2017 IEEE 24th International Conference on High Performance Computing (HiPC)

Parallelizing Hines Matrix Solver in Neuron Simulations on GPU

Dharma Teja Vooturi, Kishore Kothapalli
International Institute Of Information Technology - Hyderabad
Hyderabad, India
dharmateja.vooturi@research.iiit.ac.in,kkishore@iiit.ac.in

Upinder S Bhalla
National Center for Biological Sciences
Tata Institute of Fundamental Research
Bangalore, India
bhalla@ncbs.res.in

***Abstract*—Hines matrices arise in the simulations of mathematical models describing initiation and propagation of action potentials in a neuron. In this work, we exploit the structural properties of Hines matrices and design a scalable, linear work, recursive parallel algorithm for solving a system of linear equations where the underlying matrix is a Hines matrix, using the Exact Domain Decomposition Method (EDD). We give a general form for representing a Hines matrix and use the general form to prove that the intermediate matrix obtained via the EDD has the same structural properties as that of a Hines matrix.**

Using the above observation, we propose a novel decomposition strategy called fine decomposition which is suitable for a GPU architecture. Our algorithmic approach R-FINE-TPT based on fine decomposition outperforms the previously known approach in all the cases and gives a speedup of 2.5x on average for a variety of input neuron morphologies. We further perform experiments to understand the behaviour of R-FINE-TPT approach and show its robustness. We also employ a machine learning technique called linear regression to effectively guide recursion in our algorithm.

I. Introduction

Sparse matrices and computations on sparse matrices arise in many areas of science and engineering such as computational fluid dynamics, computational neuroscience and molecular dynamics [1]. Prominent among the computations on sparse matrices include matrix vector multiplication, matrix matrix multiplication, and solving a system of linear equations where the underlying matrix of coefficients is sparse. The importance of these computations can be gauged by the fact that these computations are included as dwarfs in the Berkeley report [2]. It is therefore not surprising that most modern libraries in the parallel setting include optimized routines for the above computations on sparse matrices [3], [4].

Several researchers have focused on improving the performance of sparse matrix computations on a variety of modern many- and multi-core architectures. Promient examples include [5], [6] and [7]. Wangdong et al. [5] and Matam et al. [6] provide efficient algorithms for sparse matrix vector multiplication and sparse matrix matrix multiplication respectively on hybrid (CPU+GPU) architectures. Agullo et al. [7] optimize direct matrix solvers for Intel KNL architectures.

In recent years, one approach that is being used to improve the efficiency of sparse matrix computations on modern parallel architectures is to understand the strucure of sparsity of the matrix and its implications to parallel algorithm design and implementation. Examples of such instances are seen in the work of Ramamoorthy et al. [8] for multiplying two scale-free sparse matrices, Vooturi et al. [9] for multiplying two quasi-band sparse matrices, Buluc et al. [10] for multiplying two hyper-sparse matrices, and Wangdong et al. [11] for multiplying a quasi-band sparse matrix with a dense vector.

In this paper, we investigate GPU algorithms for solving a system of linear equations where the underlying matrix is a sparse matrix. In particular, the sparse matrix we study is called a Hines matrix that has the following sparsity structure. A Hines matrix is a symmetric matrix where every row of the matrix has only one nonzero element with a column index bigger than the row index. Solving a system of linear equations with the underlying matrix being a Hines matrix is denoted as `HinesSolver` in the rest of the paper.

The rest of the paper is organized as follows. In Section I-A, we give the motivation for the problem and in Section I-B, we discuss related work and list our key contributions in Section I-C. In Section II, we describe how a Hines matrix is generated from the mathematical model, its structural properties and a general form. In Section III, we describe a linear $O(N)$ algorithm for `HinesSolver` using an Exact Domain Decomposition method(EDD) and also discuss how to tailor our algorithm on a GPU. Results and experiments are discussed in Section IV. Finally, we conclude and outline future work in Section V.

A. Motivation

Neuron simulations happen in a time-step manner and a `HinesSolver` is invoked in each time-step. Generally, researchers have to run these simulations for many time-steps to understand a particular behaviour. For example, a single neuron simulation running for a neuron time of one minute with 1 milli second time-step involves 60,000 `HinesSolver` instances. In a given time-step, the computation apart from a `HinesSolver` is reasonably parallel and is suitable for GPU. By having `HinesSolver` on a GPU, simulations can be made faster by making use of GPU hardware and avoiding costly memory transfers from GPU to CPU at each time-step.

In case of network simulations, which involve the study of the behavior of interconnected neurons, multiple Hines matrix systems have to be solved at each time-step. It is possible to map the computation of solving multiple Hines matrix systems

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00051

into a single big Hines matrix system, which when solved gives solutions to the individual systems.

Further, it is not uncommon for researchers to run experiments which involves changing only a single parameter while keeping other parameters fixed. This types of experiments also result in solving multiple Hines matrices in each time-step. We discuss one such case in Section IV-D4, where for a set of experiments the matrix remains the same, but the right hand side vectors vary in a given time-step.

Hence, parallelizing `HinesSolver` on a GPU can speedup neuron simulations and also enable researchers to perform rapid experimentation.

B. Related Work

`HinesSolver` is studied in the sequential setting by Hines [12]. Hines [12] proposed a modified Gauss elimination algorithm whose runtime is $O(N)$, where N is the number of rows in the Hines matrix. Hines also proposed parallel Gauss elimination algorithms for `HinesSolver` in [15] and [16]. To understand these algorithms, it is helpful to visualize a Hines matrix as a rooted tree with self-loops. These algorithms are based on the fact that suitably selected subtrees can be processed in parallel. However, this approach suffers from the following drawbacks. Firstly, there is a constraint on subtree division which limits the amount of parallelism available. Secondly, they are designed for optimizing network simulations on multi-core architectures, where multiple Hines matrices of different sizes have to be solved and the ability to break a tree allows for efficient load balancing.

In `HinesSolver`, triangularizing the segments at the same level of a tree can be done in parallel. This idea was exploited by Roy et al. in [17] which is also based on Gauss elimination. One of the drawbacks of this approach is that the parallel time of the algorithm is bounded by the depth of the tree. Another drawback is that the amount of parallelism and computation at each level is dependent on the input and hence can introduce signifcant load imbalance.

Some of the above problems are solved by Mascagni [13], Larriba Pey [14] who introduced the Exact Domain Decomposition (EDD) technique to solve matrices corresponding to undirected graphs. EDD involves solving a domain matrix of size equal to the number of nodes with degree greater than two. In case of matrices corresponding to undirected graphs, the domain matrix does not exhibit any special properties and it was suggested to solve using any direct solver, such as Gauss elimination.

C. Contributions

In this work, we first start by proving that the domain matrix obtained via the exact domain decomposition method on a Hines matrix has the same structural properties as that of a Hines matrix. This result has three immediate benefits as listed below.

1) It allows us to apply the exact domain decomposition technique recursively.
2) As the recursion bottoms out, the small size of the resulting domain matrix allows us to invoke a sequential `HinesSolver` [12] in much less time.
3) It allows us to introduce a decomposition strategy called fine decomposition which can be efficiently mapped onto a GPU.

Using the above observations, we design an efficient parallel algorithm and its GPU implementation to solve a system of linear equations where the underlying matrix is a Hines matrix. Our experimental results on an Nvidia Tesla K40c GPU over a variety of inputs indicate that our algorithmic approach R-FINE-TPT based on fine decomposition is 2.5x faster than the previously known approach. We also conduct experiments to study the effect of parameters such as amount of fineness in R-FINE-TPT, depth of recursion, compartment resolution and number of right hand sides in the matrix system to show the robustness of our approach. We also employ a machine learning technique called linear regression to find a threshold function which helps in deciding when to stop the recursion in our algorithm.

II. Hines matrix

A. The Hodgkin-Huxley Model

The Hodgkin-Huxley model is a mathematical model proposed by Hodgkin and Huxley [19] to explain ionic mechanisms and voltage behaviour involved in the initiation and propagation of action potentials in neurons. A non-linear differential equation models how potential difference (V_m) changes with respect to ion-channels, current and other properties of a neuron. To simulate the model, a neuron is discretized spatially into multiple compartments as shown in Figure 1(a). The relationships between various compartments in a compartmentalized neuron can then be represented as a rooted tree as shown in Figure 1(b) where each node in the tree corresponds to a compartment. A node V_i in the tree has a unique parent compartment V_p and child compartments as shown in Figure 1(c). The tree is then numbered using a DFS numbering scheme from leaves to root. DFS numbering ensures two things.

1) The number of a node is larger than all its children and smaller than its parent.
2) The compartments in each branch of the neuron have consecutive numbers.

The current balance equation of i^{th} compartment at j^{th} timestep is then described according to Equation 1.

$$(V_i^j - V_i^{j-1}) \times \frac{C_i}{\Delta t} = (E_i - V_i^j)/Rm_i + (V_p^j - V_i^j) \times Ga_{i,p} + \sum_{k=IonChannels(i)} G_{i,k}^j \times (E_{i,k} - V_i^j) + Iext_i + \sum_{c=children(V_i)} (V_c^j - V_i^j) \times Ga_{i,c} \quad (1)$$

where V_i^j and $G_{i,k}^j$ represent voltage and conductance of ion channel k respectively for compartment i at time step j.

The constants $Iext_i$, C_i, E_i, Rm_i, $E_{i,k}$ represent external current, membrane capacitance, membrane resting potential, membrane resistance and reverse potential of ion channel k respectively for compartment i. Ga_{t_1,t_2} represents radial conductance between compartments t_1 and t_2. Δt is the time interval between two time steps. The only unknowns in Equation 1 are V_i^j V_p^j and $\{V_c^j | c \in children(V_i)\}$. By isolating them, Equation 1 can be written in a concise manner as shown in Equation 2. For more details refer [20].

$$A_{i,p}^j V_p^j + A_{i,i}^j V_i^j + \sum_{c=children(V_i)} A_{i,c}^j V_c^j = b_i^j \qquad (2)$$

where

$$A_{i,i}^j = \left(\frac{C_i}{\Delta t} + \frac{1}{Rm_i} + \sum_{r=neigh(i)} Ga_{i,r} + \sum_{k=IonChannels(i)} G_{i,k}^j \right)$$

$$A_{i,p}^j = Ga_{i,p}, \forall_{c \in children(V_i)} A_{i,c}^j = Ga_{i,c}$$

$$b_i^j = \left(Iext_i + \frac{E_i}{Rm_i} + V_i^{j-1} \times \frac{C_i}{\Delta t} + \sum_{k=IonChannels(i)} G_{i,k}^j \times E_k \right)$$

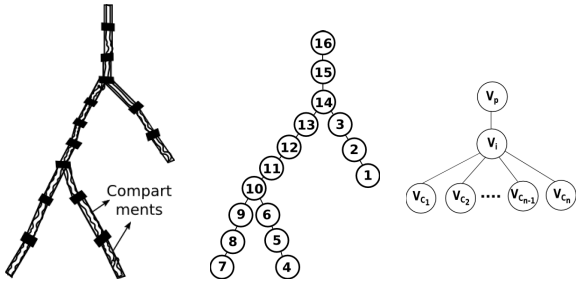
V_i^j V_p^j and $\{V_c^j | c \in children(V_i)\}$ are the voltages of i^{th} compartment, parent compartment of V_i and child compartments of V_i respectively at j^{th} time step and $A_{i,i}^j$,$A_{i,p}^j$ and $\{A_{i,c}^j | c \in children(V_i)\}$ are the corresponding coefficients. The current balance equations of all compartments can then be represented in a matrix form $A\vec{x} = \vec{b}$, where $\vec{x} = [V_1^j \cdots V_N^j]^T$ and $\vec{b} = [b_1 \cdots b_N]^T$. Solving this linear system gives the voltage values for compartments in each time-step. Figure 2 corresponds to the structure of the matrix formed by the current balance equations of compartmentalized neuron in Figure 1(b). The matrices obtained from the voltage PDE simulations fall under a class of matrices called Hines matrices. A Hines matrix has the following structural properties:

1) The matrix is symmetric. $[A_{ij} = A_{ji}]$
2) In each row i, there exists only one nonzero element with column index j such that $j > i$. $[\exists! j | (A_{i,j} \neq 0$ and $j > i)]$

From Equation 2, we can see that the off-diagonal elements of A have contributions only from the radial conductance $Ga_{t1,t2}$. As the radial conductance between any two compartments $t1$ and $t2$ is the same irrespective of order i.e, $Ga_{t1,t2} = Ga_{t2,t1}$, the matrix A is symmetric. The only non-zero element in row i after $A_{i,i}$ corresponds to the coefficient of parent compartment of V_i.

B. A General Form

A Hines matrix A can be represented in a general form along with some conditions. Let R be the set of compartments with more than one children and junction set J be a superset of R. Dividing the matrix at rows J and columns J results in a grid G of dimensions $(S+1) \times (S+1)$, where $S = |J|$. Each main diagonal entry of G is a block diagonal matrix Tr_i with k_i blocks, with each block being a symmetric tridiagonal matrix. A non main-diagonal entry of G is a zero matrix O. A Hines matrix A can then be represented in the form of Equation 3. We however note that not all matrices which can be represented in the form of Equation 3 are Hines matrices. In Section II-C, we describe the conditions that the general form should satisfy for it to represent only Hines matrices. The notation used for describing general form is described in Table I.



(a) Compartmentalized neuron (b) Neuron tree (c) Parent and children of i^{th} compartment

Fig. 1: Multi compartment neuron modelling.

$A\vec{x} = \vec{b}$	Matrix system
$Tr_i(Tr_i^1 ....Tr_i^{k_i})$	Block diagonal matrix with k_i blocks
Tr_i^j	Symmetric Tridiagonal matrix
$\overrightarrow{C_{ij}}(\overrightarrow{C_{ij}^1} ... \overrightarrow{C_{ij}^{k_i}})$	Column vector $\overrightarrow{C_{ij}}$ split into k_i vectors.
J, J_i	Junction array J and i^{th} junction.
$\vec{x_i}$	Vector $\vec{x}[J_{i-1}+1{:}J_i{-}1]$
$\vec{b_i}$	Vector $\vec{b}[J_{i-1}+1{:}J_i{-}1]$
x_i,b_i	$\vec{x}[i]$, $\vec{b}[i]$
$A_{i,j}$	$A[i][j]$
$Parent[i]$	Column index of the only non-zero entry after $A_{i,i}$
$JunctionIndex[k]$	Index of junction k in junction array J

TABLE I: Notation used in general form, algorithms and proof.

$$A = \begin{bmatrix} Tr_1 & \vec{C}_{1,1} & O & .. & \vec{C}_{1,S} & O \\ \vec{C}_{1,1}^T & A_{J_1,J_1} & \vec{C}_{2,1}^T & .. & A_{J_1,J_S} & \vec{C}_{(S+1),1}^T \\ O & \vec{C}_{2,1} & Tr_2 & .. & \vec{C}_{2,S} & O \\ : & : & : & : & : & : \\ \vec{C}_{1,S}^T & A_{J_S,J_1} & \vec{C}_{2,S}^T & .. & A_{J_S,J_S} & \vec{C}_{S+1,S}^T \\ O & \vec{C}_{S+1,1} & O & .. & \vec{C}_{S+1,S} & Tr_{S+1} \end{bmatrix} \qquad (3)$$

$$Tr_i = \begin{bmatrix} Tr_i^1 & & & & \\ & Tr_i^2 & & & \\ & & . & & \\ & & & . & \\ & & & & Tr_i^{k_i} \end{bmatrix} \quad \overrightarrow{C_{i,j}} = \begin{bmatrix} \overrightarrow{C_{i,j}^1} \\ \overrightarrow{C_{i,j}^2} \\ . \\ . \\ \overrightarrow{C_{i,j}^{k_i}} \end{bmatrix} \qquad (4)$$

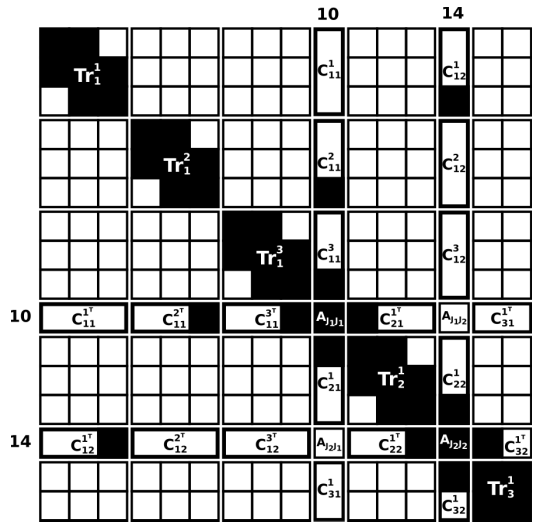


Fig. 2: The general form of Hines matrix corresponding to compartmentalized neuron in Figure 1(b) with J=[10,14].

$$\vec{x} = \begin{bmatrix} \vec{x}_1 \\ x_{J_1} \\ \vec{x}_2 \\ : \\ x_{J_S} \\ \vec{x}_{S+1} \end{bmatrix} \quad \vec{b} = \begin{bmatrix} \vec{b}_1 \\ b_{J_1} \\ \vec{b}_2 \\ : \\ b_{J_S} \\ \vec{b}_{S+1} \end{bmatrix} \quad [\vec{x_i}, \vec{b_i}] = \begin{bmatrix} \vec{x_i^1}, \vec{b_i^1} \\ \vec{x_i^2}, \vec{b_i^2} \\ : \\ \vec{x_i^{k_i}}, \vec{b_i^{k_i}} \end{bmatrix} \tag{5}$$

C. Necessary Conditions

Condition 1: *For a given Tr_i^r, where $1 \leq i \leq S+1$ and $1 \leq r \leq k_i$, only one column vector among $\{\overrightarrow{C_{ij}^r} | i \leq j \leq S\}$ is non-zero with only one non-zero element at the end.*

Each Tr_i^r is bounded by two rows *startRow* and *endRow* in matrix A. As Tr_i^r is a tridiagonal matrix, the matrix element to the right of each main diagonal element in Tr_i^r is in itself except for endRow. From general form, we know that $Parent[endRow]$ has to be in junction set J. As there is only one non-zero after each main diagonal element in a Hines matrix, the matrix element to the right of $A_{endRow,endRow}$ has to be in one of the column vectors $\{\overrightarrow{C_{ij}^r} | i \leq j \leq S\}$. As the non-zero column vector is also bounded by startRow and endRow, only the last element of that vector is non-zero. For the matrix in Figure 2, we can see that each tridiagonal matrix $(\overrightarrow{Tr_1^1}, \overrightarrow{Tr_1^2}, \overrightarrow{Tr_1^3}, \overrightarrow{Tr_2^1})$ has only one nonzero column vector $(\overrightarrow{C_{1,2}^1}, \overrightarrow{C_{1,1}^2}, \overrightarrow{C_{1,1}^3}, \overrightarrow{C_{2,2}^1})$ to its right.

Condition 2: *For a given junction J_i, $\overrightarrow{C_{i,i}^{k_i}} \neq \vec{0}$.*

Each tridiaonal matrix Tr_i^r corresponds to an unbranched segment in the tree. For a junction node J_i, $Tr_i^{k_i}$ corresponds to the segment which is numbered just before numbering junction node J_i. From a DFS numbering scheme, we can say that J_i=$Parent[endRow(\overrightarrow{Tr_i^{k_i}})]$. So, $\overrightarrow{C_{i,i}^{k_i}}$ becomes a non-zero vector. For the matrix in Figure 2, both the vectors corresponding to junctions i.e, $\overrightarrow{C_{1,1}^3}$ and $\overrightarrow{C_{2,2}^1}$ are non-zero.

Condition 3: *If $Parent[J_i] \in J$, then $(\overrightarrow{C_{i+1,i}^1})^T = \vec{0}$, else $(\overrightarrow{C_{i+1,i}^1})^T \neq \vec{0}$ and has exactly one non-zero at the first index.*

If $Parent[J_i] \notin J$, then according to DFS numbering system $Parent[J_i] = J_i + 1$. This means $A_{J_i,(J_i+1)} \neq 0$. As $A_{J_i,(J_i+1)}$ is the first element of $(\overrightarrow{C_{i+1,i}^1})^T$, $(\overrightarrow{C_{i+1,i}^1})^T[1] = A_{J_i,(J_i+1)}$. As there can be only one non-zero element to the right of A_{J_i,J_i}, all the elements of $(\overrightarrow{C_{i+1,i}^1})^T$ are zero except for the first one.

If $Parent[J_i] \in J$, then the row and column of the only non-zero element to the right of A_{J_i,J_i} belong to J. This means all the row vectors to the right of A_{J_i,J_i} are zero vectors which includes $(\overrightarrow{C_{i+1,i}^1})^T$.

For matrix in Figure 2, both $Parent[J_1]$ and $Parent[J_2]$ are not in J. Hence both $\overrightarrow{C_{2,1}^1}^T$ and $\overrightarrow{C_{3,2}^1}^T$ are non-zero.

Condition 4: *For a given junction row J_i, all row vectors after $(\overrightarrow{C_{i+1,i}^1})^T$ are zero vectors.*

Only one non-zero element exists after each main diagonal element in Hines matrix. From condition 3, we know that for a junction row J_i, it may only be part of row vector $(\overrightarrow{C_{i+1,i}^1})^T$. So the rest of the row vectors after $(\overrightarrow{C_{i+1,i}^1})^T$ are zero vectors.

III. Our Approach

A. EDD on an Undirected Graph

The Exact Domain Decomposition (EDD) method was first employed by Mascagni [13] to solve matrices corresponding to undirected graphs. The main idea is to create subdomains by breaking the graph at nodes with degree greater than two. In such a decomposition, each subdomain corresponds to a chain graph and the matrix of the subdomain corresponds to a tridiagonal matrix. These subdomains are solved independently and the solutions are fused together based on subdomain relationships to construct the final solution. Any undirected graph can be represented in general form described in Equation 3. Thus the EDD algorithm for a matrix that can be represented in general form can be described in Algorithm 1.

Algorithm 1 Domain decomposition method for a matrix in general form corresponding to an undirected graph.

1: $\forall_{i=1:S+1} \overrightarrow{R_i} = Tr_i | \vec{b}_i$
2: $\forall_{i=1:S+1} \forall_{j=1:S} \overrightarrow{P_{i,j}} = Tr_i | \overrightarrow{C_{i,j}}$
3: $\forall_{i=1:S} \forall_{j=1:S}$
$M[i][j] = (\sum_{l=1}^{l=S+1} \sum_{r=1}^{r=k_l} \overrightarrow{C_{l,i}}^T \times \overrightarrow{P_{j,l}}) - A_{J_i,J_j}$
4: $\forall_{i=1:S}\ M_{rhs}[i] = (\sum_{l=1}^{l=S+1} \overrightarrow{C_{l,i}}^T \times \overrightarrow{R_l}) - b_{J_i}$
5: $M_x = M | M_{rhs}$
6: $\forall_{i=1:S+1} \overrightarrow{x_i} = \overrightarrow{R_i} - \sum_{k=1}^{k=S} M_x[k] \times \overrightarrow{P_{i,k}}$

In Algorithm 1, we start by solving each Tr_i with multiple right hand sides $\forall_{1\leq j\leq S}\overrightarrow{C_{i,j}}$ and $\vec{b_i}$. Subsequently, the domain matrix M and its right hand side M_{rhs} are constructed using rows at junction indices of matrix A and the tridiagonal solutions computed. Solving the system $(M|M_{rhs})$ gives solutions M_x for compartments in the junction set. The solutions for non junction nodes are then computed using M_x and the tridiagonal solutions. The complexity of this algorithm is $O(N * S + S^3)$, where N is the size of the matrix and S is the number of junctions.

B. Domain Matrix in the Exact Domain Decomposition Method

In this section, we prove that when EDD is applied on Hines matrix, the domain matrix obtained has the same structural properties as that of a Hines matrix.

Theorem 1. *The domain matrix M has the structural properties of a Hines matrix.*

Proof: We prove it by showing that the domain matrix M satisfies the structural properties of a Hines matrix as described in Section II.

(i) The matrix is symmetric, $M_{i,j} = M_{j,i}$.
From Algorithm 1, we know that any element $M_{i,j}$ of the domain matrix M can be constructed as follows:

$$M_{i,j} = \left(\sum_{l=1}^{S+1} \overrightarrow{C_{l,i}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,j}}\right) - A_{J_i,J_j}$$

and

$$M_{j,i} = \left(\sum_{l=1}^{S+1} \overrightarrow{C_{l,j}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,i}}\right) - A_{J_j,J_i}$$

If R is a symmetric matrix of size $N \times N$ and p, q are column vectors of size N, then $p^T \times R \times q = q^T \times R \times p$. So for a valid l, i, and j, $(\overrightarrow{C_{l,i}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,j}}) = (\overrightarrow{C_{l,j}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,i}})$. As a Hines matrix is a symmetric matrix, $A_{J_i,J_j} = A_{J_j,J_i}$. Hence the matrix M is symmetric.

(ii) In each row i, there exists only one nonzero element with column index j such that $j > i$. $[\exists! j | (M_{i,j} \neq 0 and j > i)]$

For a given junction row J_i, the non-zero row vectors before and after $A_{J_iJ_i}$ can be divided into two sets S_{left} and S_{right} respectively. Because a Hines matrix is symmetric, S_{left} can only contribute to the main diagonal element $M_{i,i}$ of the domain matrix M. From the Conditions 3 and 4, we know that $S_{right} = \{\overrightarrow{C^1_{i+1,i}}^T\}$ or $\emptyset$. Each element in row i of M after main diagonal element $M_{i,i}$, $j > i$, can then be represented in the following cases.

Case 1: $S_{right} = \{\overrightarrow{C^1_{i+1,i}}^T\}$
In this case, we can see that:

$$M_{i,j} = \overrightarrow{C^1_{i+1,i}}^T \times (Tr^1_{i+1})^{-1} \times \overrightarrow{C^1_{i+1,j}}$$

From condition 1, we know that for Tr^1_{i+1}, there exists only one non-zero column vector to its right i.e, $\exists! j | (\overrightarrow{C^1_{i+1,j}} \neq \vec{0} \& j > i)$.

Case 2: $S_{right} = \emptyset$

We have that $M_{i,j} = -A_{J_i,J_j}$. From conditions 3 and 4, we know that this can happen only if $Parent[J_i] \in J$. As there is only one non-zero element after A_{J_i,J_i}, only one of the elements from set $\{A_{J_i,J_{i+1}}, A_{J_i,J_{i+2}}.....A_{J_i,J_S}\}$ is non-zero.

In both cases, for a given row i in the domain matrix M, there exists only one j such that $j > i$ and $M_{i,j} \neq 0$.

■

C. An O(N) Linear Algorithm for `HinesSolver`

In this section, we describe an $O(N)$ algorithm for `HinesSolver` using EDD, where N is the total number of compartments. If R is the set of compartments with more than one child, then a junction set J is a superset of R. Let S be the number of compartments in a junction set J. Our algorithm comprises of four stages.
1) Solve independent tri-diagonal systems.
2) Generate the domain matrix.
3) Solve the domain matrix system $(M|M_{rhs})$.
4) Construct the final solution $(\vec{x})$.

Analysis: In Steps 3,6,7, and 10 of Algorithm 2, independent tridiagonal systems are being solved. The cumulative size of the tridiagonal systems from Step-3 is bounded by N and that of Steps 6,7, and 10 is bounded by $2N$. So Stage-1 involves solving tridiagonal systems with cumulative size bounded by $3N$. As a tridiagonal system can be solved in linear time, the complexity of Stage-1 is $O(N)$. The complexity for computing both the main diagonal of M and right hand side M_{rhs} is $O(\sum_{i=1}^{i=S} n_i)$, where n_i is the number of neighbours of compartment J_i. It can be seen from Step-16 that the complexity for computing non-zero off diagonal element of M is $O(1)$. As $(\sum_{i=1}^{i=S} n_i < 2N)$ and the domain matrix M has only $2(S-1)$ non-zero off diagonal elements, the complexity of Stage-2 is $O(N + S)$. As a Hines matrix can be solved in linear time [12], the complexity for solving domain matrix in Stage-3 is $O(S)$. Stage-4 has the same loop structure as that of Stage-1. In places where a tri-diagonal system is solved, vector scaling and addition is performed. Hence the complexity of Stage-4 is $O(N)$. The complexity of Stages 1-4 are $O(N)$, $O(N+S)$, $O(S)$ and $O(N)$ respectively. As the number of junctions S cannot exceed N, the complexity of our algorithm is $O(N)$.

D. EDD for `HinesSolver` *on GPU*

In this section, we show how `HinesSolver` can be efficiently mapped onto a GPU architecture using EDD. In Section III-C, we showed that the complexity of our algorithm is $O(N)$, irrespective of the number of junction compartments. We use this fact and propose a decomposition strategy called

Algorithm 2 O(N) algorithm for `HinesSolver`using EDD.

#Stage-1 Solve tridiagonal systems.
for i=1:S **do**
  Q_i = $Tr^1_{i+1}|\overrightarrow{C^1_{i+1,i}}$
  for r=1:k_i **do**
    col = JunctionIndex[Parent[endRow(Tr^r_i)]]
    P^r_i = $Tr^r_i|\overrightarrow{C^r_{i,col}}$
    R^r_i = $Tr^r_i|\overrightarrow{b^r_i}$
  end for
end for
R^1_{S+1} = $Tr^1_{S+1}|\overrightarrow{b^1_{S+1}}$ // End case

#Stage-2 Constructing the domain matrix.
for i=1:S **do**
  $M[i][i]$ += $(C^1_{i+1,i}[1] \cdot Q_i[1] - A_{J_i,J_i})$
  col = JunctionIndex[Parent[endRow(Tr^1_{i+1})]]
  $M[i][col]$ = $(C^1_{i+1,i}[1] \cdot P^1_{i+1}[1] - A_{J_i,J_{col}})$
  $M_{rhs}[i]$ += $(C^1_{i+1,i}[1] \cdot R^1_{i+1}[1] - b_{J_i})$
  for r=1:k_i **do**
    col = JunctionIndex[Parent[endRow(Tr^r_i)]]
    $M[col][col]$ += $\overrightarrow{C^r_{i,col}}[|\overrightarrow{C^r_{i,col}}|] \cdot P^r_i[|\overrightarrow{C^r_{i,col}}|]$
    $M_{rhs}[col]$ += $\overrightarrow{C^r_{i,col}}[|\overrightarrow{C^r_{i,col}}|] \cdot R^r_i[|\overrightarrow{C^r_{i,col}}|]$
  end for
end for

#Stage-3 Find solutions at junctions
$M_x = M|M_{rhs}$

#Stage-4 Construct $\vec{x}$.
for i=1:S **do**
  $\overrightarrow{x^1_{i+1}}$ -= $M_x[i] \times Q_i$
  for r=1:k_i **do**
    col = JunctionIndex[Parent[endRow(Tr^r_i)]]
    $\overrightarrow{x^r_i}$ -= $M_x[col] \times P^r_i$
    $\overrightarrow{x^r_i}$ += R^r_i
  end for
end for

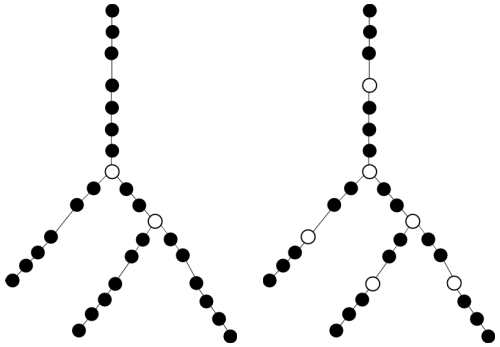
fine decomposition.

Minimal Decomposition(J_{md}): Compartments with more than one children are chosen as junctions.

Fine Decomposition(J_{fd}): The goal of this decomposition is to have equal size tridiagonal systems to solve in Stage-1. In order to achieve that, we break each branch of the tree into K sized chains and include last compartment of each chain in junction set J_{fd}. Apart from these compartments, J_{fd} includes all compartments which have more than one children i.e, junction set J_{md}.

An example of minimal decomposition and fine decomposition with $K = 4$ for a given tree are shown in Figures 3(a) and 3(b) respectively. Hollow nodes in Figures 3(a) and 3(b) correspond to junction compartments.

Algorithm 3 describes the recursive algorithm for `HinesSolver` using the Exact Domain Decomposition



(a) Minimal Decomposition (b) Fine Decomposition with K=4

Fig. 3: Decomposition strategies.

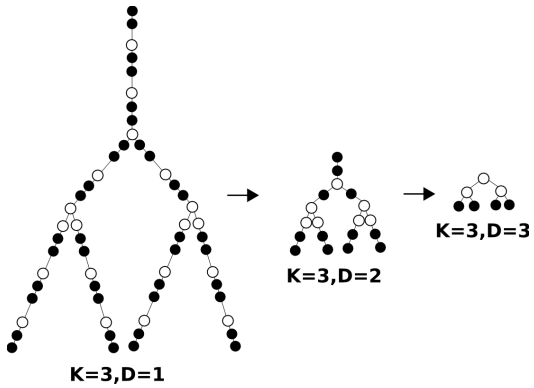


Fig. 4: Recursive application of fine decomposition with K=3

method.

1) Stage-1: In this stage, the computation involves solving many tridiagonal systems. This computation can be performed using two approaches **TRSV** and **TPT**.

TRSV: Making all junction rows and junctions columns of matrix A zero, except for main diagonal elements results in a tridiagonal matrix T. Figure 5 shows the tridiagonal matrix T obtained from the Hines matrix shown in Figure 2. From Stage-1 of Algorithm 2, we know that each tridiagonal Tr^r_i has to be solved with at least two and at most three right hand sides. We position them accordingly as shown in Figure 2 and use an optimized tridiagonal solver from NVIDIA's CUDA library CuSparse [4] to solve tridiagonal matrix T with three right hand sides.

TPT: In this approach, we map each thread to solve an independent tridiagonal system. In minimal decomposition, independent tridiagonal systems that have to be solved in Stage-1 are big and have lot of variance in size. So using TPT approach for solving tridiagonal systems suffers from less parallelism, higher load imbalance and more amount of work per thread. TRSV approach hides these to some extent and takes advantage of optimized library function for tridiagonal solver. This compatibility leads to MIN-TRSV approach.

In fine decomposition, each independent tridiagonal system

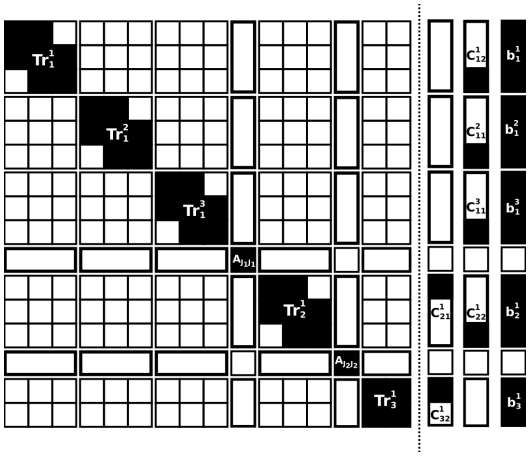


Fig. 5: Mapping Stage-1 computation to tridiagonal solver(TRSV) with three right hand sides in MIN-TRSV.

in Stage-1 is almost of same size and has equal compute. So using the TPT approach, one can take advantage of the SIMD architecture of a GPU. This compatibility leads to R-FINE-TPT approach.

2) Stage-2: In this stage, we construct the domain matrix system (M, M_{rhs}). As there is no dependency among the non-zero elements of the domain matrix system, they can be constructed in parallel.

3) Stage-3: In this stage, we have a choice to run the algorithm recursively to solve domain matrix system $(M|M_{rhs})$. In the R-FINE-TPT approach, we run the algorithm recursively and stop when using a GPU is no longer efficient. In case of MIN-TRSV, we do not run the algorithm recursively. The reason is that the tree corresponding to domain matrix when minimal decomposition is applied has no vertices of degree two. Applying any decomposition recursively on that tree will not reduce the size of domain matrix size significantly. So in the MIN-TRSV approach, the runtime is less when we do not recurse.

4) Stage-4: Stage-4 involves constructing final solution $\vec{x}$. As there is no dependency among elements of $\vec{x}$, each element of $\vec{x}$ can be constructed in parallel.

E. Implementation Details

Hines matrix A of size N is stored using two arrays, parent array P of size N and data array D of size $2N$. $P[i]$ stores the column index of the only nonzero after $A[i][i]$. As A is symmetric, we only store the nonzero values of upper triangular matrix in a row major fashion in D. By storing in the row major fashion, we get better memory coalescing while accessing a tridiagonal matrix Tr_i^j in Stage-1. We used buffer arrays to avoid replication of tridiagonal matrices in the computations of Stage-1. We employed NVIDIA's CUDA Occupancy Calculator tool to configure thread block sizes in CUDA kernels.

Algorithm 3 Recursive algorithm for `HinesSolver` using Exact Domain Decomposition method.

R-EDD(A, $\vec{b}$, decomposition, num-rhs)
{
if decomposition == MIN **then**
  Use minimal decomposition with TRSV approach to solve tridiagonal systems in Stage-1.
else if decomposition == FINE **then**
  Use fine decomposition with TPT approach to solve tridiagonal systems in Stage-1.
end if
Stage2: Construct domain matrix system (M, M_{rhs})
if decomposition == FINE and
Threshold(rows(M),num-rhs) == False **then**
  M_x = **R-EDD**(M, M_{rhs}, decomposition, num-rhs)
else
  Transfer (M,M_{rhs}) to CPU.
  Solve Domain system $M_x = M/M_{rhs}$ on CPU.
  Transfer M_x to GPU.
end if
Stage-4: Construct $\vec{x}$
return $\vec{x}$
}

IV. Results and Analysis

A. Platform

We use an Nvidia Tesla K40c GPU for all our experiments. It is mounted on an Intel i7-4790K CPU with 32GB RAM. The K40c has a total of 2880 cores organized in 15 SMx, with each core clocked at 745 MHz. It provides a peak double precision floating point performance of 1.43 Tflops and single precision floating point performance of 4.29 TFlops. Each SM also has a 64KB configurable cache to exploit data locality.

B. Dataset

All input Hines matrices come from neuron morphologies taken from `www.neuromorpho.org` [21]. We choose our dataset in such a way that they come from different parts of brain and has variation in size and the number of junctions. Some details of the chosen morphologies are shown in Table II. We group the dataset into three categories small (7K-11K), medium (29K-35K), and large (80K-120K) neurons based on size of the matrix.

C. Results

We compare our R-FINE-TPT approach with the MIN-TRSV approach which is based on minimal decomposition strategy suggested by Mascagni in [13]. These approaches differ in the decomposition strategy used for finding junctions and the computation strategy used to solve tridiagonal systems in Stage-1. All operations are carried out in double precision. From the results in Figure 6, it can be observed that R-FINE-TPT is faster than MIN-TRSV for all classes of input. It has to be noted that we achieve a speedup of 2x on EC5 neuron,

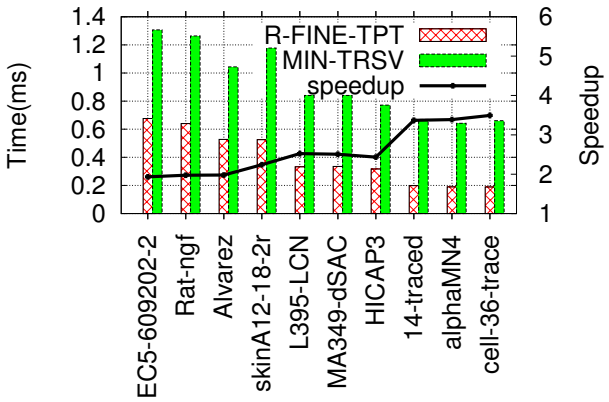


Fig. 6: Results on input dataset

Neuron	Compartments	Junctions	Branches
EC5-609202-2	123248	1321	2259
Rat-ngf-11-11-04	114193	468	941
Alvarez-Control-Cell-1	84254	185	370
skinA12-18-2r	82110	3287	6183
L395-LCN	35630	787	1554
MA349-dSAC	35190	1838	3483
HICAP3	29159	119	242
14-traced	11193	546	1042
alphaMN4	9231	69	151
cell-36-trace	7881	97	184

TABLE II: Details of neuron morphologies

which is the neuron with highest number of compartments than any other neuron in the repository [21].

The reason for the good performance of R-FINE-TPT over MIN-TRSV is that in R-FINE-TPT there is more parallelism, less work per thread, and negligible load imbalance. Whereas in case of MIN-TRSV, threads have to coordinate among themselves to solve one big tridiagonal system with three right hand sides. This requirement for coordination results in poor performance when compared to R-FINE-TPT.

D. Further experiments

In this section, we perform two sets of experiments. One set of expeirments study the impact parameters such as K in the fine decomposition, depth D in the recursion, and varying the number of right hand sides on the runtime of the algorithm. The second set of experiments are aimed at coming up with guidelines to choose appropriate values for K and D automatically based on empirical data. For some experiments we use linear neuron as our model. In linear neuron, there is only one branch and all the compartments have only one child.

1) Varying K in Fine Decomposition: In this experiment, we study how varying K in fine decomposition affects overall runtime and runtime of individual stages in R-FINE-TPT. To understand the behaviour, we take a linear neuron of size 100K as the input. For a linear neuron with N compartments, there are roughly N/K junctions and $3N/K$ tridiagonal systems of size $(K-1)$ to be solved in Stage-1. As we increase K, number of threads to be launched i.e, $3N/K$ decreases in Stage-1 and work per thread i.e, solving tridiagonal of size $(K-1)$ increases. Having few threads with more work is not good for GPU and it can be observed in run time of Stage-1 in Figure 7. As K increases, running time of Stage-1 increases. Stage-3 of EDD involves time for recursive call $T(N/K)$. It decreases with increase in K and it can be observed in Figure 7. Threads launched in Stage-2 and Stage-4 are very light and changing K has little impact on their runtime. The overall runtime decreases to a certain K and then increases. For our input linear neuron of size 100K the best performance is at $K = 3$.

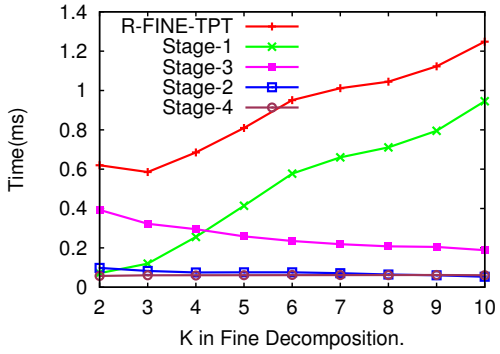


Fig. 7: Impact of varying K on R-FINE-TPT approach.

2) Varying the Depth of Recursion, D: In this experiment, we study the effect of recursion depth D on the R-FINE-TPT approach. From Figure 8, we can see that as we increase D, the speedup increases to a certain point and decreases from then on. This is due to the fact that at inflexion point it is better to solve the matrix on a CPU rather than running the algorithm recursively. For large neurons, the inflection point is at $D = 3$ and the average size of the domain matrix at $D = 3$ is 1800. For such small matrices, it is faster to solve it on a CPU despite the cost of memory transfers. As long as we have bigger matrices to solve at each level, it is beneficial to run the algorithm recursively.

3) Varying Resolution: In compartmental modelling, each branch of the neuron is divided into multiple compartments. More accurate simulations are possible by increasing the number of compartments into which a branch is divided. The morphology file contains a particular compartmentalization of a neuron. In this experiment, we obtain a P-resolution morphology by breaking an original compartment in the morphology into P compartments. If the input morphology has N compartments, P-resolution morphology contains $P \times N$ compartments. From Figure 9, we can see that R-FINE-TPT performs better than MIN-TRSV for all resolutions. The primary reason for this is that using fine decomposition enables us to have computation in Stage-1 divided in to many threads with very little work. This coupled with recursion is the reason for better performance compared to MIN-TRSV.

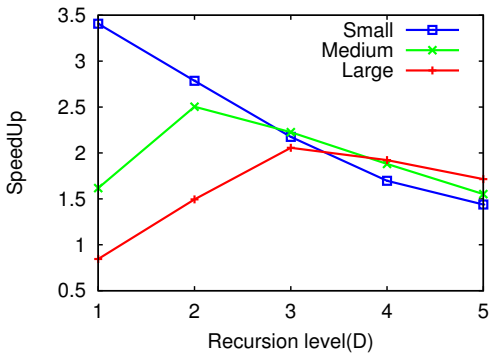


Fig. 8: Impact of varying recursion depth D on speedup.

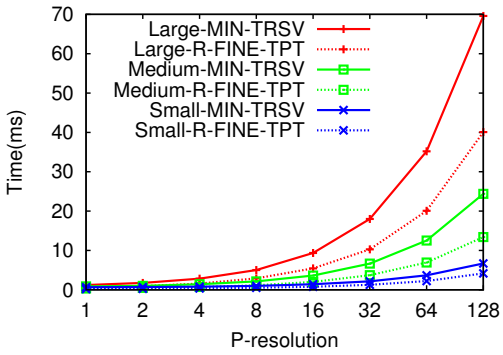


Fig. 9: Impact of varying resolution on R-FINE-TPT and MIN-TRSV approaches.

4) Varying Right Hand Sides: Voltage behaviour studies have a lot of parameters to tinker with. For example, in Equation 2, having different values of external compartment current ($Iext_i$) effects only right hand side of the matrix system. Now, it is possible to do multiple simulations for different values of $Iext_i$ at once. This is advantageous because it suffices to factorize the tridiagonal matrices in Stage-1 only once and use the factorizations for all right hand sides. In this experiment, we see how R-FINE-TPT behaves with change in the number of right hand sides. From Figure 10, it can be seen that speedup increases with respect to number of right hand sides for all classes of neurons.

5) Determining the Threshold Function: In this experiment, we find a boolean threshold function for deciding when to stop the recursion in Algorithm 3. The threshold function depends on two parameters: the size of the matrix, N, and the number of right hand sides, R. We have to break the recursion at a stage where using CPU is better than using GPU. So, we run an experiment to find out the largest matrix size at which CPU is better than GPU for each value of R. From Figure 11, we can see that the data is following $1/x$ behaviour. Hence we modeled the function as $(N = a_0/R + a_1)$ and used linear regression to find constants a_0 and a_1 at which the error is minimum. Threshold function thus obtained from the above

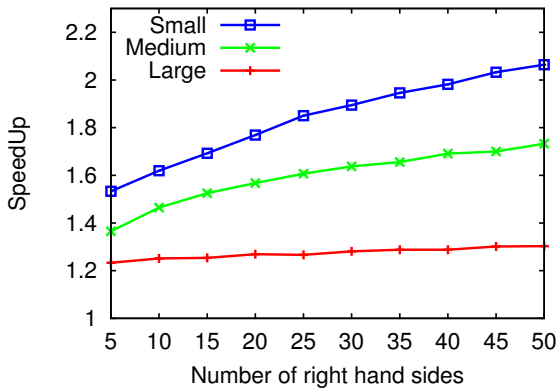


Fig. 10: Impact of varying right hand sides on speedup.

technique is $(N - (5245/R) - 40 \leq 0)$. The actual value in case of $R = 1$ is 3500 and it is recommended to use threshold function $(N \leq 3500)$ when using $R = 1$.

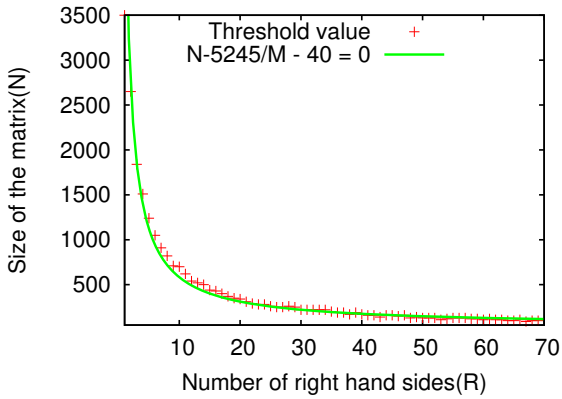


Fig. 11: Threshold function for matrix with multiple right hand sides

6) Choosing K in Fine Decomposition: In this experiment, we provide insights for choosing best value for K in fine decomposition when $R = 1$. The choice of K depends on the size of the matrix N and the K values chosen for matrices less than size N. For linear neurons of size $N > 3500$, we ran our algorithm for different values of K and chose the K which gave the best runtime. We find best K values for all matrix sizes constructively. Hence, in the lower levels of recursion, we use the computed best K values. From Figure 12, we can see that the variance of K is high for small values of N. For larger values of N, where recursion depth is greater than one, the best value of K remains constant at three. One interesting thing to observe is that for neurons around size 20000, it is possible to go down two levels in the recursion but the best K value is the one that recurses only once. To get good performance, maintain a look up table for smaller matrices and use $K = 3$ for larger matrices.

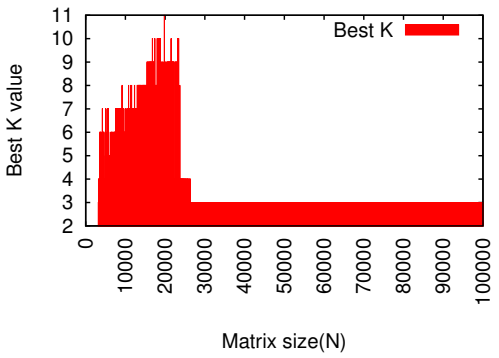


Fig. 12: Best value of K in Fine Decomposition

V. Conclusions and Future work

In scientific simulations based on ordinay and partial differential equations, matrix solvers are almost always a bottleneck and making them faster reduces simulation time considerably. In this paper, we have demonstrated that embracing the semantics of matrices into parallel algorithm design helps in designing efficient parallel solvers. The general form given for Hines matrices in this paper provides a framework for proving more results on Hines matrices. On the software front, we will consolidate our algorithms into a CUDA library which can then be used by neuron simulator frameworks such as NEURON [22], MOOSE (Multiscale Object-Oriented Simulation Environment) [23] and others.

In voltage PDE simulation of a neuron, only main-diagonal of the matrix and right hand side vector changes in every time-step. It would be interesting to see if any numerical method can take advantage of this and in turn lead to efficient parallel algorithms.

References

[1] University of Florida (2011) UF sparse matrix collection.Available at:(http://www.cise.ufl.edu/research/sparse /matrices/groups.html).

[2] Asanovic, Krste, et al. The landscape of parallel computing research: A view from berkeley. Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Intel Math Kernel Library, https://software.intel.com/en-us/articles/intel-mkl/.

[4] Nvidia sparse matrix library(cuSPARSE), http://developer.nvidia.com/cusparse.

[5] Wangdong Yang, Kenli Li, and Keqin Li. A hybrid computing method of SpMV on CPU-GPU heterogeneous computing systems, in Proc. of Journal of Parallel and Distributed Computing (2017), volume 104, 49-60.

[6] Kiran Kumar Matam, Siva Rama Krishna Bharadwaj, and Kishore Kothapalli. Sparse Matrix Matrix Multiplication on Hybrid CPU+GPU Platforms, in Proc. of 19th Annual International Conference on High Performance Computing (HiPC), Pune, India, 2012, 1-10.

[7] Emmanuel Agullo, Alfredo Buttari, Mikko Byckling, Abdou Guermouche, Ian Masliah. Achieving high-performance with a sparse direct solver on Intel KNL. [Research Report] RR-9035, Inria Bordeaux Sud-Ouest; CNRS-IRIT; Intel corporation; Universit of Bordeaux. 2017, pp. 15.

[8] Kiran Raj Ramamoorthy, Dip Sankar, Kannan Srinathan and Kishore Kothapalli, A Novel Heterogeneous Algorithm for Multiplying Scale-Free Sparse Matrices, in Proc. of IPDPS Workshops, 2016, 637-646.

[9] Dharma Teja Vooturi and Kishore Kothapalli, Parallel Algorithm for Quasi-Band Matrix-Matrix Multiplication, in Proc. of Parallel Processing and Applied Mathematics 2015, 106-115.

[10] Aydin Buluc and John Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In Proc. ICPP 2008, 503-510.

[11] Wangdong Yang, Kenli Li, Yan Liu, Lin Shi and Lanjun Wan. Optimization of quasi-diagonal matrix-vector multiplication on GPU. International Journal On High Performance Computing Applications, Vol. 28(2) 2014, 183-195.

[12] Michael Hines. Efficient computation of branched nerve equations. International journal of bio-medical computing 15.1 (1984): 69-76.

[13] Michael Mascagni. A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. Journal of Neuroscience Methods 1990, vol 36, 105-114.

[14] Osep-Lluis Larriba-Pey, Michael Mascagni, Angel Jorba, Juan J. Navarro. An Analysis of the Parallel Computation of Arbitrarily Branched Cable Neuron Models. In PPSC 1995, (373-378).

[15] Michael L. Hines, Hubert Eichner and Felix Schurmann. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. Journal of Computational Neuroscience. 2008;25(1):203-210. doi:10.1007/s10827-007-0073-3.

[16] Michael L. Hines, Henry Markram and Felix Schurmann. Fully implicit parallel simulation of single neurons. Journal of computational neuroscience 25.3 (2008): 439-448.

[17] Ben-Shalom, Roy, Gilad Liberman, and Alon Korngreen. Accelerating compartmental modeling on a graphical processing unit. Frontiers in neuroinformatics 7 (2013): 4.

[18] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. JACM 1973, 20:27-38.

[19] Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. The Journal of Physiology. 1952;117(4):500-544.

[20] James M. Bowe, Beeman David. Compartmental modeling, The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System. 1998, New York: Springer-Verlag, 7-16.

[21] Ascoli A. Giorgio, Duncan E. Donohue, and Maryam Halavi. "NeuroMorpho. Org: A central resource for neuronal morphologies." Journal of Neuroscience 27.35 (2007): 9247-9251.

[22] NEURON (https://www.neuron.yale.edu/neuron/).

[23] MOOSE(Multiscale Object-Oriented Simulation Environment) neuron simulator, (https://moose.ncbs.res.in).