Parallel Computing xxx (xxxx) xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers

Kasia Świrydowicz [b,a], Eric Darve [c], Wesley Jones [b], Jonathan Maack [b], Shaked Regev [c], Michael A. Saunders [c], Stephen J. Thomas [b], Slaven Peleš [a,*]

[a] Pacific Northwest National Laboratory, United States of America
[b] National Renewable Energy Laboratory, United States of America
[c] Stanford University, United States of America

ARTICLE INFO

Keywords:
Sparse linear equations
Solvers
GPU
Grid optimization

ABSTRACT

The linear equations that arise in interior methods for constrained optimization are sparse symmetric indefinite, and they become extremely ill-conditioned as the interior method converges. These linear systems present a challenge for existing solver frameworks based on sparse LU or LDL^T decompositions. We benchmark five well known direct linear solver packages on CPU- and GPU-based hardware, using matrices extracted from power grid optimization problems. The achieved solution accuracy varies greatly among the packages. None of the tested packages delivers significant GPU acceleration for our test cases. For completeness of the comparison we include results for MA57, which is one of the most efficient and reliable CPU solvers for this class of problem.

1. Introduction

Interior-point methods [1,2] provide an important tool for nonlinear optimization and are used widely for engineering design [3] and discovery from experimental data [4]. Most of the computational cost for interior methods comes from solving underlying linear equations. Indeed, interior methods came to prominence with the emergence of robust sparse linear solving techniques. For a general nonconvex optimization problem, the linear systems are sparse symmetric indefinite and typically ill-conditioned. Furthermore, many implementations require the linear solver to provide matrix inertia information (the number of positive, zero, and negative eigenvalues). Relatively few presently available linear solvers meet the requirements of interior methods.

We review the current state-of-the-art, with particular emphasis on solvers that can run on hardware accelerators such as GPUs. Our investigation is motivated by power grid optimization work within the US Department of Energy's Exascale Computing Project. In a large-scale security-constrained optimal power flow analysis, several coupled optimal power flow problems are solved simultaneously [5,6]. Each coupled unit runs its own interior method computation. Because more than 95% of the computational power on new and emerging computational platforms is typically in accelerators, it is desirable to run each interior method optimization, including linear solves, on those devices. We identify technical gaps in this area and suggest possible paths to bridge those gaps.

The paper is organized as follows: In Section 2, we provide a brief description of the algorithms employed in interior point method and we formulate the problem; we also present an overview of the available literature, and place the problem in a broader context. In Section 3, we discuss the main features of the linear solver packages. In Section 4, we describe the test cases. In Section 5, we present test results for each software package. These results are summarized in Section 6. In Section 7, we state conclusions and provide ideas for future work.

2. Problem statement

We are concerned with solving large linear systems

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

where A is a symmetric indefinite $n \times n$ sparse matrix, b is an $n \times 1$ right-hand-side (RHS) vector, and x is the $n \times 1$ solution vector. The *residual vector* is $r = b - Ax$.

We generate our test problems for linear solvers using the Ipopt optimization package [7], which is considered to be the "gold standard" for freely available interior method software. We instrumented Ipopt to save A and b, which are handed off to the linear solver every time A is updated. Each series of test cases corresponds to a successful optimization solver run, i.e., Ipopt converged to the optimal solution. Ipopt used MA57 [8] as the linear solver in these computations.

* Corresponding author.
E-mail address: slaven.peles@pnnl.gov (S. Peleš).

https://doi.org/10.1016/j.parco.2021.102870
Received 4 November 2020; Received in revised form 31 July 2021; Accepted 4 November 2021
Available online 11 December 2021

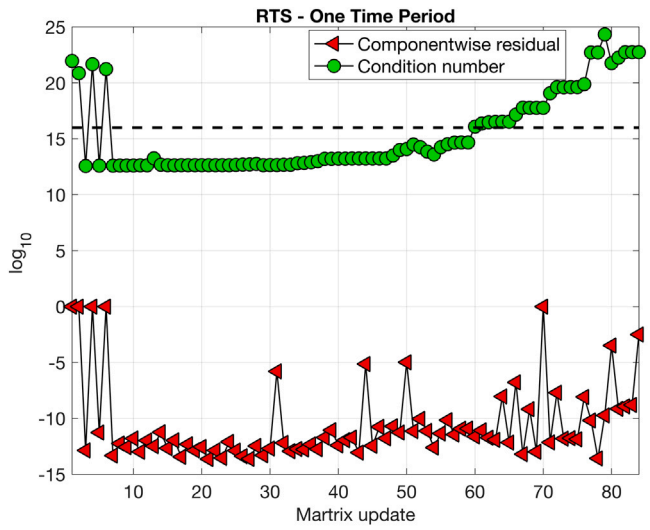

Fig. 1. Condition number and relative backward error for the linear systems solved by Ipopt during optimal power flow analysis for a 73-bus grid. The optimization problem has 1071 variables, 655 equality constraints, and 240 inequality constraints. Each matrix is 2237 × 2237 with 11,297 nonzeros. The dashed line denotes the reciprocal of floating-point double precision.

The interior method finds the minimum of a scalar objective function given equality and inequality constraints. The method extends the minimization problem with barrier functions in order to enforce inequality constraints and bounds on the variables. The extended problem is solved using a variant of Newton's method. A continuation is applied to reduce barrier parameters gradually and bring the extended problem as close to the original problem as possible. A nonlinear problem is (re)solved at each continuation step. The problem becomes singular when barrier parameters are zero, so the continuation needs to stop when the extended problem solution is close enough to the solution of the original problem, but before the problem becomes too ill-conditioned. This is why having a robust linear solver is critical for successful implementation of the interior method. For more details, the reader is referred to [7].

A typical sequence of linear systems generated by Ipopt is depicted in Fig. 1, where the matrix condition number is superposed with the component-wise relative backward error computed with STRUMPACK (see Table 3 for the exact formula) for each of 84 matrix updates during the optimization solver run. Note that four of the first six iterations produce a matrix with condition number larger than 10^{20} (numerically singular). These correspond to failed iteration steps of the linesearch Newton method employed by Ipopt. The optimization solver can recover from this, for example, by adjusting the steplength. The condition number increases significantly somewhere after the 60th matrix update—a normal feature of the continuation algorithm employed by Ipopt. It is at this stage that a robust linear solver is essential. All of our test cases are selected from the last few successful Ipopt iterations.

Literature review

There is a vast body of research on benchmarking optimization solvers for their effectiveness and performance [9,10]. However, not much attention has been devoted to the linear solvers employed within interior methods for optimization. The literature can be divided into three major categories: (a) papers that compare different optimization solvers without isolating the linear solver performance, including parameters and algorithm selection, (b) literature focusing on linear solver selection and performance, and (c) papers focusing on linear solvers for symmetric indefinite systems.

In papers from category (a), the CUTE, CUTEr, or CUTEst [11] frameworks are often employed. For example, Schenk et al. [12] applies a combinatorial (symmetric weighted matching based) pre-processing step to lower the condition number of the matrices arising in Ipopt, followed by a direct solver that uses supernodal static pivoting. The authors test this approach with CUTEr.

For category (b), authors are primarily concerned with finding suitable preconditioners [13–15] for solving the linear systems iteratively. We cannot take advantage of these results, however, because the problems described herein are extremely ill-conditioned and thus require a direct linear solver, possibly with iterative refinement (IR), to achieve acceptable backward error levels.

For category (c), there is a long lineage of research available because the scientific community has been interested in efficient numerical linear solvers for (dense and sparse) symmetric indefinite systems dating back to the 1970s [16,17]. Bunch and Kaufman [18] introduced the pivoting strategy currently used in LAPACK for dense symmetric indefinite systems. This scheme was later improved in [19], where the more stable *bounded Bunch–Kaufman pivoting* is proposed. Duff et al. pioneered much of the field with a series of papers focused on symmetric indefinite systems [8,20–23]. Gould et al. [24] reviewed the sparse symmetric solvers that are part of the mathematical software library HSL (formerly the Harwell Subroutine Library) and concluded that MA57 [8] is currently the best general-purpose package in HSL. Hogg et al. [25] improved the pivoting strategy of [22] for challenging matrices. Duff et al. [26] focused on pivoting strategies and new approaches designed for parallel computers. Similarly, several recent papers [27–29] have considered improved algorithms targeting modern multicore processors and GPUs. Becker et al. [30] proposed a supernode-Bunch–Kaufman pivoting strategy that enables high-performance implementations on modern multicore processors. For a survey of the literature on sparse symmetric systems, see Davis et al. [31]. For a benchmark of several sparse symmetric direct solvers, see Gould et al. [32].

One of the few published resources, where different linear solver strategies are compared within Ipopt, is the PhD thesis of Jonathan Hogg [33], who compares the performance of four linear solver frameworks: MA57, HSL_MA77 [34], HSL_MA79 [35], and Pardiso [36]. Hogg devotes some consideration to the IR approach, and also analyzes parallel performance and scaling of the linear solver strategies, although when the thesis was published, the use of GPUs in scientific computing was still in its infancy.

Our study is similar to the work of Tasseff et al. [37] and can be viewed as an extension. In that study, the authors analyze the performance of nine different linear solver frameworks (MA27, MA57, HSL_MA77, HSL_MA86, HSL_MA97, MUMPS, PARDISO, SPRAL and WSMP) that can be applied in Ipopt to solve symmetric indefinite linear systems. All the linear solver packages taken into consideration by [37] are based on a symmetric LDL^T factorization. One of them is SPRAL [29], which is also employed in our study.

Several factors differentiate our study from [37]: (a) we consider linear solver software packages that use an unsymmetric LU factorization (which is potentially more stable), (b) we analyze GPU performance of the packages and compare the GPU performance to the CPU performance (the only GPU-accelerated software package tested in [37] is SPRAL), (c) we focus on only one, not multiple test cases (they all come from the same application), (d) we extract matrices and right-hand sides from Ipopt instead of running the entire Ipopt code for each test case, and (e) in our comparisons, we not only compare performance but also check the solution accuracy.

The major contribution of this paper is an in-depth comparison of well known linear solver packages for symmetric indefinite saddle-point systems arising from an interior method for optimization. The analysis includes CPU scaling (if applicable), GPU performance, and accuracy.

3. Linear solver packages

We compare the performance and usability of five widely available linear solver packages: SuperLU [38,39], STRUMPACK [40,41], SPRAL/SSIDS [42], cuSolver [43], PaStiX [44], with MA57 [8] as a reference. Of these packages, cuSolver is a proprietary product that can be obtained free of charge. MA57 is licensed as part of the HSL library [45], while the others are open source software. We identified these solvers as best candidates to be used within the interior method context, based on results available in peer-reviewed literature. We found that GPU support for PaStiX is less mature than for the other solvers and were not able to run all of our tests with it. PaStiX performance results are therefore not presented here, but partial results are included in the supplementary material at https://github.com/kswirydo/linear-solver-review.

3.1. SuperLU

SuperLU is the most mature and established of the five packages evaluated. For the purpose of generating results presented in this study we used SuperLU_dist, which is accelerated by MPI and CUDA.

SuperLU is a direct linear solver for large, sparse unsymmetric systems. It employs Gaussian elimination (super-nodal) to compute the factorization

$$P_r D_r A D_c P_c = LU,$$

where D_r and D_c are diagonal matrices used to equilibrate (scale) the matrix A, and P_c, P_r are column/row permutation matrices. If the solution is not accurate enough, it follows with an iterative refinement (IR) phase based on the Richardson iteration.

Unlike the serial version of SuperLU (which uses partial pivoting), the distributed version uses static pivoting based on graph matching [46], which has better scaling properties [47]. The pivoting strategy determines the choice of P_r. Distributed SuperLU employs an $A^T + A$-based sparsity ordering, which determines P_c. SuperLU reorders both columns and rows; however, the user does not have the freedom to select a particular algorithm.

3.2. STRUMPACK

STRUMPACK (STRUctured Matrix PACKage was developed by Pieter Ghysels, Xiaoye Li, Yang Liu, Lisa Claus and other contributors at Lawrence Berkeley National Laboratory. STRUMPACK is intended to solve sparse and dense systems. STRUMPACK is targeted toward matrices that exhibit an underlying low-rank structure (such as block structure) that is exploited to construct the L and U factors. Unlike in SuperLU, the computed factors can be used as a preconditioner within an iterative linear solver. A multifrontal factorization is employed.

STRUMPACK uses MC64 [48] to permute and scale matrix columns (as does SuperLU). It follows the MC64 step with a nested dissection re-ordering [49], which controls the amount of fill-in. The third pre-processing step is a reordering that helps reduce the ranks used for the HSS (hierarchically semi-separable) compression steps. After the pre-processing, there is a symbolic factorization phase to construct the elimination tree, and then actual multifrontal factorization. During the factorization, HSS is used to numerically compress the fronts. This strategy is typically only applied to large fronts (near the end of the factorization). Depending on the settings and the matrix, the solve can be done through back substitution combined with either IR or an iterative solver (GMRES or BiCGStab).

STRUMPACK is parallelized through MPI and CUDA. It uses cuBLAS and cuSolver, and relies on SLATE for LAPACK functions.

3.3. cuSolver

`cuSolver` is a library provided by NVIDIA and distributed with CUDA Toolkit The functions provided by the library allow the user to solve a system (or multiple systems) of linear equations $Ax = b$, where A does not have to be square. It computes a QR, LU, or LDL^{T} factorization of A, then performs upper and lower triangular solves. The library consists of three main components: `cuSolverDN` (for dense matrices), `cuSolverSP` (for sparse matrices) and `cuSolverRF` (for matrix series in which the matrix values change but the nonzero pattern stays the same). The library comes with many helper functions and provides several algorithms for solving linear systems (using LU, QR, Cholesky, or least squares).

In this study, we tested `cuSolverSP` because our matrices are sparse. However, the sparse interface appears to be less mature and to have limited functionality. For example, some of the functions are not implemented to run on GPUs, and LDL^{T} is available only through the dense interface.

cuSolver is proprietary software and its source code is not available. The user supplies the matrix and right-hand side and the functions return a solution along with the error. The only parameter that can be chosen is the reordering with options: RCM (reverse Cuthill–McKee), AMD (approximate minimum degree), Metis, or none.

3.4. SSIDS

SSIDS is part of a large library known as SPRAL (Sparse Parallel Robust Algorithms Library). SPRAL was developed by the Computational Mathematics Group at Rutherford Appleton Laboratory in the UK.

Unlike SuperLU and STRUMPACK, SSIDS uses an LDL^{T} factorization to solve $Ax = b$ with A symmetric but indefinite. In a first pass, SSIDS computes the decomposition $A = PLD(PL)^T$, where P is a permutation matrix, L is unit lower triangular, and D is either diagonal or block diagonal with 1×1 or 2×2 blocks. A clear advantage of the LDL^{T}factorization is that it requires less storage, as only one triangular factor needs to be stored. SSIDS applies an *a posteriori* threshold pivoting [29] to implement a scalable GPU pivoting strategy.

3.5. Comparison

The information concerning the basic functionality of the packages is collected in Table 1.

4. Test case descriptions

All our test cases were generated by running optimal power flow analysis for synthetic power grid models developed by Texas A&M University. The analysis was performed using Ipopt with MA57 as the linear solver. We present results for three families of test cases named ACTIVSg 2000, 10k and 70k [50]. They closely resemble grids of Texas, US Western Interconnection and US Eastern Interconnection, respectively, and are representative of proprietary grid models used by power industry. Eastern Interconnection is the largest publicly available transmission grid model. Details on how these models are created are provided in [51]. Each family contains a set of linear problems (1) from the last few Ipopt iterations. The families of matrices and a more detailed description of the optimization problems are available in [52]. The test case family characteristics (matrix size, number of matrices in the family, average number of nonzeros) are given in Table 2.

The results for other test cases are available in the repository: https://github.com/kswirydo/linear-solver-review.

5. Test results

All performance results presented in this section were obtained on one node of Summit, the OLCF (Oak Ridge Leadership Computing Facility) supercomputer [53]. Each node of Summit is equipped with two Power9 CPUs, each having 21 cores and six NVIDIA V100 GPUs. In all the test cases, we either used one rank and one GPU or multiple ranks with each rank having its own GPU.

Table 1
Summary of linear solver package capabilities. The line labeled *Preconditioner* says if the packages compute an LU or LDL^T decomposition to be used as a preconditioner for an iterative linear solver. The capitalized names in the last row are commonly used names of BLAS routines.

Solver	SuperLU	STRUMPACK	SSIDS	cuSolver	PaStiX
Exploits matrix symmetry	NO	NO	YES	NO	YES
Primary algorithm	LU	LU	LDL^T	LU, QR	LDL^T
Follows with IR	YES	YES	NO	NO	YES
Preconditioner	NO	YES	NO	NO	YES
Supports reading RHS	NO	NO	NO	NO	NO
GPU-accelerated	YES	YES	YES	YES	YES
Multi-node	YES	YES	NO	NO	YES
Open source	YES	YES	YES	NO	YES
GPU-accelerated parts	GEMM	GETRF, LASWP, GETRS, TRSM	SYRK, reord., fact., solve	QR fact., QR solve	fact.

Table 2
Characteristics of the problems evaluated. Because the matrices are symmetric, only the triangular part of a matrix is stored in a Matrix Market file. *NNZ un* shows the number of nonzeros after unpacking (forming the full matrix with all of its nonzeros). The last column shows the number of matrices in each test case.

		Matrix characteristics				
Model	Grid	Rows	Cols	NNZ	NNZ un	#
2000	ACTIVSg2000	56k	56k	150k	268k	3
10k	ACTIVSg10k	238k	238k	626k.8	1112k	5
70k	ACTIVSg70k	1640k	1640k	4320k	7672k	5

Table 3
Different errors computed by the solver packages to measure solution accuracy. By x_{true} we understand a known, true solution used in SuperLU testing.

Name	Error formula	Used in
Relative forward error (RFE)	$\frac{\|x - x_{\text{exact}}\|_\infty}{\|x\|_\infty}$	SuperLU
Normwise Backward Residual Error (NBRE) v1	$\max_i \frac{\|r_i\|}{s_i}$, where $s = \|A\|\|x\| + \|b\|$	SuperLU
Componentwise Relative Backward Error (CRBE)	$\max_{i=1,\ldots,n} \frac{\left\|b_i - \sum_{j=1}^n a_{ij}x_j\right\|}{\|b_i\| + \sum_{j=1}^n \|a_{ij}\|\|x_j\|}$	STRUMPACK
Relative residual	$\frac{\|b - Ax\|}{\|b\|}$	STRUMPACK, cuSolver, SSIDS, PaStiX
Normwise Backward Residual Error (NBRE) v2	$\frac{\|b - Ax\|_\infty}{\|\|b\| + \|A\|\|x\|\|_\infty}$	cuSolver, SSIDS, PaStiX

For each of the linear solver packages, we first present the results in terms of achieved (backward) error, followed by a short discussion. Each of the solver packages measures the solution accuracy differently. Table 3 collects the error descriptions. Afterwards, we provide the performance results.

The results for all the tested packages are shown in Fig. 2 together with results obtained using MA57, which is considered a standard solver for the type of linear systems discussed in this manuscript. In the figure, only best results obtained using each of the packages are shown.

The results in the Figure are averaged per system. The systems for which the solver failed are excluded from the averages.

5.1. SuperLU

5.1.1. Convergence and the quality of solutions

The SuperLU driver provided with the library computes NBRE v1 and RFE (see Table 3). Obtaining the second quantity is not possible when the exact solution is not known *a priori*. Therefore, two sets of tests have been performed. In the first set, the right-hand side was generated by SuperLU based on an exact solution vector consisting of 1s. This provides important information about the factorization quality (we call this *default RHS*). The second set was performed using right-hand sides matching the matrices (extracted from Ipopt) and read from

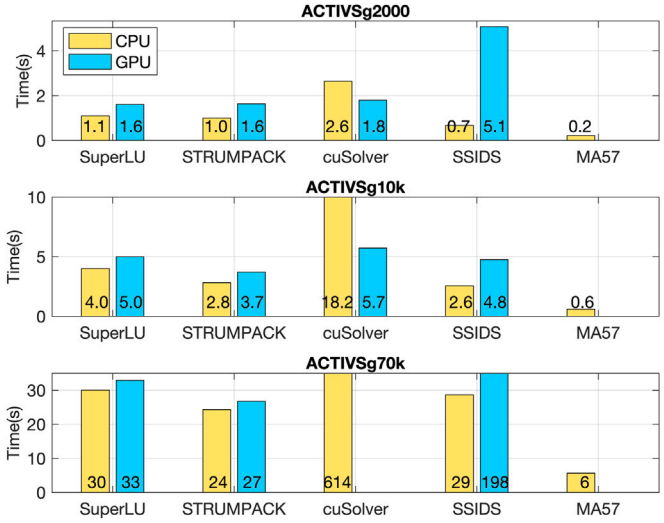

Fig. 2. Comparison of the performance of all the solver packages tested. Times shown on the bars do not include matrix I/O nor error computation. Left: times without GPU acceleration. Right: times with GPU acceleration.

Table 4
The error levels achieved by SuperLU_dist. *NNZ inc*: increase in nonzeros in L and U in relation to nonzeros in unpacked matrix; *Av. RFE*: average relative forward error (average through all matrices in each test case family); *Av. NBRE*: average NBRE v1. (error reported after last step of IR).

Name	NNZ inc.	av. RFE	av. NBRE
ACTIVSg2000	7.12x	$5.38 \cdot 10^{-7}$	$2.73 \cdot 10^{-15}$
ACTIVSg10k	6.52x	$2.74 \cdot 10^{-8}$	$5.78 \cdot 10^{-6}$
ACTIVSg70k	6.85x	$7.20 \cdot 10^{-3}$	1.045

files. In the second case, the reported error is the NBRE v1 achieved after IR.

If a zero pivot is encountered, SuperLU stops the LU factorization. In such a case, if the exact solution is available, the error equals 1 (vector of computed solutions is initialized to b) and no IR is performed.

The results are collected in Table 4, where:

NNZ inc. is the ratio of the nonzeros in the L and U factors to the nonzeros in the original matrix;
Av. RFE is the average relative forward error (average through all matrices in each test case family);
Av. NBRE is the average norm-wise relative backward error (reported after the last step of IR).

The matrices for which SuperLU failed were excluded from the averages.

From Table 4 one can infer that SuperLU produces solutions with sufficient accuracy for the optimization algorithm, even if the problem is close to being singular, and fails only on singular cases. We expect that these cases would be handled by the optimization algorithm

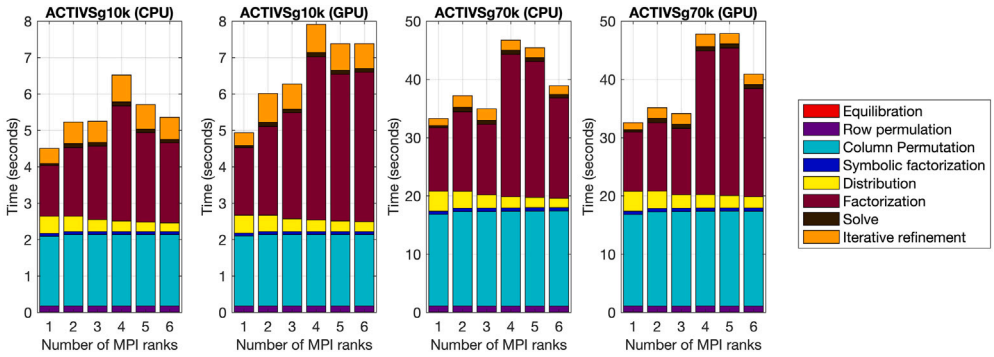

Fig. 3. SuperLU performance results using default right-hand sides. All times are given in seconds and all runs have one MPI rank per GPU. Left: results for ACTIVs10k, without and with GPU acceleration. Right: results for ACTIVSg70k without and with GPU acceleration.

Table 5
SuperLU results for ACTIVSg2000 on one rank and one GPU. Eq = equilibration, RP = row permutation, CP = column permutation, SF = symbolic factorization, Dist = distribution, Fact = factorization, Ref = refinement. TOT is the total time (without I/O). The results show that only Fact and Ref use the GPU.

SuperLU									
	Eq	RP	CP	SF	Dist	Fact	Sol	Ref	TOT
CPU	0.00	0.04	0.41	0.02	0.12	0.42	0.01	0.08	1.10
GPU	0.00	0.04	0.41	0.02	0.13	0.91	0.01	0.04	1.56

controls, which adjust the iteration step and recompute the search direction.

5.1.2. Performance results

The computations performed by SuperLU can be divided into 8 phases: (a) equilibration (scaling), (b) row permutation, (c) column permutation, (d) symbolic factorization, (e) data distribution (through MPI), (f) (full) factorization, (g) solve, (h) IR.

SuperLU can be compiled with or without GPU acceleration. Only phases (f) and (h) use acceleration. Fig. 3 and Table 5 give results for both variants. In this section, we only show results that were obtained using a fabricated RHS (generated by SuperLU). The extended set of results, including the results with correct RHS, are available in the repository.

While analyzing Fig. 3, we observe that certain parts of the linear solver scale quite well (e.g., the distribution for ACTIVSg70k decreases with the increase in the number of ranks). In the majority of cases, the highest computational cost is (as expected) the matrix factorization; however, for low MPI rank counts the computation is dominated by the column permutation step. A similar pattern can be observed for the GPU accelerated performance tests. The GPU times are much worse than CPU times for the smaller cases and comparable to the CPU results for the two largest test cases (as expected from the matrix sizes). The reason for the time increase observed in the smallest test cases is due to data allocation/deallocation and movement. SuperLU uses a threshold value to decide whether the matrix is large enough to apply GPU acceleration. However, if the GPU accelerated version was used, SuperLU allocated and deallocated GPU memory regardless of the matrix size.

To understand which part of the solve process occurs on the GPU, we launched compute jobs with `nvprof` to obtain profiling results. We consider three categories of GPU operations: computations (computational kernels, either written by the SuperLU team or calling libraries such as cuBLAS), communication (explicit copies from Host to Device and from Device to Host) and API operations (cudaFree, cudaMalloc, cudaStreamCreate/Synchronize, cudaMallocHost, cudaMallocManaged, etc.). All the detailed timing tests were performed on one GPU. The results are presented in Fig. 4.

A detailed GPU profiling explains why the time increases significantly compared to the CPU-only case; the increase is largely due to GPU memory allocations, copies, and frees leading to a large startup cost.

We conclude that for all of our matrices, the GPU does not improve the overall performance. While it can be argued whether or not the API calls should be a part of the statistics (code analysis shows that these occur inside the linear solver functions), it is clear that the computations on the GPU are dominated by the cost of communication, as shown in Fig. 4. Both are several orders of magnitude cheaper than the API operations (see the figure caption).

5.2. STRUMPACK

5.2.1. Convergence and quality of the solution

STRUMPACK evaluates accuracy using component-wise relative backward error (CRBE) and relative residual; see Table 3. The relative residual comes from using IR (Richardson by default).

We note that during the numerical phase of its multifrontal factorization, STRUMPACK pads matrices with zeros, which is not done in SuperLU and produces a factorization with more nonzeros compared to SuperLU (see [54] for details of the factorization) .

For ACTIVSg2000, ACTIVSg10k, and ACTIVSg70k STRUMPACK with default settings (default depth of nested dissection (8), default iterative refinement options (Richardson)) fails because of zero pivots. For ACTIVSg2000, out of the three runs, one converged on the second matrix in the series, and the remaining two failed. This may be due to the randomized sampling applied to create the HSS structure. Linear systems as ill-conditioned as our test cases are very sensitive to a poor sampling and this may be the reason why some of the test cases from the same family fail while some produce correct solutions.

The lack of convergence can be prevented by adjusting the STRUMPACK parameters. It was determined that switching the IR algorithm to GMRES improved the relative residual. However, we needed to increase the value of a parameter that determined the depth of nested dissection reordering (and resulting level of fill) to prevent zero pivots. For ACTIVSg2000, this parameter was set to 25; for ACTIVSg10k it was set to 50; and for ACTIVSg70k it was set to 64.

The numerical errors, summarized in Table 6, are generally comparable with SuperLU.

5.2.2. Performance results

STRUMPACK computations can be split into 5 phases: (a) re-ordering, (b) symmetrization, (c) symbolic factorization, (d) factorization, (e) solve. STRUMPACK can be compiled with and without GPU acceleration.

Because the end-to-end compute time increased significantly with the number of MPI ranks, we only present results for one rank (using four cores). A full set of results, including scaling up to 42 ranks, can

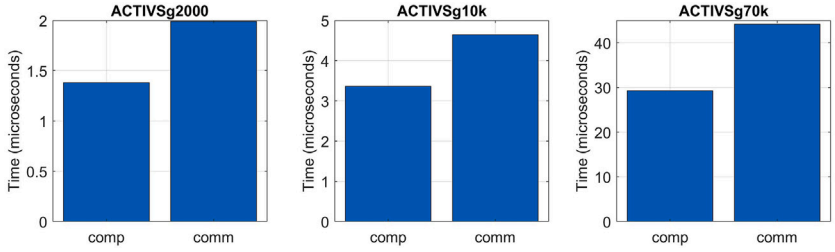

Fig. 4. Details of profiling SuperLU performance on one GPU. Only computation (comp) and communication (comm) times are shown. API calls took 722 ms (ACTIVSg2000), 861 ms (ACTIVSg10k), and 2230 ms (ACTIVSg70k).

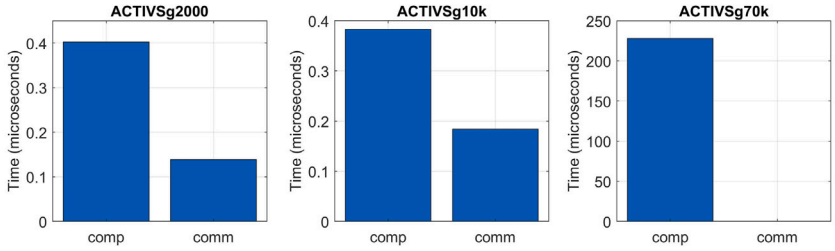

Fig. 5. Details of profiling STRUMPACK performance on the GPU. API calls are not shown; they took 1674 ms (ACTIVSg2000), 1649 ms (ACTIVSg10k), and 2234 ms (ACTIVSg70k). The discrepancy in the timing between the full stack timing as in Table 7 and the detailed GPU timing shown here originates in the difference between timing strategies. In the first case, `MPI_Wtime()` timer is used. In the second case, the code was run with `nvprof`, which adds overhead due to the nature of cuda profiling.

Table 6
The error levels achieved in STRUMPACK. The results for ACTIVSg test cases were achieved by increasing the nested dissection parameter to 25, 50, 64, respectively. All numbers computed using the default RHS.

Name	NNZ inc	av. RR	av. CRBE
ACTIVSg2000	22.30x	$5.26 \cdot 10^{-10}$	$2.05 \cdot 10^{-11}$
ACTIVSg10k	36.42xx	$2.48 \cdot 10^{-3}$	$1.07 \cdot 10^{-12}$
ACTIVSg70k	46.29x	$7.19 \cdot 10^{-3}$	$7.50 \cdot 10^{-11}$

Table 7
STRUMPACK results for all the test cases. All computations were performed on a single CPU (and single GPU). The nested dissection parameter was increased for the 3 test cases to 25, 50, and 64. Richardson IR was used for ACTIVSg10k on the GPU. For the remaining results, GMRES IR was used. ND=nested dissection, Sym=symmetrization, SF=symbolic factorization, Fact=numeric factorization, Sol=solve, TOT=total time (without I/O).

STRUMPACK						
	ND	Sym	SF	Fact	Sol	TOT
ACTIVSg2000 (CPU)	0.45	0.08	0.01	0.07	0.03	0.64
ACTIVSg2000 (GPU)	0.45	0.07	0.01	0.84	0.02	1.39
ACTIVSg10k (CPU)	1.86	0.33	0.024	0.50	0.11	2.82
ACTIVSg10k (GPU)	1.86	0.33	0.025	1.41	0.08	3.71
ACTIVSg70k (CPU)	15.14	2.30	0.14	5.69	1.08	24.35
ACTIVSg70k (GPU)	15.15	2.34	0.15	7.93	1.19	26.76

be found in the repository. When we examine the timings for particular components, the scaling properties are much better. For example, the solve time improves even for very small test cases, and the factorization time decreases for larger test cases. Results using a single rank are in Table 7.

Next, we profile GPU performance. STRUMPACK uses a GPU if the matrix is large enough or more than one rank is specified. In order to observe any computations on the GPU, we need to use at least two MPI ranks or a large enough test case. All results in Fig. 5 were obtained using two MPI ranks, except the last case which was tested with one MPI rank and one GPU. See Fig. 5.

For all problems except the last, the only computation that occurs on the GPU is the triangular solve. For these small cases, the factorization is not GPU-accelerated, but the memory allocation and memory copying that happen during the factorization are expensive and can explain the poor performance. For the last test case, many other computations (e.g., GEMM) are off-loaded to the GPU.

We conclude that there is no benefit to using GPU-accelerated STRUMPACK. In other cases, the overhead arising from the memory allocations outweighs any benefits. If a sequence of matrix equations is solved, and the GPU memory allocation (and free) happens only once, this cost is amortized over the optimization solver iterations.

Table 8
The error levels achieved in cuSolver. Metis reordering was used for all the test cases. For ACTIVSg70k test case, QR method fails regardless of reordering.

ACTIVSg2000		
LU	$9.07 \cdot 10^{-17}$	$6.10 \cdot 10^{-23}$
QR	$2.96 \cdot 10^{-16}$	$3.30 \cdot 10^{-22}$
ACTIVSg10k		
LU	$1.41 \cdot 10^{-10}$	$1.30 \cdot 10^{-21}$
QR	$3.47 \cdot 10^{-11}$	$3.24 \cdot 10^{-22}$
ACTIVSg70k		
LU	$9.98 \cdot 10^{-7}$	$4.53 \cdot 10^{-21}$
QR	–	–

5.3. cuSolver

We tested linear solvers based on LU factorization and QR factorization. The linear solver function based on least squares was also appropriate but its performance was poor and therefore not included here.

For a given matrix and right-hand side, the LU and QR linear solvers both return the solution x and an integer that equals -1 if the matrix is nonsingular, and the index of the first zero row otherwise.

All results in this section were computed using cuSolver from CUDA Toolkit version 10.1.243.

5.3.1. Convergence and quality of solution

The error is measured as relative residual, and as NBRE v2; see Table 3. Note that for the last test case, the QR factorization on GPU fails. The results are shown in Table 8.

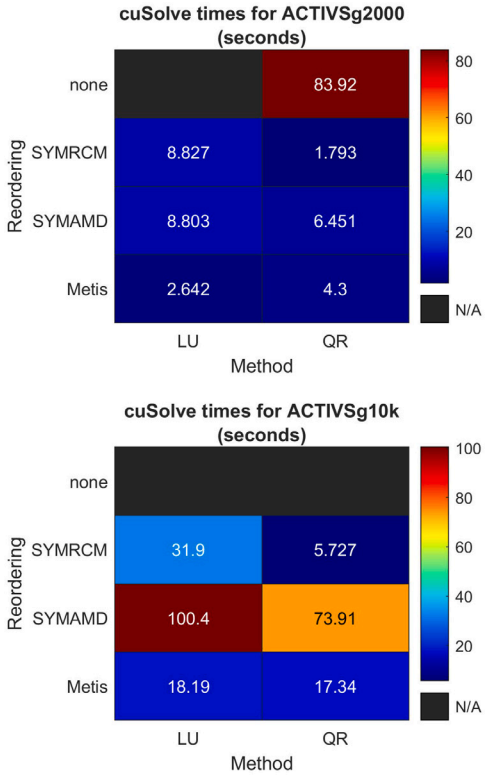

Fig. 6. Performance results for cuSolver with different re-ordering options. Note that the ratio between timings for best and worst reordering is sometimes 5, and larger than 1000 for very small test cases.

5.3.2. Performance results

In addition to providing dimensions and CSR structure pointers, the user can choose the reordering (0: no reordering, 1: symrcm, 2: symamd, or 3: csrmetisnd) and the *tolerance* to decide if singular or not.

While the tolerance has a rather marginal influence on both performance and solution quality, the type of reordering has a huge impact on the time. Thus, we set the tolerance to 10^{-12} for all tests and vary the reordering.

As discussed earlier, both LU- and QR-based solves are *black box*. For a given matrix problem, they output the solution vector without information on the factors; thus we are unable to present detailed timing results as we did for SuperLU and STRUMPACK.

While cuSolver is a GPU-accelerated library, only the QR solve function has a GPU interface. LU factorization is performed on the CPU. Hence the compute time for the LU solver was measured with a system timer and the compute time for the QR solver was measured with CUDA events. Results are shown in Fig. 6.

For ACTIVSg2000, CPU-based LU is faster than GPU-based QR, and again reordering has a huge impact on the time. Since even for the relatively small ACTIVSg2000, cuSolver was not able to complete the computation in 100 min on Summit, we did not try to run ACTIVSg10k without reordering. The only reordering that works for the ACTIVSg70k case is 3 (metis). The time is ≈ 614 s per system (for LU) and the QR factorization fails regardless of the setting. The LU factorization is much slower than STRUMPACK. For the 70k case, the LU time in cuSolver can be lowered by a factor of 3.5× if the matrix is scaled using Ruiz scaling [55] prior to solving; however, STRUMPACK is still faster. The best results for the two largest cases are shown in Fig. 2.

The results in Fig. 6 display the cost of solving the systems, but do not include the cost of data allocation, matrix read, and error evaluation. These costs were (averaged per system): 1.86 s for ACTIVSg2000, 2.67 s for ACTIVSg10k, and 9.97 s for ACTIVSg70k.

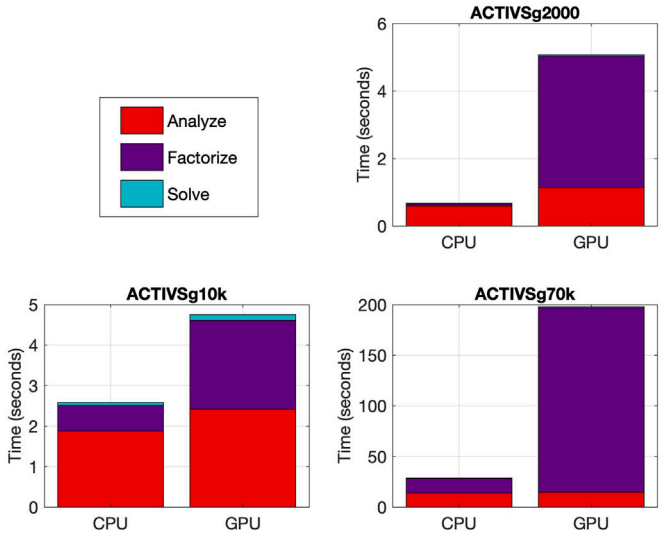

Fig. 7. SSIDS performance on CPU and GPU.

Table 9
The error levels achieved in SSIDS.

Name	av. RR	av. NRBE
ACTIVSg2000		
CPU	$1.85 \cdot 10^{+6}$	$2.35 \cdot 10^{-12}$
GPU	$2.51 \cdot 10^{-5}$	$2.25 \cdot 10^{-12}$
ACTIVSg10k		
CPU	$1.67 \cdot 10^{+4}$	$7.98 \cdot 10^{-11}$
GPU	$5.99 \cdot 10^{-1}$	$2.84 \cdot 10^{-10}$
ACTIVSg70k		
CPU	$3.67 \cdot 10^{+6}$	$2.55 \cdot 10^{-12}$
GPU	$1.86 \cdot 10^{0}$	$2.69 \cdot 10^{-12}$

5.4. SSIDS

5.4.1. Convergence and quality of solution

The error is measured in the same way as for cuSolver, as the scaled residual and NBRE v2; see Table 3. The results are summarized in Table 9:

ACTIVSg2000. For the 2000 series, in most cases GPU-accelerated SSIDS resulted in a segmentation fault. However, if run multiple times, it would eventually produce results, suggesting a possible write-past-the-end-of-array bug. For this series of problems, SSIDS exhibits good performance as long as no segmentation fault occurs.

ACTIVSg10k. While SSIDS is generally fast for these matrices, the relative residuals remain large. However, the norm-wise relative backward error is consistently very low.

ACTIVSg70k. Only four of the five systems can be solved with the GPU-accelerated version. The CPU version again produces solutions with high scaled residuals and low relative backward error.

We conclude that SSIDS works very well for small problems (see the full set of results in the repository) It produces solutions with very small norm-wise relative backward errors. For the RTS test cases, SSIDS is able to solve systems that none of the other packages can solve, which is impressive considering the lack of IR. After carefully studying the available documentation, we did not find a good explanation for the discrepancy between the relative residual values achieved on the CPU and on the GPU SSIDS.

5.4.2. Performance results

SSIDS computations can be split into three basic phases: (a) analysis, (b) factorization, and (c) solve; this is how we time SSIDS. The results in

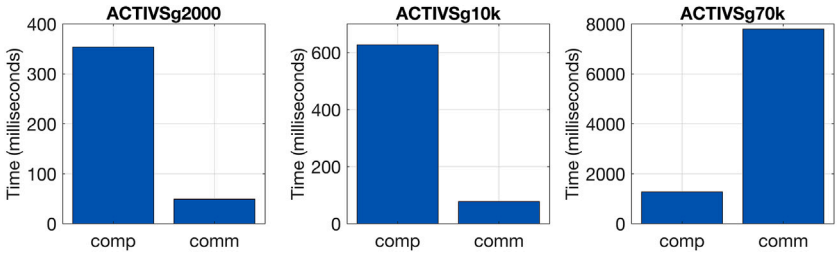

Fig. 8. Details of profiling SSIDS performance on one GPU. Only computation (comp) and communication (comm) times are shown. API calls took ≈ 11 s (ACTIVSg2000), ≈ 10 s (ACTIVSg10k), and ≈ 35 s (ACTIVSg70k).

Fig. 7 indicate that the GPU-accelerated version is much slower than the CPU-only version. In comparison to STRUMPACK and SuperLU, SSIDS offloads more work to the GPU. However, because of the nature of the systems solved, the created *fronts* are very small. To factor them on the GPU, SSIDS allocates separate piece of memory at every level, which explains the huge increase in the time cost of the factorization compared to the CPU version. Profiling details are given in Fig. 8.

Random segmentation faults suggest the package is not quite mature.

6. Comparison

The performance comparison of tested linear solvers is shown in Fig. 2. The results displayed do not include I/O and error evaluation.

CPU results: For the smallest test cases (available in the repository), cuSolver leaves other packages behind, but as the matrix size grows, the situation changes. SSIDS is the fastest for ACTIVSg2000; as the matrix size grows, STRUMPACK becomes the fastest.

GPU results: As for the CPU, cuSolver can solve the smallest test case linear systems the fastest, but other packages become faster as the matrix sizes increases. SuperLU, STRUMPACK and cuSolver take roughly the same time for ACTIVSg2000, and STRUMPACK becomes a winner for the two largest test cases.

The most robust linear solver, able to process singular or close-to-singular systems, is SSIDS.

All the linear solver packages exhibit better CPU than GPU performance, which can be possibly mitigated if the memory used to store the matrices and factors is allocated once and reused over and over. None of the packages scales well in distributed environments, which is expected from our problem sizes. However, individual components of the solution process—such as solve and factorization—scale very well for some of the linear solvers (e.g., STRUMPACK).

7. Conclusions and future work

In this study we tested five linear solver packages on challenging test problems arising from optimal power flow analysis for power grids. The test cases are linear problems (1) that an interior-point optimization method hands off to the linear solver.

A common observation for the linear solver software is the lack of parallel scalability. If we look at the cost of the solution process (without I/O), the time decreases with the number of ranks, but only for certain components of the solution process. This lack of parallel scaling is due to the properties of the matrices, which stress-test the linear solver packages beyond what they were designed for. The authors of all codes used in this paper were contacted to ensure the tests were run correctly.

Based on the data collected for factorization times, we conjecture that the pivot strategies in LU and LDL^T solvers are among the main factors that explain the GPU performance. This is due to the irregular memory accesses and data movement associated with pivoting. The computations are memory bound: there is not enough computation per data movement, as can be inferred from our GPU profiling results for SuperLU and STRUMPACK.

In the case of SuperLU, STRUMPACK and SSIDS, there are excessive GPU memory allocations associated with solving multiple systems, leading to larger than necessary factorization times being reported. Therefore it is difficult to separate the cost of memory allocations from the effects of pivoting on performance for these solvers. For SSIDS, we observe that factorization times increase by 10× over the CPU times; profiling results strongly suggests this happens because of repeated device memory allocations.

The two largely robust linear solver packages are STRUMPACK and SSIDS. To solve highly ill-conditioned systems, STRUMPACK might need to have its parameters adjusted to allow for more fill, but it is capable of producing high-quality solutions that have both low normwise relative backward error and low residual error. SSIDS is able to solve some of the systems that cause all other packages to fail. The advantage of SSIDS is that it respects matrix symmetry, and hence has lower storage requirements.

The cuSolver from NVIDIA consistently produces the lowest normwise relative backward error (NBRE v2) across all the test problems and thus represents the most accurate suite of algorithms evaluated in our study. The cuSolver for GPU is based on QR factorization, which in general exhibits better backward stability properties than LU [56].

Future work will focus on FGMRES IR [57,58], with the possible application of mixed precision [59] for increased execution speed on the GPU. To accelerate the solver phase we plan to explore iterative solution of sparse triangular systems as proposed by Chow et al. [60].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We thank Chris Oehmen and Lori Ross O'Neil for critical reading of the manuscript and helpful suggestions. We also thank Tim Carlson and Kurt Glaesmann of Research Computing for their support, as well as Kestor Gokcen and Jiajia Li for providing utilities for matrix format conversion (all from Pacific Northwest National Laboratory). Finally, we acknowledge the support from the Oak Ridge Leadership Computing Facility, in particular a great deal of help from Philip Roth.

References

[1] Andreas Wächter, Lorenz T. Biegler, Line search filter methods for nonlinear programming: Motivation and global convergence, SIAM J. Optim. 16 (1) (2005) 1–31.

[2] Andreas Wächter, Lorenz T. Biegler, Line search filter methods for nonlinear programming: Local convergence, SIAM J. Optim. 16 (1) (2005) 32–48, http://dx.doi.org/10.1137/S1052623403426544.

[3] Lorenz T Biegler, A survey on sensitivity-based nonlinear model predictive control, IFAC Proc. Vol. 46 (32) (2013) 499–510.
[4] Zhijun Duan, Mirela Andronescu, Kevin Schutz, Sean McIlwain, Yoo Jung Kim, Choli Lee, Jay Shendure, Stanley Fields, C. Anthony Blau, William S. Noble, A three-dimensional model of the yeast genome, Nature 465 (7296) (2010) 363–367.
[5] Sambuddha Chakrabarti, Matt Kraning, Eric Chu, Ross Baldick, Stephen Boyd, Security constrained optimal power flow via proximal message passing, in: 2014 Clemson University Power Systems Conference, IEEE, 2014, pp. 1–8.
[6] Cosmin G. Petra, Olaf Schenk, Mihai Anitescu, Real-time stochastic optimization of complex energy systems on high-performance computers, Comput. Sci. Eng. 16 (5) (2014) 32–42.
[7] Andreas Wächter, Lorenz T. Biegler, On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming, Math. Program. 106 (1) (2006) 25–57.
[8] Iain S. Duff, Ma57—a code for the solution of sparse symmetric definite and indefinite systems, ACM Trans. Math. Software 30 (2) (2004) 118–144.
[9] Elizabeth D. Dolan, Jorge J. Moré, Benchmarking optimization software with performance profiles, Math. Program. 91 (2, Ser. A) (2002) 201–213.
[10] Elizabeth D. Dolan, Jorge J. Moré, Todd S. Munson, Benchmarking Optimization Software with COPS 3.0, Technical report,, Argonne National Laboratory, 2004.
[11] Nicholas I.M. Gould, Dominique Orban, Philippe L. Toint, Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization, Comput. Optim. Appl. 60 (3) (2015) 545–557.
[12] Olaf Schenk, Andreas Wächter, Michael Hagemann, Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization, Comput. Optim. Appl. 36 (2–3) (2007) 321–341.
[13] Luciana Casacio, Christiano Lyra, Aurelio Ribeiro Leite Oliveira, Cecilia Orellana Castro, Improving the preconditioning of linear systems from interior point methods, Comput. Oper. Res. 85 (2017) 129–138.
[14] Jerome W. O'Neal, The Use of Preconditioned Iterative Linear Solvers in Interior-Point Methods and Related Topics (Ph.D. thesis), Georgia Institute of Technology, 2005.
[15] Tim Rees, Chen Greif, A preconditioner for linear systems arising from interior point optimization methods, SIAM J. Sci. Comput. 29 (2007) 1992–2007.
[16] J.R. Bunch, B.N. Parlett, Direct methods for solving symmetric indefinite systems of linear equations, SIAM J. Numer. Anal. 8 (4) (1971) 639–655.
[17] R. Fletcher, Factorizing symmetric indefinite matrices, Linear Algebra Appl. 14 (3) (1976) 257–272.
[18] James R. Bunch, Linda Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Math. Comp. (1977) 163–179.
[19] Cleve Ashcraft, Roger G. Grimes, John G. Lewis, Accurate symmetric indefinite linear equation solvers, SIAM J. Matrix Anal. Appl. 20 (2) (1998) 513–561.
[20] Iain S. Duff, John K. Reid, N. Munksgaard, Hans B. Nielsen, Direct solution of sets of linear equations whose matrix is sparse, symmetric and indefinite, IMA J. Numer. Anal. 23 (2) (1979) 235–250.
[21] Iain S. Duff, John K. Reid, The multifrontal solution of indefinite sparse symmetric linear, ACM Trans. Math. Software 9 (3) (1983) 302–325.
[22] Iain S. Duff, Nick I.M. Gould, John K. Reid, Jennifer A. Scott, Kathryn Turner, The factorization of sparse symmetric indefinite matrices, IMA J. Numer. Anal. 11 (2) (1991) 181–204.
[23] Iain S. Duff, Stéphane Pralet, Strategies for scaling and pivoting for sparse symmetric indefinite problems, SIAM J. Matrix Anal. Appl. 27 (2) (2005) 313–340.
[24] Nicholas I.M. Gould, Jennifer A. Scott, A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations, ACM Trans. Math. Software 30 (3) (2004) 300–325.
[25] Jonathan D. Hogg, Jennifer A. Scott, Pivoting strategies for tough sparse indefinite systems, ACM Trans. Math. Software 40 (1) (2013) 1–19.
[26] Iain S. Duff, Stéphane Pralet, Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems, SIAM J. Matrix Anal. Appl. 29 (3) (2007) 1007–1024.
[27] Jonathan D. Hogg, Jennifer A. Scott, Compressed threshold pivoting for sparse symmetric indefinite systems, SIAM J. Matrix Anal. Appl. 35 (2) (2014) 783–817.
[28] J.D. Hogg, J.A. Scott, An Indefinite Sparse Direct Solver for Large Problems on Multicore Machines, Technical Report RAL-TR-2010-011, Rutherford Appleton Laboratory, Chilton, 2010.
[29] Iain Duff, Jonathan Hogg, Florent Lopez, A new sparse symmetric indefinite solver using a posteriori threshold pivoting, NLAFET Working Note, 2018.
[30] Dulceneia Becker, Marc Baboulin, Jack Dongarra, Reducing the amount of pivoting in Symmetric Indefinite Systems, Technical report, INRIA, 2011.
[31] Timothy A. Davis, Sivasankaran Rajamanickam, Wissam M. Sid-Lakhdar, A survey of direct methods for sparse linear systems, Acta Numer. 25 (2016) 383–566.
[32] Nicholas I.M. Gould, Jennifer A. Scott, Yifan Hu, A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations, ACM Trans. Math. Software 33 (2) (2007) 10–es.
[33] Jonathan D. Hogg, High Performance Cholesky and Symmetric Indefinite Factorizations with Applications (Ph.D. thesis), University of Edinburgh, 2010.
[34] John K. Reid, Jennifer A. Scott, An out-of-core sparse cholesky solver, ACM Trans. Math. Software 36 (2) (2009).
[35] J.D. Hogg, J.A. Scott, A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems, ACM Trans. Math. Software 37 (2) (2010).
[36] https://www.pardiso-projectorg/.
[37] Byron Tasseff, Carleton Coffrin, Andreas Wächter, Carl Laird, Exploring benefits of linear solver parallelism on modern nonlinear optimization applications, 2019.
[38] Xiaoye S. Li, An overview of superlu: Algorithms, implementation, and user interface, ACM Trans. Math. Software 31 (3) (2005) 302–325.
[39] https://portal.nersc.gov/project/sparse/superlu/.
[40] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, Artem Napov, A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization, ACM Trans. Math. Software 42 (4) (2016).
[41] https://portal.nersc.gov/project/sparse/strumpack/.
[42] http://www.numerical.rl.ac.uk/spral/.
[43] https://docs.nvidia.com/cuda/cusolver/index.html.
[44] Pascal Hénon, Pierre Ramet, Jean Roman, Pastix: A high-performance parallel direct solver for sparse symmetric definite systems, Parallel Comput. 28 (2) (2002) 301–321.
[45] The HSL Mathematical Software Library, 0000. http://www.hsl.rl.ac.uk/.
[46] Xiaoye S. Li, James W. Demmel, A scalable sparse direct solver using static pivoting, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing 1999 (San Antonio, TX), SIAM, Philadelphia, PA, 1999, p. 10.
[47] Edward Jason Riedy, Making Static Pivoting Scalable and Dependable (Ph.D. thesis), University of California, Berkeley, 2010.
[48] Iain Duff, Jacko Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, SIAM J. Matrix Anal. Appl. 22 (2001) 973–996.
[49] Alan George, Nested dissection of a regular finite element mesh, SIAM J. Numer. Anal. 10 (2) (1973) 345–363.
[50] A.B. Birchfield, T. Xu, T.J. Overbye, Electric grid test case repository, 2016, https://electricgrids.engr.tamu.edu/electric-grid-test-cases/.
[51] A.B. Birchfield, T. Xu, K.M. Gegner, K.S. Shetye, T.J. Overbye, Grid structural characteristics as validation criteria for synthetic networks, IEEE Trans. Power Syst. 32 (4) (2017) 3258–3265.
[52] Jonathan Maack, Shrirang Abhyankar, ACOPF sparse linear solver test suite, 2020, https://github.com/NREL/opf_matrices.
[53] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.
[54] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, Artem Napov, An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling, SIAM J. Sci. Comput. 38 (5) (2016) S358–S384.
[55] Daniel Ruiz, A Scaling Algorithm to Equilibrate Both Rows and Columns Norms in Matrices, ENSEEIHT-IRIT Technical Report RT/APO/01/4, 2001.
[56] C.C. Paige, Some aspects of generalized QR factorization, in: M.G. Cox, S. Hammarling (Eds.), Reliable Numerical Computation, Clarendon Press, 1990.
[57] Mario Arioli, Iain Duff, Serge Gratton, Stephane Pralet, A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting, SIAM J. Sci. Comput. 29 (5) (2006) 2024–2044.
[58] Mario Arioli, Iain Duff, Using FGMRES to obtain backward stability in mixed precision, ETNA 33 (2009) 31–44.
[59] Erin Carson, Nicholas J. Higham, Accelerating the solution of linear systems by iterative refinement in three precisions, SIAM J. Sci. Comput. 40 (2) (2018) A817–A847.
[60] Edmond Chow, Hartwig Antz, Jennifer Scott, Jack Dongarra, Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning, J. Parallel Distrib. Comput. 119 (2018) 219–230..