Performance Benchmark of A Parallel Sparse Linear System Solver Using Conjugate Gradient

Sushant Kumar*
kumars12@rpi.edu
MSE, RPI
Troy, New York

Narendra Nanal†
nanaln@rpi.edu
MANE, RPI
Troy, New York

Vignesh Vittal-Srinivasaragavan‡
vittav@rpi.edu
MANE, RPI
Troy, New York

Abstract

Numerical simulation of systems governed by partial differential equations ultimately leads to solving massive systems of linear equations. We present a parallel conjugate gradient solver which employs a memory efficient *Compressed Row Storage* data-structure. Two versions of the solver, a CPU only (MPI) and CPU-GPU (CUDA) are developed and compared in this study. The CPU only version of the solver exhibits speedup with increase in number of processors. The CPU-GPU solver although is faster, doesn't show any scaling which needs further investigation. But overall, we are able to demonstrate the effectiveness of parallel conjugate gradient solver which is a key to faster simulations.

Keywords: parallel linear solver, conjugate gradient algorithm, MPI programming, MPI I/O, CUDA reduction, CUDA-MPI

1 Introduction

Partial differential equations or PDEs can be used to describe phenomena such as heat diffusion, electrostatics, electrodynamics, fluid dynamics, elasticity, quantum mechanics etc. Tremendous efforts have been made over the years to solve theses equations numerically and simulate these phenomena. Finite difference, finite volume or finite element are some of the most popular methods. These methods discretize the domain into small parts and the governing equations are solved approximately. This leads to a system of linear equation. For large simulations, solving this system of equations is computationally most expensive operation. Parallel implementation of this operation can reduce the simulation time significantly. The objective of the project is to develop a parallel solver for system of linear equations using conjugate gradient algorithm and benchmark its performance for weak and strong scaling.

Dekker *et al.* [**?**] has conducted a survey of different parallel algorithms for systems of linear equations. Lof [**?**] has developed a parallel conjugate gradient solver based on shared memory architecture. There are multiple studies like [**?**], [**?**] which demonstrate parallel conjugate gradient solver on multiple GPU platform. Schneider *et al.* [**?**] outlined a strategy for parallel conjugate gradient method and investigated influence of number of processors and different floating point precision on the convergence. We have used similar approach in this study to create a parallel conjugate gradient solver.

In this study we have created a conjugate gradient solver in C programming language which employs a *Compressed Row Storage* (CRS) data-structure. The solver is parallelized using MPI and CUDA. We have considered a simple 1D heat equation problem which is solved using finite difference technique. By varying number of grid points we can get systems of linear equations of varying sizes. The performance of the solver is tested by running the only MPI version and hybrid CUDA/MPI version across multiple ranks. The rest of the report is organized as follows. In section 2 the conjugate gradient algorithm and the storage data structure is discussed. In section 3 the parallelization strategy for the solver is explained. Finally, in section 4 performance analysis of the solver is presented.

2 Conjugate Gradient Solver

Solving PDEs numerically using finite difference or finite element technique leads to large system of matrix with sparse, symmetric and positive definite coefficient matrix. Conjugate gradient method [17] is suitable for solving such systems. The conjugate gradient algorithm and the corresponding storage data structure is explained in this section.

2.1 Conjugate Gradient Method

Consider the following system of linear equations.

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

Here, $\mathbf{A}$ is a known matrix of size $n \times n$. It symmetric, positive definite (i.e. $\mathbf{x}^T\mathbf{Ax} > 0$ for all non-zero vector $\mathbf{x}$ in $\mathbf{R}^n$). Vector $\mathbf{b}$ is known while $\mathbf{x}$ is a solution vector. Conjugate gradient method treats this as an optimization problem and finds the solution $\mathbf{x}_*$ attractively. The objective function used in conjugate gradient method is as follows:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b} \tag{2}$$

Since matrix $\mathbf{A}$ is positive definite, the objective function given in the Eq. (2) has a unique minimum. Considering

Project, Parallel Computing, Spring '20

the symmetric nature of matrix, the gradient of the given objective function can be calculated as follows:

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} \tag{3}$$

Hence, optimum of the Eq. (2) is the solution for the given system of linear equations. Let $\mathbf{P} = \{\mathbf{p}_0, \mathbf{p}_1, ...\mathbf{p}_{(n-1)}\}$ be sequence of n linearly independent directions. As $\mathbf{P}$ forms basis in R^n, we can express the solution of the system $\mathbf{x}_*$ as follows:

$$\mathbf{x}_* = \mathbf{x}_0 + \sum_{i=0}^{n-1} \alpha_i \mathbf{p}_i \tag{4}$$

Here, $\mathbf{x}_0$ is the initial guess. This can be considered as finding the solution from an initial guess by taking steps in the sequence of n linearly independent directions given by P. The step size for each direction is denoted by α.
Given the initial guess $\mathbf{x}_0$, the gradient at this point is $\mathbf{A}\mathbf{x}_0 - \mathbf{b}$. According to steepest descent, the first step will be in the direction of negative gradient i.e. $\mathbf{p}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$. The new optimal location of $\mathbf{x}$ at any given $(k+1)$th step is given as:

$$\mathbf{x}_{(k+1)} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{5}$$

The conjugate gradient algorithm insists that directions $\mathbf{P}$ are mutually orthogonal. This can be achieved by enforcing following condition:

$$\mathbf{p}_k = \mathbf{r}_k - \sum_{i<k} \frac{\mathbf{p}_i^T \mathbf{A}\mathbf{r}_k}{\mathbf{p}_i^T \mathbf{A}\mathbf{p}_i} \mathbf{p}_i \tag{6}$$

Here, $\mathbf{r}_k$ is the residual for the kth step given as $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$. The optimal step size α_k is give as:

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k} \tag{7}$$

The resulting algorithm is as follows:

Algorithm 1 Conjugate Gradient Method

Initialize:
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
while $\mathbf{r}_{k+1} >$ tolerance **do**
  $\alpha_k := \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}$
  $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
  $\mathbf{r}_{k+1} := \mathbf{x}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
  $\beta_k := \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
  $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
  $k := k + 1$
Return: $\mathbf{x}_{k+1}$

2.2 Compressed Row Storage Data-structure

The coefficient matrix obtained from finite difference or finite element method is sparse in nature i.e. most of the entries in the matrix are zero. Number of non-zero entries in the sparse matrix varies from 4% for larger systems to 25% for smaller systems. In such cases storing all zero values can lead to excessive wastage of memory. This problem can be circumvented by using a special data-structure to store the matrix values. Compressed Row Storage (CRS) [4] is a very popular data-structure used to store the sparse matrices efficiently.
In CRS data-structure a sparse matrix $\mathbf{M}$ of size $n \times n$ is stored using 3 one dimensional arrays. The array which stores non-zero entries in the matrix is denoted as *m_vec* while their corresponding column indices are stored in an array called *col_index*. The third vector called here as *row_index* is used to extract row elements from *m_vec* vector. If the number of non-zero entries in the matrix is *nnz* then *m_vec* and *col_index* has size *nnz*. The size of *row_index* is $(nnz + 1)$ as one extra padding element is added for the first row. To extract ith row, first we calculate $row\_start = row\_index[i]$ and $row\_start = row\_index[i]$. The ith row elements are extracted by slicing *m_vec* starting from *row_start* and ending at *row_end*. The number of non-zero entries in ith row is $(row\_end - row\_start)$. This process is explained using a following example. Consider a 4×4 matrix:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

Using CRS data-structure matrix $\mathbf{M}$ is represented as follows:

$$m\_vec = \begin{bmatrix} 5 & 8 & 3 & 6 \end{bmatrix}$$
$$col\_index = \begin{bmatrix} 0 & 1 & 2 & 1 \end{bmatrix}$$
$$row\_index = \begin{bmatrix} 0 & 0 & 2 & 3 & 4 \end{bmatrix}$$

To extract elements in the first row, *row_start* and *row_end* are calculated as follows:

$$row\_start = row\_index[1] = 0$$
$$row\_end = row\_index[2] = 2$$

Using these indexes row elements are extracted as $m\_vec[0 : 2] = [5 \;\; 8]$ and the their corresponding column indices are $[0 \;\; 1]$.
The CRS data-structure does not make any assumptions about the sparsity pattern of the matrix. Buluc *et al.* [3] have demonstrated efficiency of matrix-vector multiplication operation using CRS data-structure in parallel environment. All the operations presented in the algorithm 2.1 are performed on the three arrays used in the data-structure.

3 Parallel Conjugate Gradient Algorithm

The conjugate gradient solver is implemented in parallel using both MPI and CUDA so that the solver can be run across multiple multiple processors and GPUs. In this section MPI and CUDA implementation of parallel algorithm is explained. We also discuss use of MPI I/O for reading and writing input/optupt files across different MPI ranks.

3.1 MPI Implementation

To implement the algorithm in parallel, the first step is to read data from an input file and distribute it across the processors. The obvious choice for data distribution is to divide the coefficient matrix row wise. The coefficient matrix is divided into blocks based on the number of processors and each processor will be allocated with a particular block of rows. As explained in Section 2.2, we use CRS data-structure to store the rows i.e. we only store non-zero elements in the row. Each processor will store the global row numbers allocated to it, non-zero elements in those rows and corresponding column indices. The matrix data-structure for an example matrix **M** (given in the Section 2.2) using two processors is illustrated in Fig. 1. The solution vector **x** is stored completely

Processor	Global Row Number	Non-zero Elements	Column Index
0	0	-	-
	1	5	0
		8	1
1	2	3	2
	3	6	1

Figure 1. Schematic representation of matrix data-structure for each processor

on each processors.
The most time consuming linear algebra operations in the conjugate gradient algorithm are as follows:

1. Matrix-vector multiplication
2. Scaled Addition of two vectors
3. Dot product of two vectors

Theses operations should be parallelized in an efficient way to get the maximum speedup. The rest of the algorithm can be used unchanged.

3.1.1 Matrix-vector Multiplication. Matrix-vector multiplication is used to calculate the residual in the conjugate gradient algorithm. Consider a matrix-vector multiplication $\mathbf{y} = \mathbf{Ax}$.

$$\begin{bmatrix} & \mathbf{A} & \end{bmatrix} \begin{Bmatrix} \mathbf{x} \end{Bmatrix} = \begin{Bmatrix} \mathbf{y} \end{Bmatrix}$$

The ith element in $\mathbf{y}$ can be expressed as:

$$y_i = \mathbf{A}_i \mathbf{x} = \sum_{k=1}^{n} \mathbf{A}_{i,k} \mathbf{x}_k \tag{8}$$

In CRS data-structure we only store non-zero elements and their corresponding column indices. Hence, the Eq. 8 can be expressed as:

$$y_i = \sum_{k=1}^{nnz} \mathbf{A}_{i,k} \mathbf{x}_{c_k} \tag{9}$$

Here, c_k represents the column index for kth non-zero element in ith row. After parallelization the matrix vector product will be computed on different ranks as

$$\begin{bmatrix} \begin{bmatrix} \mathbf{A}_1 \end{bmatrix} \\ \hline \begin{bmatrix} \mathbf{A}_2 \end{bmatrix} \\ \hline \begin{bmatrix} \mathbf{A}_3 \end{bmatrix} \end{bmatrix} \begin{Bmatrix} \mathbf{x} \end{Bmatrix} = \begin{Bmatrix} \begin{Bmatrix} \mathbf{y}_1 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{y}_2 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{y}_3 \end{Bmatrix} \end{Bmatrix}$$

In the above example, the serial matrix vector product is parallelized with 3 processors. Each processor handles a portion of the matrix and multiplies it with full vector **x** to obtain the portion of the result vector. After the computation is completed by all the processors, all the locally computed elements are sent to all other processors. Better efficiency is achieved if the rows are distributed evenly across the processors.

3.1.2 Vector Addition. Consider an addition of two vectors $\mathbf{x} + \mathbf{y} = \mathbf{z}$ i.e.

$$\begin{Bmatrix} \mathbf{x} \end{Bmatrix} + \begin{Bmatrix} \mathbf{y} \end{Bmatrix} = \begin{Bmatrix} \mathbf{z} \end{Bmatrix}$$

Parallelizing the elementwise operations such as this with multiple processors very straightforward

$$\begin{Bmatrix} \begin{Bmatrix} \mathbf{x}_1 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{x}_2 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{x}_3 \end{Bmatrix} \end{Bmatrix} + \begin{Bmatrix} \begin{Bmatrix} \mathbf{y}_1 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{y}_2 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{y}_3 \end{Bmatrix} \end{Bmatrix} = \begin{Bmatrix} \begin{Bmatrix} \mathbf{z}_1 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{z}_2 \end{Bmatrix} \\ \hline \begin{Bmatrix} \mathbf{z}_3 \end{Bmatrix} \end{Bmatrix}$$

Each processor will compute the sum of the specific set of rows which it operates on and will store the partial results in a local variable.

3.1.3 Dot Product. Dot product operation between two vectors is required for calculation of step size α. The dot product between two vectors of size n can be expressed as: $\mathbf{x}.\mathbf{y} = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i$.

$$\left\{ \mathbf{x} \right\} . \left\{ \mathbf{y} \right\} = sum \left\{ \mathbf{x} \circ \mathbf{y} \right\}$$

Both of the vectors are similarly distributed across the processors. Each processor calculates the partial dot product corresponding to its allocated elements as

$$\left\{ \begin{array}{c} \left\{ \mathbf{x}_1 \right\} \\ \hline \left\{ \mathbf{x}_2 \right\} \\ \hline \left\{ \mathbf{x}_3 \right\} \end{array} \right\} . \left\{ \begin{array}{c} \left\{ \mathbf{y}_1 \right\} \\ \hline \left\{ \mathbf{y}_2 \right\} \\ \hline \left\{ \mathbf{y}_3 \right\} \end{array} \right\} = sum \left\{ \begin{array}{c} sum \left\{ \mathbf{x}_1 \circ \mathbf{y}_1 \right\} \\ \hline sum \left\{ \mathbf{x}_2 \circ \mathbf{y}_2 \right\} \\ \hline sum \left\{ \mathbf{x}_3 \circ \mathbf{y}_3 \right\} \end{array} \right\}$$

The partial dot product for kth processor can be expressed as follows:

$$s_k = \sum_{i=f_k}^{l_k} \mathbf{x}_i \mathbf{y}_i \tag{10}$$

Here, f_k and l_k are respectively first and last indices of elements stored on kth processors. These partial dot products are sent to the master processor using MPI_Send and MPI_Rec operations. The final dot product for m processors is calculated on the mater processor as follows:

$$\mathbf{x}.\mathbf{y} = \sum_{k=1}^{m} s_k \tag{11}$$

Finally, every processor requires a copy of complete dot product value for further operations. This achieved by MPI broadcast operation (MPI_Bcast) where dot product value is sent to all the processors from master processor.

3.2 CUDA Implementation

Graphical Processing Unit (GPU) is a relatively new technology which has brought a paradigm shift in the field of high performance computing. It was originally developed for rendering graphics albeit over the years it has become indispensable for solving simulation problems hitherto considered computationally prohibitive. Here, we try to exploit the putative computational power of GPU programming for accelerating the performance of our conjugate gradient solver. The strength of GPUs predicates on the multitude of cores available for computation which empowers the user to run numerous threads concurrently without context switching. CUDA, which is a general purpose parallel computing platform and programming model developed by NVIDIA is commonly used for this purpose [14].

However, GPU programming is not apt for every computational problem and it needs to be carefully evaluated as to how one could exploit the potential of this tool. Lee et al. have debunked the myth of the hundred-fold improvement in the performance of GPUs over CPUs through their expository work [10]. It was shown that intelligent optimization techniques could boost the performance of CPUs and deliver results at a speed comparable to GPUs. Karunadasa & Ranasinghe [9] implemented the Stranssen algorithm for matrix multiplication and conjugate gradient for solving a system of linear equations using a combination of MPI and CUDA. MPI was used to deliver data to the GPUs which in turn performed the computations to yield the final result. While significant improvement was observed in the performance of the Stranssen algorithm, the conjugate gradient algorithm failed to show an noteworthy speedup. The authors attributed this to a lack of second level of parallelism in the algorithm. However, certain application specific CUDA implementations of the conjugate gradient solver have yielded promising results. Maringnati et al [12] employed GPU programming for circuit simulation using a preconditioned conjugate gradient algorithm and found a speedup of upto 10 times relative to single-threaded CPU implementation. Zhang et al. [19] observed similar speedups for their implementation of a CUDA polynomial preconditioned conjugate gradient for elasticity related FEM problems. CUDA implementation for fluid flow problems has also bolstered the efficacy of the algorithm [1, 8]. Favorable performance in variegated areas [6, 11, 13, 15, 16] has made the technique a hugely popular tool for numerical computing.

In contrast to the aforementioned CUDA implementations of conjugate gradient algorithm which make use of sophisticated tricks, we employ simpler techniques like reduction and seek to enhance the performance of the conjugate gradient algorithm used here. In the following section, we describe the algorithms we used for the CUDA implementation.

3.2.1 Matrix-vector implementation. As explained in previous sections, we use the CSR data structure for representing the sparse matrices. Several pedagogical works have expounded the usage of CUDA for sparse matrix-vector product (SpMV) computation using this representation [2, 5, 18]. Figure 3 illustrates how work is distributed for GPU computation of SpMV. This implementation functions at the level of rows per thread. As the kernel is launched, each thread computes the product corresponding to a single row of the sparse matrix. We use MPI to allocate a chunk of the full sparse matrix to a rank which is subsequently processed by the GPU threads. As a common practice in CUDA programming to make the code more portable, readable and scalable, one can also use grid-stride loops for parallelizing the loop using the kernel [7]. Schematically shown in Figure

3, this becomes imperative for solving problems on hardware where the thread grid is not large enough to cover the input array in entirety.

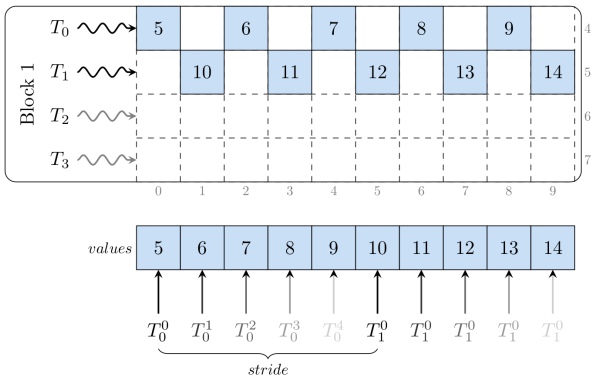


Figure 2. Work distribution for SpMV implementation using CUDA [5]

Nonetheless, issues of load balancing and thread divergence beset this simple and easy-to-understand algorithm for SpMV [5]. The pattern of memory access which needlessly processes threads with zeros impinges upon the efficacy of this method. The performance of our code manifestly suffers due to this inefficiency and more sophisticated techniques like CSR-Vector reduction, CSR-Stream and CSR-Adaptive implementations are needed to observe significant speedups [5].

3.2.2 Vector dot product. The algorithm, as implemented, needs to compute the dot product of multiple pairs of vectors on any given MPI rank. The concomitant computational load engenders the need to use GPU-enhanced processing of the dot product. A cursory glance at the problem suggests that CUDA is aptly suited for this job wherein each thread computes the product of the corresponding components of the vectors in parallel. However, calculating a vector dot product entails the reduction from vectors to a scalar.

$$c = \vec{a} \cdot \vec{b} \tag{12}$$

$$= (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3) \tag{13}$$

$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{14}$$

To calculate the sum in the last equation, one could use different approaches. A rather straightforward implementation involves parallel pairwise multiplication of array elements followed by serial addition. The so-called "atomic" operations corral the products of elements computed by each thread and sum them up to produce a scalar value. This operation is analogous `MPI_Sum` and `MPI_Reduce`. In one of our implementations, we explored the use of `atomicAdd()` for this purpose. A caveat is the necessity of using `__()syncthreads` to avoid any inadvertent race conditions. However, this approach, as discussed later, turns out to be relatively inefficient.

We use reduction technique, a tree-based approach to accelerate dot product computation. As shown in Figure **??**, this approach uses multiple thread blocks and sums up partial results sequentially, yielding significant gain in run time.

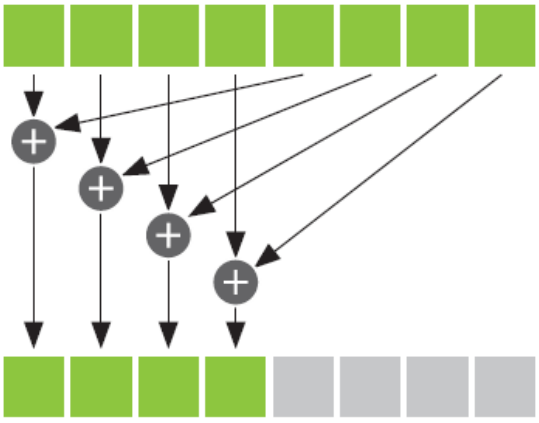


Figure 3. Vector dot product: A step of summation reduction using CUDA [5]

3.3 MPI I/O

MPI I/O modules are optimally implemented in the algorithm at three instances – loading all partitioned input data in the right processor, load the full update vector on to each rank (after it is partially updated by each rank) and to write the partial final results from each rank to a common output file.

3.3.1 Reading Partitioned Inputs. The following binary files are required as inputs to solve the system of equations

1. `Mvec` – File which stores the sparse matrix as a vector
2. `rowp` – File storing the auxiliary array pointing to the index of first non-zero element of each row in `Mvec`
3. `colm` – File storing the column index of all non-zero entries
4. `rhs` – File storing the full RHS vector (i.e. **b** in **Ax** = **b**)

All the processors will have its own copy of the full RHS vector. The other 3 vectors are partitioned and read into each processor to ensure that values corresponding to the set of rows assigned to a processor are only read.

The routine `MPI_File_read_at()` is used to ensure that each process reads the right chunk of data from the input files and load them in relevant variables.

3.3.2 Loading Update Vector. The algorithm can be realized by using only one full-vector (i.e. a vector that is stored in its entirety on each rank). The update vector $\mathbf{p_k}$ in our programme is the only full-vector as it is the only one involved in the matrix-vector product. On each processor, the

partial vector result of the partial matrix full vector product is computed as in fig 4

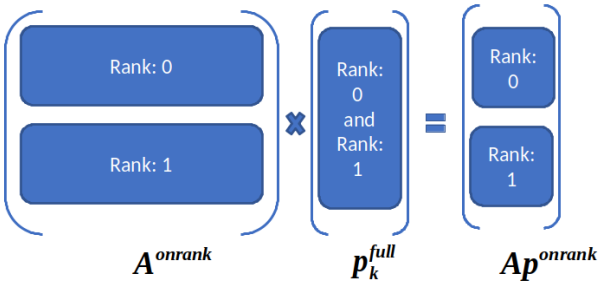


Figure 4. Matrix-vector product on multiple ranks

The subsequent operations operations to get the new update vector $\mathbf{p_{k+1}}$ are done using the scaled vector addition and dot product kernels. The new update vector, however is stored partially on each rank based on the calculations from that rank.

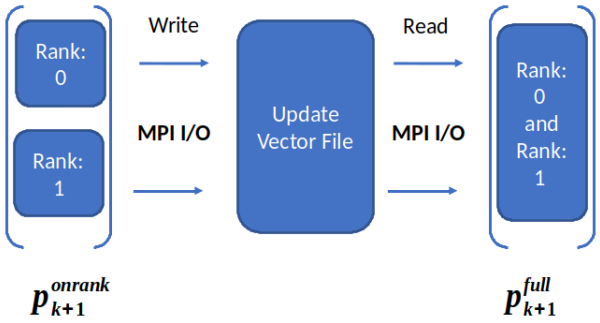


Figure 5. Partial update vector on each rank to full update vector using MPI I/O

`MPI_File_write_at()` routine is used to write the partial update vector on each rank to a common file `update_vector`. After all ranks finish writing the update vector to the file, using `MPI_File_read_all()` the full update vector is read into a variable. Therefore, all ranks will have a copy of the full update vector for the matrix vector multiplication in the next iteration.

3.3.3 Writing Final Result. The solution to given linear system i.e. the result vector is also stored in each processor partially. The results were written to output file(s) in two ways –

1. To one common result file using `MPI_File_write_at()` ensuring each processor writes the partial result in the right location
2. To multiple result files (one from each rank) using `MPI_File_write_all()`

4 Performance Analysis

The performance of parallel conjugate gradient solver is tested on *AiMOS* supercomputer for different systems with varying sizes. We tested two versions of the solver, the CPU only version and hybrid CPU-GPU version. Both strong and weak scaling analysis are conducted. These test results are discussed in this section.

4.1 Problem Definition

Numerical analysis of steady 1D heat equation is considered for the performance analysis of the parallel conjugate gradient solver. Steady state heat equation in 1D is given as:

$$-\kappa \frac{\partial^2 T}{\partial x^2} = f(x) \tag{15}$$

Here, T is the temperature, κ is thermal conductivity and $f(x)$ is the source term. Numerical analysis of Eq. (15) involves discretizing the domain into small segments and solving the equation using finite difference technique. This ultimately leads to solving a system of linear equations. The coefficient matrix obtained here is a sparse matrix with tri-diagonal banded structure. Based on number of discretized segments, systems with varying sizes are obtained for performance analysis.

4.2 MPI Performance

In this subsection performance analysis of MPI-only version of the solver is discusses. Strong scaling study is conducted on 7 different systems of size $1K \times 1K$, $2.5K \times 2.5K$, $5K \times 5K$, $10K \times 10K$, $20K \times 20K$, $50K \times 50K$ and $100K \times 100K$. The number of MPI ranks is varied from 1-6 and 12.

Plots for run-time analysis of strong scaling study are displayed in Fig. 6. In Fig. 7, run-time analysis for all the cases is displayed. It can be observed that for size $1K \times 1K$, $2.5K.5K$ and $5K \times 5K$, there is a nominal speedup by using 2 ranks. But as we increase the ranks from 2 the run-time is increasing. For the larger systems we can observe the speedup with increase in number of ranks from 1-6. But for 12 ranks all the systems show different degrees of slowdown.

To get better understanding of these results, we did performance analysis of the solver using an application *GProf*. The results for system of size $50K \times 50K$ for 1, 2 and 12 ranks are shown in Table 1, 2, 3 respectively. The operations which take maximum time are matrix-vector multiplication, vector-vector addition and vector-vector dot product. As we increase the ranks from 1-2 and eventually to 12, the speedup in all three operations can be observed. But all the ranks need synchronization before moving to the next iteration. If each rank handles relatively small number of rows, the idle time will increase because of synchronization. Hence using more than 2 ranks in only justified for large systems.

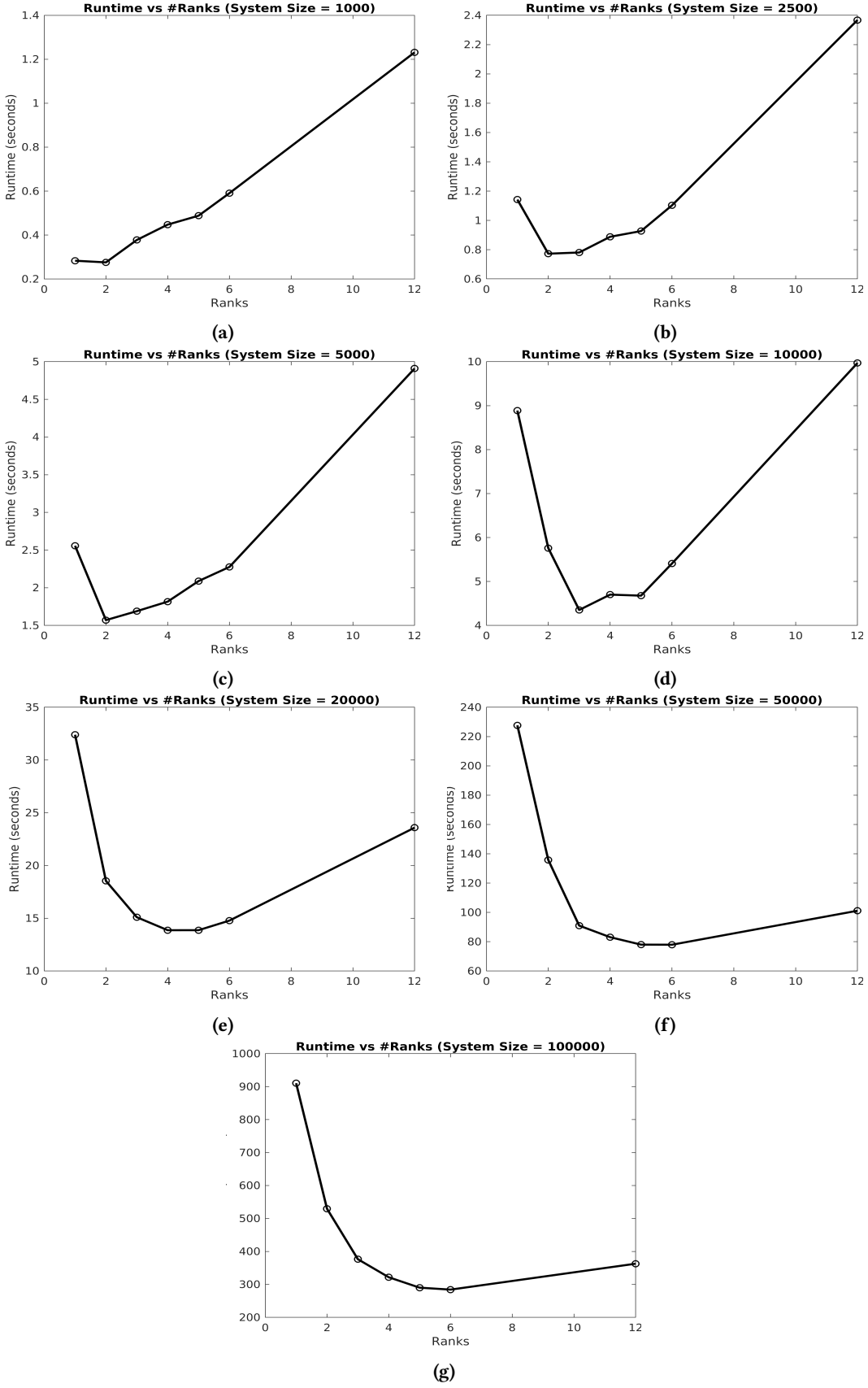


Figure 6. Strong scaling MPI-only solver

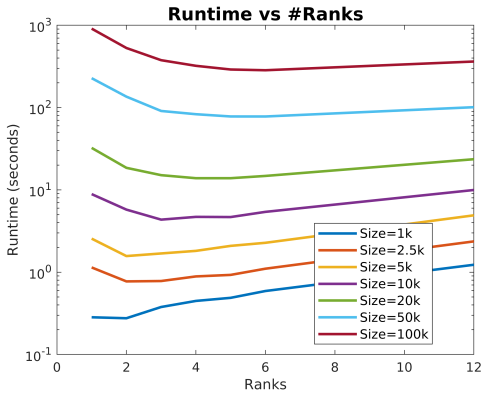


Figure 7. Strong Scaling MPI-only solver: run-time vs. no. of ranks for all sizes

Table 1. Profiling for 50k system size with 1 rank

% time	Cumulative seconds	self seconds	self calls	Function
44.68	92.54	92.54	50010	matrix_vector_product_onrank
27.54	149.58	57.04	150027	vector_scaled_addition_onrank
26.80	205.08	55.50	150027	vector_dot_product_onrank

Table 2. Profiling for 50k system size with 2 ranks

% time	Cumulative seconds	self seconds	self calls	Function
47.01	48.63	48.63	50009	matrix_vector_product_onrank
26.44	75.98	27.35	150024	vector_scaled_addition_onrank
25.76	102.63	26.65	150024	vector_dot_product_onrank

Table 3. Profiling for 50k system size with 12 ranks

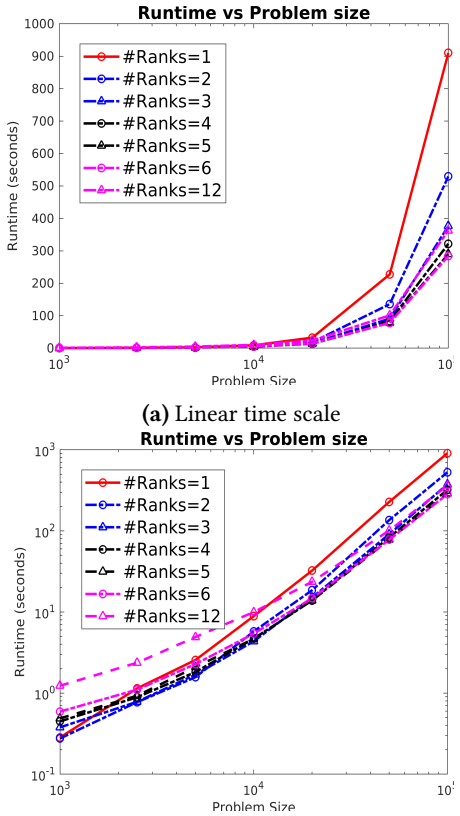
% time	Cumulative seconds	self seconds	self calls	Function
42.51	7.36	7.36	50006	matrix_vector_product_onrank
30.63	12.67	5.30	150015	vector_scaled_addition_onrank
25.80	17.14	4.47	150015	vector_dot_product_onrank

The weak scaling study analyzes speedup with number of processors for fixed problem size per processor. This type of study is not possible on our solver as it would lead to an invalid system (number of rows and columns different) of equation. Here, weak scaling is run-time analysis vs. size of the problem by keeping number of processors constant. There results for different number of ranks are displayed in Fig. 8.

Weak scaling results from Fig. 8 show a greater % increase in the speedup with increase in number of ranks for larger systems.

4.3 MPI-CUDA Performance

Similar test cases as Section 4.2 are used to analyze the performance of the hybrid MPI-CUDA version of the solver. The



(a) Linear time scale

(b) Log time scale

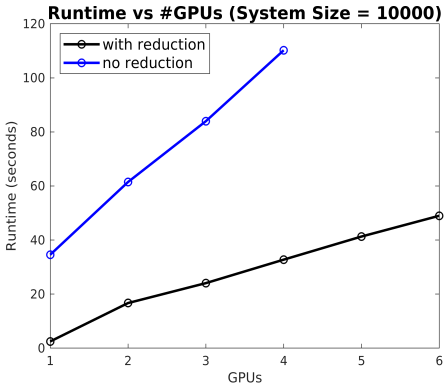
Figure 8. Weak scaling MPI-only solver: run-time vs. problem size for all ranks

results for both versions are also compared in this section. As explained in the Section 3.2.2, we are employing a CUDA reduction technique for vector-dot product. We first analyze the performance gain obtained by using the reduction technique.

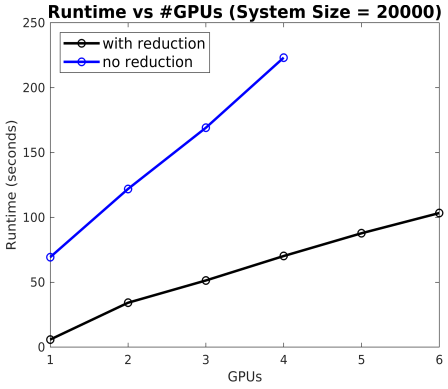
Fig. 9 illustrates that the run-time significantly decreases after applying CUDA reduction technique for vector-dot product. All the subsequent MPI-CUDA results presented are with the reduction technique.

Strong scaling studies for different systems are done on the hybrid solver. The run-time analysis vs. number of GPUs for all systems is presented in Fig. 10. Also we compare theses results with the MPI-only solver's performance. These comparison for system size $50K \times 50K$ and $100K \times 100K$ are shown in Fig. 11.

Fig. 11 clearly illustrates that for small number of ranks, the MPI-CUDA solver is much faster than MPI-only solver. But the MPI-CUDA solver doesn't exhibit any scaling. On the contrary run-time increases with number of GPUs. We conducted performance analysis on a $20K \times 20K$ using application *NVProf*. The performance analysis results for GPUs 1,2 and 6 are presented in Tables 4, 5 and 6 respectively.



(a)



(b)

Figure 9. Run-time comparison after applying reduction technique

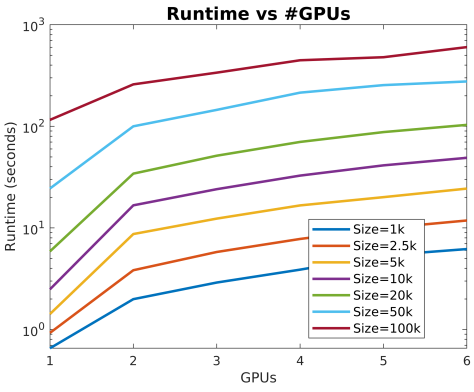
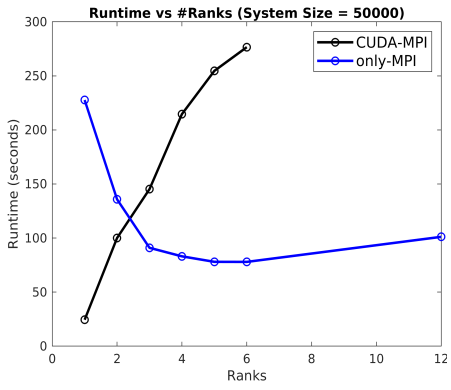


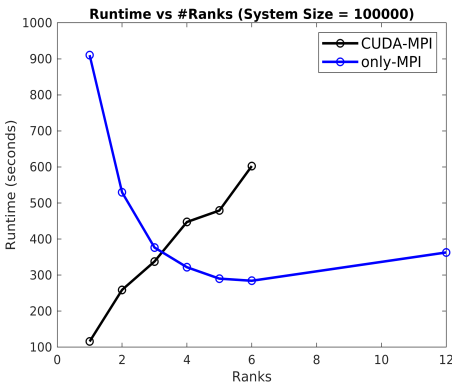
Figure 10. Strong scaling MPI-CUDA solver: runt-time vs. no. of GPUs for all sizes

Table 4. Profiling for 20k system size with 1 GPU

Rank	% time	self seconds	self calls	Kernel
0	62.34	1.93	60003	dot_product
	28.66	0.89	60003	vector_addition
	9.00	0.28	20002	matrix_vector_multiplication



(a)



(b)

Figure 11. Strong scaling: MPI-CUDA vs. MPI only comparison

Table 5. Profiling for 20k system size with 2 GPU

Rank	% time	self seconds	self calls	Kernel
0	63.43	3.71	20002	matrix_vector_multiplication
	27.69	1.62	60003	dot_product
	8.88	0.52	60003	vector_addition
1	61.25	3.71	20002	matrix_vector_multiplication
	27.73	1.68	60003	dot_product
	11.02	0.67	60003	vector_addition

Table 6. Profiling for 20k system size with 6 GPU

Rank	% time	self seconds	self calls	Kernel
0	72.84	5.01	20002	matrix_vector_multiplication
	21.15	1.45	60003	dot_product
	6.01	0.41	60003	vector_addition
2	67.13	3.76	20002	matrix_vector_multiplication
	26.05	1.46	60003	dot_product
	6.82	0.38	60003	vector_addition
5	51.70	2.28	20002	matrix_vector_multiplication
	35.93	1.58	60003	dot_product
	12.37	0.54	60003	vector_addition

For the serial execution (Table 4) of MPI-CUDA solver, matrix-vector multiplication only takes 9% of the run-time.

But for more than 1 GPU it is the most expensive operation. This indicates that computing resources are wasted on the synchronization causing parallel operations be more expensive. Further is needed to understand the cause for bad scaling performance of the hybrid solver.

We also performed weak scaling analysis keeping number of GPUs constant while changing the problem size. These analysis for different ranks is illustrated in Fig. 12. The comparison of weak scaling between MPI-CUDA solver and MPI-only solver for ranks 1, 2, 4 and 6 is presented in the Fig. 13.

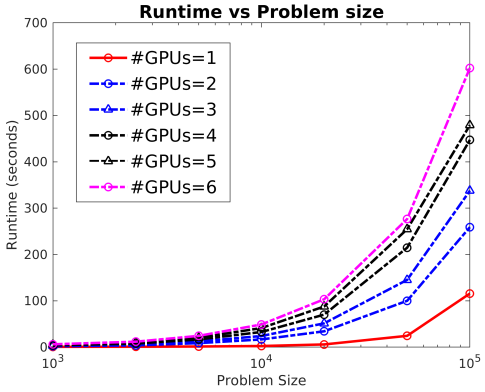


Figure 12. Weak Scaling MPI-CUDA solver: run-time vs. problem size for all ranks

It can be observed in Fig. 13 that for small number of ranks MPI-CUDA solver is much faster than MPI-only solver. But as we increase number of rank MPI-only solver is faster due to inefficient scaling of MPI-CUDA solver.

We have separately tested MPI-only solver and MPI-CUDA solver on different problems. In summary, the MPI solver shows good scaling for larger problems only. For smaller systems, run-time is increasing with processors due to time consumed in synchronization. Overall, MPI-CUDA solver performs better than MPI-only solver for small number of ranks. But it doesn't exhibit any scaling with more processors which needs further investigation.

5 Summary

In the scope of this project we have developed a parallel solver for solving large systems of linear equation. The solver is based on conjugate gradient algorithm. A compresses row storage data-structure is used to store large sparse matrices efficiently.

Two versions of the parallel solver are developed, a CPU only version using MPI and a hybrid CPU-GPU version using CUDA. Matrix vector multiplication and vector-vector dot product are the most critical operations in conjugate gradient algorithm. These operations can be performed in parallel by distributing the coefficient matrix row wise across different

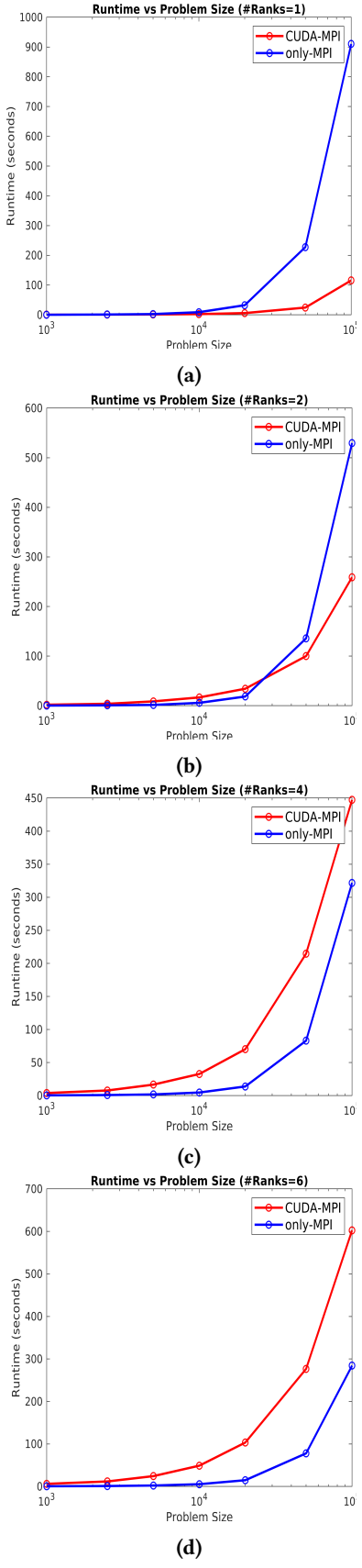


Figure 13. Weak scaling: MPI-CUDA vs. MPI-only comparison

processors. Operations like MPI broadcast are used to communicate the results between different processors. We also use MPI I/O to read partitioned input files and distribute the data across processors. The ability to write different block of solutions form different processors to a single file or multiple files is also demonstrated.

The performances of both MPI-only solver and MPI-CUDA solver are tested with systems of sizes $1K \times 1K$, $2.5K \times 2.5K$, $5K \times 5K$, $10K \times 10K$, $20K \times 20K$, $50K \times 50K$ and $100K \times 100K$. The number of processors is varied from 1-6 and 12. The MPI-only solver shows speedup with increase in number of processors for large systems only(size > 5K). The MPI-CUDA solver shows speedup associated with use of GPUs. But it doesn't show scaling as we increase number of processors.

Performance analysis applications like *NVProf* and *Gprof* are used to get insights about the behaviour of solvers. We found that with parallelization, all the operations are faster but significant amount of time is consumed by synchronization and communication between processors. This cost is only justified for larger systems.The performance of MPI-CUDA solver can be improved by employing efficient reduction technique. Additionally, more sophisticated techniques like CSR-Vector reduction, CSR-Stream and CSR-Adaptive for SpMV could boost the performance of the algorithm. In our implementation, we didn't use preconditioning which could significantly impact the efficacy of the method. It needs more investigation to find reasons for its poor scaling behaviour. Nonetheless, we are able to demonstrate the effectiveness of a parallel conjugate gradient solver to solve large systems of linear equations.

References

[1] G Amador and A Gomes. 2010. CUDA-based linear solvers for stable fluids. In *2010 International Conference on Information Science and Applications*. IEEE, 1–8.

[2] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

[3] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[4] Jack Dongarra. 1995. Compressed row storage (crs). *Survey of Sparse Matrix Storage Formats* (1995).

[5] Georgii Evtushenko. 2019 (accessed May 4, 2020). *Sparse Matrix-Vector Multiplication with CUDA*. https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f

[6] George A Gravvanis, CK Filelis-Papadopoulos, and Konstantinos M Giannoutakis. 2012. Solving finite difference linear systems on GPUs: CUDA based parallel explicit preconditioned biconjugate conjugate gradient type methods. *The Journal of Supercomputing* 61, 3 (2012), 590–604.

[7] Mark Harris. 2013 (accessed May 4, 2020). *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

[8] Xiaohui Ji, Tangpei Cheng, and Qun Wang. 2012. CUDA-based solver for large-scale groundwater flow simulation. *Engineering with Computers* 28, 1 (2012), 13–19.

[9] NP Karunadasa and DN Ranasinghe. 2009. Accelerating high performance applications with CUDA and MPI. In *2009 International Conference on Industrial and Information Systems (ICIIS)*. IEEE, 331–336.

[10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*. 451–460.

[11] Xue Li and Fangxing Li. 2014. GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method. *Electric Power Systems Research* 116 (2014), 87–93.

[12] Anirudh Maringanti, Viraj Athavale, and Sachin B Patkar. 2009. Acceleration of conjugate gradient method for circuit simulation using CUDA. In *2009 International Conference on High Performance Computing (HiPC)*. IEEE, 438–444.

[13] Dominik Michels. 2011. Sparse-matrix-CG-solver in CUDA. In *Proceedings of the 15th central European seminar on computer graphics*.

[14] CUDA Nvidia. 2011. Nvidia cuda c programming guide. *Nvidia Corporation* 120, 18 (2011), 8.

[15] G Oyarzun, R Borrell, A Gorobets, and A Oliva. 2014. MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids* 92 (2014), 244–252.

[16] Everett Phillips and Massimiliano Fatica. 2014. A CUDA implementation of the High Performance Conjugate Gradient benchmark. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 68–84.

[17] Jonathan Richard Shewchuk et al. 1994. An introduction to the conjugate gradient method without the agonizing pain. (1994).

[18] Zhuowei Wang, Xianbin Xu, Wuqing Zhao, Yuping Zhang, and Shuibing He. 2010. Optimizing sparse matrix-vector multiplication on CUDA. In *2010 2nd International Conference on Education Technology and Computer*, Vol. 4. IEEE, V4–109.

[19] Jianfei Zhang and Lei Zhang. 2013. Efficient CUDA polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. *Mathematical Problems in Engineering* 2013 (2013).