A Parallel Jacobi-Embedded Gauss-Seidel Method

Afshin Ahmadi, *Student Member, IEEE*, Felice Manganiello, Amin Khademi, and Melissa C. Smith, *Senior Member, IEEE*

Abstract—A broad range of scientific simulations involve solving large-scale computationally expensive linear systems of equations. Iterative solvers are typically preferred over direct methods when it comes to large systems due to their lower memory requirements and shorter execution times. However, selecting the appropriate iterative solver is problem-specific and dependent on the type and symmetry of the coefficient matrix. Gauss-Seidel (GS) is an iterative method for solving linear systems that are either strictly diagonally dominant or symmetric positive definite. This technique is an improved version of Jacobi and typically converges in fewer iterations. However, the sequential nature of this algorithm complicates the parallel extraction. In fact, most parallel derivatives of GS rely on the sparsity pattern of the coefficient matrix and require matrix reordering or domain decomposition. In this article, we introduce a new algorithm that exploits the convergence property of GS and adapts the parallel structure of Jacobi. The proposed method works for both dense and sparse systems and is straightforward to implement. We have examined the performance of our method on multicore and many-core architectures. Experimental results demonstrate the superior performance of the proposed algorithm compared with GS and Jacobi. Additionally, performance comparison with built-in Krylov solvers in MATLAB showed that in terms of time per iteration, Krylov methods perform faster on CPUs, but our approach is significantly better when executed on GPUs. Lastly, we apply our method to solve the power flow problem, and the results indicate a significant improvement in runtime, reaching up to 87 times faster speed compared with GS.

Index Terms—Linear systems, iterative methods, parallel, PJG, Gauss-Seidel, Jacobi, Krylov, SpMV performance, power flow

1 Introduction

Techniques for solving linear algebraic systems have always been a subject of interest in the scientific community because the simulation of many complex physical problems relies on the solution of these equations. Generally, there are two different techniques for this purpose: direct and iterative. Direct methods can provide the exact solution within a finite number of steps, while iterative methods start with an initial guess and compute iteratively to find a close approximation. The computation cost of iterative methods for full $n \times n$ matrices is on the order of n^2 for each iteration, compared with the overall $\frac{2}{3}n^3$ required by direct techniques [1]. Moreover, iterative methods require less computational resources, such as memory, compared with direct methods. Therefore, iterative methods are generally preferred when it comes to large systems because the dense fill-in factor of direct methods scales intensely with respect to the matrix size. For instance, solving boundary integral equations and the radiosity equation often leads to large dense matrices, and the solution of partial differential equations often results in large sparse systems. Iterative methods for solving linear systems generally fall into the following categories:

- Stationary methods (e.g., Gauss-Seidel (GS), Jacobi, Successive Over-Relaxation (SOR), etc.)
- Methods based on Krylov subspace (e.g., Conjugate Gradient, Generalized Minimal Residual, etc.)

Among the stationary methods, the convergence rate of GS is typically better than Jacobi but lower than SOR. However, finding the optimal relaxation parameter of SOR is difficult and computationally expensive for many problems [2]. The GS approach is widely employed as a smoother for the sparse linear system arising from the discretization of the Poisson Equation and is considered superior to other preconditioners such as Jacobi [3].

Inspired by the Jacobi and GS methods, this research demonstrates a new algorithm for solving systems of linear equations that exploits the convergence property of GS and adapts the parallel structure of Jacobi. We base our iteration on the execution of a sequence of partial Jacobi iterations, focusing on a partition of the equations in the system. The parallel structure of Jacobi improves the computation time of each iteration, and the superior convergence rate of GS helps reduce the number of iterations to reach the solution. These characteristics allow our method to achieve better performance compared with these two methods alone.

Recent advancements in computational resources and their widespread availability have allowed researchers to utilize parallel computing approaches to develop fast linear solvers that can address the growth in size and complexity of scientific problems [4], [5], [6], [7], [8]. Among the iterative

- *Afshin Ahmadi and Melissa C. Smith are with the Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634 USA. E-mail: {aahmadi, smithmc}@clemson.edu.*
- *Felice Manganiello is with the School of Mathematical and Statistical Sciences, Clemson University, Clemson, SC 29634 USA. E-mail: manganm@clemson.edu.*
- *Amin Khademi is with the Department of Industrial Engineering, Clemson University, Clemson, SC 29634 USA. E-mail: khademi@clemson.edu.*

Manuscript received 30 May 2020; revised 10 Jan. 2021; accepted 12 Jan. 2021. Date of publication 15 Jan. 2021; date of current version 28 Jan. 2021. (Corresponding author: Afshin Ahmadi.) Recommended for acceptance by K. Mohror. Digital Object Identifier no. 10.1109/TPDS.2021.3052091

methods, Jacobi is a parallel algorithm that compared with GS, is inherently sequential due to the dependencies in the variables. However, various parallel derivatives of GS exist as well, mostly for sparse matrices and based on a multi-color ordering of grid points. A well-known parallel implementation of GS based on the coloring scheme is called red-black (RBGS), which is one of the most widely used smoothers for the problems [3]. However, exploiting parallelism in this method depends on the sparsity pattern of the system matrix due to the ordering requirement of the matrix rows/columns to achieve a specific structure [9].

Parallel GS implementations are regularly developed for either directly solving a problem [10], [11], [12], [13], [14], [15], [16], [17], [18], [19] or as a smoother in multigrid methods [3], [20], [21], [22], [23], [24]. A distributed memory implementation was presented in [12] for solving linear algebraic systems ($Ax = b$), where the $n \times n$ matrix A and the right-hand side vector b are first split among different computational nodes as row-wise blocks. The GS iteration is then performed with a natural ordering starting from Node 1, and only the updated values of the x vector are exchanged in the cluster to reduce the impact of communication delay. A $2\times$ speedup for a linear system with $n = 24,000$ was observed. This technique is useful for very large systems that cannot be stored in the memory of a single machine. However, the overall efficiency is low since the majority of the processors are idle at the same time.

In [14], Courtecuisse and Allard developed a block-based parallelization strategy for dense GS, in which the reliance on global synchronizations was reduced. The average speedup reached $10\times$ for a system size of 10,000. The approach applies to both dense and sparse matrices. Parallel solutions of SOR (relaxed GS) on the graphics processing unit (GPU) were also obtained in [15]. This method consists of a new domain decomposition and an alternate loop tiling technique. The maximum speedup achieved for a problem size of 28,672 was $2.8\times$. Another parallel implementation based on domain decomposition for the solution of linear equation systems related to the discretization of partial differential equations was developed in [20]. The advantage of this method is that the convergence rate is close to the original GS. However, no analysis regarding execution time was provided in the paper.

Brike *et al*. [25] developed a parallel block-smoothing algorithm based on block Jacobi and chaotic block GS and achieved a $20\times$ speedup on multicore CPUs and around $45\times$ speedup on GPUs. A parallel GS algorithm for sparse power systems based on matrix ordering techniques reached a relative speedup of $11.6\times$ for a matrix on the order of 6,000 [16]. However, the performance of this approach deteriorates when applied to multigrid problems due to the increase in the number of required colors [12], [21]. A variant of parallel block GS for block tri-diagonal linear systems based on block reordering of the coefficient matrix has been developed in [17]. Compared with multicoloring methods, parallel block GS has the same spectrum as block GS; thus, the asymptotic rate of convergence remains close to the original approach.

This literature review shows that synchronization primitives (e.g., mutex, locks, signals, events, etc) and communication delay between distributed nodes are the main factors affecting the scalability and speed of parallel GS implementations. For these reasons, when it comes to large scale systems, naturally parallelizable but slow convergent methods such as Jacobi are sometimes favored over methods with a better convergence rate that are a sequential or semi-parallel algorithms. In this case, there is no guarantee that the selected method will converge to the solution or that the execution time will improve. Our Parallel Jacobi-Embedded Gauss-Seidel (PJG) method is designed to address these challenges by exploiting the parallel structure of Jacobi and superior convergence rate of GS. The PJG method differs from the block versions of GS and Jacobi as we do not divide the matrix A and the vectors into blocks.

2 Contributions

The main contributions of this study are summarized as follows:

- The proposed algorithm applies to both sparse and dense linear systems and no precomputation stage is needed. Most parallel implementations of GS are designed for sparse systems and require matrix reordering or domain decomposition.
- Implementation in both multicore and many-core architectures is straightforward since synchronization primitives are not required. In distributed memory architectures, blocks can be distributed among different nodes, where each machine can concurrently work on the problem and utilize parallel computing to finish the task.
- The proposed method converges for the class of strictly diagonally dominant and irreducibly diagonally dominant matrices (Theorems 6 and 7). One key step is to provide a novel representation of involved matrices in the algorithm structure (Eqs. (16) and (17)). Our analysis sheds light on the potential impact of variable choice for parallelization and speed of convergence (Corollary 8). The proposed method also has a faster convergence rate than Jacobi in a subclass of the above-mentioned matrices (Proposition 10).
- The proposed method is tested on the current state-of-the-art multicore and many-core hardware architectures and a comprehensive performance study is provided. Experimental results for dense linear systems demonstrate that the average number of iterations for the proposed algorithm is observably close to GS and the average execution time is up to $7\times$ and $85\times$ faster than GS on CPU and GPU, respectively. However, due to some technical limitations in MATLAB, which will be discussed in Section 4, the GPU implementation of the proposed algorithm is not well optimized and more speedup is expected if the kernel is developed in other programming platforms.
- A regression model is developed to estimate the performance of matrix-vector multiplication given the matrix characteristic and the processor specifications. This information is useful when it comes to selection of an optimal hardware platform or estimating the performance of various algorithms, including iterative solvers.

- We modified the GS implementation of power flow algorithm to substitute the GS solver with the proposed method, and the execution time was reduced from 79 minutes to less than a minute for a 2736-bus system. Power flow analysis is a widely used study in power systems.

3 Proposed Method

3.1 Formulation and Convergence Analysis

Let $A \in \mathrm{Mat}_{n\times n}(\mathbb{R})$ and $b \in \mathbb{R}^n$ and consider the linear system $Ax = b$. Let $A = D - L - U$ be the decomposition of matrix A in a diagonal matrix D, a strictly lower triangular matrix L and a strictly upper triangular matrix U. The following Jacobi (Eq. (1)) and GS (Eq. (2)) algorithms are used to iteratively solve the equation $Ax = b$ when $A = (a_{i,j})_{1\le i,j\le n}$ is strictly diagonally dominant, i.e.,

$$|a_{i,i}| > \sum_{j=1, j\neq i}^{n} |a_{i,j}|,$$

for $i \in [n] := \{1, \ldots, n\}$, or irreducibly diagonally dominant, meaning that A is irreducible,

$$|a_{i,i}| \geq \sum_{j=1, j\neq i}^{n} |a_{i,j}|,$$

for all $i \in [n]$, and the inequality is strict for at least one $i \in [n]$. Given $x_0 \in \mathbb{R}^n$

$$x_{k+1} = D^{-1}(L+U)x_k + D^{-1}b \tag{1}$$

$$x_{k+1} = (D-L)^{-1}Ux_k + (D-L)^{-1}b. \tag{2}$$

We propose a generalization of these iterative methods that can be implemented via parallel computing. The method sequentially updates the entries of a vector as follows. Given a partition of variables x, the Jacobi method restricted to the variables of the first partition is used to update the entries of the vector belonging to the same partition. This step can be performed in parallel. The updated vector is then passed as an input to the Jacobi method restricted to the variables of the second partition to update the entries of the vector belonging to the same partition. The updated vector is then used to update its entries belonging to the third partition of variables and this procedure continues until all of the entries are updated. This sequence is considered one iteration of the algorithm. We move to the next iteration until a stopping criteria is triggered. An appealing feature of our algorithm is that it uses the Jacobi method, which can be implemented in parallel, and enjoys a convergence rate closer to the GS method, as we will show in Section 7. That is, our algorithm combines the advantages of both the Jacobi and GS methods.

To formalize this idea we begin with the following definition that facilitates variable partitioning.

Definition 1. *The set of logical matrices $\mathcal{P} = \{P_i \,|\, i \in [t]\} \subset \mathrm{Mat}_{n\times n}(\mathbb{R})$ is a* decomposition of the identity (DoI) in $\mathbb{R}^n$, *or simply DoI, if $\sum_{i=1}^{t} P_i = I$. We call* partial identities *the elements of a decomposition of the identity.*

Let $\mathcal{P} = \{P_i \,|\, i \in [t]\}$ be a DoI. At the beginning of iteration k, let $x_{k,0} := x_k$ and define for $\ell \in [t]$

$$\begin{aligned} x_{k,\ell} &= P_\ell\big[D^{-1}(L+U)x_{k,\ell-1} + D^{-1}b\big] + (I - P_\ell)x_{k,\ell-1} \\ &= \big(P_\ell D^{-1}(L+U) + I - P_\ell\big)x_{k,\ell-1} + P_\ell D^{-1}b, \end{aligned} \tag{3}$$

and

$$x_{k+1} := x_{k,t}. \tag{4}$$

Hereafter, given the matrix decomposition $A = D - L - U$, we use the notation $M = D^{-1}(L+U)$. One can show that Eqs. (3) and (4) can be written as

$$\begin{aligned} x_{k+1} = &\prod_{i=t}^{1}(P_i M + I - P_i)x_k \\ &+ \left(\sum_{j=1}^{t}\left[\prod_{i=t}^{j+1}(P_i M + I - P_i)\right]P_j\right)D^{-1}b, \end{aligned} \tag{5}$$

which shows the iterative nature of the proposed algorithm.

Next, we show that our proposed method is a generalization of both the Jacobi and the GS methods. Clearly, for $t = 1$, the recursion defined by Eq. (5) is the same recursion of the Jacobi method defined by Eq. (1). Therefore, the Jacobi method is a special case of the proposed, more general method. The same is true for the GS method but first we present several lemmas that will be used to prove this claim. In particular, Lemmas 2, 3, and 4 explore some properties of the matrices that appear in the proof of the main theorems.

Lemma 2. *Let $\mathcal{P} = \{P_i \,|\, i \in [t]\} \subset \mathrm{Mat}_{n\times n}(\mathbb{R})$ be a DoI. Then the following relations hold.*

$$P_i P_j = 0, \ \textit{for } 1 \le i, j \le t. \tag{6}$$

$$P_i^2 = P_i, \ \textit{for } 1 \le i \le t. \tag{7}$$

$$P_i\left(I - \sum_{j=1}^{i-1} P_j\right) = 0, \ \textit{and} \tag{8}$$

$$P_i\left(I - \sum_{j=1}^{i} P_j\right) = P_i, \ \textit{for } 1 \le i \le t. \tag{9}$$

The proof of Lemma 2 is based on standard results in linear algebra and properties of P_is.

Lemma 3. *Let $P_i \in \mathrm{Mat}_{n\times n}(\mathbb{R})$ be such that $(P_i)_{i,i} = 1$ and the remaining entries are 0 for $i \in [n]$ and $M \in \mathrm{Mat}_{n\times n}(\mathbb{R})$. For $d \geq 1$ it holds that*

$$M^d = \sum_{i_1,\ldots,i_{d+1}=1}^{n} P_{i_1} M P_{i_2} M P_{i_3} \cdots P_{i_d} M P_{i_{d+1}}. \tag{10}$$

This lemma can be proven by induction on the exponent and by using the relations of Lemma 2.

Lemma 4. *Let $\mathcal{P} = \{P_i \,|\, i \in [t]\} \subset \mathrm{Mat}_{n\times n}(\mathbb{R})$ be a DoI with $t \geq 2$. For a matrix $M \in \mathrm{Mat}_{n\times n}(\mathbb{R})$, the matrix*

$$T = \sum_{\ell < j \le t} P_j M P_\ell,$$

is a nilpotent matrix with index less than or equal to t.

Proof. We will prove that $T^{t-1} = P_t M P_{t-1} M P_{t-2} \cdots P_2 M P_1$, which implies that $T^t = 0$ by Eq. (6). We prove this by the induction on t, meaning the cardinality of the DoI.

If $t = 2$, then trivially $T = P_2 M P_1$. Let the statement be true for any DoI of cardinality t, let us prove that it holds true also for a DoI $\mathcal{P} = \{P_i \mid i = 1, \ldots, t+1\} \subset \mathrm{Mat}_{n\times n}(\mathbb{R})$ of cardinality $t+1$. Define the DoI $\tilde{\mathcal{P}}$ of cardinality t with partial identities $\tilde{P}_i = P_i$ for $i = 1, \ldots, t-1$ and $\tilde{P}_t = P_t + P_{t+1}$. By induction hypothesis we have that

$$\tilde{T}^{t-1} = \left(\sum_{\ell < j \le t} \tilde{P}_j M \tilde{P}_\ell\right)^{t-1} = \tilde{P}_t M \tilde{P}_{t-1} M \tilde{P}_{t-2} \cdots \tilde{P}_2 M \tilde{P}_1.$$

It follows that

$$\begin{aligned} T^t &= (\tilde{T} + P_{t+1} M P_t)\left(\sum_{i=0}^{t-1} \binom{t-1}{i} \tilde{T}^i (P_{t+1} M P_t)^{t-1-i}\right) \\ &\overset{(6)}{=} (\tilde{T} + P_{t+1} M P_t)\tilde{T}^{t-1} = \tilde{T}^t + P_{t+1} M P_t \tilde{T}^{t-1} \\ &\overset{(7)}{=} P_{t+1} M P_t M P_{t-1} \cdots P_2 M P_1. \end{aligned}$$

□

The following property holds for nilpotent matrices. Let $T \in \mathrm{Mat}_{n\times n}(\mathbb{R})$ be a nilpotent matrix with nilpotency index t, then $I - T$ is invertible and

$$(I - T)^{-1} = \sum_{i=0}^{t-1} T^i. \tag{11}$$

Next, we show that the GS method is a special case of the proposed method.

Proposition 5. *For $P_i \in \mathrm{Mat}_{n\times n}(\mathbb{R})$ be such that $(P_i)_{i,i} = 1$ and the remaining entries are 0 for all $i \in [n]$, the recursion defined by Eq. (4) is the same recursion of the GS method defined by Eq. (2).*

Proof. The matrices P_i for $1 \le i \le n$ satisfies the hypotheses of Lemma 3. By induction on $\ell \ge 2$ and with the help of the previous lemmas, it holds that Eq. (3) is equivalent to

$$x_{k,\ell} = \delta(\ell) x_{k,0} + \gamma(\ell) b, \tag{12}$$

where, if

$$\Delta := \sum_{\substack{j=1 \\ \ell \ge i_1 > \cdots > i_{j+1}}}^{\ell-1} P_{i_1} D^{-1} L P_{i_2} \cdots P_{i_j} D^{-1} L P_{i_{j+1}} + \sum_{j=1}^{\ell} P_j,$$

$$\delta(\ell) = \Delta(M - I) + I, \tag{13}$$

and

$$\gamma(\ell) = \Delta D^{-1}. \tag{14}$$

Finally it holds that

$$\begin{aligned} \delta(n) &= \left(\sum_{j=1}^{n-1} (D^{-1}L)^j + I\right)(M - I) + I \\ &= (I - D^{-1}L)^{-1}(M - I) + I = (D + L)^{-1} U, \end{aligned}$$

and

$$\gamma(n) \overset{(10)}{=} \left(\sum_{j=1}^{n-1} (D^{-1}L)^j + I\right) D^{-1} = (D + L)^{-1},$$

since $D^{-1}L$ is an nilpotent matrix with nilpotent exponent at most n. This concludes the proof. □

It is essential to prove that our proposed method converges to the solution of the linear system. In [26], the author shows that if A is either a strictly diagonally dominant or irreducibly diagonally dominant, then the Jacobi and GS methods converge to the solution. We prove that our method converges for the same class of matrices for *any DoI*. We establish this result in two steps. First, we prove in Theorem 6 that the solution $A^{-1}b$ of $Ax = b$ is a fixed point of the iterative procedure presented in Eq. (5). Then, we show in Theorem 7 that the iterative procedure converges. Therefore, our proposed algorithm converges to the solution of the system. The difficulty in proving these results is that matrices $P_i M + I - P_i$ *do not commute* and standard results do not apply. For example, the bounds on spectral radius of summation and multiplication of matrices hold true only for commuting matrices. We address these challenges by exploiting the special structure of the produced matrices, by developing a novel representation of the involved terms, and by invoking Gersgorin Circle Theorem and Taussky's Theorem.

Theorem 6. *Let A be an invertible $n \times n$ matrix with decomposition $A = D - L - U$ with D invertible. Furthermore, let $\{P_i \mid i \in [t]\}$ be a DoI. Then $A^{-1}b$ is a fixed point of Eq. (5).*

Proof. In order to prove that

$$\begin{aligned} A^{-1}b = &\prod_{i=t}^{1} (P_i M + I - P_i) A^{-1} b \\ &+ \left(\sum_{j=1}^{t} \left[\prod_{i=t}^{j+1} (P_i M + I - P_i)\right] P_j\right) D^{-1} b, \end{aligned}$$

it is sufficient to prove that

$$\prod_{i=t}^{1} (P_i M + I - P_i) + \left(\sum_{j=1}^{t} \left[\prod_{i=t}^{j+1} (P_i M + I - P_i)\right] P_j\right) D^{-1} A = I. \tag{15}$$

By induction and with the help of Lemma 2, it is possible to prove that

$$\begin{aligned} \sum_{j=1}^{t} \left[\prod_{i=t}^{j+1} (P_i M + I - P_i)\right] P_j &= \sum_{\substack{j=0 \\ i_1 < \cdots < i_{j+1} \le t}}^{t-1} P_{i_{j+1}} M P_{i_j} \cdots P_{i_2} M P_{i_1} \\ &= \sum_{j=0}^{t-1} T^j = (I - T)^{-1}, \end{aligned} \tag{16}$$

where $T = \sum_{\ell < j \le t} P_j M P_\ell$ and the last equality is a consequence of Lemma 4 and Theorem 11. Moreover, by recursively using the relation

$$\prod_{i=t}^{j}(P_iM+I-P_i)=\prod_{i=t}^{j+1}(P_iM+I-P_i)P_jM + \prod_{i=t}^{j+1}(P_iM+I-P_i)-\prod_{i=t}^{j+1}(P_iM+I-P_i)P_j,$$

we obtain that

$$\begin{aligned}\prod_{i=t}^{1}(P_iM+I-P_i) &= \sum_{j=1}^{t}\left[\prod_{i=t}^{j+1}(P_iM+I-P_i)\right]P_jM+I \\ &\quad -\sum_{j=1}^{t}\left[\prod_{i=t}^{j+1}(P_iM+I-P_i)\right]P_j \qquad (17)\\ &\overset{(16)}{=} (I-T)^{-1}M+I-(I-T)^{-1} \\ &= (I-T)^{-1}(M-I)+I.\end{aligned}$$

Since $D^{-1}A=I-M$, we obtain that the right-hand side of Eq. (15) becomes

$$(I-T)^{-1}(M-I)+I+(I-T)^{-1}(I-M)=I.$$

□

Theorem 7. *Let A be either a strictly diagonally dominant or an irreducibly diagonally dominant matrix and $\{P_i \mid i\in[t]\}$ be a DoI. Then*

$$\rho\left(\prod_{i=t}^{1}P_iM+I-P_i\right)<1,$$

where ρ denotes the spectral radius.

Proof. First, we show the result for strictly diagonally dominant matrices. Recall that in the proof of Theorem 6, we showed that

$$\prod_{i=t}^{1}P_iM+I-P_i=(I-T)^{-1}(M-I)+I.$$

Matrix $M=\left(m_{ij}\right)_{1\le i,j\le n}$ has zero diagonal and since A is strictly diagonally dominant, $\sum_{j=1}^{n}|m_{ij}|<1$ for $i\in[n]$.

For an eigenvalue λ of $(I-T)^{-1}(M-I)+I$ it holds that

$$\begin{aligned}0&=\det\Big(\lambda I-(I-T)^{-1}(M-I)-I\Big)\\ &=\det(I-T)^{-1}\det(\lambda(I-T)-M+T).\end{aligned}$$

Since by Lemma 4 $\det((I-T)^{-1})\neq 0$, then $\det(\lambda(I-T)-M+T)=0$. Let $B_\lambda=\lambda(I-T)-M+T$. We show that if $|\lambda|\ge 1$, B_λ is not singular, which gives us the result. Note that the entries of matrix T are either 0 or the entries of M based on P_is. It follows that B_λ is a matrix with λ on all diagonal entries and either $-m_{ij}$ or $-\lambda m_{ij}$ based on matrix T otherwise. Also, if $\lambda>1$, then the sum of the absolute values of all non-diagonal entries in each row of $B_\lambda=(\,b_{ij}\,)_{1\le i,j\le n}$ is less than λ because

$$\sum_{j=1}^{n}|b_{ij}|\le|\lambda|\sum_{j=1}^{n}|m_{ij}|<|\lambda|.$$

Let $K_i:=\{z\in\mathbb{C}:|z-\lambda|\le\sum_{j=1}^{n}|b_{ij}|<1\}$ denote the Gershgorin disk of B_λ. By the Gershgorin Circle Theorem, all eigenvalues of B_λ belong to $K=\bigcup_{i=1}^{n}K_i$. However, since $|\lambda|\ge 1$, 0 does dot belong to K. Therefore, if $|\lambda|>1$ then

$$\det\Big(\lambda I-(I-T)^{-1}(M-I)-I\Big)\neq 0.$$

The theorem follows by contraposition.

For irreducibly diagonally dominant matrices, all of the previous arguments hold, but in the last step, 0 belongs to the boundary of K. However, by Taussky's Theorem, 0 must belong to all K_is for irreducible matrices, which is impossible because by the definition given at the beginning of the section for an irreducibly diagonally dominant matrix, there exists at least a row for which the absolute value of the diagonal is strictly greater than the sum of the absolute values of the non-diagonal entries. □

The proof of Theorem 7 is insightful in that it can help identify partitions that may have a faster convergence rate. First, we state the following corollary.

Corollary 8. *Let $K^P=\bigcup_{i=1}^{n}K_i^P$ and $K^J=\bigcup_{i=1}^{n}K_i^J$ denotes respectively the union of the Gershgorin disks that contain all of the eigenvalues of $D^{-1}(L+U)$, the iteration matrix of Jacobi, and $(I-T)^{-1}(M-I)+I$, the iteration matrix of the proposed method. Then, $K^P\subset K^J$.*

Proof. All eigenvalues corresponding to Jacobi satisfy $\det(\lambda I-M)=0$ and to the proposed method satisfy $\det(\lambda(I-T)-M+T)=0$. For $i\in[n]$, all disks K_i^J and K_i^P are centered at λ. The radius of K_i^J is: $\sum_{j\neq i}|m_{ij}|$ whereas the radius of K_i^P is

$$\sum_{j\neq i,\ j\notin S_i^T}|m_{ij}|+|\lambda|\sum_{j\neq i,\ j\in S_i^T}|m_{ij}|,$$

where $S_i^T:=\{j\in[n]\mid t_{i,j}\neq 0\}$. Since $\lambda<1$ it follows that $K_i^J\subsetneq K_i^P$. □

As can be seen from the proof above, the radius of our proposed disk is smaller due to the term $\sum_{j\neq i,j\in S_i^T}\lambda|m_{ij}|$, which depends on S_i^T. Therefore, if matrix T is chosen properly, the radius of the disk is smaller and a faster convergence may be achieved.

Nonetheless, Corollary 8 does not guarantee that the convergence rate of the proposed method is better than Jacobi. Next, we use a generalization of Stein-Rosenberg Theorem to show that indeed our method has a faster convergence rate than Jacobi. To that end, we use the following result.

Theorem 9 ([27]). *Let $A=M_1-N_1=M_2-N_2$ be two M-splittings of A (i.e., M_is are M-matrices and $N_i\ge 0$ for $i=1,2$), $N_1\neq N_2$, $N_2\neq 0$, and $N_1\ge N_2$. Then, exactly one of the following holds:*

1) $0\le\rho(M_2^{-1}N_2)<\rho(M_1^{-1}N_1)<1$;
2) $\rho(M_2^{-1}N_2)=\rho(M_1^{-1}N_1)=1$;
3) $1<\rho(M_1^{-1}N_1)<\rho(M_2^{-1}N_2)$.

Recall that the basic Stein-Rosenberg takes irreducible $A = I - L - U$, where $L, U \geq 0$, and $L, U \neq 0$. We also consider this class of matrices to show that the proposed method produces a faster convergence rate than Jacobi. In particular, we have the following result.

Proposition 10. *Let A be a strictly diagonally dominant or irreducibly diagonally dominant matrix with diagonal entries equal to 1 and $L, U \geq 0$, and $L, U \neq 0$. Given a DoI $\mathcal{P} = \{P_i \mid i \in [t]\}$, it follows that*

$$\rho\left(\prod_{i=t}^{1} P_i M + I - P_i\right) < \rho(L+U).$$

Proof. Since $D = I$, for Jacobi the iterative procedure is given by $x_{k+1} = (L+U)x_k + b$, and for the proposed methods is $(I-T)x_{k+1} = (M-T)x_k + b$ from Eq. (5). For the class of matrices $A = I - L - U$. First, we need to show that $I - T$ is an M-matrix. Note that all of the non-diagonal elements of $I - T$ are less than or equal to 0 and, since T is nilpotent, that all of the eigenvalues of $I - T$ are equal to 1. Therefore, $I - T$ is an M-matrix. Also, by construction, $M - T \leq L + U$, $M - T \neq 0$, and $M - T \geq 0$. Clearly, the identity is an M-matrix and $L + U \geq 0$ for Jacobi. Taking $M_1 = I$, $N_1 = L + U$, $M_2 = (I - T)$, $N_2 = (M - T)$, and applying Theorem 9 yields the desired result. □

3.2 Computational Complexity and Storage Analysis

Let the number of non-zero elements in matrix $A_{n\times n}$ be *NNZ*. The number of operations for solving the $Ax = b$ using Jacobi and Gauss-Seidel is approximately $NNZ + 2n$ in each iteration [5]. Since the PJG method is basically an embedded Jacobi inside the GS algorithm, it has the same number of multiplication operations. Likewise, the computational complexity of PJG is $7kn$, where k is the number of iterations to solve the linear system. The storage requirement to store the A matrix in a sparse CSR format is $2NNZ + n + 1$. Additionally, vectors x^k, x^{k+1}_{block}, and b must be stored during the iteration process of PJG. Therefore, the total amount of storage required for our proposed method is approximately

$$2NNZ + n + 1 + 2n + l \approx 2NNZ + 3n + l,$$

TABLE 1
Computational Complexity and Storage Analysis

Algorithm	Number of Operations Per Iteration	Computational Complexity	Storage (CSR)
PJG	$NNZ + 2n$	$7kn$	$2NNZ + 3n + l$
Jacobi	$NNZ + 2n$	$7kn$	$2NNZ + 4n$
Gauss-Seidel	$NNZ + 2n$	$7kn$	$2NNZ + 3n$
BiCG	$2NNZ + 8n$	$18kn$	$2NNZ + 8n$
BiCGStab	$2NNZ + 11n$	$21kn$	$2NNZ + 7n$
TFQMR	$2NNZ + 8n$	$18kn$	$2NNZ + 7n$
GMRES	$NNZ + (2v+1)n$	$6kn + k^2n$	$2NNZ + 2kn + k^2$

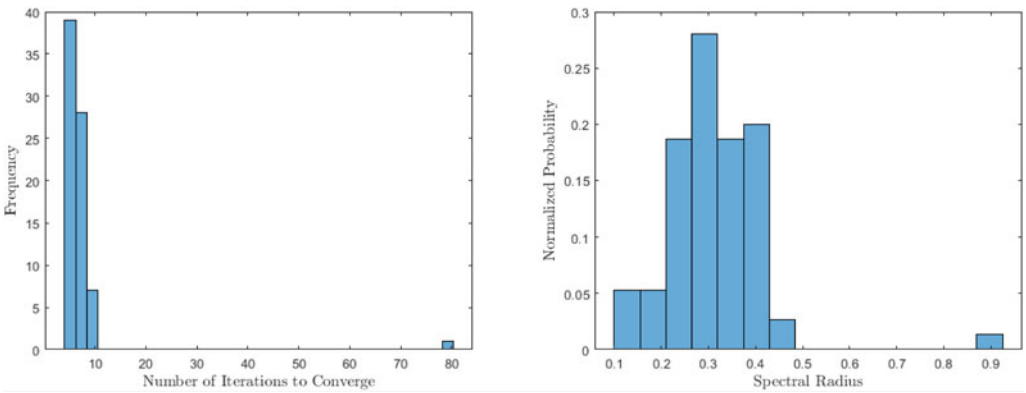
where l is the size of the Jacobi block (between 1 to n). The computational complexity and storage requirement of several iterative solvers has been discussed in [5], [28], [29], [30] and a summary is presented in Table 1. Here, BiCG, BiCGSTAB, GMRES, and TFQMR refers to Biconjugate Gradients, Biconjugate Gradients Stabilized, Generalized Minimum Residual, and Transpose-free Quasi-minimal Residual methods, respectively.

3.3 Sensitivity to the Choice of DoI

Recall that our method converges for all of the DoIs and Corollary 8 suggests the possibility of a faster convergence by properly choosing matrix T. However, it is difficult to predict a priori which partition produces the best result. Here, we investigate the distribution of the spectral radius of our method on (0,1) numerically. To that end, we consider the following diagonally dominant linear system, construct all possible DoIs, and calculate the spectral radius of the proposed method for each DoI.

$$\begin{bmatrix} 9 & 2 & 1 & 5 \\ 5 & 14 & 1 & 7 \\ 4 & 8 & 15 & 2 \\ 3 & 4 & 5 & 13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{18}$$

Fig. 1a shows the frequency plot of the number of iterations to reach convergence. As can be seen, the proposed method mostly converged in 11 or fewer iterations, while only one permutation required 80 iterations. Further investigation revealed that the permutation with 80 iterations corresponds to Jacobi (recall that a DoI in our method retrieves Jacobi), which generally requires more iterations to



(a) Frequency of Iteration Number (b) Normalized Probability of Spectral Radius

Fig. 1. Convergence analysis of the numerical example.

converge compared with GS. This discovery is in line with the theoretical position that the convergence rate of the proposed algorithm is generally better than Jacobi. We also assess the change in the spectral radius of Eq. (5) for all of the possible permutations. The normalized probability distribution shown in Fig. 1b confirms that the spectral radius always remained less than one. The radii value of 0.9 relates to the Jacobi method, and 0.24 refers to the GS. It is noticeable that the spectral radius of the iteration matrix in the proposed method has a very high probability to remain close to GS. Generally, a smaller spectral radius means fewer iterations, which results in faster execution time.

4 Parallel Implementation in MATLAB

In this section we examine the performance of the proposed method on CPU and GPU architectures and provide a fair comparison with Krylov solvers. The built-in numerical and linear algebra functions in MATLAB are suitable for this purpose because they are highly efficient, developed to exploit parallelism by default, and can target different architectures. Also, since partial matrix-vector multiplication is the core of our algorithm, and the vectorization techniques in MATLAB allow this part of the code to be parallelized efficiently, we implement and examine our method in this platform.

4.1 Algorithm

In general, each iteration of the proposed algorithm can be implemented in t "mini-steps," where t is the number of submatrices in the set $\mathcal{P} = \{P_i \mid i = 1, \ldots, t\} \subset \mathrm{Mat}_{n\times n}(\mathbb{R})$. In the first step, we chose a block of vector x according to P_1 and perform the Jacobi operation to update the respective unknown values independently and in parallel. The updated values are stored in a separate vector, $\hat{x}$, to avoid overwriting the initial x values. In the second step, we replace the old values of x with the updated ones in $\hat{x}$, and perform the same operation as in the previous step but with P_2. This operation continues until all of the P_i matrices are applied. The order in which the unknown variables are updated independently (i.e., Jacobi mini-steps) depends on how the P_i matrices are defined and applied. In our implementation, we perform the Jacobi updates in fixed-size contiguous blocks, starting from the first entry of vector x in each block, for two reasons:

- Algebraic operations are performed on continuous chunks, and because MATLAB stores the array data in a contiguous block of memory, higher performance is achieved.
- The code is less complex and more efficient.

Algorithm 1 shows the vectorized implementation of the PJG iterative method in MATLAB. The number of computational threads can be controlled by changing the value of N in $maxNumCompThreads(N)$ function. As it can be seen, process synchronization primitives are not required because all threads are working on a single section (i.e., block) with no data dependency. Synchronization mechanisms such as mutex, signal, locks, etcetera are needed when threads are to access a shared resource that might be updated by another thread at the same time. Synchronization between processors has the unfortunate side effect of increased latency, which leads to a longer execution time.

4.2 GPU Implementation

GPUs are equipped with a massive number of processors that can deliver extremely high computational throughput for computationally intensive and inherently parallel applications. However, programming for GPUs is notably different from traditional CPUs and requires additional effort and time. MATLAB provides tools and functions for this purpose that significantly reduce the programming effort. Basically, Algorithm 1 will automatically execute on a GPU by converting A, b, and x to GPU arrays using *gpuArray* function. However, there are two limitations with MATLAB linear algebra libraries for GPU implementation in this case. First, array indexing is not supported for sparse matrices. Second, there is significant performance loss in matrix operations when array indexing is applied because MATLAB makes a strided memory copy of the array to perform the operation. The latter issue negatively impacts the performance of the algorithm because it highly relies on the array indexing.

Algorithm 1. Vectorized Implementation of the PJG Algorithm in MATLAB for Solving $Ax = b$

```
Input: matrix A = (a_{i,j})_{1≤i,j≤n}
       vector b = (b_i)_{1≤i≤n}
       psize // size of Jacobi partition
Output: solution vector x = (x_i)_{1≤i≤n}
x ← (0, 0, ..., 0)_n // initial guess
d ← −diag(A) // diagonal entries of A
while residual > ε and iter ≤ maxiter do
   iter = iter + 1
   for begin ← 1 to n by psize do
      end ← begin + psize
      if end > n then end ← n
      sum ← A(begin:end,1:n) ∗x
            + d(begin:end) ·∗ x(begin:end)
      x(begin:end) ← (sum - b(begin:end))
                     ./ d(begin:end)
   end
   residual ← norm(A ∗ x − b)/norm(b)
end
```

Several strategies to overcome this problem were tested, and among them, the transposed-multiply operated much faster. First, matrices in MATLAB are stored in column-major format, and picking out specific columns is more efficient than picking out certain rows. Second, according to the MATLAB support team, *gpuArray* matrix multiplication is optimized for the transposed-times case. Therefore, we replaced the A(begin:end, :)$*x$ operation with A_{trans}(:, begin:end)$*x$ in Algorithm 1, where A_{trans} contains the transpose of coefficient matrix A. As illustrated in Fig. 2, row-block multiplication using the transposed matrix runs about 3-5× faster than the non-transposed one. MATLAB creates a copy of that block in both cases, but clearly, the entire operation is faster when using transposed matrices. Interestingly, the execution time for full matrix-vector multiplication is identical in both cases. Although our strategy improves the overall performance, it is expected to exhibit more speedup if the code is developed in other programming platforms such as CUDA.

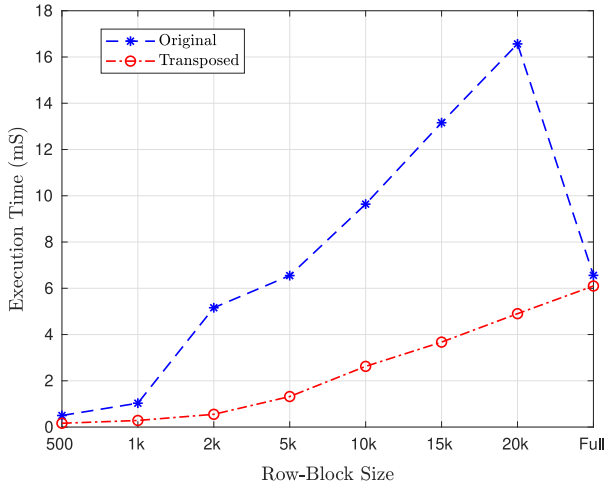


Fig. 2. Performance comparison of Matrix-Vector multiplication strategies with the change in row-block size on Tesla V100. Average execution time is measured by $gputimeit$ function for 100 trials, and matrix dimension is 25 000.

5 Performance Estimate of SpMV

Sparse matrix-vector multiplication (SpMV) and the number of iterations are the main factors that affect an iterative solver's performance. Accurately estimating the performance of SpMV on CPU and GPU is challenging because of the diverse property of sparse matrices and differences in processors architecture. Also, SpMV operation's performance for a specific sparse matrix may vary depending on the employed sparse storage format. [31] proposed a simple method to estimate the performance of SpMV on CPUs for the coordinate list (COO) format. Given the number of nonzero elements in COO format (NNZ), frequency of the CPU (f), and the number of CPU cores (nc) assigned to perform the SpMV operation, the SpMV compute time can be approximately estimated by $(2 * NNZ)/(f \times nc)$. This information was used to enhance the SpMV performance by adopting a hybrid CPU-GPU parallel programming model. Other hybrid methods for enhancing the speed of SpMV have been investigated in [32], [33]. An auto-tuning system was developed in [34] to automatically determine the optimal sparse storage format for faster SpMV operation given a sparse matrix in CSR format. Similarly, an auto-tuning library for the CSR format based on machine learning and retargetable back-end library was proposed in [35], and performance improvement of up to 2.5 times was achieved compared with Intel MKL. Several attempts have been made to model and optimize the SpMV operation on GPUs based on probabilistic models [36], [37]. Partitioning of the coefficient matrix is a technique used to address the imbalance in the computational load of processors due to sparse matrices' irregular sparsity pattern. However, this method adds communication overhead during the parallel computation of SpMV. [7] proposed a novel method to reduce the latency overhead of partitioned based SpMV and scale the performance of iterative solvers.

The most time-consuming element in iterative solvers is the matrix-vector (MV) operation because it is repeatedly performed in each iteration. Because our proposed method's performance largely depends on this operation, we have developed a simple regression model to estimate the speed of MV and SpMV for compressed sparse column (CSC) format on CPU and GPU platforms. Our dataset is unique in terms of size and sparsity pattern of matrices (more than 10,000 matrices with sparsity ranging from 0 to 90 percent), the number of utilized CPU cores (2 to 20 cores), and the diversity of processor architectures (six CPUs and four GPUs). Double-precision was selected to store the matrices and perform the computations. This dataset is available for download as a part of the supplementary materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3052091. This proposed regression model can be used to 1) estimate the performance of a single iteration in an iterative solver, 2) help in modeling and optimizing SpMV operation.

TABLE 2
Coefficients for Linear Regression Models for Performance Estimation of Matrix-Vector Multiplication on CPU and GPU

	Constant	a	b	c	d	e
CPU	−100.1	0.02052	-15.79	0.9664	30.64	-2.19
GPU	4.116	6.318E-4	-6.761	-0.9741	-	-

For example, the regression formula for CPU performance is $-100.1+0.02052a-15.79b+0.9664c+30.64d-2.19e$.

We have performed comprehensive testing to determine the best fitting variables. These parameters and their respective coefficients are presented in Table 2. R-squared, which is a measure of how close the data are to the fitted line, for both models is close to 65 percent. That is, the model explains about 65 percent of the variability of the CPU (GPU) performance data around its mean. For CPU platforms, a refers to the dimension of the coefficient matrix (use n for $A_{n\times n}$), b is the sparsity level of the A matrix (between 0 and 1, where zero refers to a fully dense matrix), c is the ratio of available cores to the utilized cores, d is the ratio of maximum processor clock to the base clock, and e is the cache size (in megabyte). For GPUs, a and b are similar to the CPU model, but c refers to the device double-precision performance in TeraFLOPS (available in the device manual). Refer to the dataset file for details about hardware devices used in the testing, as well as the complete information about the regression models.

6 Experiment Setup

6.1 Hardware and Software Configuration

The computer system used in this study is equipped with two Intel Xeon Gold 6148 processors and four NVIDIA Tesla V100-SXM2-16GB GPUs that are connected via NVIDIA NVLink. Each CPU has 20 cores with Advanced Vector Extension 512 (AVX-512) instruction set, which provides 512-bit wide vector operations, and two Fused Multiply Add instructions (FMA). The base operating frequency of each core is 2.4 GHz, which can be boosted up to 3.7 GHz. Tesla V100-SXM2 is currently one of the most advanced GPUs in the market. This GPU has 5,120 CUDA cores with a boost clock frequency of 1530 MHz, 16 GB of memory, and can achieve up to 7.8 TFLOPS in double-precision arithmetic. The operating system of the machine is Oracle Linux

TABLE 3
Performance Comparison of Iterative Solvers on Multicore CPU

	Average Iteration				Average Time (s)				Avg. Time Per Iteration (s)			
n	10K	20K	30K	40K	10K	20K	30K	40K	10K	20K	30K	40K
GS	**10.97**	**12.2**	**12.78**	**13.39**	**7.03**	**46.02**	**168.9**	**416.7**	**0.64**	**3.77**	**13.21**	**31.11**
PJG-500	11.25	12.33	12.87	13.48	3.64	16.46	37.99	72.99	0.32	1.33	2.95	5.41
PJG-1500	11.78	12.66	13.10	13.66	3.52	15.83	35.8	67.65	0.30	1.25	2.73	4.95
PJG-2500	12.42	12.95	13.31	13.83	3.67	16.00	35.6	67.35	0.30	1.24	2.68	4.87
PJG-3500	**12.98**	**13.22**	**13.53**	**14.01**	**3.81**	**16.26**	**35.94**	**66.88**	**0.29**	**1.23**	**2.66**	**4.78**
Jacobi	**22.54**	**23.72**	**24.54**	**26.77**	**6.67**	**29.33**	**69.83**	**138.38**	**0.30**	**1.24**	**2.85**	**5.17**
CGS	2.15	2.16	2.13	2.16	0.08	0.30	0.64	1.21	0.04	0.14	0.30	0.56
BiCG	3.02	3.01	3.01	3.01	0.10	0.40	0.87	1.57	0.03	0.13	0.29	0.52
BiCGSTAB	1.51	1.51	1.50	1.50	0.06	0.25	0.54	0.97	0.04	0.17	0.36	0.65
GMRES	**3.01**	**3.00**	**3.00**	**3.00**	**0.06**	**0.26**	**0.54**	**0.97**	**0.02**	**0.09**	**0.18**	**0.32**
QMR	3.02	3.01	3.01	3.01	0.10	0.42	0.88	1.57	0.03	0.14	0.29	0.52
TFQMR	1.95	1.90	1.83	1.81	0.08	0.31	0.64	1.18	0.04	0.16	0.35	0.65
A\b	–	–	–	–	**2.34**	**14.33**	**41.07**	**92.42**	–	–	–	–

Server 7.7, and all coding and testing have been accomplished in MATLAB 2020a.

6.2 Simulation Settings

Our simulation setting covers a wide range of problems and methods to provide a comprehensive performance report. For this reason, we set up the simulations as follows:

- Algorithms are tested on randomly generated systems of dense linear equations with the dimension ranging from 10,000 to 40,000.
- The number of test matrices is limited to 1,500 and 1,000 for 10K and 20K systems, respectively. For larger systems, this number is set to 500.
- Simulation stops when the relative residual becomes less than 10^{-5}, or the iteration number reaches 50.
- The test computer has a total of 40 cores. The number of threads is set to eight for each simulation to allow four simulations to run on the machine at the same time.
- Hyper-threading is disabled on this machine to limit the number of threads per core to one since the simulations are CPU-bound.
- Test matrices are generated and stored in double-precision format. All computations are also carried in double precision.

Memory and simulation time are the two main factors in choosing the parameters mentioned above. In fact, a single simulation in our study contains more than twenty sub simulations. Completing the whole experiment takes about four days, even though we utilized all hardware resources to simulate multiple cases at the same time.

6.3 Generating Problem Instances

The non-symmetric, diagonally dominant matrices in this study are randomly generated according to the following rules. Let $X \sim U[-1, 1]$ and $Y \sim U[1, n]$ be uniform and independent random variables. Let δ_1 and δ_2 be sampled from X with $\delta_1 < \delta_2$, and δ_0 be sampled from Y. The coefficient matrix A is formed as,

$$A = (a_{i,j})_{1 \le i,j \le n} = \begin{cases} \text{sampled in } [\delta_1 n, \delta_2 n] & \text{if } i \neq j, \\ \sum_{i \neq j} |A_{ij}| + \delta_0 & \text{otherwise.} \end{cases}$$

All random values are generated using MATLAB's uniformly distributed random number generator. For simplicity, the right-hand side vector b is set to one, and vector x is initialized with zeros.

7 Experimental Results

In this section, we investigate the performance of the PJG algorithm for various settings on both manycore and multicore architectures. The analysis is focused on the number of iterations to reach the solution, average run time, and per iteration computation time. We compare our algorithm with Krylov subspace methods and refer the interested reader to [38] and [2] for details. In this paper, CGS, BiCG, BiCGSTAB, GMRES, QMR, and TFQMR refers to Conjugate Gradients Squared, Biconjugate Gradients, Biconjugate Gradients Stabilized, Generalized Minimum Residual, Quasi-minimal Residual, and Transpose-free Quasi-minimal Residual methods, respectively.

7.1 Performance Evaluation on Multicore CPUs

Performance results on multicore CPU are presented in Table 3. The value following PJG refers to the Jacobi block size ($psize$ variable) in Algorithm 1. Changing the $psize$ value to one or n corresponds to the GS and Jacobi solvers, respectively. Important observations from the results are:

1) PJG outperformed GS with 2-7 times faster computation time. Parallelization of the Jacobi block significantly reduces the effort per iteration compared with GS. In terms of iteration count, PJG convergence is observably close to GS, with a maximum of three more iterations when the Jacobi block size is 3,500. Indeed, there is a trade-off between the block size and the number of iterations, and parameters should be optimized for different problems and hardware architectures.

TABLE 4
Performance Comparison of Iterative Solvers on GPU (Includes Host-GPU-Host Data Transfer Time)

	Average Time (s)				Avg. Time Per Iteration (s)			
n	10K	20K	30K	40K	10K	20K	30K	40K
PJG-500	0.58	1.49	2.86	5.19	0.05	0.12	0.21	0.38
PJG-1500	0.52	1.38	2.84	4.86	0.04	0.11	0.19	0.36
PJG-2500	0.51	1.37	2.87	4.85	0.04	0.11	0.18	0.35
PJG-3500	**0.51**	**1.37**	**2.87**	**4.84**	**0.04**	**0.10**	**0.17**	**0.35**
Jacobi	**0.54**	**1.46**	**2.96**	**5.24**	**0.02**	**0.06**	**0.12**	**0.20**
CGS	1.23	1.79	2.66	3.99	0.57	0.82	1.26	1.84
BiCG	1.22	1.79	2.73	4.26	0.40	0.59	0.91	1.41
BiCGSTAB	1.22	1.78	2.71	4.19	0.81	1.18	1.81	2.78
GMRES	1.22	1.78	2.73	4.16	0.41	0.59	0.91	1.39
QMR	1.23	1.80	2.77	4.21	0.41	0.60	0.92	1.40
TFQMR	1.22	1.78	2.74	4.14	0.63	0.94	1.51	2.28

2) PJG and Jacobi have similar computational time per iteration, but Jacobi requires twice the number of iterations, which results in a around twice the simulation time. Jacobi time can be further improved if more CPU cores are utilized, but in general, this is not always the case due to the limitation in the hardware resources and the problem size. Also, GS has better convergence property and is more stable than Jacobi, especially for weakly diagonally dominant matrices.
3) In the simulations, the Krylov methods perform considerably better than stationary methods in terms of iteration count and time. Iteration count is always problem dependent and can substantially change [5]. However, Krylov methods are computationally cheap and can easily beat PJG even if they need 8-15 times more iterations to converge. According to the MATLAB documentation, Generalized Minimum Residual (GMRES) is the most stable among the Krylov methods, but the work and memory storage requirement grows linearly with iteration count. Provided that GMERS converge in around three iterations in this study, it is possible that the reported average time for this method dramatically changes for more difficult problems.
4) Lastly, MATLAB's A\b command is used to measure the performance of direct solvers. Although these techniques provide an exact solution, the computational effort and memory requirement exponentially grow with the system size. These reasons make iterative solvers more favorable for solving large linear systems.

7.2 Performance Evaluation on Many-Core GPUs

We also analyze the performance of iterative methods on Tesla V100 GPUs for the same test cases. In this scenario, GS only performed 30 percent better than its CPU implementation because of the overhead from memory copy in MATLAB (discussed in Section 4.2) and the sequential nature of this algorithm. Also, we were not able to examine direct methods for systems larger than 10K because the storage requirement of the algorithm was beyond the device memory. The performance for all other methods is shown in Table 4, and findings are as follows:

1) Compared with GS on CPU, PJG performed 13, 33, 58, and 87 times faster for 10K, 20K, 30K, 40K systems, respectively, which is exceptional. To the best of our knowledge, this is the highest reported speedup in the literature at the time of this writing for algorithms that belong to this category.
2) An average of 7 to 14 times speed improvement can be observed between the PJG implementation on GPU and CPU. Accounting for the overhead associated with MATLAB's GPU library for array indexing, we believe that more speedup could be achieved using C and cuBLAS, or CUDA.
3) On the CPU, Jacobi requires twice the simulation time as PJG. However, the average run time of Jacobi on GPU is only slightly higher than PJG. The key reason behind this improvement is the superior performance of GPUs in algebraic operations that resulted in much lower computation time per iteration for Jacobi.
4) Interestingly, the MATLAB built-in Krylov solvers performed 1.5-3.5× slower on GPU compared with their CPU implementation. We investigated this performance degradation by profiling each algorithm but could not determine a common part of the code that causes this issue. In terms of average simulation time on GPU, both PJG and Krylov methods have similar

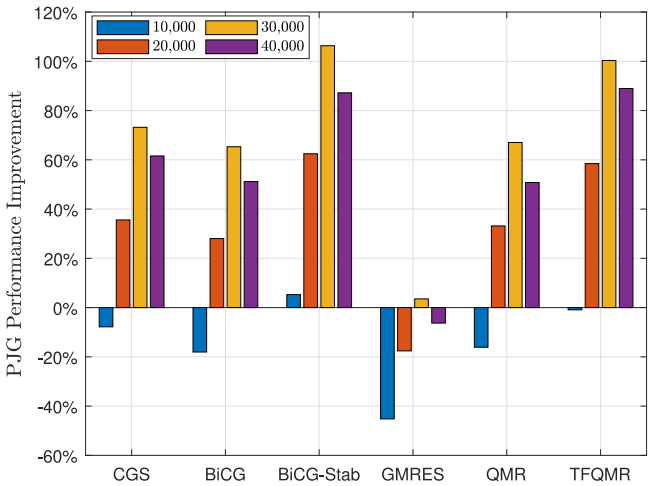


Fig. 3. Comparison of iteration time improvement between PJG-GPU and Krylov-CPU.

TABLE 5
Comparison of Iterative Methods for Solving Quasi- and Block- Tridiagonal Systems (Time and Iteration Number)

n	Quasi-Tridiagonal						Block-Tridiagonal	
	$\approx 50\%$		$\approx 70\%$		$\approx 90\%$		Ref [39]	PJG
	Ref [5]	PJG	Ref [5]	PJG	Ref [5]	PJG		
50,176	0.175 (18)	0.971 (44)	0.378 (37)	0.716 (26)	1.600 (152)	0.740 (12)	0.133 (19)	0.264 (20)
200,704	0.785 (19)	7.017 (76)	1.515 (31)	2.088 (19)	10.52 (125)	3.026 (12)	0.464 (19)	1.077 (19)
640,000	2.288 (17)	73.66 (250)	4.210 (26)	10.58 (30)	16.00 (82)	7.986 (12)	1.471 (20)	3.817 (19)
1,000,000	3.336 (16)	52.42 (113)	8.289 (31)	11.97 (20)	34.65 (95)	15.35 (12)	2.366 (21)	5.787 (18)
4,000,000	15.88 (17)	1831.06 (878)	37.86 (31)	59.29 (22)	163.32 (89)	61.94 (12)	10.39 (45)	25.55 (19)

results. However, PJG is 4-8 times faster in terms of iteration time, which makes it more desirable for problems that are also difficult for Krylov methods.

5) The average iteration time of Krylov methods on CPU is comparable with PJG on GPU, which is worth further investigation. As can be seen in Fig. 3, PJG-3500 outperformed the majority of Krylov techniques for systems larger than 10K. GMRES is the only algorithm that achieved slightly better performance. However, as we mentioned earlier, the work and memory storage requirement for this method grows linearly with iteration count. The legends in Fig. 3 refer to system size.

8 Comparison With Special Solvers

The performance of iterative solvers depends on the structural and spectral properties of the linear system. Some algorithms are specifically designed to solve systems with input matrices that have diagonal, tridiagonal, Hermitian, etc., property to achieve higher performance. In this section, we compare our method's performance with proposed methods in [39] and [5] for solving block-tridiagonal and quasi-tridiagonal linear systems, respectively.

Test matrices are generated using the *gallery* function of Matlab to allow for reproducibility of the results. Sparse block-tridiagonal matrices are derived from the Poisson equation and quasi-tridiagonal systems are created in the form of $A = T + S$, where T is a tridiagonal matrix and S is a random sparse matrix with non-zero values outside the three diagonal. Because the density of sparse matrix S could vary, three problem instances have been produced for each system size where the ratio of the number of non-zero elements (nnz) in S ($nnz(S)$) over $nnz(A)$ is close to 50, 70, and 90 percent. Additionally, because these iterative algorithms are only guaranteed to converge for strictly diagonally dominant matrices, the diagonal values in the resulting quasi-tridiagonal matrix A have been set to the sum of non-diagonal elements in each row. The generated block-tridiagonal matrices are diagonally dominant. Thus, each diagonal element is scaled by a factor of 1.1 to make them strictly diagonally dominant. Also, vector b in $Ax = b$ has been set to an array of all ones.

Different implementations of block-tridiagonal and quasi-tridiagonal solvers have been proposed in the reference papers to improve the execution speed by performing part of the calculations in parallel (i.e., CPU, GPU, block parallel, etc.), which has minor or no impact on the number of iterations. Our comparison aims to observe these algorithm's general behavior for solving tridiagonal systems and not to compare the exact execution time. In fact, it is always possible to further fine-tune these implementations and slightly improve the overall performance. In this study, we only implement and compare the originally proposed algorithms in these papers (Algorithm 1) on multicore CPUs. Additionally, to avoid any performance degradation due to an inefficient code, only high-performance MATLAB functions together with array indexing, are utilized to develop the iterative solvers.

Table 5 shows the performance of each solver in terms of computation time in seconds and the number of iterations to reach the stopping tolerance of 10^{-6} (values in parentheses). Observations are as follows:

1) The Quasi-Tridiagonal solver is significantly faster than PJG when the S matrix is very sparse. However, when the ratio of $nnz(S)$ over $nnz(A)$ approaches one, PJG outperforms the proposed solver. In particular, PJG requires fewer iterations for denser matrices, which explains the improved performance.
2) The block-tridiagonal solver is $2\times - 2.5\times$ faster than PJG in all cases. In fact, the PJG performance declined with an increase in the system size, although the number of iterations remained almost the same.

Although the examined solvers performed better than PJG in most cases, note that they only work for strictly diagonally dominant systems with unique matrix structure while PJG only requires the matrix to be strictly diagonally dominant and does not need any matrix structure. That is, PJG is designed for a more general case.

9 Application in Power System Analysis

To further demonstrate the advantage of the proposed algorithm, we apply it to the power flow (PF) problem in the electric industry. PF is one of the most widely used numerical studies in the planning and steady-state analysis of power systems. The primary objective of PF calculations is to determine the voltage values at different nodes, real and reactive

TABLE 6
Power System Test Cases

Test Case	System Size	nnz
IEEE 118	118 x 118	476
IEEE 300	300 x 300	1,118
PEGASE 1,354	1354 x 1354	4,774
POLISH 2,736	2736 x 2736	9,262

TABLE 7
Performance Comparison of Gauss-Seidel With PJG for Solving the PF Problem

Case	Iterations		Time (s)	
	GS	PJG	GS	PJG
IEEE 118	282	315	0.24	0.07 **(3.4x)**
IEEE 300	1,537	1,851	3.33	0.37 **(9.0x)**
PEGASE 1354	10,184	11,915	291.7	8.3 **(34.9x)**
POLISH 2736	34,766	51,236	4768.9	54.6 **(87.3x)**

power flows in transmission lines, and network losses. These variables are found by solving a set of non-linear nodal power balance equations. Among different techniques to obtain the solution of PF, Newton-Raphson, Fast Decoupled, and Gauss-Seidel are the most common. These methods are well discussed in power system analysis books (see, e.g., [40]).

The convergence of GS becomes relatively slow for large power systems due to its sequential nature. To evaluate the effectiveness of the proposed algorithm in addressing this issue, we integrated the PJG code into the MATPOWER[41] application. MATPOWER is a well-known, high performance, and open-source power system simulation package based on MATLAB. The benchmark test cases in Table 6 are available in the MATPOWER 7.0 package. Typically, power system matrices are very sparse because each bus is connected to an average of three other nodes. Comparing the number of nonzero elements to the system size indicates a sparsity level of 70 to 99 percent in the test cases.

By default, MATPOWER utilizes sparse matrix storage formats to maximize performance. This matter will not impact the implementation of our algorithm since MATLAB performs all operations for sparse matrices seamlessly. After carrying out multiple simulations, we observed that using 50 percent of the system size as the length of the Jacobi block will result in peak performance for almost all cases. Since these systems are highly sparse, increasing the number of threads to more than four did not improve the runtime. Results in Table 7 show that despite the increase in the number of iterations, PJG is considerably faster than the original GS. For instance, solving the POLISH system using GS required 79 minutes while PJG completed the computation in less than one minute.

10 Conclusion

In this research, we develop a new iterative method based on Jacobi and GS for solving linear systems of equations. The proposed algorithm's convergence rate is observably close to GS, but the performance is up to 7× faster on multicore CPUs and 87× on many-core GPUs for dense systems in this study. The performance can further be improved by tuning the number of threads and adjusting the Jacobi block size. Selecting the proper solver is problem dependent and requires prior knowledge of the system. For the scientific problems that are directly solved by the GS method, such as power flow analysis, we recommend using the PJG algorithm. For other problems, if a GPU is available, our proposed solver is a good candidate; otherwise, we suggest using Krylov techniques because of their small computational requirement. Additionally, parallel Jacobi is the simplest approach to compute a preconditioner, but it is not very useful due to the resulting larger number of iterations compared to other variants such as SSOR. In this case, PJG is a good alternative since it has superior convergence property and comparable execution time. Our algorithm's bottleneck is the speed of matrix-vector multiplication, and we believe that with the advancement of computing resources, PJG may become a viable choice in the future. Further work will focus on developing our algorithm using C, Intel MKL, and cuBLAS to obtain maximum performance since MATLAB overhead, especially for GPU libraries, is high. Our future work will also cover solving sparse systems on GPUs.

Acknowledgments

The authors would like to thank Dr. Timo Heister and Dr. Edward R. Collins for their valuable comments. This research was supported in part by the NSF-MRI Award 1725573.

References

[1] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics*, vol. 37. Berlin, Germany: Springer, 2010.
[2] D. M. Young, *Iterative Solution of Large Linear Systems*. Amsterdam, The Netherlands: Elsevier, 2014.
[3] M. Kawai, T. Iwashita, H. Nakashima, and O. Marques, "Parallel smoother based on block red-black ordering for multigrid poisson solver," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.*, 2012, pp. 292–299.
[4] I. Yamazaki, J. Kurzak, P. Wu, M. Zounon, and J. Dongarra, "Symmetric indefinite linear solver using OpenMP task on multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1879–1892, Aug. 2018.
[5] K. Li, W. Yang, and K. Li, "A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2795–2808, Oct. 2016.
[6] D. Zaitsev, S. Tomov, and J. Dongarra, "Solving linear diophantine systems on parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1158–1169, May 2019.
[7] R. O. Selvitopi, M. M. Ozdal, and C. Aykanat, "A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 632–645, Mar. 2015.
[8] T. Tsuburaya, Y. Okamoto, and Z. Meng, "Improvement of block IC preconditioner using fill-in technique for linear systems derived from finite-element method including thin elements," *IEEE Trans. Magn.*, vol. 54, no. 3, pp. 1–4, Mar. 2018.
[9] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*. Berlin, Germany: Springer, 2013.
[10] Z. Wu, Y. Xue, X. You, and C. Zhang, "Hardware efficient detection for massive MIMO uplink with parallel gauss-seidel method," in *Proc. 22nd Int. Conf. Digit. Signal Process.*, 2017, pp. 1–5.
[11] A. Bartel, M. Brunk, M. Gunther, and S. Schops, "Dynamic iteration for coupled problems of electric circuits and distributed devices," *SIAM J. Sci. Comput.*, vol. 35, no. 2, pp. B315–B335, 2013.
[12] Y. Shang, "A distributed memory parallel gauss–seidel algorithm for linear algebraic systems," *Comput. Math. Appl.*, vol. 57, no. 8, pp. 1369–1376, 2009.
[13] Z. Feng, Z. Zeng, and P. Li, "Parallel on-chip power distribution network analysis on multi-core-multi-GPU platforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 10, pp. 1823–1836, Oct. 2011.
[14] H. Courtecuisse and J. Allard, "Parallel dense gauss-seidel algorithm on many-core processors," in *Proc. 11th IEEE Int. Conf. High Perform. Comput. Commun.*, 2009, pp. 139–147.
[15] P. Di, H. Wu, J. Xue, F. Wang, and C. Yang, "Parallelizing SOR for GPGPUs using alternate loop tiling," *Parallel Comput.*, vol. 38, no. 6/7, pp. 310–328, 2012.
[16] D. P. Koester, S. Ranka, and G. C. Fox, "A parallel gauss-seidel algorithm for sparse power system matrices," in *Proc. ACM/IEEE Conf. Supercomputing*, 1994, pp. 184–193.

[17] P. Amodio and F. Mazzia, "A parallel Gauss–Seidel method for block tridiagonal linear systems," *SIAM J. Sci. Comput.*, vol. 16, no. 6, pp. 1451–1461, 1995.
[18] G. H. Golub and J. M. Ortega, *Scientific Computing: An Introduction With Parallel Computing*. Amsterdam, The Netherlands: Elsevier, 2014.
[19] A. Ahmadi, S. Jin, M. C. Smith, E. R. Collins, and A. Goudarzi, "Parallel power flow based on OpenMP," in *Proc. North Amer. Power Symp.*, 2018, pp. 1–6.
[20] R. Tavakoli and P. Davami, "A new parallel Gauss-Seidel method based on alternating group explicit method and domain decomposition method," *Appl. Math. Comput.*, vol. 188, no. 1, pp. 713–719, 2007.
[21] M. F. Adams, "A distributed memory unstructured gauss-seidel algorithm for multigrid smoothers," in *Proc. ACM/IEEE Conf. Supercomputing*, 2001, pp. 14–14.
[22] J. Zhang, "Acceleration of five-point red-black Gauss-Seidel in multigrid for poisson equation," *Appl. Math. Comput.*, vol. 80, no. 1, pp. 73–93, 1996.
[23] K. S. Kang, "Scalable implementation of the parallel multigrid method on massively parallel computers," *Comput. Math. Appl.*, vol. 70, no. 11, pp. 2701–2708, 2015.
[24] X. Yang and R. Mittal, "Efficient relaxed-jacobi smoothers for multigrid on parallel computers," *J. Comput. Phys.*, vol. 332, pp. 135–142, 2017.
[25] M. Rodriguez, B. Philip, Z. Wang, and M. Berrill, "Block-relaxation methods for 3D constant-coefficient stencils on GPUs and multicore CPUs," 2012, *arXiv:1208.1975*.
[26] R. Bagnara, "A unified proof for the convergence of jacobi and gauss-seidel methods," *SIAM Rev.*, vol. 37, no. 1, pp. 93–95, 1995. [Online]. Available: https://www.jstor.org/stable/2132758
[27] W. Li, L. Elsner, and L. Lu, "Comparisons of spectral radii and the theorem of stein–rosenberg," *Linear Algebra Appl.*, vol. 348, no. 1/3, pp. 283–287, 2002.
[28] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.
[29] D. Bertaccini and F. Durastante, *Iterative Methods and Preconditioning for Large and Sparse Linear Systems With Applications*. Boca Raton, FL, USA: CRC Press, 2018.
[30] K. Abe, T. Sogabe, S. Fujino, and S. Zhang, "A product-type krylov subspace method based on conjugate residual method for nonsymmetric coefficient matrices," *IPSJ Trans. Adv. Comput. Syst.*, vol. 48, pp. 11–21, 2007.
[31] W. Yang, K. Li, and K. Li, "A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 104, pp. 49–60, 2017.
[32] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.
[33] A. Elafrou, G. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on modern multi-and many-core processors," in *Proc. 46th Int. Conf. Parallel Process.*, 2017, pp. 292–301.
[34] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 117–126.
[35] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive SpMV library for multicore and many-core architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, pp. 1–25, 2018.
[36] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1112–1123, May 2014.
[37] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, May 2014.
[38] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2003. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003
[39] W. Yang, K. Li, and K. Li, "A parallel solving method for block-tridiagonal equations on CPU–GPU heterogeneous computing systems," *The J. Supercomputing*, vol. 73, no. 5, pp. 1760–1781, 2017.
[40] J. J. Grainger and W. D. Stevenson, *Power System Analysis*. New York, NY, USA: McGraw-Hill, 1994.
[41] R. D. Zimmerman, C. E. Murillo-Snchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

Afshin Ahmadi (Student Member, IEEE) received the MS degree in electrical engineering from the University of the Philippines Diliman, Philippines, in 2012, and the PhD degree in computer engineering from the Clemson University, Clemson, South Carolina, in 2020. He is currently a power system application developer at Electric Reliability council of Texas (ERCOT). His main research interests are high performance computing in power and energy systems, smart grid, renewable energy, and power system planning and optimization.

Felice Manganiello received the MS degree from the University of Pisa, Italy, in 2005, and the PhD degree from the University of Zurich, Switzerland, in 2011, all in mathematics. He is an associate professor with the School of Mathematical and Statistical Sciences, Clemson University, Clemson, South Carolina. His research interests are in applied algebra and communication with focus on coding theory, network coding, and cryptography. He was the recipient of a Swiss National Science Foundation Postdoctoral Fellowship from 2011 to 2013, which he spent at the Department of Electrical and Computer Engineering, University of Toronto, Canada. He was awarded with one of the 2019 Simons Visiting Professorships and during the academic year 2019/20, he held a visiting scholar position at the Cybersecurity Research Lab of Ryerson University, Toronto, Canada.

Amin Khademi received the PhD degree from the University of Pittsburgh, Pittsburgh, Pennsylvania. He is an associate professor of industrial engineering at Clemson University, Clemson, South Carolina. He is interested in decision making under uncertainty and optimal learning. He has worked on applications such as adaptive design of clinical trials, optimal allocation rules for organ transplantation, and infectious disease control. He is a recipient of NSF CAREER Award.

Melissa C. Smith (Senior Member, IEEE) received the BS and MS degrees in electrical engineering from Florida State University, Tallahassee, Florida, in 1993 and 1994, respectively, and the PhD degree in electrical engineering from the University of Tennessee, Knoxville, Tennessee, in 2003. She is currently an associate professor of electrical and computer engineering at Clemson University, Clemson, South Carolina. She has more than 25 years of experience developing and implementing scientific workloads and machine learning applications across multiple domains, including 12 years as a research associate at Oak Ridge National Laboratory. Her current research focuses on the performance analysis and optimization of emerging heterogeneous computing architectures (GPGPU- and FPGA-based systems) for various application domains including machine learning, high-performance or real-time embedded applications, and image processing. Her group collaborates with researchers in other fields to develop new approaches to the application/architecture interface providing interdisciplinary solutions that enable new scientific advancements and/or capabilities.