Performance Assessment of Parallel Algorithms on Partial Differential Equation Solver with IBM Blue Gene Q

Bowen Gong [a], Sylvia Hua [b], Zhepeng Luo [b], Siwen Zhang [c]

[a] Department of Mechanical Engineering
[b] Department of Computer Science
[c] Department of Chemical and Biological Engineering
Rensselaer Polytechnic Institute

***Abstract*--Partial differential equation (PDE) is a type of differential equation that is well known and has been widely studied due to their vast existence and broad application in mathematical, engineering, finance fields. PDEs contain multivariable functions associated with their partial derivatives, which describe and govern the behavior of laws of nature. In the macro scale, PDE defines the wave transmission of oceans, the formation of clouds, the windage on planes, etc. In the micro scale, PDE controls the heat generated in a smartphone, the quantum mechanics of an atom, the flow rate of the blood, etc. For some PDE systems, an analytical solution is available. However, for the majority of the PDE systems, analytical solutions are not feasible. Numerical solutions become vital at this moment, which provides a second-best approach to find the state of a system at a given time and conditions. Traditionally, the numerical solutions of a PDE system involves serial algorithms that iterates in time and computes value of a particular position, with a complexity of $\Theta(N_t)$, comparing to the complexity of a serial algorithm with a complexity of $\Theta(N_s \cdot N_t)$, where N_s is the size of the grid, while N_t is the iteration numbers [1]. With the development of the supercomputers, it becomes possible and applicable to obtain the numerical solutions of a PDE with a large grid size and a large number of time steps within a reasonable time. In this paper, we present two algorithms that we implemented for solving a specific family of PDE, i.e., initial boundary-value problem (IBVP). A series of parallel techniques were applied to the implementation, including CUDA, MPI, and parallel I/O. The performance of these two algorithms were systematically assessed and compared.**

Index Terms--CUDA, IBVP problem, MPI, Parallel Programming, PDE solver

I. INTRODUCTION

Most of the physical phenomena in the domain of motion dynamics, thermodynamics, fluid dynamics, electromagnetics and solid vibrations can be modeled by partial differential equations (PDEs) [2-4], which are equations that contain multivariable unknown functions and related partial derivatives. The partial derivatives in the PDEs represent rich natural information like velocity, acceleration, force, current and flux in either time or space domain. The order of a PDE is the order of the highest partial derivative in the equation. All linear second-order PDEs can be categorized into three basic classes: hyperbolic, parabolic, and elliptic equations [2, 5], describing vibration systems and wave motions, heat flow and diffusion processes, and steady-state phenomena, respectively.

To solve a PDE, the most important method is to reduce the PDE into an ordinary differential equation (ODE) that contains only one independent variable, which can be solved by familiar tractable ODE methods [2]. Some useful techniques for changing PDEs to ODEs have been developed, including separation of variables, integral transforms, Fourier transformation, eigenmode expansion, and numerical methods. Thanks to the increasingly powerful computation capability of CPUs and GPUs in past few decades [6], numerical methods [7] that change a PDE to a system of difference equations which could be solved by iterative techniques on a high-speed computer have become popular and efficient for many applications, especially for wave motions. There are many types of wave motions concerned in nature and engineering, such as the propagation of acoustic waves, water waves, the vibrations in a string or a thin membrane or even in solids, and the electromagnetic waves of light and electricity, as well as probability waves in quantum mechanics. These wave motions can be generally described by a second-order linear PDE [8] given by equation (1) below, where x, y, z are the three-dimensional coordinates in space, and t is the time variable.

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)+f(x,y,z,t) \quad (1)$$

The dependent variable $u(x,y,z,t)$ in equation (1) may represent the pressure or density in a liquid or gas, or the displacement of the particles in a vibrating solid along some direction, or any other possible characteristic for the wave, while $f(x,y,z,t)$ could be external disturbance or inputs to the wave motion, such as force, pressure or power that would change the motion dynamics. And c is the coefficient constant which denotes the propagation speed of the wave. Specifically, in the two-dimensional space domain where $x \in [a_x, b_x]$ and $y \in [a_y, b_y]$, $t \in [0, t_{final}]$, the wave equation can be written as

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+f(x,y,t) \quad (2)$$

Additionally, the wave equation might be constrained by some initial conditions shown in equation (3) [9], along with some known surrounding boundary conditions of the region depicted by equation (4-5).

$$u(x,y,0)=\phi(x,y), \frac{\partial u}{\partial t}(x,y,t)|_{t=0}=\psi(x,y) \quad (3)$$
$$u(a_x,y,t)=g_{ax}(y,t), u(b_x,y,t)=g_{bx}(y,t) \quad (4)$$
$$u(x,a_y,t)=g_{ay}(x,t), u(x,b_y,t)=g_{by}(x,t) \quad (5)$$

For this kind of 2-D second-order wave equation with given initial value and boundary value, numerical method often starts from discretization of the region into a uniform Cartesian grid according to the surrounding boundaries, then uses the 2-D central difference method to approximate the second-order space partial derivatives for $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$, and a second-order accurate approximation for $\frac{\partial^2 u}{\partial t^2}$, thus the wave equation becomes a finite difference equation [10] in the recursive form, which could be solved by numerical computation.

Specifically, for the IBVP we are going to solve in this project, its discrete approximation is as follows in equation (6), along with its initial conditions and boundary conditions in equation (7-12):

$$D_{+t}D_{-t}U_i^n = c^2\left[D_{+x}D_{-x}+D_{+y}D_{-y}\right]U_{i,j}^n + f \quad (6)$$
$$U_{i,j}^0 = u_0\left(x_{i,j}, y_{i,j}\right) \; i=0,1,2,\dots N_y-1 \quad (7)$$
$$U_{i,j}^1 = U_{i,j}^0 + \Delta t u_1 + \frac{1}{2}\Delta t^2 u_0\left[c^2\partial_{xx}+c^2\partial_{yy}\right]+f \quad (8)$$
$$U_{0,j}^{n+1} = g_{ax}\left(y_{0,j}, t^{n+1}\right) \quad (9)$$
$$U_{i,0}^{n+1} = g_{ay}\left(x_{0,j}, t^{n+1}\right) \quad (10)$$
$$U_{N_x,j}^{n+1} = g_{bx}\left(y_{N_x,j}, t^{n+1}\right) \quad (11)$$
$$U_{i,N_y}^{n+1} = g_{by}\left(x_{i,N_y}, t^{n+1}\right) \quad (12)$$

The current finite difference method has been well studied on large, high-performance computer systems. Inspired by massively parallel computer systems which use multiple processing elements simultaneously for solving problems, we came up with the idea of dividing the whole working space into pieces. These pieces, assigned to particular processors in the parallel system, can be handled at the same time. In this way, the execution of processes is carried out simultaneously [11]. Compared to using one large processor computing in serial, this parallel processing approach can compute the wave equation, the partial differential equation we study on, with higher speed and lower power consumption. In this paper, we proposed a novel and effective solution with the utilization of GPUs on CUDA and the adoption of MPI and pthreads for the second-order wave equation in two dimensions considering the initial boundary-value problem.

Although various algorithms were developed to solve PDEs, including elliptic, parabolic, and hyperbolic types, the basic principle is still to split or divide the time and space variables into individual intervals. A virtual grid can be generated, with x axis being spatial variable while y axis being variable for time. The serial algorithm is straightforward, which first of all creates the grid with specified values of x and t, and then for each iteration/tick, apply the update rule to each node on the grid. The algorithm is such that when computing the value at time $t[i+1]$, the values at its left bottom, bottom, and right bottom, are required. The communication between each rank is such that ranks have the demand to exchange node values with their neighbors. In this case, each rank will need two extra columns to store their neighbors' edge value, except for the columns at boundary since no wrap is needed for this type of problem. To further reduce the message passing cost, two columns of ghost cells can be added to each side of the grid for a single rank. In this way, redundant computations are executed in exchange of message passing. It was reported that when the number of ghost columns increases, the cost decreases first and then increases.

The system we used for this project is the Blue Gene/Q (BG/Q), which is the most advanced supercomputer from the Blue Gene family. The IBM Blue Gene supercomputers play an essential role in the high-performance computing field with their superior performance. Blue Gene/L (BG/L) is the first generation released in 2004, dramatically improving the cost/performance for a large number of applications with good scaling behavior [12, 13]. Blue Gene/P (BG/P) is basically an improved version of BG/L, doubling the core count, memory capacity, and also the bandwidth; slightly increasing core frequency; and employing a coherent cache [14]. BG/Q is the third and also the latest generation of the IBM Blue Gene massively parallel supercomputer systems. BG/Q made great progress on processor and interconnect technologies. The peak performance is 209 TF per rack, and the power efficiency is approximately 2.1 GFlops/W without optical interconnect [14]. The key architectural features of Blue Gene systems are the packaging and system management [15, 16]. Every rack of the Blue Gene system is organized in a hierarchical manner, consisting of two mid-planes, eight link cards, and two service cards. For BG/Q, there are 5 racks with up to 5120 nodes. Each node contains a 16-core PowerPC A2 processor [17], with 4 hardware threads per core, and DDR3 memory. The architecture of BG/Q allows an execution mode of a combination of MPI and threads with faster thread context switching. It can satisfy high-performance processing needs and can reach higher operating speed with low power cost. The BG/Q supercomputer was designed to dramatically improve cost/performance for expanding a broad class of applications reaching ease of programming, broader application front, and increasing capability compared to Blue Gene L and Blue Gene P. BG/Q can perform the most accurate and precise results even with complexity and unpredictable workloads. Its ultra-scalability and high reliability greatly help researchers for developing breakthrough science [17, 18].

Our project is built based on the CUDA architecture, which exposes GPU parallelism for solving wave equations in high-performance computing. The CUDA platform gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels [19]. In order to perform high-efficient communication in x-direction, we use the message passing interface (MPI) to pack data into a contiguous buffer before sending [20]. Combining parallel I/O technologies with MPI would greatly improve both the portability and convenience for performing parallel file Input/output [21].

II. Implementation of Parallel Algorithms

A. Sequential Program Design

As depicted in the introduction section, the schema of the computation of $U_{i,j}^{n+1}$ is associated with the value of $U_{i,j}^{n}$, $U_{i,j}^{n-1}$, $U_{i+1,j}^{n}$, $U_{i,j+1}^{n}$, $U_{i-1,j}^{n}$, and $U_{i,j-1}^{n}$, which naturally connects the updated value of $U_{i,j}$ at a new time step with the value at location (i,j) and its four neighbors in a previous time step, as well as the value of location (i,j) in two time steps ahead. Thus, three variables were needed to store the value of $U_{i,j}$ at time step n-1, n, and n+1. An illustration shown below vividly demonstrates the core idea, where the blue mark denotes the value at time step n+1, location (i,j). The time step formula (13) is used, which can ensure the convergence of the algorithm, where λ_{CFL} is known as Courant-Friedrichs-Lewy parameter and is taking a value of 0.9 in the current study. In a typical serial program, the grid was firstly defined, with desired size. Then, the grid was initialized with the initial condition (7) and (8). A single iteration would be necessary to go through time from 0 to *final_time*. In each iteration, the interior values on the grid at time *n+1* would be calculated based on formula (6), followed by the application of boundary conditions (9-12). Finally, a swap between variable U^n and U^{n-1} and another swap between U^n and U^{n+1} is needed.

$$c^2\Delta t^2\left(\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}\right)=\lambda_{CFL}^2 \qquad (13)$$

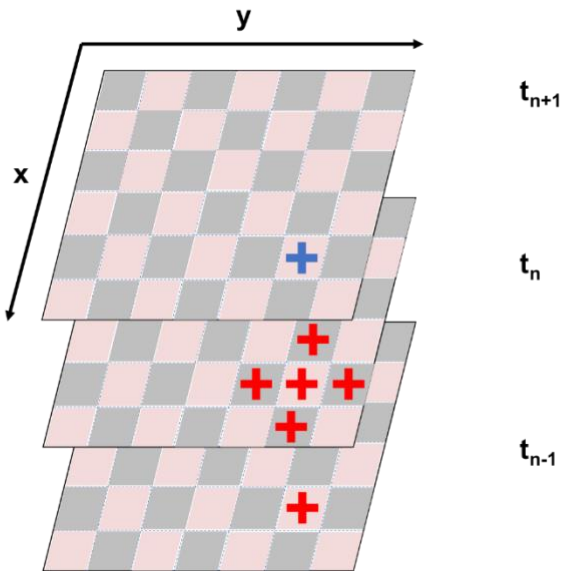


Figure 1: Schema showing the update rule for computing the value at time step n+1, location (i,j) $U_{i,j}^{n+1}$ based on previous data.

B. Parallel Program Design without MPI

The original parallel program was designed based on the serial design, where CPU was used to allocate memories for variables, initialize the grid, coordinate data allocation and apply boundary conditions, while GPUs were used to take care of the core computations, where the variables of current data $U_{i,j}^{n+1}$, previous data $U_{i,j}^{n-1}$, and data from two steps $U_{i,j}^{n-2}$ ahead were passed as reference to GPU, which is in charge of the update of the data at time step n+1. Apart from that, each GPU can also have multiple threads, each conducts a computation on a single value of $U_{i,j}^{n}$. The Listing 1 below displays the core part of the CUDA code for updating $U_{i,j}^{n+1}$.

Listing 1: Core CUDA code for updating $U_{i,j}^{n+1}$, along with a device function to calculate the forcing term f.

```
// A device function that is used to calculate forcing term f
based on the current location x and y
__device__ double f(double x, double y, double t){
   double b0 = 0.5; double b1 = 0.7; double b2 = 0.9;
   double c00 = 1; double c10 = 0.8; double c20 = 0.6;
   double c11 = 0.3; double c01 = 0.25; double c02 = 0.2;
   double c = 1;
   return 2*b2*(c20*y*y + c11*y*x + c10*y + c02*x*x +
c01*x + c00) - c*c * (2*c20*(b2*t*t + b1*t +
b0)+2*c02*(b2*t*t + b1*t + b0));
}

// CUDA function to compute the value of U_{i,j}^{n+1}based on
previous data.
__global__ void gol_kernel(const double* g_data,
unsigned int worldWidth, unsigned int worldHeight,
double* g_resultData, double* g_dataOld, double t,
double dt, double dx, double dy, double c)
{
   int x, y;
   double fvalue = 0;
   unsigned long y0, y1, y2, x0, x1, x2;
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   while (index < worldHeight*worldWidth) {
      y = index / worldWidth;
      x = index % worldWidth;
      y0 = ((y + worldHeight - 1) % worldHeight) *
worldWidth;
      y1 = y * worldWidth;
      y2 = ((y + 1) % worldHeight) * worldWidth;
      x0 = (x + worldWidth - 1) % worldWidth;
      x1 = x;
      x2 = (x + 1) % worldWidth;
      fvalue = f(y*dx,x*dx,t-dt);
      g_resultData[x1+y1] = 2 * g_data[x1+y1] -
g_dataOld[x1+y1] + dt*dt*(c*c/(dx*dx)*(g_data[y2+x1]-
2*g_data[y1+x1]+g_data[y0+x1]) +
c*c/(dy*dy)*(g_data[y1+x2]-
2*g_data[y1+x1]+g_data[y1+x0])) + dt*dt * fvalue;
      index += blockDim.x * gridDim.x;
   }
}
```

C. Parallel Program Design with stripe MPI, Algorithm I

After bringing MPI into consideration, the algorithms become slightly more complex, since the program would have steps to initialize MPI, to assign memories to each MPI rank, to take care of data exchange among neighboring ranks. However, the merit of using MPI is that multiple processors can work on the individual subtasks at the same time, each with multiple threads to improve the computing efficiency. During the implementation phase, several aspects shall be paid attention to. First of all, the size of each rank may not be the same, which arises when the size of the grid cannot be evenly distributed to each rank. In this scenario, the rows of the last rank will be slightly more comparing to the other ranks. Secondly, the exchange between ghost rows of the first rank and the last rank can be different than the other ranks. For the ghost row exchange of the current PDE problem, such consideration was arranged: since the first row and the last row were the boundaries and there are separate boundary condition functions working on them, the value of boundaries was not given special attention during the GPU computation, thus avoiding multiple if statement inside the CUDA code to waste cycles. Instead, the values of the boundaries were updated again after the job on GPU finishes. On the two boundary columns at the left and right side of the grid, no ghost data is needed since the values at these locations can be updated later in CPU with boundary functions. Thus, their values were computed based on wrapping approach, which were “fake” and may cost some computing resources. However, comparing to a series of if statement that every thread will go through, the trade-off is beneficial. Thirdly, the number of ranks shall not be too small to increase the communication overheads, or too large to make the implementation of MPI rank not meaningful. The number of ranks will greatly depend on the size of the PDE grid, the best value of which will be discussed in the following chapters. A schematic was shown in Figure 2 to illustrate the ghost row exchange between ranks, where the solid squares represent the data for each rank, while the dashed lines represent the ghost rows to be exchanged with the neighboring ranks. The communication is relatively easy, each rank firstly starts receiving from their prior rank and next rank, and subsequently send their first row and last row to the prior rank and next rank.

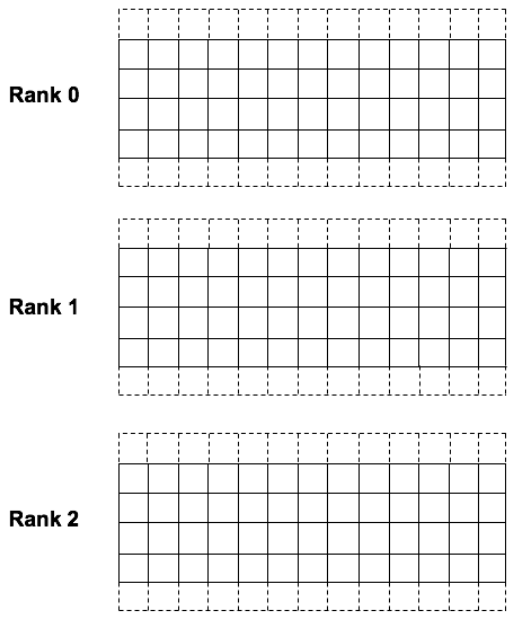


Figure 2: Schematics illustrating the data and ghost rows of each rank.

The pseudocode of Algorithm I is shown below.

```
Input: m: PDE world size magnitude; t: final time
Output: A: a 1D array with PDE solution at time t
Begin
    Step1: Initialize MPI with input arguments.
    Step 2: Set up devices to determine the number of
        ranks and the current rank.
    Step 3: Determine the number of iterations
        iteration_time based on the size of the PDE
        world worldSize and the time final_time.
    Step 4: Allocate enough space for the grid of the
        rank.
    Step 5: Allocate enough memories for ghost
        variables ghost_up and ghost_bottom.
    Step 6: Initialize the grid of each rank by applying
        initial functions.
    Step 7: Begin iteration until the counter i reaches
        specified iterations.
    Step 8: Get current time if myrank is 0.
    For counter $i = 1, 2, \ldots,$ iteration_time – 1 do
        Step 9: Exchange data among different ranks
        and store in ghost rows.
        For MPI rank $i = 0, 1, \ldots,$ numranks-1 do
            Step 9.1: Receive ghost rows from the rank
                underneath and above and store in
                ghost_data_up and ghost_data_bottom.
            Step 9.2: Send the first row to the prior rank
                and the last row to the next rank.
            Step 9.3: Wait until all rows were exchanged
                using MPI_Wait()
        END
        Step 10: Launch kernel with required
            arguments.
        Step 11: After the world is updated to new time
            step, apply boundary conditions to 4
            surrounding borders.
        Step 12: Switch the content between g_data ($t_n$)
            and g_dataOld ($t_{n-1}$).
        Step 13: Switch the content between
            g_resultData ($t_{n+1}$) and g_data ($t_n$).
        Step 14: Get current time if myrank is equal to
            0.
    End
End
```

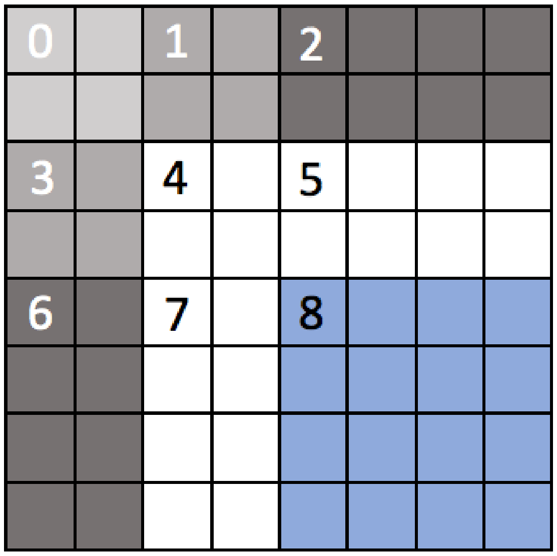
Algorithm I: Stripe MPI Algorithm

D. Parallel Program Design with cube MPI, Algorithm II

The cube algorithm was designed based on the idea of dividing the whole grid into square-like cubes and assign to each rank. The implementation of this algorithm takes much more efforts comparing to the MPI algorithm with stripe-shape rank, which is due to the fact that four ghost rows were required for each rank. Similar to the previous algorithm, the troublemaker of this algorithm is the determination of the rows and columns of each rank. Since the grid size may not be evenly divided among each rank, the size of ranks will generally have discrepancies. For example, when the magnitude of the grid is taken a value of 4, the rows and the columns of the grid will be 161. If 9 (3 on each row and 3 on each column) ranks were used, then the first 2 ranks (0th and 1st) will have 53 columns each, the 2nd rank will have 55 columns. The same applies at the row direction, which will cause the 0th and 3rd rank to possess 53 rows each, which the 8th rank to possess 55 rows and columns. An illustration is shown below, where the ranks will generally possess different size, so does the size of the ghost row up/bottom/left/right. It is also problematic when determining the size of the data to be transmitted to its neighboring ranks. Different from the stripe MPI algorithm that only two ghost rows were needed to exchange in each iteration, the cube MPI algorithm requires 4 ghost rows to exchange, which is expected to increase the overheads during communication, not to mention the extra work required for memory allocation, grid initialization, more complex updating rules, etc. It was anticipated that only if the grid size is large enough and more computing nodes were allowed, the overheads can be compensated.

Figure 3: Schematic illustrating the problem of determining the size of each rank during the implementation of cube MPI algorithm.

Fig. 4 below shows an illustration of each rank and their associated ghost rows/columns. The size of each



rank is determined based on several rules:

1) The initial size of row and column is determined based on $n/\sqrt{p}$, where n is the total row/column, while p is the number of processors. Each process will manage a total of $(n/\sqrt{p})$ x $(n/\sqrt{p})$ squares.

2) Check the location of the rank to determine whether the rows per rank or columns per rank needs to be adjusted. Make corresponding changes to these variables.

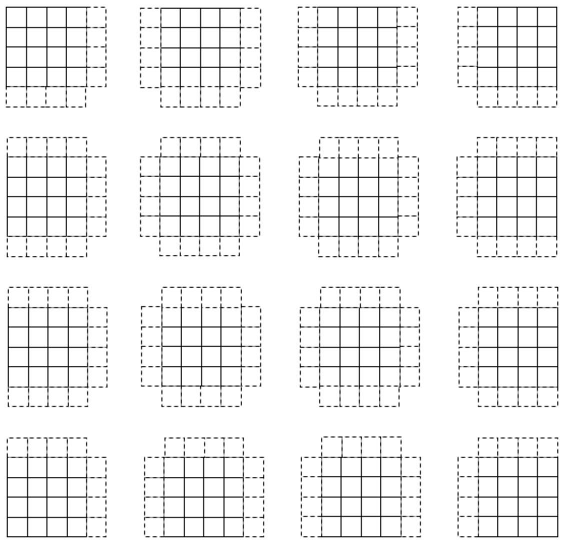


Figure 4: Schematic showing the allocation of each rank and their associated ghost rows/columns.

Fig. 5 shows the details in the ghost data exchange for cube MPI algorithm, where dark squares represent the current data to be computed, the gray squares represent the data that are available from the current rank, while the blue squares represent the data from ghost rows. For the two columns and two rows at the boundaries, their neighbors were determined with the wrapping approach, such that the rank at first row will exchange data with the rank in the last row, while the left-most rank will exchange data with the right-most rank.

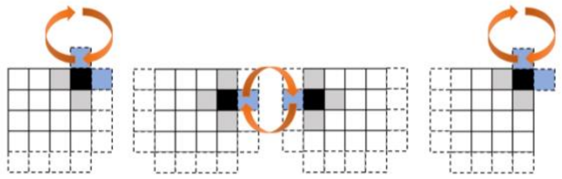


Figure 5: Schematic illustrating data exchange among ranks in the cube MPI algorithm. Each rank will receive data from four directions and need to send data to its neighboring 4 ranks.

The pseudocode of Algorithm II is shown below.

```
Input: m: determine PDE world size; t: final time
Output: a 1D array with PDE solution at time t
Begin
  Step1: Initialize MPI with input arguments;
  Step 2: Set up devices to determine the number of
    ranks and the current rank. The number of ranks
    has to be square number.
  Step 3: Determine the number of iterations
    iteration_time based on the size of the PDE
    world worldSize and the time interval final_time.
```

```
Step 4: Allocate enough space for the grid of the rank.
Step 5: Allocate enough space for ghost variables ghost_data_up, ghost_data_bottom, ghost_data_left, and ghost_data_right.
Step 6: Initialize the grid of each rank by applying initial functions.
Step 7: Begin iteration until the counter i reaches specified iterations.
Step 8: Get current time if myrank is 0.
For counter $i$ = 1, 2, …, iteration_time – 1 do
    Step 9: Exchange data among different ranks and store in ghost rows. Different from the above algorithm that only needs to exchange among above and below ranks, this new algorithm requires data exchange to ranks in all four directions.
    For MPI rank $i$ = 0, 1, …, numranks-1 do
        Step 9.1: Receive ghost rows from the rank underneath and above and store in ghost_data_up and ghost_data_bottom.
        Step 9.2: Receive ghost rows from the rank at left and right of the current rank. Store the data received in ghost_data_left and ghost_data_right.
        Step 9.3: Send the left column to the left rank and the right column to the right rank.
        Step 9.4: Wait until all rows were exchanged using MPI_Wait()
    End
    Step 10: Launch kernel with required arguments.
    Step 11: After the world is updated to new time step, apply boundary conditions to 4 surrounding borders.
    Step 12: Switch the content between g_data ($t_n$) and g_dataOld ($t_{n-1}$).
    Step 13: Switch the content between g_resultData ($t_{n+1}$) and g_data ($t_n$).
    Step 14: Get current time if myrank is equal to 0.
  End
End
```

Algorithm II: Cube MPI Algorithm

III. Compute Performance Evaluation

As stated in section II, three algorithms were implemented, the first one is the one with only CUDA, the second and the third are associated with CUDA and MPI. Specifically, the second one is implemented with stripe-geometry MPI ranks, while the third is implemented with cube-geometry MPI ranks.

A. Implementation with CUDA

To evaluate the performance with CUDA, the program was executed with matrices sizes ranging from 10×2^1 to 10×2^8 and with the number of threads being 64, 128, 256, 512 and 1024 respectively.

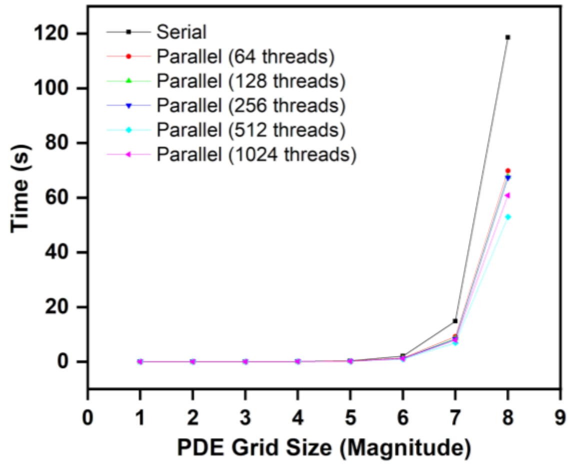


Figure 6: Execution time at different thread numbers and matrix sizes

Table I: Speedup Ratio at Different Threads and Matrix Sizes

Grid size (magnitude)	Number of threads				
	64	128	256	512	1024
1	1.877	2.016	2.046	2.074	**2.432**
2	1.953	1.991	1.964	2.125	2.231
3	1.358	1.436	1.704	2.139	2.369
4	0.908	0.938	0.815	0.847	0.972
5	1.340	1.564	1.510	2.085	2.032
6	1.478	1.757	1.695	2.141	1.874
7	1.609	1.738	1.802	2.145	1.873
8	1.698	1.752	1.760	2.237	1.949

The performance of our first implementation is shown in Fig. 6 and Table I. Several observations can be made. First of all, the time cost to execute the program remains almost constant when the magnitude of the PDE grid varies from 1 to 5. This mainly because the majority of the time is used to initialize the world, to set up the memories, and the actual time to update the world is relatively small when the world size is not large. When further increase the world size magnitude, the time to solve the program increased drastically, especially for serial algorithm. This is because there is only one thread working for the serial program, which does not fully exploit the computing resources. For parallel programs, increasing the number of threads generally reduces the execution time for solving the problem. This is because each thread can work simultaneously and thus improving the efficiency. Secondly, taking a look at the performance of the scenario with 512 threads and 1024 threads, one can notice that after the magnitude of the world increased to 5, the speedup ratio of the former exceeds the latter. This is because when 1024 threads were involved, a noticeable portion of the time were used to allocate necessary resources for each thread such as space and registers. The coordination among these threads also cost cycles. Finally, it was noticed that the maximum speedup ratio was achieved when the grid size is 1 (a grid of 21 x 21), and when the number of threads is 1024.

B. Implementation with CUDA & MPI

Two different algorithms were implemented with MPI. The first stripe algorithm divided the grid into multiple strips, with rank 0 to rank n arranged in order from above to below.

In our CUDA & MPI implementation, we use two different methods to decide the ranks, dividing the matrix by rows or by submatrices. (see Fig. 2 and 4). The performance evaluation is designed in such a fashion as listed below. For all evaluations, only one parameter is changing, and the others remain the same. The baseline is set such that each process will use 500 threads, 4 GPUs were allocated, each launch 4 processes. The node is 1.

1) Strong scaling analysis of both strip MPI and cube MPI;

2) Weak scaling analysis of both strip MPI and cube MPI;

3) The comparison of execution time of strip MPI and cube MPI vs. various number of ranks;

4) The comparison of execution time of strip MPI and cube MPI vs. various number of GPUs.

(1) Strong scaling results

Fig. 7 shows the plot of strong scaling, where the number of nodes increases but the grid size remains unvaried. The execution time decreases almost linearly for both algorithms, which is because increasing the number of nodes will proportionally increase the number of ranks and thus increase the number "workers" that simultaneously working together. On the other hand, the execution time for cube MPI is constantly higher than that of stripe MPI, but their difference was significantly narrowed when the number of nodes increases to 4. This is because when the number of nodes is not large enough to compensate the overheads of allocating extra ghost rows and communication cost, the computing efficiency will not be ideal. When the number of nodes increases and the number of ranks increases, the extra workers can make a difference.

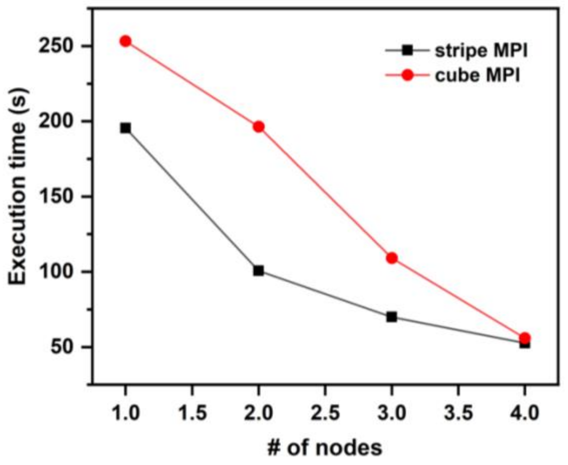


Figure 7: plot of strong scaling results showing improvement in computing efficiency with the increasement of the number of nodes while maintaining the same grid size.

To further understand the difference between these two algorithms, cycle counter was brought into consideration for the assistance of the performance analysis. The data was summarized in Table II. It can be seen that for strip MPI algorithm, both communication overheads and kernel time decrease. This is because when the number of nodes increases, more ranks will come into being. Each rank will have a smaller-size subtask to complete, thus smaller-size ghost row/columns to exchange, which leads to a reduction in the communication overheads for each rank. Since each rank will have a smaller task, it will take less time for the kernel to execute. It was also observed that the cube MPI algorithm possesses a faster reduction rate in execution time, comparing to the strip MPI, which might suggest that when further increase the number of nodes, the performance of cube MPI algorithm might transcend that of the strip MPI algorithm.

Table II: The communication overhead and kernel cycle count for strong scaling of both algorithms.

Strong	Strip MPI		Cube MPI	
# nodes	Comm time(s)	Kernel time(s)	Comm time(s)	Kernel time(s)
1	7.172	151.071	17.526	130.190
2	3.428	67.607	8.708	128.060
3	3.189	41.949	4.126	62.666
4	1.369	35.048	2.241	38.391

(2) Weak scaling Results

Different from strong scaling, weak scaling involves increasing the number of nodes while increasing the size of the grid. Fig. 8 shows the plot of the results for weak scaling, where the execution time of both algorithms increases with the increase of the number of nodes and grid size. Ideally, the execution time shall not fluctuate too much, which would suggest a great scalability. However, since the number of nodes can be allocated has a limit of 4, while the grid size increasement is exponential, the fast increasement in execution time is not surprising. It is expected, though, that when the number of nodes further increases, the performance will be significantly ameliorated.

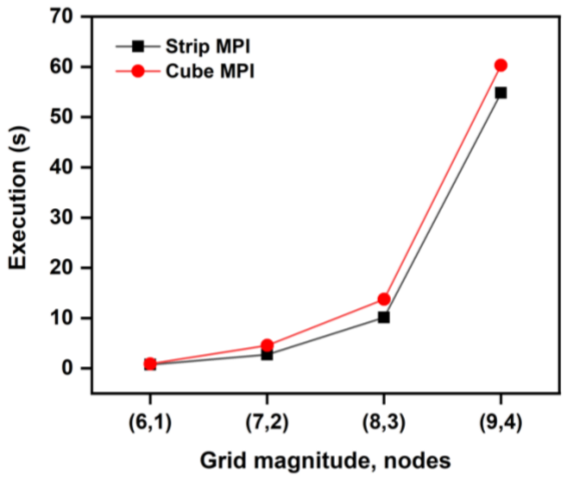


Figure 8: Plot of weak scaling results, showing increase in execution time when the number of nodes and the grid size increased at the same time.

The overheads of communication and the actual time cost to execute the kernel were listed in Table III. It can be noticed that both communication overheads and the actual kernel execution cycles increases with the increase of the grid size and the number of nodes. This is reasonable since the nodes increases linearly while the grid size increases exponentially. In overall, the ratio of communication overheads to the summation of communication overheads and kernel counts decreases for both algorithms. However, the cube MPI algorithm constantly possesses a higher communication overheads ratio comparing to the strip MPI, which is around bifold that of the strip MPI. This is due to the double workload in the ghost row/column exchange for the latter.

Table III: The communication overhead and kernel cycle count for weak scaling of both algorithms.

Weak	Strip MPI		Cube MPI	
Grid mag, nodes	Comm time (s)	Kernel time (s)	Comm time (s)	Kernel time (s)
(6,1)	0.067	0.408	0.192	0.477
(7,2)	0.257	1.630	0.779	2.678
(8,3)	0.835	6.406	1.663	8.676
(9,4)	4.923	34.245	10.251	35.263

(3) Impact of the number of ranks

When the grid size remains unvaried, the increase in the number of ranks will ideally lead to the shrinkage of execution time. However, it is not always true, since the initialization of MPI ranks, the communication overheads among ranks, and the allocation of extra memories for ghost data will lead to the consumption of resources and will slow down the execution if the number of ranks exceeds a certain value. As shown in Fig. 9, for strip MPI algorithm, the execution time decreases fast when the number of ranks increases from 1 to 4. However, the execution time increases when the number of ranks further increases. This is due to the communication overheads mentioned above. For cube MPI algorithm, the execution time remains almost constant at beginning and decreases with the increase of the number of ranks. It might suggest that the bottleneck of this algorithm has not reached. It was anticipated that when further increasing the number of ranks for cube MPI algorithm, the execution time will increase as well.

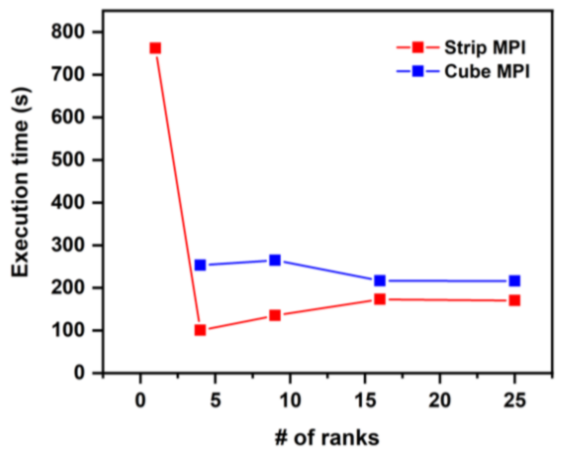


Figure 9: Plot showing the impact of the number of ranks to the execution time.

Table IV: The communication overhead and kernel cycle count for the impact of number of ranks

	Strip MPI		Cube MPI	
Rank	Comm time (s)	Kernel time (s)	Comm time (s)	Kernel time (s)
1	0.602	527.839	/	/
4	3.428	67.607	17.526	130.19
9	72.555	47.152	13.943	222.434
16	85.987	63.439	59.099	141.603
25	60.375	98.924	24.257	181.133

(4) Impact of the number of GPUs

Generally speaking, when the number of GPUs increases, the overall performance of the program shall be increased due to the fact that the computing resources increase. Fig. 10 below shows the plot of execution time vs. the number of GPUs for both algorithms. It can be observed that it is indeed the case that the performance of the both algorithms get improved with the number of GPUs. However, the rate of the reduction in execution time becomes flat after the number of GPUs reaches 4 to 5. This is mainly due to the overheads of communication among MPI ranks because the number of total ranks also rises proportionally with the number of GPUs.

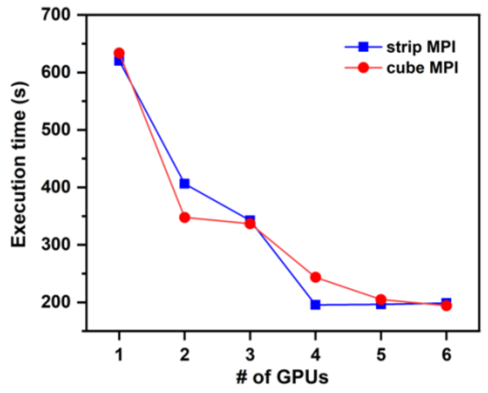


Figure 10: Plot of the impact of the number of GPUs on the performance of two algorithms. The execution time keeps decreasing with the increase of the number of GPUs.

(5) Strong scaling efficiency

Based on strong scaling efficiency $\frac{t_1}{N \times tN} \times 100\%$, we can calculate the strong scaling efficiency of our implementation. The result is shown in Fig. 11 below, which shows that both methods have the same trend when applying the strong scaling configuration. However, deciding the MPI ranks by rows performs. To understand how the same algorithm is affected by different methods of deciding MPI ranks, we need to take the communication overhead into consideration.

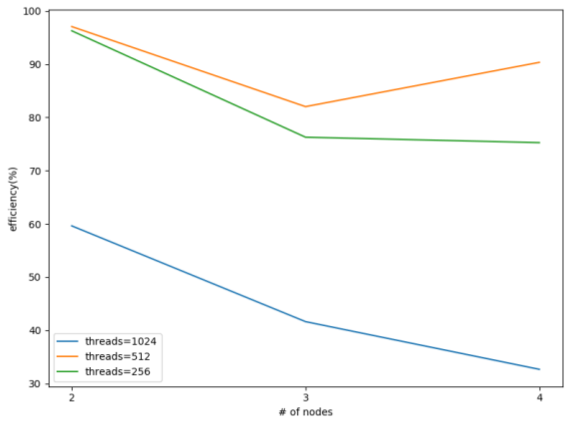
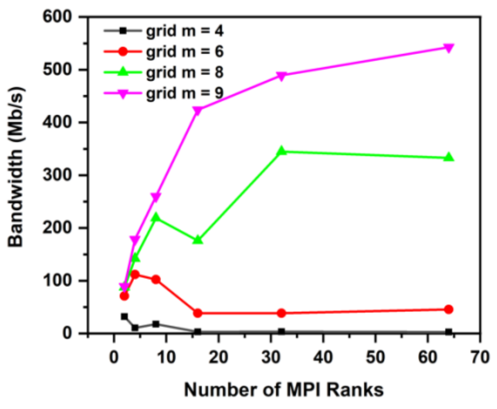
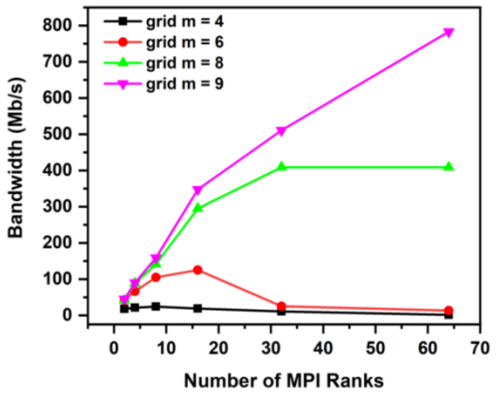


Figure 11: Strong Scaling Efficiency

V. PARALLEL I/O PERFORMANCE EVALUATION

To evaluate the I/O performance, the strip algorithm with parallel I/O was executed with 2, 4, 8, 16 MPI ranks, and the grid magnitude being 9, 10, 11, 12. The actual grid size is thus 5120 x 5120, 10240 x 10240, 20480 x 20480, and 4060 x 40960, respectively. The file size of our grid is 0.36 Mb, 5.75 Mb, 91.84 Mb, 366 Mb, respectively.





It can be observed that both the read I/O and write I/O bandwidth increase with the number of MPI ranks in the beginning. For larger grid size, the bandwidth keeps increasing for both read and write. It can be expected that the increasing trend will slow down gradually and then becomes flat, which seems to suggest that when there are more ranks to collaborate together, the efficiency will be improved. When further increasing the number of ranks, however, due to the communication overheads, the benefits of having multiple ranks will be compensated by the cost in the ghost row/column exchange. For smaller grids, there bottleneck was reached at a very early time because they do not need that many ranks to work together, such that their bandwidth starts to drop when increasing the number of ranks.

VI. CONCLUSIONS AND FUTURE WORK

This project demonstrates a hybrid parallel computing model for accelerating wave equation solvers in two-dimensional spaces, dealing with the initial boundary-value problem. Two algorithms with CUDA and MPI, strip MPI algorithm and cube MPI algorithm, were implemented and their performance was evaluated based on the results of strong scaling and weak scaling. The impact of number of ranks, the number of ghost columns, the number of GPUs, the number of computing nodes, and many other factors were considered and systematically addressed. Several conclusions can be made based on the findings:

(1) The parallel algorithm is superior than the serial algorithm, especially when the size of the grid is large. The advantages will be enlarged if further increase the size of the grid.

(2) Strip MPI algorithm is superior than the cube MPI algorithm, based on the strong scaling and weak scaling analysis. However, if the grid size remains unchanged while increase the number of computing nodes, cube MPI algorithm might surpass the strip MPI algorithm.

(3) The overheads in the cube MPI algorithm occupies a great portion of the total execution time, which is around bifold that of the strip MPI algorithm. This is because the cube MPI algorithm requires 4 ghost rows/columns to be exchanged with the neighboring rank while the strip MPI algorithm only requires 2. Only if the grid size is large enough the overheads can be compensated.

(4) The execution time of both algorithms decreases with the increase in the number of ranks per GPU, due to the better exploitation in the computing resources. When further increasing the number of ranks, the performance of the strip MPI algorithm deteriorated, due to the increase in the communication overheads.

(5) The execution time of both algorithms decreases with the increase of the number of GPUs since more computing resources were allocated for the program, thus more ranks were coming into work. However, the performance improvement becomes flat due to the increase in the communication overheads.

In the future, we plan to further extend the algorithm to explore the impact on the number of ghost row/column. Currently, we have one single line of data transmitted with the neighboring ranks. Another possibility is to exchange multiple lines of data to reduce the computation cost. In this way, the algorithm is able to compute two time steps without further message passing. It was reported that the parallel overheads will decrease first with the increase in the number of ghost rows/columns [22] and then increase again. It would be exciting to validate these statements and further exploit the charm of parallem computing.

VII. ACKNOWLEDGMENT

This research was supported by the AiMOS supercomputer at the center for computational innovations of Rensselaer Polytechnic Institute.

VIII. CODE STRUCTURE

The absolute path of the code for the PDE solver is /gpfs/u/home/PCP9/PCP9gngb/scratch/submission

The gol.cu CUDA code in the no_MPI folder is the original CUDA file with no MPI implementation. To compile the code, first load the essential moduls, and then use "nvcc -O3 -gencode arch=compute_70, code=sm_70 gol.cu -o gol" to compile. To run the code, use "./gol grid magnitude, final_time, threads count, plot_or_not", for example: "./gol 4 1 500 0"

To execute the algorithm of strip MPI, go to with_strip folder. Compile the file with Makefile and then execute using the sbatch and slurmSpectrum.sh

To execute the algorithm of cube MPI, go to with_cube folder. Compile the file with Makefile and then execute using the sbatch and slurmSpectrum.sh

To execute the algorithm of parallel I/O, go to with_IO folder. Compile the file with Makefile and then execute using the sbatch and slurmSpectrum.sh

References

[1] P. Worley, "Parallelizing across time when solving time-dependent partial differential equations."

[2] V. Grigoryan, "Math 124A–Fall 2010," 2010.

[3] M. Renardy, and R. C. Rogers, *An introduction to partial differential equations*: Springer Science & Business Media, 2006.

[4] R. Courant, and D. Hilbert, "Methods of Mathematical Physics, Volume II: Partial Differential Equations," *American Journal of Physics*, vol. 31, no. 3, pp. 221-221, 1963.

[5] S. J. Farlow, *Partial differential equations for scientists and engineers*: Courier Corporation, 1993.

[6] R. Mehra, N. Raghuvanshi, L. Savioja, M. C. Lin, and D. Manocha, "An efficient GPU-based time domain solver for the acoustic wave equation," *Applied Acoustics*, vol. 73, no. 2, pp. 83-94, 2012.

[7] L. Olsen-Kettle, "Numerical solution of partial differential equations," *Lecture notes at University of Queensland, Australia*, 2011.

[8] A. Salih, "Second-Order Wave Equation," 2016.

[9] V. Korzyuk, V. Erofeenko, and J. Sheika, "Classical Solution for Initial–Boundary Value Problem for Wave Equation with Integral Boundary Condition," *Mathematical Modelling Analysis*, vol. 17, no. 3, pp. 309-329, 2012.

[10] R. Alford, K. Kelly, and D. M. Boore, "Accuracy of finite-difference modeling of the acoustic wave equation," *Geophysics*, vol. 39, no. 6, pp. 834-842, 1974.

[11] A. Gottlieb, and G. Almasi, *Highly parallel computing*: Benjamin/Cummings Redwood City, CA, 1989.

[12] M. Gilge, *IBM system blue gene solution blue gene/Q application development*: IBM Redbooks, 2014.

[13] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, and G. V. Kopcsay, "Overview of the Blue Gene/L system architecture," *IBM Journal of research development*, vol. 49, no. 2.3, pp. 195-212, 2005.

[14] K. Yoshii, K. Iskra, R. Gupta, P. Beckman, V. Vishwanath, C. Yu, and S. Coghlan, "Evaluating power-monitoring capabilities on IBM Blue Gene/P and Blue Gene/Q." pp. 36-44.

[15] G. Almasi, S. Asaad, R. E. Bellofatto, H. R. Bickford, M. A. Blumrich, B. Brezzo, A. A. Bright, J. R. Brunheroto, J. G. Castanos, and D. Chen, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research Development*, vol. 52, no. 1-2, pp. 199-220, 2008.

[16] P. Coteus, H. R. Bickford, T. M. Cipolla, P. G. Crumley, A. Gara, S. A. Hall, G. V. Kopcsay, A. P. Lanzetta, L. S. Mok, and R. Rand, "Packaging the blue gene/L supercomputer," *IBM Journal of Research Development*, vol. 49, no. 2.3, pp. 213-248, 2005.

[17] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, and R. Wisniewski, "The ibm blue gene/q compute chip," *Ieee Micro*, vol. 32, no. 2, pp. 48-60, 2011.

[18] G. Chiu, "The IBM blue gene project," *IBM Journal of Research Development*, vol. 57, no. 1, pp. 1-6, 2013.

[19] F. Abi-Chahla, "Nvidia's CUDA: The End of the CPU?'," *Tom's Hardware*, pp. 1954-7, 2008.

[20] W. Gropp, W. D. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*: MIT press, 1999.

[21] H. Yu, and K.-L. Ma, "A study of I/O methods for parallel visualization of large-scale data," *Parallel Computing*, vol. 31, no. 2, pp. 167-183, 2005.

[22] M. Quin, “parallel programming in Cwith MPI and OpenMP,” *Tata McGraw Hills edition,* 2000.