

Solution step involves going through the *nodes* array and calculating and updating the solution variable, also called as displacement, for the next iteration by calling the *solve()* function. We use Pthreads to perform this solve step in parallel. The number of Pthreads created as specified by

NUM_THREADS constant. Considering that the main thread (zero thread) invoking other threads also performs a portion of the update calculations, we create only NUM_THREADS – 1 extra Pthreads. These Pthreads call *solve_at_thread()* function from within the *solve()* function executed at thread 0.

The number of update iterations to be performed is stored in *sol* data structure. In each iteration, every rank or process posts receive requests for the solution values of the ghost nodes for the present iteration through *request_ghost_nodes()* function. Following which every Pthread in an MPI process starts performing the update of nodes allocated to it. The start and end of the update calculations at thread level are synchronized through *pthread_barrier_wait()* function. When all the nodes at all threads are updated for any MPI process, the process sends the current values of the ghosted nodes by executing *copy_ghosted_nodes()* function followed by *send_ghosted_nodes()* and *update_ghost_nodes()* in this order. *MPI_Waitall()* is performed on send requests from *send_ghosted_nodes()* to block the next iteration until ghosted nodes' data has been successfully send. Fig 2 shows the schematic of the solution algorithm described above.

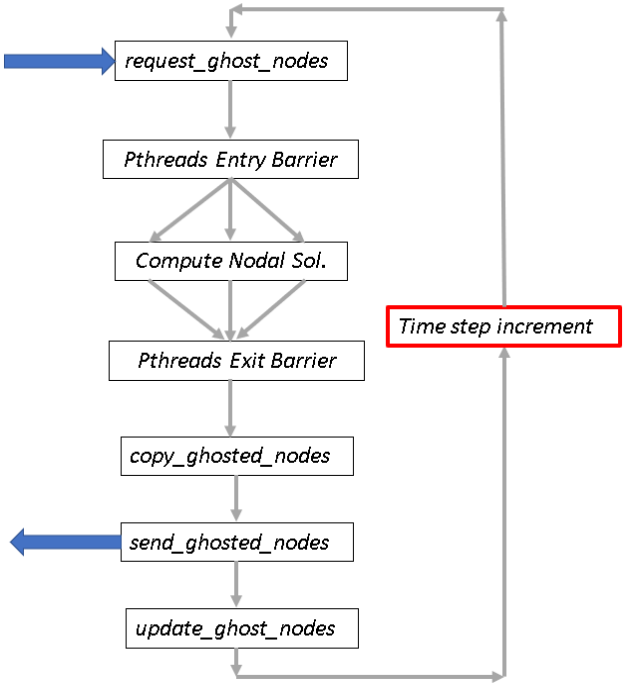


Fig. 2. A Schematic showing the Parallel Grid Solver algorithm.

F. Output Data & Postprocessing

The output of the solution step, i.e. the nodal solution or displacement value at the last iteration, are written to a binary results file using MPI_IO. Each MPI process uses *MPI_File_write_at()* function with offset calculated as *node_offset*sizeof(*real_t*). Thus, the order of the nodal output written to the results file is dependent on the partitioning and corresponds to node ids' order in *g_global_vert_ids* array. Therefore, *g_global_vert_ids* array is required to properly read the binary results file. The plotting of the results is done using a MATLAB script (*view_res.m*) which needs the binary output and the binary *g_global_vert_ids* array as inputs.

V. Problem Specification

The main features of the solver developed in this work are:

1) Works on domains of arbitrary shape
2) Good scaling for high processor counts
3) Flexibility to be modified to accept any type of grid and physics

To demonstrate feature (1), we chose an annular domain, as shown in fig 3. The inner and outer radii of this domain are 0.25 m and 0.5 m respectively. The physics is of heat conduction (without advection) and material properties are arbitrarily chosen as k = 1 ($W \cdot m^{-1} \cdot K^{-1}$), ρ = 1 (Kg/m^3), and c_p = 1 ($J/(K \cdot Kg)$). The initial temperature is 300 K throughout the body. The inner boundary is that 1300 K and outer boundary is at 300 K. A grid of mesh size $\Delta x = \Delta y$ = 0.0005 m is taken (2356220 grid points) and the evolution is performed for 10^6 time increments of $\Delta t = 5 \times 10^{-8}$ secs each. It is expected that this much time would be sufficient for the body to achieve its steady state and hence comparison can be made with analytical steady state solution for this problem.

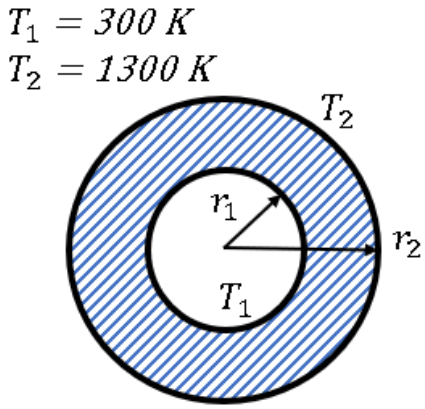


Fig. 3. A schematic of problem 1. The domain is annular with fixed temperature inside and outside. The problem is of heat conduction.

To test the advection physics, we create another problem involving a square domain of size 1 m x 1 m. The material properties of the domain are same as problem 1. Further, the grid size and the time increment size is also same as problem 1. However, the number of iterations is 10^5. Moreover, this problem involves a predefined fixed velocity field of V_x = -100 m/s and V_y = 0 m/s. All the boundaries are fixed at 300 K and the initial temperature of the domain is also 300 K. A circular heat source of diameter 0.1 m and heat generation rate of 10^6 W/m^2 is present at the center of the square domain. This problem involves over 4 million nodes and hence it is large enough for performance studies with large processor counts. The schematic of the problem is shown in fig 4.

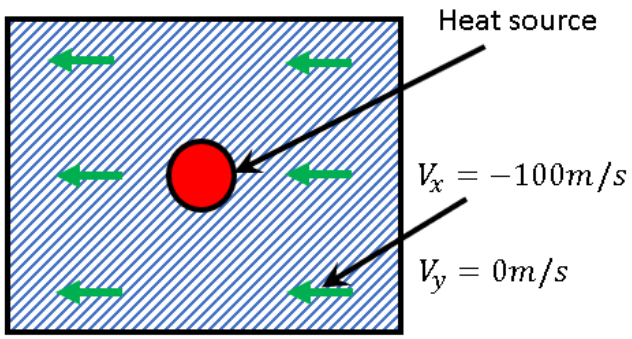


Fig. 4. A schematic of problem 2. The domain is square with fixed outside boundary temperature at 300 K. The problem is of heat conduction with advection.

VI. Results

A. Verification

Problem 1 (annular domain) has an easily accessible analytical solution which can be used to verify the code. The analytical solution for the temperature distribution in an annular domain with temperature T_1 and T_2 at r_1 and r_2 where $r_1 < r_2$ and $T_1 > T_2$ is given by eqn 3.

$$T(r) = T_1 - \frac{T_1 - T_2}{\log\left(\frac{r_2}{r_1}\right)} \cdot \log\left(\frac{r}{r_1}\right) \qquad (3)$$

Fig. 5(a) shows a domain of problem 1 partitioned into 512 regions of nearly equal size (determined by METIS). The GS code presented here is parallel deterministic and the solution is indifferent to parallelization. Fig. 5(b) shows the numerical solution obtained by GS code run on 32 BG/Q nodes (512 processes). The solution took about 820 sec to finish. Fig. 5(c) shows the radial variation of temperature in the solution. It can be seen that the result is in close agreement with the analytical solution, thereby validating the accuracy of the code for heat conduction problems.

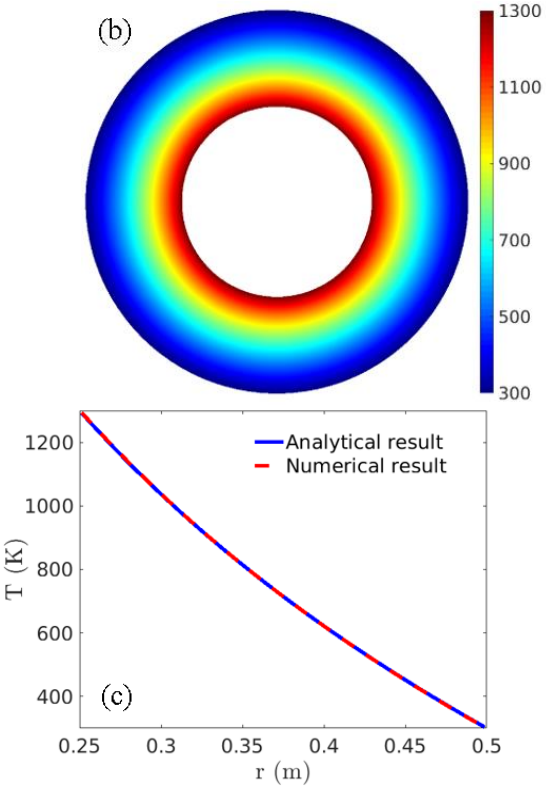
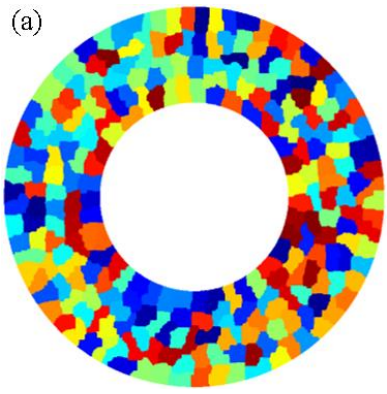
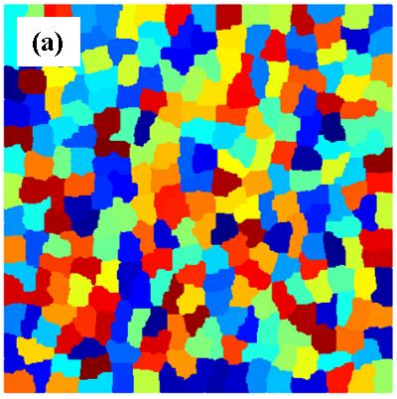


Fig. 5. (a) Annular domain of problem 1 partitioned into 512 regions. (b) The numerical solution of problem 1 obtained using the GS code. The color-bar shows the temperature in kelvin. (c) A plot showing a good agreement between the numerical and analytical result for radial temperature distribution (T vs r).

The solution for problem 2 demonstrating the capability of the code to handle advection problems is shown in fig 6. Fig. 6(a) shows the partitioned domain while 6(b) shows the solution on this domain. We used various levels of partitions 16, 32, 64, … 1024 are found no difference in the solution, thus validating the indifference of the solution to the parallelism. The result, as shown in fig 6(b) is difficult to verify quantitatively due to lack of an analytical solution. However, visual inspection doe matches the qualitative expectation of the solution.





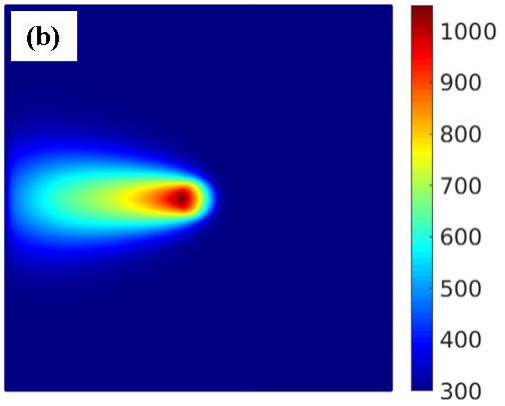


Fig. 6. (a) Square domain of problem 2 partitioned into 512 regions. (b) The numerical solution of heat-conduction with advection problem 2 obtained using the GS code. The color-bar shows the temperature in kelvin.

B. *Performance Analysis*

For performance analysis of the code, we choose problem 2 but with only 1000 iterations to make the solution time and resources required fit within the usage restrictions (for this course - max time is 30 mins and max nodes is 128) on the Blue Gene Q supercomputer for all run configuration, i.e. all values of ranks and threads. The GS code was run on 1, 2, 4, 8, 16, 32, 64, and 128 BG/Q nodes with 1, 2, and 4 threads per MPI process. The runtime of the parallel code is measured from rank 0 using *GetTimeBase()* before and after the *solve()* function. Furthermore, the runtime for only the computation of update of the grid solution, i.e. eliminating the communication overhead, is measured by placing *GetTimeBase()* within the *solve()* function just after the entry barrier and the exit barrier of the Pthreads, and then summing the net computation time from all ranks using *MPI_Reduce()*. This essentially measures the time elapsed between the starting of all Pthreads (synchronized) and ending of the last Pthread.

Thus, we get two measurements of time – T_p is the total parallel execution time and T_{comp} is the cumulative computation (only) time for all ranks. The cumulative communication overhead time, T_o can be calculated as $T_o = r \times t \times T_p - t \times T_{comp}$, where r is the total number of MPI processes and t is the number of threads per process. Note that T_{comp} is the cumulative computation time for all ranks, so the mean execution time per rank would be T_{comp}/r. Furthermore, the mean computation time per rank would decrease with increasing threads per rank, t. Therefore, the total computation (only) time is $t \times T_{comp}$. T_s is the serial execution time measured as the total parallel execution time of the code for 1 rank and 1 thread run configuration, it is found to be $T_s = 628.44$ seconds for this problem.

Fig 7 shows the parallel execution time T_p as a function of total parallel computational resources, $r \times t$, for various t cases. It can be seen that execution time reduces substantially with increase in parallel computational resources suggesting a good scaling behavior of the code. Further, no significant change is observed for different threads per rank values t.

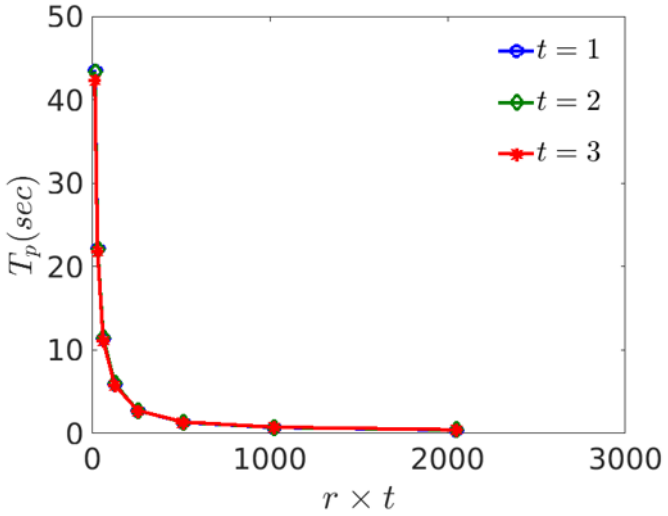


Fig. 7. Variation of parallel execution time T_p with computational resources, i.e. MPI processes (r) x threads per process (t), for various t cases.

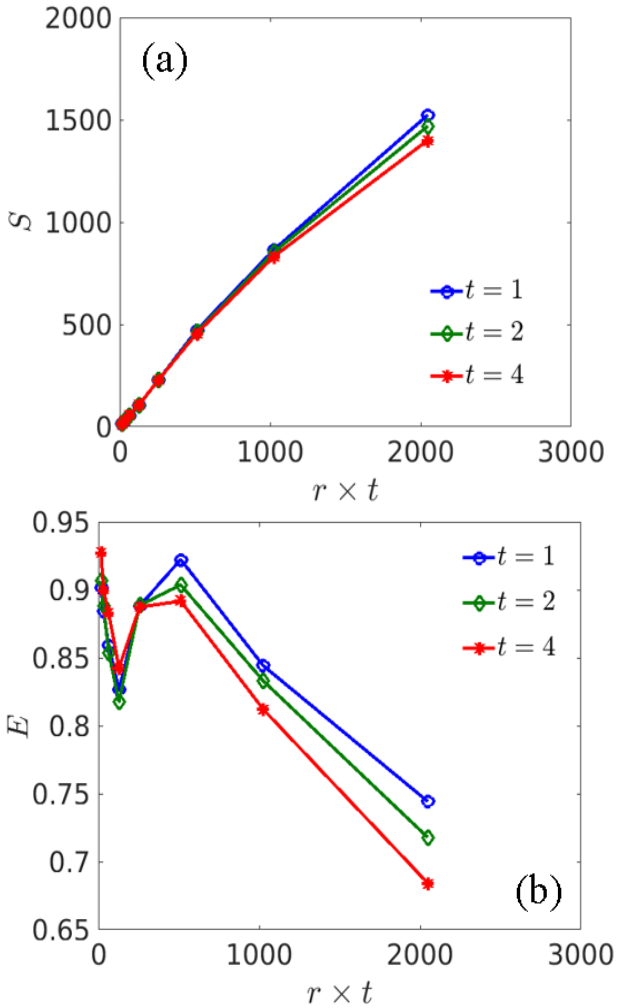


Fig. 8. (a) Variation of speedup S with computational resources, i.e. MPI processes (r) x threads per process (t), for various t cases. (b) Variation of parallel efficiency E with computational resources, i.e. MPI processes (r) x threads per process (t), for various t cases.

The speedup, S, measured as T_s/T_p is shown in fig 8(a). It can be observed that speedup is slightly superlinear from 64 to 512 processors (r x t) after which it becomes sublinear. There is not much variation in the speedup due to number of threads per rank, t. The parallel efficiency, E, measured as $S/(r$ x $t)$ is shown in fig 8(b). Interestingly, the plot shows an unexpected increase in the parallel efficiency from 64 to 512 processors (r x t). This clearly supports the super linearity observed in the speedup plots. The unusual behavior is further investigated by cumulative computation time measurements T_{comp} as well as the overhead time T_o.

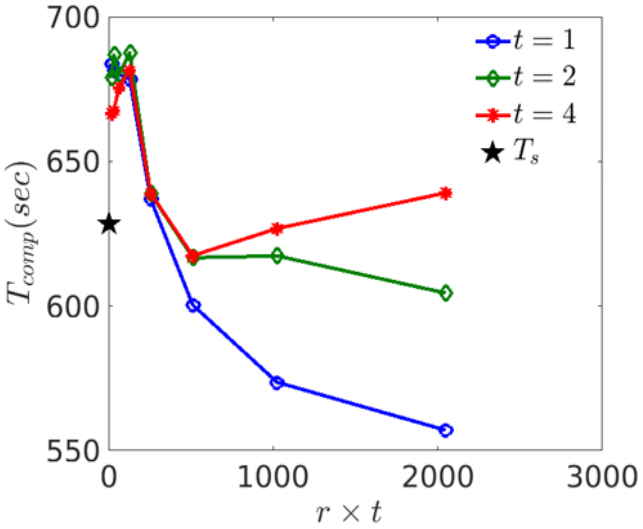


Fig. 9. Variation of cumulative computation time T_{comp} with computational resources, i.e. MPI processes (r) x threads per process (t), for various t cases.

Fig 9 shows the cumulative computation time T_{comp} for all MPI processes as well as for the serial case where T_{comp} = 628.43 seconds ~ T_s. T_{comp} increases as we move from serial to parallel execution. A possible reason for this increase could be the fact that in parallel execution the grid point solution computation for ghosted nodes would necessarily require access to ghost nodes which are defined in a separate array. Thus, the first reference to ghost nodes (*ghost_nodes* array) during the grid point update computation for any ghosted node would result in a cache miss as the currently loaded memory would correspond to *nodes* array. This would certainly increase the computation time as cache misses increase due to parallelization. However, since the ratio of elements in the *nodes* array (within a process sub-domain) to the *ghosts_nodes* array (sub-domain boundary) would be constant for a process, dependent only on the shape of sub-domain rather than its size, for increasing number of MPI processes. The implies that the cache miss penalty effect on T_{comp} is asymptotically constant and it would be minimum for circular sub-domains or sub-domains of even aspect ratio. Similarly, decrease in the sub-domain size would reduce the cache misses by fitting more sub-domain in the cache. The reduction in cache misses is expected to cause superlinear scaling by reducing the computation time, T_{comp}, as clearly observed in fig 9.

The cumulative communication overhead time, T_o, is an important performance measure. The absolute inter-process communication time is T_o/r. This time should be mostly constant as the number of neighbors for any process asymptotically remain same. Therefore, it is expected that T_o is proportional to r. This behavior is clearly observed in fig 10 thus validating the analysis.

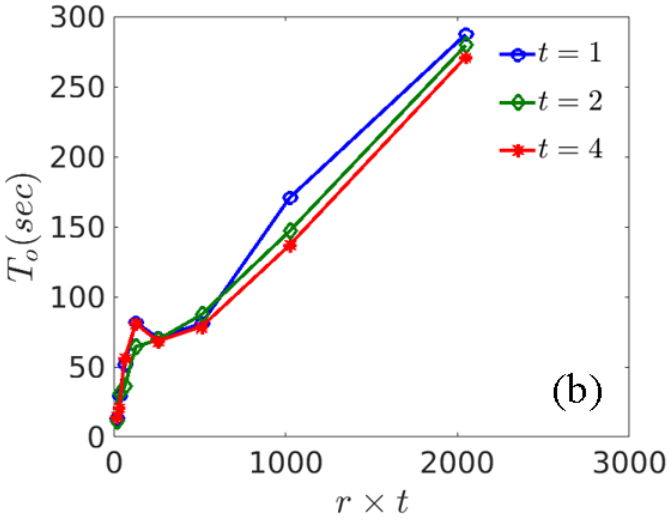


Fig. 10. Variation of overhead time T_o with computational resources, i.e. MPI processes (r) x threads per process (t), for various t cases.

To understand the effect of mapping of the MPI processes to physical processors, we used *MPI_Dist_graph_create_adjacent()* function to create the graph topology based on the partition adjacencies obtained from *adj_processes* data structure. The simulations were run for only 1 thread per rank case and compared with the results using the default MPI_COMM_WORLD. Fig 11 showing the parallel execution time T_p for the two cases shows that for the Blue Gene Q supercomputer the effect is negligible for the present code application.

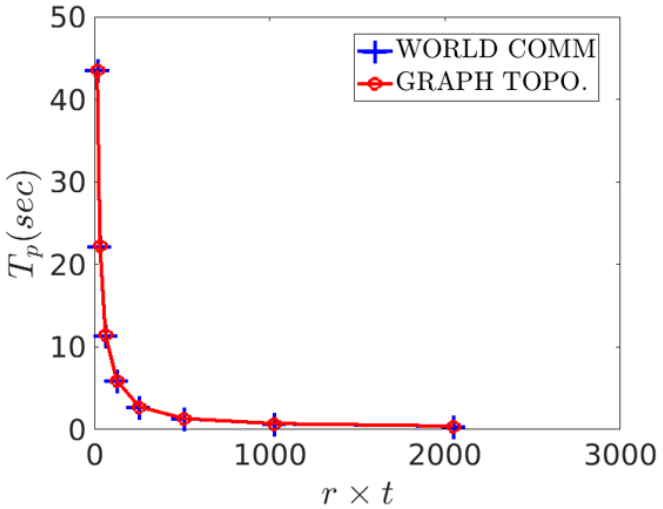


Fig. 11. Effect of MPI communicator's graph topology on parallel execution time T_p, for t = 1 case.

We also studied the effect of r on the MPI_IO time as shown in fig 12. It can be seen that the IO time decreases

slightly as the number of MPI processes are increased from 1. However, at large r values the IO time increases sharply due to possible IO layer network congestion.

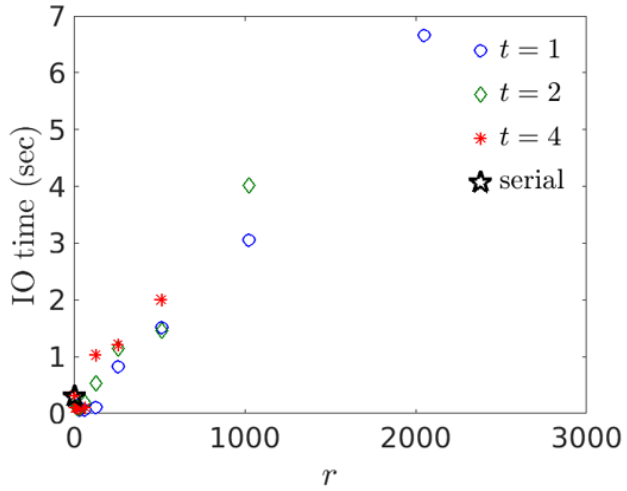


Fig. 12. Variation of MPI IO time T_o with number of MPI processes (r).

VII. Conclusion & Future Work

In this work, we developed a framework for solving numerical problems involving grid domains of arbitrary shapes, e.g. Finite Difference, Finite Volume, Lattice Boltzmann Method etc. We used this framework to solve heat conduction (with and without Advection) problem on an annular and a square domain. We demonstrated the validity of the code by showing that the numerical results are in good agreement with the analytical results for the problems considered.

The performance analysis of the code revealed a good scaling behavior even for large processor counts, i.e. 2048 processors. The parallel efficiency decreased slowly with the number of processors reaching ~0.75 at 2048 processors. Detailed investigation into actual computation time revealed the problem of cache misses during computation due to grid nodes residing on two separate data structures, *nodes* array and *ghost_nodes* array, during parallel execution. Superlinear speedup due to increase in the cumulative cache availability with increase in processor count was also observed up to 512 processors. It was found that threading and MPI communicator topology had negligible influence on the runtimes. The MPI_IO time was found to decrease slightly up till 32 processors after which it increases sharply. Overall, all performance metrics considered pointed to a good parallel scalability of the code.

In future, cache optimizations by modifying the data structures, e.g. combining the *nodes* array and *ghost_nodes* array, and the grid update rules to reduce the penalty for parallelization can be studied. In terms of application, Lattice Boltzmann Method which employs a grid of nodal connectivity 8 can be implemented. The effect of the grid nodal connectivity on the code performance and consequent data structure optimizations has not been studied here but could be a good area for future work on this code.

References

[1] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*. 2011.

[2] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2011.

[3] C. N. Dawson, Q. Du, and T. F. Dupont, "FOR NUMERICAL SOLUTION OF THE HEAT EQUATION WM - MzM," vol. 57, no. 195, pp. 63–71, 1991.

[4] T. Bohlen, "Parallel 3-D viscoelastic finite difference seismic modelling," *Comput. Geosci.*, vol. 28, no. 8, pp. 887–899, 2002.

[5] R. Tavakoli and P. Davami, "2D parallel and stable group explicit finite difference method for solution of diffusion equation," *Appl. Math. Comput.*, vol. 188, no. 2, pp. 1184–1192, 2007.

[6] P. Krysl and Z. Bittnar, "Parallel explicit finite element solid dynamics with domain decomposition and message passing: Dual partitioning scalability," *Comput. Struct.*, vol. 79, no. 3, pp. 345–360, 2001.

[7] S. Succi, M. Sbragaglia, and S. Ubertini, "Lattice Boltzmann Method," *Scholarpedia*, vol. 5, no. 5, p. 9507, 2010.

[8] D. Kandhai, A. Koponen, A. G. Hoekstra, M. Kataja, J. Timonen, and P. M. A. Sloot, "Lattice-Boltzmann hydrodynamics on parallel systems," *Comput. Phys. Commun.*, vol. 111, no. 1–3, pp. 14–26, 2002.

[9] C. Methods, *Large Scale Lattice-Boltzmann Simulations door*, no. december. 1999.

[10] J. C. Desplat, I. Pagonabarraga, and P. Bladon, "Ludwig: A parallel Lattice-Boltzmann code for complex fluids," *Comput. Phys. Commun.*, vol. 134, no. 3, pp. 273–290, 2001.

[11] E. Urnberg, T. Pohl, N. Th, F. Deserno, R. Ulrich, P. Lammers, G. Wellein, and T. Zeiser, "FRIEDRICH-ALEXANDER-UNIVERSIT AT Performance Evaluation of Parallel Large-Scale Lattice Boltzmann

Applications on Three Supercomputing Architectures
Performance Evaluation of Parallel Large-Scale
Lattice Boltzmann Applications on Three
Supercomputing Arch," 2004.