# A Feature-complete SPIKE Dense Banded Solver

BRAEGAN S. SPRING and ERIC POLIZZI, University of Massachusetts, Amherst
AHMED H. SAMEH, Purdue University

This article presents a parallel, effective, and feature-complete recursive SPIKE algorithm that achieves near feature-parity with the standard linear algebra package banded linear system solver. First, we present a flexible parallel implementation of the recursive SPIKE scheme that aims at removing its original limitation that the number of cores/processors be restricted to powers of two. A new transpose solve option for SPIKE is then developed to satisfy a standard requirement of most numerical solver libraries. Finally, a pivoting recursive SPIKE strategy is presented as an alternative to the non-pivoting scheme to improve numerical stability. All these new enhancements lead to the release of a new black-box feature-complete SPIKE-OpenMP package that significantly improves upon the performance and scalability obtained with other state-of-the-art banded solvers.

CCS Concepts: • **Mathematics of computing** → **Solvers**; *Mathematical software performance*; • **Theory of computation** → *Shared memory algorithms*;

Additional Key Words and Phrases: SPIKE, banded matrices, linear system solver

**ACM Reference format:**
Braegan S. Spring, Eric Polizzi, and Ahmed H. Sameh. 2020. A Feature-complete SPIKE Dense Banded Solver. *ACM Trans. Math. Softw.* 46, 4, Article 36 (October 2020), 35 pages.
https://doi.org/10.1145/3410153

## 1 INTRODUCTION

Linear systems (i.e., find $X$ solution of $AX = F$ for a given square matrix $A$ and right-hand-side vectors $F$) are a fundamental tool, frequently used to express our understanding of the natural and engineering world. Because of the importance of linear systems in applications, high-quality linear algebra software is a cornerstone of computational science. Two well-known examples of software for performing dense and banded linear algebra are Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) [Anderson et al. 1990]. These collections of subroutines provide a consistent interface to high-performance linear algebra building blocks across hardware platforms and operating systems.

Many recent improvements in available computational power have been driven by increased use of parallelism [Gallopoulos et al. 2016]. The development of new parallel algorithms for

This work is supported by NSF Grants No. CCF-1510010 and No. OAC-1739423.
Authors' addresses: B. S. Spring and E. Polizzi, Department of Electrical and Computer Engineering, Department of Mathematics and Statistics, University of Massachusetts, Amherst, MA 01060; emails: bspring@umass.edu, polizzi@ecs.umass.edu; A. H. Sameh, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907; email: sameh@cs.purdue.edu.

0098-3500/2020/10-ART36 $15.00
https://doi.org/10.1145/3410153

solving linear systems aims at achieving scalability and performance over LAPACK Lower-Upper (LU) algorithms on either shared memory or distributed memory architectures. In shared memory systems, the parallelism in LAPACK LU can directly benefit from the threaded implementation of the low-level BLAS routines. To achieve further scalability improvement, however, it is necessary to move to a higher level of parallelism based on divide-and-conquer techniques. The latter are mandatory with the use of distributed memory systems, but they are also becoming increasingly important if one aims at fully exploiting shared memory machines composed of a large number of cores. The LU factorization paradigm could be adapted to address a high-level parallelism implementation as it is the case for the algorithms proposed in the ScaLAPACK library package [Blackford et al. 1997]. However, in many instances, it can become more advantageous to design algorithms that are inherently better suited for parallelism such as the SPIKE algorithm for solving banded linear systems.

This article focuses on one particular class of structured sparse linear systems that are banded with bandwidth of size $b \ll n$, where $n$ is the matrix size, and that are also dense between the extreme non-zero diagonals below and above the main diagonal. Very often, banded systems arise after a general sparse system is reordered in some fashion [Cuthill and McKee 1969] or they can naturally arise from applications (e.g., [Polizzi and Ben Abdallah 2005]). In other instances, they are constructed as effective preconditioners for iterative methods [Manguoglu et al. 2010].

SPIKE is a very effective banded solver that can significantly outperform the ScaLAPACK package on distributed memory systems, as well as LAPACK on shared memory systems. A SPIKE–Message-passing Interface (MPI) package was released in collaboration with Intel in 2008 [Polizzi and Sameh 2006; SPIKE-MPI-library 2011; Polizzi 2011], and a SPIKE-OpenMP solver was completed in 2015 and included into the distribution of FEAST eigenvalue solver [Polizzi 2009, 2020; FEAST-library 2020] (where SPIKE is used as a kernel for solving banded eigenvalue problems). GPU implementations of SPIKE have also been proposed by other authors [Venetis et al. 2015; Chang et al. 2012; Li et al. 2014].

This work presents essential enhancements to the SPIKE algorithm that are required to achieve a feature-complete SPIKE library package. The development of a competitive library package must not only be motivated by good performance results but should also integrate as much as possible all the main features offered by standard packages such as LAPACK. Among the large number of variants available for SPIKE, we are focusing our efforts to expand the capabilities of the recursive SPIKE algorithm. The recursive scheme demonstrates parallel efficiency and is applicable to both diagonally and non-diagonally dominant systems. However, it lacked the flexibility to adapt to some key situations. In this work, new features and usability enhancements for recursive SPIKE will be considered to address the issues listed below.

(1) In practice, the standard SPIKE recursive scheme is prone to potential waste of parallel resources if the number of cores/processors is not a power of two. For instance, if SPIKE runs on 63 cores, then only 32 would be effectively used (i.e., the lowest nearest power of two). Here, this restriction is removed using a new flexible partitioning scheme and load-balancing strategy that will be presented in Section 3.
(2) Most library solvers include the "transpose solve" option as a standard feature. The same factorization of the matrix $A$ can then be used to solve either $AX = F$ or $A^TX = F$ (i.e., there is no need to factorize $A^T$). This feature is important in many practical situations including the efficient use of non-transpose free iterative solvers (where $A$ is a preconditioner), and the capability to achieve a $\times 2$ speedup while solving complex Hermitian and non-Hermitian eigenvalue problems using FEAST [Kestyn et al. 2016]. The transpose solve option for the SPIKE algorithm is successfully derived in Section 4.

(3) The SPIKE recursive scheme is usually associated with a non-pivoting factorization strategy applied to each matrix partition. The non-pivoting option in SPIKE helps maintaining the banded structure of the matrix, which simplifies the implementation of the algorithm and improves performance of the factorization stage. For systems with very low diagonal dominance, however, partial pivoting may become a necessity to impove the numerical stability and obtain solutions with small residuals (without the need to perform iterative refinements). An efficient pivoting scheme for the recursive SPIKE is presented in Section 5.

All these new enhancements participate to create a feature-complete SPIKE algorithm. Without loss of generality (since both MPI and OpenMP implementation are possible), the presentation terminology and all numerical results are considering a SPIKE OpenMP implementation and the use of threading. A broader impact of this work has been the development and released of a new stand-alone SPIKE-OpenMP package (v1.0) [SPIKE-library 2018]. To the extent possible, this solver has been designed as an easy to use, "black-box" replacement to the standard LAPACK banded solver. For example, the library includes support for single and double precision arithmetic using real or complex matrices. Sections 4–6 of this article are accompanied with extensive numerical experiments that demonstrate that the SPIKE solver significantly outperforms the performance and parallel scalability obtained using the LAPACK banded solver in Intel-MKL. The basic SPIKE algorithm using the recursive scheme is first summarized in Section 2.

## 2 SPIKE BACKGROUND

The SPIKE algorithm can be traced back to work done by A. Sameh and D. Kuck on tridiagonal systems in the late 1970s [Sameh and Kuck 1978], which was later extended to address banded systems [Chen et al. 1978; Gallivan et al. 2012]. It can be viewed as a domain decomposition method [Eijkhout and van de Geijn 2012] for solving block tridiagonal systems. The central idea in SPIKE departs from the traditional $LU$ factorization with the introduction a new $DS$ factorization, which is better suited for parallel implementation as it naturally leads to lower communication cost. Several enhancements and variants of the SPIKE algorithm have since been proposed by Sameh and coauthors [Dongarra and Sameh 1984; Lawrie and Sameh 1984; Berry and Sameh 1988; Sameh and Sarin 1999; Polizzi and Sameh 2006, 2007; Manguoglu et al. 2009; Naumov et al. 2010; Manguoglu et al. 2010, 2011]. Parallelism is extracted by decoupling the relatively large blocks along the diagonal, solving them independently, and then reconstructing the system via the use of smaller reduced systems. There are a number of versions of the SPIKE algorithm, which handle the specifics of those steps in different ways. Two main families of algorithms have been proposed in recent years [Polizzi and Sameh 2006; Mikkelsen and Manguoglu 2009; Mendiratta and Polizzi 2011]: (i) the truncated SPIKE algorithm for diagonally dominant systems; and (ii) the recursive SPIKE algorithm for general non-diagonally dominant systems. This article describes improvements to the recursive SPIKE algorithm for solving banded matrices, which can either be diagonally or non-diagonally dominant.

### 2.1 Central Concept of SPIKE

This section presents the basic SPIKE algorithm. The goal is to find $X$ in the equation

$$AX = F, \tag{1}$$

where $A$ is a banded, $n \times n$ matrix. For clarity, the number of super and sub-diagonals is assumed to be the same and equal to $k$. The matrix bandwidth is $b = 2k + 1$ where $k$ denotes then the "half-bandwidth". The modifications to allow for matrices with non-symmetric bandwidth consist

primarily of padding various small submatrices in the SPIKE reduced system with zeroes. The size of matrices $F$ and $X$ is $n \times n_{rhs}$.

The banded structure may be exploited to enable a block tridiagonal partitioning. $A$ is partitioned along the diagonal into $p$ main diagonal submatrices $A_i$ and their interfaces, as follows:

$$A = \begin{bmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & B_2 & & \\ & \ddots & \ddots & \ddots & \ddots \\ & & & C_p & A_p \end{bmatrix}. \tag{2}$$

Each $A_i$ is a square matrix of size $n_i$. Because the matrix is banded, $B_i$ and $C_i$ can be considered tall and narrow matrices of size $n \times k$, which contain primarily zeroes, i.e.,

$$B_i = \begin{bmatrix} 0 \\ \hline \hat{B}_i \end{bmatrix}; \quad C_i = \begin{bmatrix} \hat{C}_i \\ \hline 0 \end{bmatrix}, \tag{3}$$

where $\hat{B}_i$ and $\hat{C}_i$ are small dense square matrices of size $k$.

We can now factorize the $A$ matrix into the $D$ and $S$ matrices. $D$ contains the diagonal blocks of the matrix $A$. $S$ (a.k.a. the spike matrix) relates the partitions to one another as follows:

$$A = DS = \begin{bmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_p \end{bmatrix} \begin{bmatrix} I_1 & V_1 & & & & \\ W_2 & I_2 & V_2 & & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & & W_p & I_p \end{bmatrix}, \tag{4}$$

where $I_i$ denotes an identity matrix of size $n_i$ and $D_i \equiv A_i$. The $V_i$ and $W_i$ matrices give the SPIKE algorithm its name, because their non-zero elements form tall, narrow submatrices of size $n_i \times k$ (a.k.a., spikes). The equations for these matrices are

$$V_i = {A_i}^{-1} B_i; \quad W_i = {A_i}^{-1} C_i. \tag{5}$$

One source of SPIKE variants is the treatment of the matrices $V_i$ and $W_i$. In the recursive version of SPIKE that is outlined in this article, only the bottom $k \times k$ tips of $V_i$ and $W_i$ need to be explicitly computed. Whenever necessary, the forms ${A_i}^{-1} B_i$ and ${A_i}^{-1} C_i$ will be used in the place of the corresponding $V_i$ and $W_i$ spikes.

Using the $DS$ on the original problem $AX = DSX = F$, it can now be broken up into two subproblems, the D stage and the S stage, i.e.,

$$DY = \begin{bmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_p \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_p \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_p \end{bmatrix}, \tag{6}$$

$$SX = \begin{bmatrix} I_1 & V_1 & & & & \\ W_2 & I_2 & V_2 & & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & & W_p & I_p \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_p \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_p \end{bmatrix}. \tag{7}$$

The submatrices of $D$ are decoupled, so the D-stage is straightforward. Each partition in Equation (6) is solved independently, since

$$Y_i = {D_i}^{-1} F_i. \tag{8}$$

In turn, the vectors and matrices involved in the $S$ stage can be partitioned as follows:

$$V_i = \begin{bmatrix} V_{it} \\ \tilde{V}_i \\ V_{ib} \end{bmatrix}; \quad W_i = \begin{bmatrix} W_{it} \\ \tilde{W}_i \\ W_{ib} \end{bmatrix}, \tag{9}$$

$$X_i = \begin{bmatrix} X_{it} \\ \tilde{X}_i \\ X_{ib} \end{bmatrix}; \quad Y_i = \begin{bmatrix} Y_{it} \\ \tilde{Y}_i \\ Y_{ib} \end{bmatrix}, \tag{10}$$

where each submatrix denoted with a subscript $t$ or $b$ has a height of $k$ rows. The non-zero partitions of $W_i$ and $V_i$ are $k$ columns wide. Essentially, we have broken out the values coupling the partitions of $A$. Equation (7) can be rewritten as

$$\begin{bmatrix} Y_{1t} \\ \tilde{Y}_1 \\ Y_{1b} \end{bmatrix} = \begin{bmatrix} X_{1t} \\ \tilde{X}_1 \\ X_{1b} \end{bmatrix} + \begin{bmatrix} V_{1t} \\ \tilde{V}_1 \\ V_{1b} \end{bmatrix} X_{2t}, \tag{11}$$

$$\begin{bmatrix} Y_{it} \\ \tilde{Y}_i \\ Y_{ib} \end{bmatrix} = \begin{bmatrix} X_{it} \\ \tilde{X}_i \\ X_{ib} \end{bmatrix} + \begin{bmatrix} V_{it} \\ \tilde{V}_i \\ V_{ib} \end{bmatrix} X_{i+1t} + \begin{bmatrix} W_{it} \\ \tilde{W}_i \\ W_{ib} \end{bmatrix} X_{i-1b}, \quad \text{for} \quad i \in 2 \ldots p-1, \tag{12}$$

$$\begin{bmatrix} Y_{pt} \\ \tilde{Y}_p \\ Y_{pb} \end{bmatrix} = \begin{bmatrix} X_{pt} \\ \tilde{X}_p \\ X_{pb} \end{bmatrix} + \begin{bmatrix} W_{pt} \\ \tilde{W}_p \\ W_{pb} \end{bmatrix} X_{p-1b}. \tag{13}$$

Interestingly, the large middle sections of these vectors may be ignored at first. This will lead to the following definition of the tops and bottoms of these vectors that is amenable to reduced system formation:

$$\begin{bmatrix} Y_{1t} \\ Y_{1b} \end{bmatrix} = \begin{bmatrix} X_{1t} \\ X_{1b} \end{bmatrix} + \begin{bmatrix} V_{1t} \\ V_{1b} \end{bmatrix} X_{2t}, \tag{14}$$

$$\begin{bmatrix} Y_{it} \\ Y_{ib} \end{bmatrix} = \begin{bmatrix} X_{it} \\ X_{ib} \end{bmatrix} + \begin{bmatrix} V_{it} \\ V_{ib} \end{bmatrix} X_{i+1t} + \begin{bmatrix} W_{it} \\ W_{ib} \end{bmatrix} X_{i-1b}, \quad \text{for} \quad i \in 2 \ldots p-1, \tag{15}$$

$$\begin{bmatrix} Y_{pt} \\ Y_{pb} \end{bmatrix} = \begin{bmatrix} X_{pt} \\ X_{pb} \end{bmatrix} + \begin{bmatrix} W_{pt} \\ W_{pb} \end{bmatrix} X_{p-1b}. \tag{16}$$

One common source of SPIKE variants is the specific method of solving this reduced system. The "recursive method" for solving the reduced system is discussed in the next section.

Once the reduced system is solved, we obtain the values for $X_{ib}$ and $X_{it}$ with $i \in 1 \ldots p$. In turn, the values for $\tilde{X}_i$ for all $i$ can be straightforwardly recovered using Equations (11), (12), and (13) (a.k.a., the retrieval stage). In some implementations of SPIKE, once the factorization stage is complete, the middle part of the spikes $V$ and $W$ (respectively, $\tilde{V}$ and $\tilde{W}$) are not stored in memory, so they are not available during the retrieval stage. In addition, we note that the spikes $V_1$ and $W_p$ are never explicitly computed providing further optimization of the algorithm (cf. Section 2.3). Consequently, the spikes can instead be replaced by their expression in Equation (5), leading to the following solve operations:

$$X_1 = Y_1 - A_1^{-1}(B_1 X_{2t}), \tag{17}$$

$$X_i = Y_i - A_i^{-1}(B_i X_{i+1t} + C_i X_{i-1b}), \quad \text{for} \quad i \in 2 \ldots p-1, \tag{18}$$

$$X_p = Y_p - A_p^{-1}(C_p X_{p-1b}). \tag{19}$$

At this point, $X$ has been found and the computation is complete.

### 2.2 Recursive SPIKE Scheme

The reduced system described in Equations (14)–(16), represents the inter-domain relationships for the partitioning performed on $A$, it is of size of $2pk$, which scales linearly with the number of partitions $p$. To fully capitalize on the performance gained by exploiting parallelism in the factorization and retrieval stages, the reduced system should not be explicitly formed. Among the multiple techniques that are available for solving the reduced system in parallel, the recursive SPIKE technique provides the best trade-off between generality and parallel efficiency. A full derivation of the recursive method for solving the reduced system is shown in Polizzi and Sameh [2006]. The essential observation is that the reduced system is banded, and, as a result, SPIKE may be used to solve it. From the original reduced system, a new spike matrix $S$ will then be generated, which, in turn, could be solved by SPIKE with half the number of partitions. The process can be repeated recursively, where the number of partitions to consider is divided by two at each recursion level, and until only two partitions are left.

For clarity, an extra superscript index has been added to all the submatrices in the following equations to designate the level of recursion. Here, the process will be illustrated using a four-partition example (i.e., $p = 4$), which is sufficient to provide one level of recursion and show the central concept of the scheme. Our starting point is the original four-partition reduced system:

$$Y^{[1]} = \begin{bmatrix} Y_{1t}^{[1]} \\ Y_{1b}^{[1]} \\ Y_{2t}^{[1]} \\ Y_{2b}^{[1]} \\ Y_{3t}^{[1]} \\ Y_{3b}^{[1]} \\ Y_{4t}^{[1]} \\ Y_{4b}^{[1]} \end{bmatrix} = \begin{bmatrix} I & & V_{1t}^{[1]} & & & & & \\ & I & V_{1b}^{[1]} & & & & & \\ & W_{2t}^{[1]} & I & & V_{2t}^{[1]} & & & \\ & W_{2b}^{[1]} & & I & V_{2b}^{[1]} & & & \\ & & & W_{3t}^{[1]} & I & & V_{3t}^{[1]} & \\ & & & W_{3b}^{[1]} & & I & V_{3b}^{[1]} & \\ & & & & & W_{4t}^{[1]} & I & \\ & & & & & W_{4b}^{[1]} & & I \end{bmatrix} \begin{bmatrix} X_{1t}^{[1]} \\ X_{1b}^{[1]} \\ X_{2t}^{[1]} \\ X_{2b}^{[1]} \\ X_{3t}^{[1]} \\ X_{3b}^{[1]} \\ X_{4t}^{[1]} \\ X_{4b}^{[1]} \end{bmatrix} = S^{[1]} X^{[1]}, \tag{20}$$

where we use the notation $Y^{[1]} = Y_{red}$, $S^{[1]} = S_{red}$, and $X^{[1]} = X_{red}$ to emphasize the current level of recursion (level one here). We then perform a new $DS$ SPIKE factorization of the reduced system using half the number of partitions (so two partitions here),

$$S^{[1]} = \left[\begin{array}{cc:cc} I & V_{1t}^{[1]} & & \\ I & V_{1b}^{[1]} & & \\ W_{2t}^{[1]} & I & & \\ W_{2b}^{[1]} & & & I \\ \hdashline & & I & V_{3t}^{[1]} \\ & & I & V_{3b}^{[1]} \\ & & W_{4t}^{[1]} & I \\ & & W_{4b}^{[1]} & & & I \end{array}\right] \left[\begin{array}{cc:cc} I & & V_{1t}^{[2]} & \\ & I & V_{1b}^{[2]} & \\ & & I & V_{2t}^{[2]} \\ & & & I & V_{2b}^{[2]} \\ \hdashline & & W_{3t}^{[2]} & I & \\ & & W_{3b}^{[2]} & & I \\ & & W_{4t}^{[2]} & & & I \\ & & W_{4b}^{[2]} & & & & I \end{array}\right] = D^{[1]} S^{[2]}, \tag{21}$$

with

$$\begin{bmatrix} I & & V_{1t}^{[1]} & \\ & I & V_{1b}^{[1]} & \\ W_{2t}^{[1]} & & I & \\ W_{2b}^{[1]} & & & I \end{bmatrix} \begin{bmatrix} V_{1t}^{[2]} \\ V_{1b}^{[2]} \\ V_{2t}^{[2]} \\ V_{2b}^{[2]} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ V_{2t}^{[1]} \\ V_{2b}^{[1]} \end{bmatrix} \rightarrow \begin{cases} \begin{bmatrix} I & V_{1b}^{[1]} \\ W_{2t}^{[1]} & I \end{bmatrix} \begin{bmatrix} V_{1b}^{[2]} \\ V_{2t}^{[2]} \end{bmatrix} = \begin{bmatrix} 0 \\ V_{2t}^{[1]} \end{bmatrix} \\ V_{1t}^{[2]} = -V_{1t}^{[1]} V_{2t}^{[2]} \\ V_{2b}^{[2]} = V_{2b}^{[1]} - W_{2b}^{[1]} V_{1b}^{[2]} \end{cases} \tag{22}$$

and

$$\begin{bmatrix} I & & V_{3t}^{[1]} & \\ & I & V_{3b}^{[1]} & \\ W_{4t}^{[1]} & & I & \\ W_{4b}^{[1]} & & & I \end{bmatrix} \begin{bmatrix} W_{3t}^{[2]} \\ W_{3b}^{[2]} \\ W_{4t}^{[2]} \\ W_{4b}^{[2]} \end{bmatrix} = \begin{bmatrix} W_{3t}^{[1]} \\ W_{3t}^{[1]} \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{cases} \begin{bmatrix} I & V_{3b}^{[1]} \\ W_{4t}^{[1]} & I \end{bmatrix} \begin{bmatrix} W_{3b}^{[2]} \\ W_{4t}^{[2]} \end{bmatrix} = \begin{bmatrix} W_{3t}^{[1]} \\ 0 \end{bmatrix} \\ W_{3t}^{[2]} = W_{3t}^{[1]} - V_{3t}^{[1]} W_{4t}^{[2]} \\ W_{4b}^{[2]} = -W_{4b}^{[1]} W_{3b}^{[2]} \end{cases}. \tag{23}$$

It should be noted that the widths of the $V$ and $W$ spikes in $S^{[2]}$ are equal to the widths of $V_2^{[1]}$ and $W_3^{[1]}$, respectively. The matrix $S^{[2]}$ is already in the form of a two-partition S-matrix, so the recursion stops at this step. The reduced system factorization is then complete. Solving the reduced system Equation (21) can be performed in two stages: (i) Obtain the intermediate solution $Y^{[2]}$,

$$D^{[1]} Y^{[2]} = Y^{[1]}, \tag{24}$$

and (ii) Solve for $X^{[1]}$,

$$S^{[2]} X^{[1]} = Y^{[2]}. \tag{25}$$

First, we will look at Equation (24). The blocks of the $D^{[1]}$ matrix are uncoupled, so they can be solved in parallel. In addition, the individual blocks take a form similar to that of a two-partition S-matrix, so an even smaller reduced system can be extracted from each.

$$\left[\begin{array}{cccc|cccc} I & & V_{1t}^{[1]} & & & & & \\ & I & V_{1b}^{[1]} & & & & & \\ W_{2t}^{[1]} & & I & & & & & \\ W_{2b}^{[1]} & & & I & & & & \\ \hline & & & & I & & V_{3t}^{[1]} & \\ & & & & & I & V_{3b}^{[1]} & \\ & & & & W_{4t}^{[1]} & & I & \\ & & & & W_{4b}^{[1]} & & & I \end{array}\right] \begin{bmatrix} Y_{1t}^{[2]} \\ Y_{1b}^{[2]} \\ Y_{2t}^{[2]} \\ Y_{2b}^{[2]} \\ Y_{3t}^{[2]} \\ Y_{3b}^{[2]} \\ Y_{4t}^{[2]} \\ Y_{4b}^{[2]} \end{bmatrix} = \begin{bmatrix} Y_{1t}^{[1]} \\ Y_{1b}^{[1]} \\ Y_{2t}^{[1]} \\ Y_{2b}^{[1]} \\ Y_{3t}^{[1]} \\ Y_{3b}^{[1]} \\ Y_{4t}^{[1]} \\ Y_{4b}^{[1]} \end{bmatrix} \tag{26}$$

$$\begin{bmatrix} I & & V_{1t}^{[1]} & \\ & I & V_{1b}^{[1]} & \\ W_{2t}^{[1]} & & I & \\ W_{2b}^{[1]} & & & I \end{bmatrix} \begin{bmatrix} Y_{1t}^{[2]} \\ Y_{1b}^{[2]} \\ Y_{2t}^{[2]} \\ Y_{2b}^{[2]} \end{bmatrix} = \begin{bmatrix} Y_{1t}^{[1]} \\ Y_{1t}^{[1]} \\ Y_{2t}^{[1]} \\ Y_{2b}^{[1]} \end{bmatrix} \rightarrow \begin{cases} \begin{bmatrix} I & V_{1b}^{[1]} \\ W_{2t}^{[1]} & I \end{bmatrix} \begin{bmatrix} Y_{1b}^{[2]} \\ Y_{2t}^{[2]} \end{bmatrix} = \begin{bmatrix} Y_{1t}^{[1]} \\ Y_{2t}^{[1]} \end{bmatrix} \\ Y_{1t}^{[2]} = Y_{1t}^{[1]} - V_{1t}^{[1]} Y_{2t}^{[2]} \\ Y_{2b}^{[2]} = Y_{2b}^{[1]} - W_{2b}^{[1]} Y_{1b}^{[2]} \end{cases}, \tag{27}$$

$$\begin{bmatrix} I & & V_{3t}^{[1]} & \\ & I & V_{3b}^{[1]} & \\ W_{4t}^{[1]} & I & & \\ W_{4b}^{[1]} & & & I \end{bmatrix} \begin{bmatrix} Y_{3t}^{[2]} \\ Y_{3b}^{[2]} \\ Y_{4t}^{[2]} \\ Y_{4b}^{[2]} \end{bmatrix} = \begin{bmatrix} Y_{3t}^{[1]} \\ Y_{3t}^{[1]} \\ Y_{4t}^{[1]} \\ Y_{4b}^{[1]} \end{bmatrix} \rightarrow \begin{cases} \begin{bmatrix} I & V_{3b}^{[1]} \\ W_{4t}^{[1]} & I \end{bmatrix} \begin{bmatrix} Y_{3b}^{[2]} \\ Y_{4t}^{[2]} \end{bmatrix} = \begin{bmatrix} Y_{3t}^{[1]} \\ Y_{4t}^{[1]} \end{bmatrix}. \\ Y_{3t}^{[2]} = Y_{3t}^{[1]} - V_{3t}^{[1]} Y_{4t}^{[2]} \\ Y_{4b}^{[2]} = Y_{4b}^{[1]} - W_{4b}^{[1]} Y_{3b}^{[2]} \end{cases} \tag{28}$$

Therefore, the $D_1$ linear system solve has been reduced to two $2k \times 2k$ solve operations, which are performed in parallel, and some recovery operations. Next, Equation (25) must be solved. This is simply a two-partition S-matrix, so we will extract a reduced system and perform recovery sweeps as usual,

$$\left[\begin{array}{cccc|cccc} I & & & & V_{1t}^{[2]} & & & \\ & I & & & V_{1b}^{[2]} & & & \\ & & I & & V_{2t}^{[2]} & & & \\ & & & I & V_{2b}^{[2]} & & & \\ \hline & & & & I & & & \\ & & & & & I & & \\ & & & & & & I & \\ & & & & & & & I \end{array}\right] \begin{bmatrix} X_{1t}^{[1]} \\ X_{1b}^{[1]} \\ X_{2t}^{[1]} \\ X_{2b}^{[1]} \\ X_{3t}^{[1]} \\ X_{3b}^{[1]} \\ X_{4t}^{[1]} \\ X_{4b}^{[1]} \end{bmatrix} = \begin{bmatrix} Y_{1t}^{[2]} \\ Y_{1b}^{[2]} \\ Y_{2t}^{[2]} \\ Y_{2b}^{[2]} \\ Y_{3t}^{[2]} \\ Y_{3b}^{[2]} \\ Y_{4t}^{[2]} \\ Y_{4b}^{[2]} \end{bmatrix} \tag{29}$$

$$\begin{aligned} \begin{bmatrix} I & V_{2b}^{[2]} \\ W_{3t}^{[2]} & I \end{bmatrix} \begin{bmatrix} X_{2b}^{[1]} \\ X_{3t}^{[1]} \end{bmatrix} &= \begin{bmatrix} Y_{2t}^{[2]} \\ Y_{3t}^{[2]} \end{bmatrix}, \\ \begin{bmatrix} X_{1t}^{[1]} \\ X_{1b}^{[1]} \\ X_{2t}^{[1]} \end{bmatrix} &= \begin{bmatrix} Y_{1t}^{[2]} \\ Y_{1b}^{[2]} \\ Y_{2t}^{[2]} \end{bmatrix} - \begin{bmatrix} V_{1t}^{[2]} \\ V_{1b}^{[2]} \\ V_{2t}^{[2]} \end{bmatrix} X_{3t}^{[1]}, \\ \begin{bmatrix} X_{3b}^{[1]} \\ X_{4t}^{[1]} \\ X_{4b}^{[1]} \end{bmatrix} &= \begin{bmatrix} Y_{3b}^{[2]} \\ Y_{4t}^{[2]} \\ Y_{4b}^{[2]} \end{bmatrix} - \begin{bmatrix} W_{3b}^{[2]} \\ W_{4t}^{[2]} \\ W_{4b}^{[2]} \end{bmatrix} X_{2b}^{[1]}. \end{aligned} \tag{30}$$

At this point the $X^{[1]}$ vectors have been found, so the reduced system is solved. The total number of $2k \times 2k$ solve operations is the same as the number of partition interfaces, $p - 1$. The total computational cost spent on solve operations is $O(p \times k \times n_{rhs})$. However, all the solve operations in each recursive level may be performed in parallel. Because the system is split in half with each recursive level, the total number of recursive levels is $\log_2(p)$. Therefore, the combined critical path length of all the solve operations in the solve stage is $O(\log_2(p) \times k \times n_{rhs})$. For the same reason, the reduced system factorization stage solve operations have a critical path length of $O(\log_2(p) \times k^2)$. So, the total cost of the solve operations is $O(\log_2(p) \times k \times \max(k, n_{rhs}))$. There is also some overhead involved with the solution recovery operations and communication, but this has not been found to be significant.

This completes the description of the recursive method of solving the reduced system. This method of solving the reduced system can significantly improve performance by exploiting

parallelism in the problem. However, because the procedure progresses through recursive levels by repeatedly splitting submatrices in half, this recursive approach limits the number of partitions allowable to a power of two. A method of decoupling the number of threads used from the number of partitions will be shown in Section 3. Next, we look at optimizations specific to the banded structure.

### 2.3 Optimizing Per-partition Costs

In Section 2.1, we neglected the specifics of the factorization performed on the blocks, $D_i$. The primary computational costs for SPIKE are the matrix operations performed on each block. The goal, then, is to reduce the number of solve operations performed.

The $D_i$ matrices are factorized into triangular matrices. For a total number of partitions $p$, partitions 1 to $p-1$ use an LU factorization. For the final partition, a UL factorization is used. In the following, we will be working with the non-pivoting SPIKE algorithm using the diagonal boosting strategy originally introduced in Polizzi and Sameh [2006] that is applied to provide a good trade-off between accuracy and performance. In Section 5 a partial pivoting algorithm that does not require diagonal boosting will be shown.

The first detail to look at is the creation of the V spikes,

$$V_i = A_i^{-1}B_i = U_i^{-1}L_i^{-1}\begin{bmatrix} 0 \\ \hline \hat{B}_i \end{bmatrix}. \tag{31}$$

The matrix $L_i^{-1}$ is lower triangular. The solve operation for a lower triangular matrix begins by identifying the topmost row in each solution vector, and proceeds downward. For this reason, we label this a "downward sweep." In the case of Equation (31), the downward sweep is simply passing over zeroes until the topmost rows of $\hat{B}_i$ are reached. So, this sweep may be shortened by beginning it at that point. This shortens the downward sweep from a height of $n_i$ to a height of $k$, rendering it relatively inconsequential in terms of computational cost.

The final partition is UL factorized. The optimization is similar, but it instead avoids the zeroes in the upward sweep:

$$W_p = A_p^{-1}C_p = L_p^{-1}U_p^{-1}\begin{bmatrix} \hat{C}_p \\ \hline 0 \end{bmatrix}, \tag{32}$$

The next important variation from the basic version of SPIKE discussed earlier is the treatment of the V and W spikes. Using the definitions for $V_i$ and $W_i$ above, and the fact that $Y_i = D_i^{-1}F_i$, we may rewrite the retrieval stage shown previously in Equations (17), (18), and (19), as follows:

$$\begin{bmatrix} X_{1t} \\ \tilde{X}_1 \\ X_{1b} \end{bmatrix} = \begin{bmatrix} Y_{1t} \\ \tilde{Y}_1 \\ Y_{1b} \end{bmatrix} - V_1 X_{2t} = A_1^{-1}\left(\begin{bmatrix} F_{1t} \\ \tilde{F}_1 \\ \hline F_{1b} \end{bmatrix} - \begin{bmatrix} 0 \\ \hline \hat{B}_1 \end{bmatrix} X_{2t}\right) = U_1^{-1}\left(L_1^{-1}\begin{bmatrix} F_{1t} \\ \tilde{F}_1 \\ \hline F_{1b} \end{bmatrix} - L_1^{-1}\begin{bmatrix} 0 \\ \hline \hat{B}_1 \end{bmatrix} X_{2t}\right), \tag{33}$$

Table 1. Total Number of Sweeps Needed

| Number of full sweeps | Factorization stage | Solve stage |
|---|---|---|
| First & Last partition | 0 | 2 |
| Middle partitions | 3 | 4 |

For the inner partitions, three solve sweeps are performed to create the spikes in the factorization, and four solve sweeps are performed in the solve stage. For the first and last partitions, two solve sweeps are performed in the solve stage, and none are required in the factorization stage.

$$\begin{bmatrix} X_{it} \\ \tilde{X}_i \\ X_{ib} \end{bmatrix} = \begin{bmatrix} Y_{it} \\ \tilde{Y}_i \\ Y_{ib} \end{bmatrix} - V_i X_{i+1t} - W_i X_{i-1b} = A_i^{-1} \left[ \begin{array}{c} F_{it} \\ \hline \tilde{F}_i \\ \hline F_{ib} \end{array} \right] - A_i^{-1} \left( \left[ \begin{array}{c} 0 \\ \hline \hat{B}_i \end{array} \right] X_{i+1t} + \left[ \begin{array}{c} \hat{C}_i \\ \hline 0 \end{array} \right] X_{i-1b} \right), \quad (34)$$

$$\begin{bmatrix} X_{pt} \\ \tilde{X}_p \\ X_{pb} \end{bmatrix} = \begin{bmatrix} Y_{pt} \\ \tilde{Y}_p \\ Y_{pb} \end{bmatrix} - W_p X_{p-1b} = A_p^{-1} \left( \left[ \begin{array}{c} F_{pt} \\ \hline \tilde{F}_p \\ F_{pb} \end{array} \right] - \left[ \begin{array}{c} \hat{C}_p \\ \hline 0 \end{array} \right] X_{p-1b} \right) = L_p^{-1} \left( U_p^{-1} \left[ \begin{array}{c} F_{pt} \\ \hline \tilde{F}_p \\ F_{pb} \end{array} \right] - U_p^{-1} \left[ \begin{array}{c} \hat{C}_p \\ \hline 0 \end{array} \right] X_{p-1b} \right). \quad (35)$$

The solve stage for the first and last partitions can be performed with just two large sweeps, and a collection of small sweeps and multiplications with practically no cost [Mendiratta and Polizzi 2011]. For all other partitions, a total of four sweeps per partition are needed in the solve stage.

The reduced system only needs $V_{1b}$ for the first partition, and $W_{pt}$ for the last partition. As a result the upward sweep in Equation (31) can also be truncated. Similarly, the downward sweep in Equation (32) is truncated. This results in no full sweeps in these partitions during the factorization stage. For the middle partitions, the tips of $V$ and $W$ can be obtained using three full sweeps in the SPIKE factorization stage, one full sweep to generate the spike $V$ and two full sweeps to generate $W$.

The total number of full sweeps needed for the factorization and solve stages is summarized in Table 1. We note that in the case where only two partitions are present (i.e., the first and last partition), SPIKE performs the same number of total sweeps as a traditional LU factorization and solve would require on solving the original linear system. Since each partition contains half of the elements of the total matrix, a two-partition SPIKE solver that uses one processor/core by partition is expected to run twice as fast as a single processor/core LU applied to the whole system [Mendiratta and Polizzi 2011; Spring 2014]. This is a remarkable result of near-perfect parallelism, which is often difficult to obtain for complex algorithms due to the cost of overhead and additional preprocessing stage. This case is known as the SPIKE 2×2 kernel and it will be used as building block in the next sections.

## 3 FLEXIBLE PARTITIONING SCHEME FOR RECURSIVE SPIKE

The recursive SPIKE algorithm can only be applied as described if the number of partitions is a power of two. Indeed, the recursive solver repeatedly applies SPIKE to the reduced system, splitting in half the number of partitions at each step. In previous implementations of recursive SPIKE using OpenMP for shared memory [Mendiratta and Polizzi 2011] or MPI for distributed memory [Polizzi 2011], the number of threads (respectively, MPI processes) was tied to the number of partitions, with one thread (respectively, one MPI process) working on each partition. As a result,

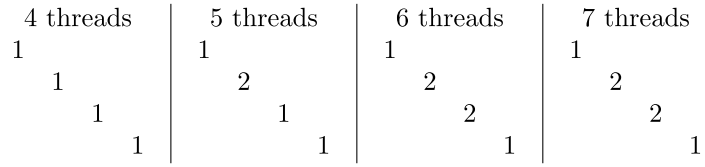



Fig. 1. Distribution of 4 to 7 threads using four partitions.

the power-of-two restriction for the number of partitions would result in a waste of parallel computing resources. For example, if 60 cores/processors were available, then only 32 cores/processor (the lowest nearest power of two) could be utilized by the standard recursive SPIKE. The approach discussed in the following waives this restriction by exploiting further the potential for parallelism. For clarity and without loss of generality (since both MPI and OpenMP SPIKE are possible choices), the presentation terminology and numerical results are considering a SPIKE OpenMP implementation and the use of threading.

The method relies on dedicating not strictly one but possibly two threads to some or all of the interior partitions when the total number of threads is not a power of two. The SPIKE 2×2 kernel is then used to perform the factorization and solve operations on the two-thread partitions. As mentioned in Section 2.3, the SPIKE 2×2 kernel has twice the performance of a single-threaded banded solver. Because the factorization and solve operations make up the majority of the computational cost for SPIKE, the 2×2 kernel has the potential to provide a significant speedup for the partitions on which it is used.

### 3.1 Distribution of Threads

This section discusses how threads are allocated to partitions. The overall plan is to start by selecting the greatest power of two below the number of available threads to generate the SPIKE partitions, as is usually the case with recursive SPIKE. From there, threads will be added to the middle partitions until we have reached the total number of threads given by the environment. Not all partitions will benefit from the addition of threads. Specifically, the first and last partitions benefit greatly from exploiting the structure of the LU and UL factorizations, respectively, as seen in Section 2.3. So, conventional LU and UL factorizations are always used for these partitions. For all other partitions 2×2 SPIKE may be useful.

Threads are allocated sequentially, starting at the second partition, as shown in Figures 1 and 2. The number one designates a partition that is given a single thread, and the number two designates one given a pair of threads. Note that seven threads are distributed as if there were six. This is because neither of the remaining single threaded partitions would benefit from using 2×2 SPIKE [Spring 2014]. Similarly, in Figure 2 one thread is wasted when there are fifteen total threads. In comparison with the standard recursive SPIKE that allows only one thread per partitions, up to three threads would be wasted in Figure 1 and up to seven in Figure 2.

Replacing the LAPACK LU solver with a 2×2 SPIKE solver is, algorithmically, trivial. The derivation of SPIKE given in Section 2.1 did not rely on the specifics of the LU factorization, with the exception of a couple of optimizations. So, neglecting these optimizations, the 2×2 SPIKE solver may be plugged into place with no changes.

Of the two main optimizations, only one requires our attention. The first optimization was used to reduce the number of solve sweeps in the first and last partitions, shown in Section 2.3. As stated previously, we simply avoid using the SPIKE 2×2 solver on those partitions, so this is not a problem. The more interesting optimization allows for the generation of the $V$ spike beginning

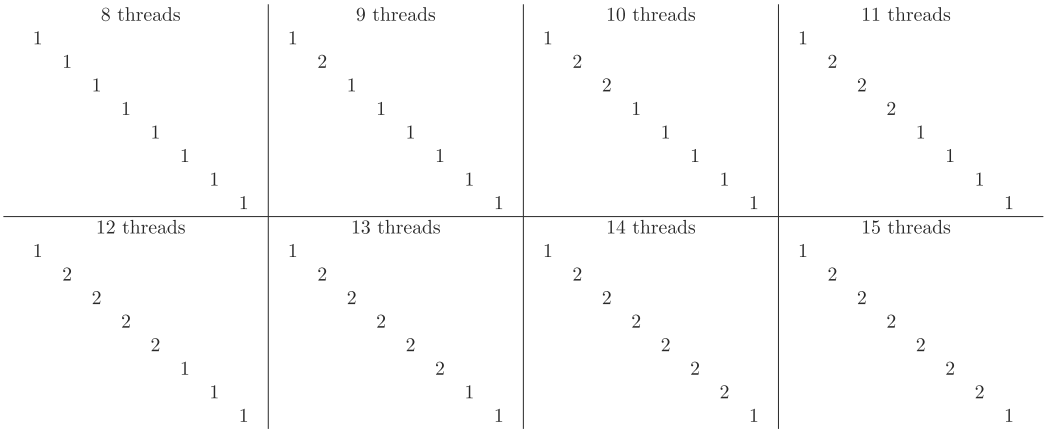



Fig. 2. Distribution of 8 to 15 threads using eight partitions.

with a truncated solve operation, for a total of only one solve-sweep. The next section describes how to perform a nearly equivalent optimization, but with the 2×2 SPIKE solver.

### 3.2 Reducing Factorization Stage Sweeps

In Section 2.3, a method of generating the $V$ spikes with just one sweep was shown. The essential observation is that the submatrix used to generate $V_i$ is comprised mainly of zeroes, and non-zero elements are restricted to the bottom $k$ rows. As a result, the sweep using $L_i$ (L-sweep) may start at the beginning of the non-zero elements. This reduces the size of the solve operation from asymptotically equal to the matrix size, to the bandwidth. As a result it is computationally inexpensive enough to be ignored.

A similar observation can also be applied to the spikes generated with the $2 \times 2$ SPIKE partitions. In this case, we will exploit the shape of the $B$ and $C$ matrices to avoid performing solve operations over a large number of zeroes. The operations to be performed are

$$A_i^{-1} \begin{bmatrix} 0 \\ \hline \hat{B}_i \end{bmatrix} = V_i; \quad A_i^{-1} \begin{bmatrix} \hat{C}_i \\ \hline 0 \end{bmatrix} = W_i. \tag{36}$$

$A_i$ is a submatrix of $A$ for which we would like to use $2 \times 2$ SPIKE. It has a half bandwidth of $k$ and a size of $n_i$. The relevant equation is

$$\left[\begin{array}{c|c} A_{i1} & \begin{matrix} 0 \\ \hat{B}_{i1} \end{matrix} \\ \hline \begin{matrix} \hat{C}_{i2} & 0 \end{matrix} & A_{i2} \end{array}\right] \begin{bmatrix} \tilde{X}_{i1} \\ X_{i1b} \\ \hline \tilde{X}_{i2t} \\ \tilde{X}_{i2} \end{bmatrix} = \begin{bmatrix} \tilde{F}_{i1} \\ F_{i1b} \\ \hline \tilde{F}_{i2t} \\ \tilde{F}_{i2} \end{bmatrix}, \tag{37}$$

where we can extract

$$A_{i1} \begin{bmatrix} \tilde{X}_{i1} \\ X_{i1b} \end{bmatrix} + \begin{bmatrix} 0 \\ \hat{B}_{i1} \end{bmatrix} X_{i2t} = \begin{bmatrix} \tilde{F}_{i1} \\ F_{i1b} \end{bmatrix}, \tag{38}$$

$$\begin{bmatrix} \tilde{X}_{i1} \\ X_{i1b} \end{bmatrix} = A_{i1}^{-1} \begin{bmatrix} \tilde{F}_{i1} \\ F_{i1b} \end{bmatrix} - \begin{bmatrix} 0 \\ \hat{B}_{i1} \end{bmatrix} X_{i2t} = U_{i1}^{-1} \left( L_{i1}^{-1} \begin{bmatrix} \tilde{F}_{i1} \\ F_{i1b} \end{bmatrix} - L_{i,1}^{-1} \begin{bmatrix} 0 \\ \hat{B}_{i1} \end{bmatrix} X_{i2t} \right). \tag{39}$$

Table 2. Computational Cost Summary for Each Partition Type

| Partition Type | Operation Count | | |
|---|---|---|---|
| | Factorize Stage | | Solve Stage |
| | Factorize | Solve Sweeps (over $k$ vectors) | Solve Sweeps (over $n_{rhs}$ vectors) |
| First & Last | 1 | 0 | 2 (LU) |
| Inner Two-Thread | 1 | 3 (SPIKE $2 \times 2$) | 4 (SPIKE $2 \times 2$) |
| Inner Single-Thread | 1 | 3 (LU) | 4 (LU) |

We may observe that, when solving for $V_i$, $F_{i1} = 0$. The initial L-sweep over this matrix is thus unnecessary. This saves a solve sweep of height $n_i/2$,

$$\begin{bmatrix} \tilde{V}_{i1} \\ V_{i1b} \end{bmatrix} = U_{i1}^{-1} \left( -L_{i1}^{-1} \begin{bmatrix} 0 \\ \hat{B}_{i1} \end{bmatrix} V_{i2t} \right). \tag{40}$$

A similar optimization is possible for $W_i$, This saves another solve sweep of height $n_i/2$, i.e.,

$$A_{i2} \begin{bmatrix} X_{i2t} \\ \tilde{X}_{i2} \end{bmatrix} + \begin{bmatrix} \hat{C}_{i2} \\ 0 \end{bmatrix} X_{i1t} = \begin{bmatrix} F_{i2b} \\ \tilde{F}_{i2} \end{bmatrix}, \tag{41}$$

$$\begin{bmatrix} X_{i2b} \\ \tilde{X}_{i2} \end{bmatrix} = A_{i2}^{-1} \begin{bmatrix} F_{i2t} \\ \tilde{F}_{i2} \end{bmatrix} - \begin{bmatrix} \hat{C}_{i2} \\ 0 \end{bmatrix} X_{i1t} = L_{i2}^{-1} \left( U_{i2}^{-1} \begin{bmatrix} F_{i2t} \\ \tilde{F}_{i2} \end{bmatrix} - U_{i2}^{-1} \begin{bmatrix} \hat{C}_{i2} \\ 0 \end{bmatrix} X_{i1b} \right), \tag{42}$$

$$\begin{bmatrix} V_{i2b} \\ \tilde{V}_{i2} \end{bmatrix} = L_{i2}^{-1} \left( -U_{i2}^{-1} \begin{bmatrix} \hat{C}_{i2} \\ 0 \end{bmatrix} V_{i1b} \right). \tag{43}$$

As a result, an amount of work equal to two half-sweeps is saved. This means that the total work performed on the SPIKE 2×2 partitions is equal to that of the normal, single threaded partitions. In other words, the SPIKE 2×2 kernel may still be used to form the $V$ and $W$ submatrices with three sweeps.

### 3.3 Load-balancing Scheme

For optimal load balancing, we would like to have each partition take the same amount of time to complete. This will be approximated by setting equal the sums of the computational costs for the partitions. The computational costs considered will be those incurred by the large factorization and solve operations.

Let us continue using the same banded matrix $A$ with a size of $n \times n$ and a half bandwidth of $k$, as well as our collections of vectors $F$ and $X$, sized $n \times n_{rhs}$. The costs incurred for each partition are summarized in Table 2. Note that in the factorization stage, the $V$ and $W$ spikes must be created for the reduced system. These require performing solve operations on blocks with widths equal to the lower and upper bandwidths, respectively. Because the matrix is considered structurally symmetric (for clarity), these operations are recorded as solve sweeps of width $k$.

Table 2 suggests that one may want to consider three partition sizes, $n_1$, $n_2$, and $n_3$. Respectively, they are the sizes of the first/last partitions, the middle partitions on which the two threaded SPIKE is used, and the middle partitions, which receive the single threaded LU factorization. Both types of middle partitions have the same total number of solve sweeps in each stage. The SPIKE $2 \times 2$ solver should require half of the computation time used by the standard LU solver. So, we may set $n_2 = 2n_3$. The relationship between $n_1$, $n_2$, and $n_3$ can be defined as ratios: $R_{12} = \frac{n_1}{n_2}$ and $R_{13} = \frac{n_1}{n_3}$.

The SPIKE implementation uses a blocked LU factorization and solve, based on the BLAS-3 and LAPACK implementation provided by the system. Similar to the banded LAPACK operations,

the factorization has an asymptotic performance of $O(n \times k^2)$, and the solve has a performance of $O(n \times k \times n_{rhs})$. These costs can be approximated as $K_1 \times n \times k^2$, the cost of factorization, and $K_2 \times n \times k \times n_{rhs}$, the cost of two full sweeps. The ratio between $K_2$ and $K_1$ will be called $K$. The coefficients $R_{12}$ and $R_{13}$ may be computed by balancing the factorization and solve performance costs between the first/last partition and the inner partitions described in Table 2 as follows:

$$K_1 n_1 k^2 + K_2 n_1 k n_{rhs} = K_1 n_3 k^2 + 3\frac{K_2}{2} n_3 k^2 + 2K_2 n_3 k n_{rhs}, \tag{44}$$

$$K_1 n_1 k + K_2 n_1 n_{rhs} = K_1 n_3 k + (3/2) K_2 n_3 k + 2K_2 n_3 n_{rhs}. \tag{45}$$

Now it is possible to obtain $R_{13}$ in terms of $K$, $n_{rhs}$, and $k$:

$$K_1 n_1 k^2 + K_2 n_1 k n_{rhs} = K_1 n_3 k^2 + (3/2) K_2 n_3 k^2 + 2K_2 n_3 k n_{rhs}, \tag{46}$$

$$\begin{aligned} R_{13} = \frac{n_1}{n_3} &= \frac{K_1 k + (3/2) K_2 n_3 k + 2K_2 n_{rhs}}{K_1 k + K_2 n_{rhs}} \\ &= \frac{1}{1 + (K_2/K_1)(n_{rhs}/k)} + \frac{3/2 + 2n_{rhs}/k}{K_1/K_2 + n_{rhs}/k} \\ &= \frac{1}{1 + (K)(n_{rhs}/k)} + \frac{3/2 + 2n_{rhs}/k}{1/K + n_{rhs}/k}. \end{aligned} \tag{47}$$

For $R_{12}$, we have

$$n_2 = 2n_3, \tag{48}$$

$$R_{12} = \frac{1}{2} R_{13} = \frac{1}{2 + 2(K)(n_{rhs}/k)} + \frac{3/4 + n_{rhs}/k}{1/K + n_{rhs}/k}. \tag{49}$$

The derivation of $K$ requires that the size of the partitioned sub-matrices is large enough for the asymptotic computational costs to dominate over data movement costs (which implies an effective maximum number of partitions for a given matrix). In this situation, the $K$ value becomes a machine specific tuning constant that depends on the system hardware and the underlying LAPACK and BLAS implementations. Due to the myriad of existing hardware and software, it is unlikely that an universally good value for $K$ exists. However, for a given machine, $K$ may be easily found by performing a matrix factorization and solve on a matrix and set of vectors for which $n_{rhs} = k$. Using the same approximations as above,

$$\text{factorization time} = K_1 \times n \times k^2, \tag{50}$$

$$\text{solve time} = K_2 \times n \times k \times n_{rhs}, \tag{51}$$

$$K = \frac{K_2}{K_1} = \frac{\text{solve time}}{n \times k \times n_{rhs}} \times \frac{n \times k^2}{\text{factorization time}} \tag{52}$$

$$= \frac{\text{solve time}}{\text{factorization time}}. \tag{53}$$

The other variable to consider when determining $R_{12}$ and $R_{13}$ is $n_{rhs}/k$. In general, if this value is known before the DS factorization is performed, then $R_{12}$ and $R_{13}$ may be calculated. If the value is not known, then the problem might be characterized as similar to one of two limiting cases, $n_{rhs}/k \to 0$ and $n_{rhs}/k \to \infty$.

In the first case, the matrix bandwidth is much greater than the number of vectors in the solution. Intuitively, this indicates that the factorization stage will dominate the computational cost. In this case, we obtain

$$\lim_{n_{rhs}/k \to 0} \quad R_{12} = (1/2) + (3/4)K \text{ and } \quad R_{13} = 1 + (3/2)K. \tag{54}$$

This can be seen simply by plugging the value $n_{rhs}/k = 0$ into Equation (49) for $R_{12}$.

In the second case, where the number of solution vectors is much greater than the matrix bandwidth, the solve stage dominates. For this type of problem, we obtain constant ratios that are independent of the value of $K$, i.e.,

$$\lim_{n_{rhs}/k \to \infty} \quad R_{12} = \frac{1}{2 + 2(K)(n_{rhs}/k)} + \frac{1 + n_{rhs}/k}{1/K + n_{rhs}/k} = 1, \text{ and } \quad R_{13} = 2. \tag{55}$$

Once the ratios between partition sizes have been decided upon, sizing the partitions is simple. The main requirement is that the partition sizes must sum to the size of $A$. Assume next that there are $x = r - 2$ partitions of size $n_2$, $y \equiv q$ of size $n_3$, and that the first and last partitions, are of size $n_1$ each. Overall, this gives the following constraints, which can be trivially solved for the size of each type of partition:

$$n = 2n_1 + xn_2 + yn_3 = 2n_1 + \frac{xn_1}{R_{12}} + \frac{yn_1}{R_{13}}, \tag{56}$$

$$\frac{nR_{12}R_{13}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_1, \tag{57}$$

$$\frac{nR_{13}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_2, \tag{58}$$

$$\frac{nR_{12}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_3. \tag{59}$$

This concludes the description of the increased parallelism scheme for recursive SPIKE. In summary, this scheme allows the use of almost any number of threads, without dramatically modifying the recursive SPIKE algorithm. Overall computational time is decreased by carefully sizing the partitions into which the matrix $A$ is distributed. The information required for the sizing process has been separated into hardware/library-dependent factors and problem-dependent ones. Finally, the sizing task is simple enough that it may be performed automatically, and the SPIKE OpenMP library package [SPIKE-library 2018] includes utility routines to do so.

### 3.4 Performance Measurements

To show the effects of the previously described enhancements, a number of measurements were taken on a large shared memory machine. The first set of measurements explore the partition sizing method, as described in the previous section. The second set of measurements shows the overall performance and scalability of the algorithm. The hardware and software used for these experiments is as follows:

- 8×Intel Xeon E7-8870: 10 cores @ 2.40 GHz with 30 MB cache
- Intel Fortran 16.0.1
- Intel MKL 11.3.1

The E7-8870 also exploits the "hyperthreading" simultaneous multithreading strategy. Hyperthreading is generally considered to be detrimental for dense numerical linear algebra. In most

cases, for these experiments hyperthreads have been avoided using the following environment variable:

—KMP_AFFINITY=granularity=fine,compact,1,0

The KMP affinity interface is a feature of the Intel implementation of OpenMP.[1]

Finally, SPIKE is also making extensive use of LAPACK/BLAS3, so any improvements in the kernel library (e.g., Intel MKL) would be as well beneficial to SPIKE and it would not change the relative scalability and speed-up performances between SPIKE-OpenMP and MKL that are presented here.

*3.4.1 Partition Ratio Accuracy.* In Section 3.3 equations to determine the appropriate sizes of the various submatrices used are derived. To measure the accuracy of this technique, an exploration of many possible partition size ratios was performed in Figures 3 and 4. For these measurements, the matrix size $n$ and bandwidth $b$ remain constant (respectively, $n = 10^6$ and $b = 321$ with $k = 160$), while the number of solution vectors changes from $n_{rhs} = 320$ in Figure 3 to $n_{rhs} = 80$ in Figure 4. In these figures, the X and Y axes correspond to the ratios $R_{12}$ and $R_{13}$, as defined in Section 3.3. By keeping the bandwidth constant and varying the number of solution vectors, the effect of these ratios can be observed. Each figure has a map for the cost, in time, of the factorization and solve stages, as well at the overall computation time. In addition, the best measured runs as well as the location of the pre-calculated values of the best partition size ratios, have been marked along with their times. The pre-computed values for the factorization and solve stages use the most favorable ratios derived in Equations (54) and (55), respectively. The pre-computed value for the combined factorization/solve measurement is obtained using the "compromise ratios" given in Equations (47) and (49). Because the matrix does not change from one run to the next, the factorization stage is identical for each run. As such, the first map in each figure is largely identical, with some small variation due to noise. The excellent agreement between the results indicates that $K$, the machine specific tuning constant, is accurately computed. The method of determining the solve stage favoring partition ratios is even more reliable than the factorization stage. Indeed, for Figure 3 the measured and calculated values are identical. This is likely because the solve stage partition ratio formula can be simplified to a pair of constant values, so whatever imprecision was introduced in the discovery of $K$ is no longer present.

Finally, a band of good performance can be visually observed starting at the origin and continuing along the path of $2R_{12} = R_{13}$. Within those areas, the primary concern is that the computation times produced by using the calculated partition ratios are not too far from the optimal measured ones. The percentage improvement from using the measured optimal, rather than calculated, partition ratios is summarized in Table 3, which also includes the cases for $n_{rhs} = 40$ and $n_{rhs} = 160$. In general, the gains of the measured optimal partition ratios over the computed ones are in the low single-digit percentages.

*3.4.2 Scalability and Performance Comparisons.* We propose to observe some aspects of the overall performance of the new implementation of recursive SPIKE. Figures 5 and 6 contain two sets of measurements. On the left, we see the scalability of SPIKE. On the right, we see absolute time measurements, as well as a comparison to MKL (note that the time axes in these measurements are on a logarithmic scale). All measurements for SPIKE (including factorization, solve, and

[1]The "compact" command instructs the OpenMP runtime to pack threads as closely as possible. The "1,0" command shifts the core hierarchy, so that the pair of hyperthreads on a given core are considered very far away from one another, while the cores inside a given CPU package are considered nearest neighbors. By using this strategy and employing less than eighty threads, a pair of hyperthreads that share a core are never considered close enough to employ both simultaneously.

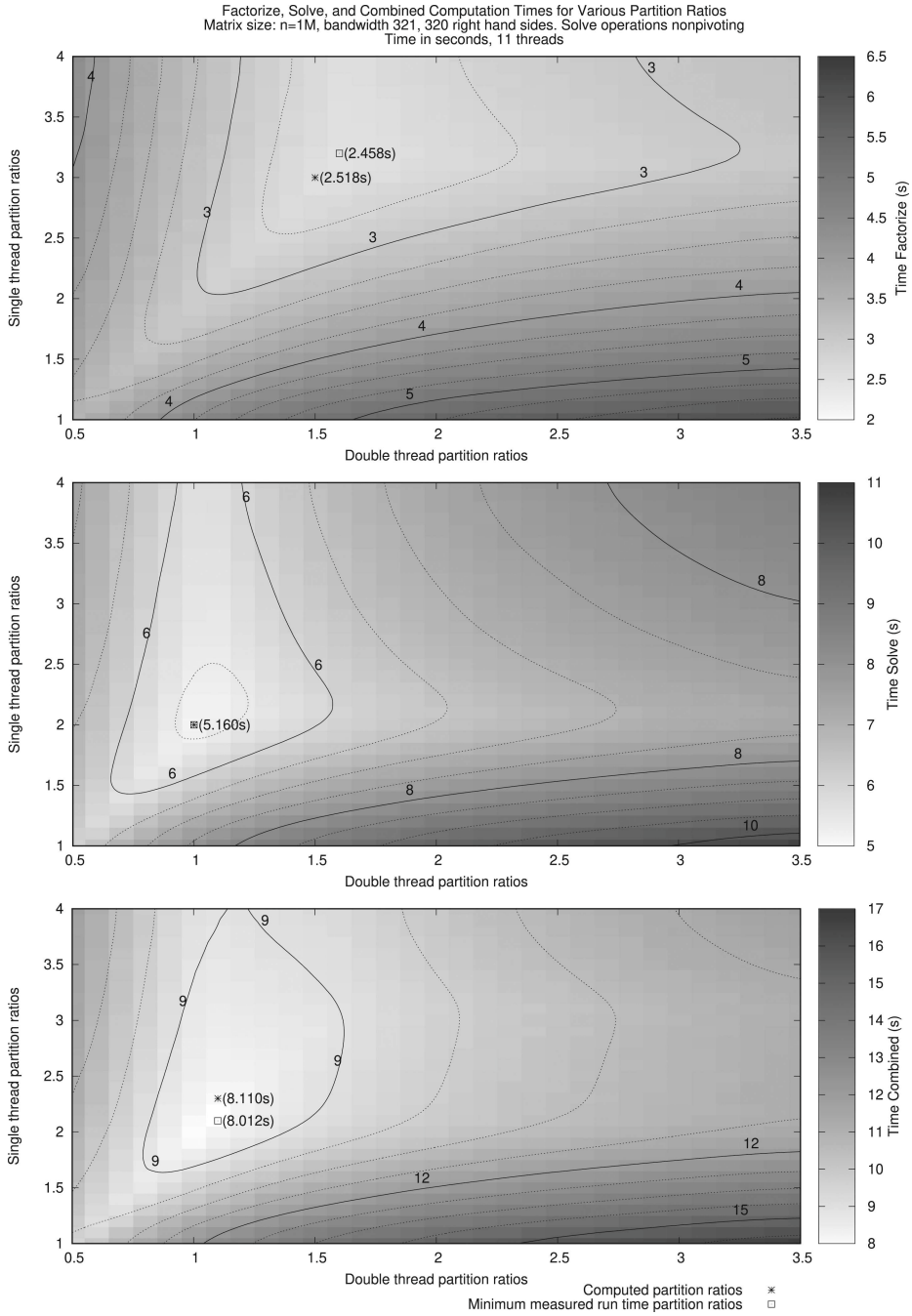



Fig. 3. Partition ratio "heatmaps" for 320 right-hand sides.

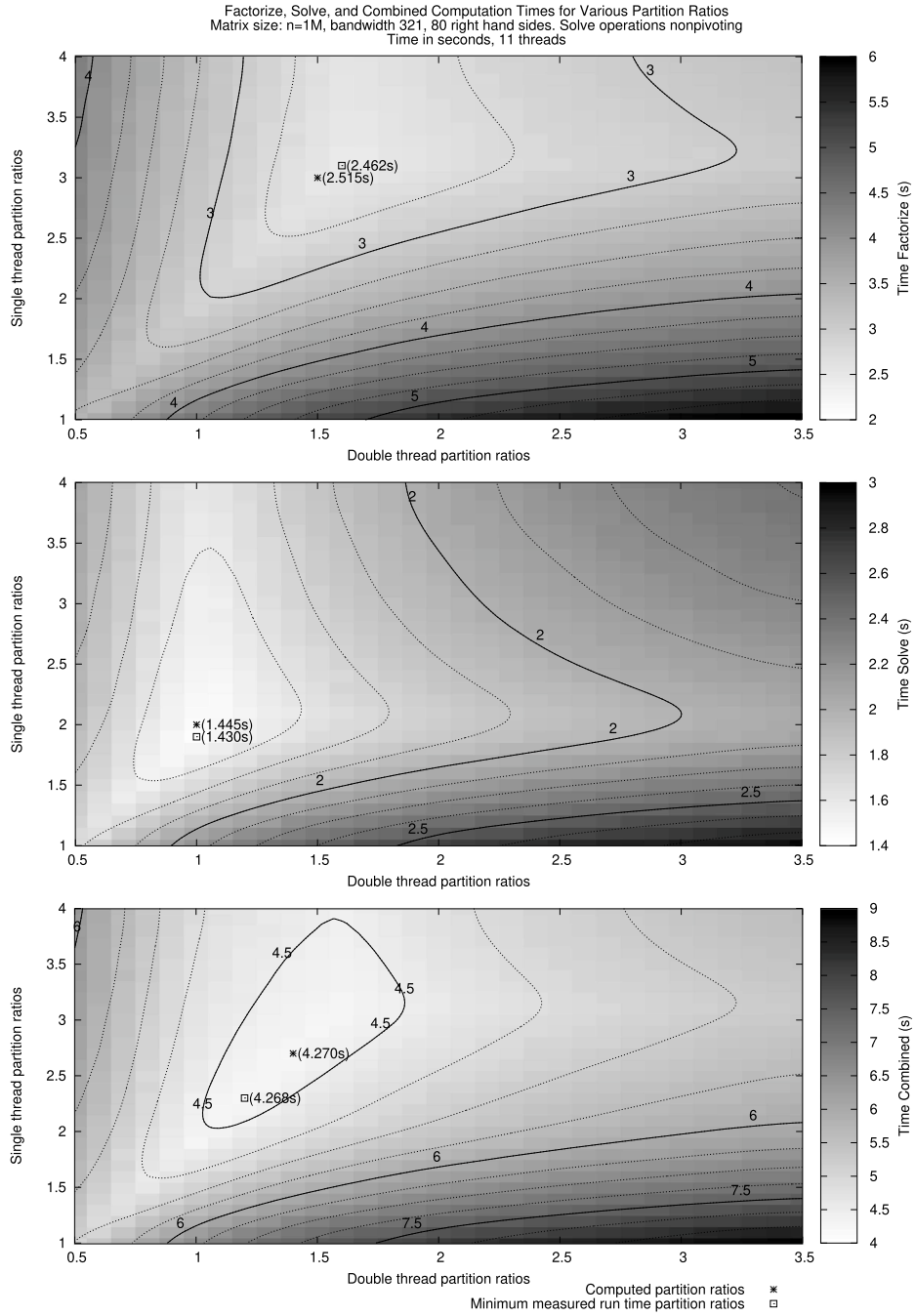



Fig. 4. Partition ratio "heatmaps" for 80 right-hand sides.

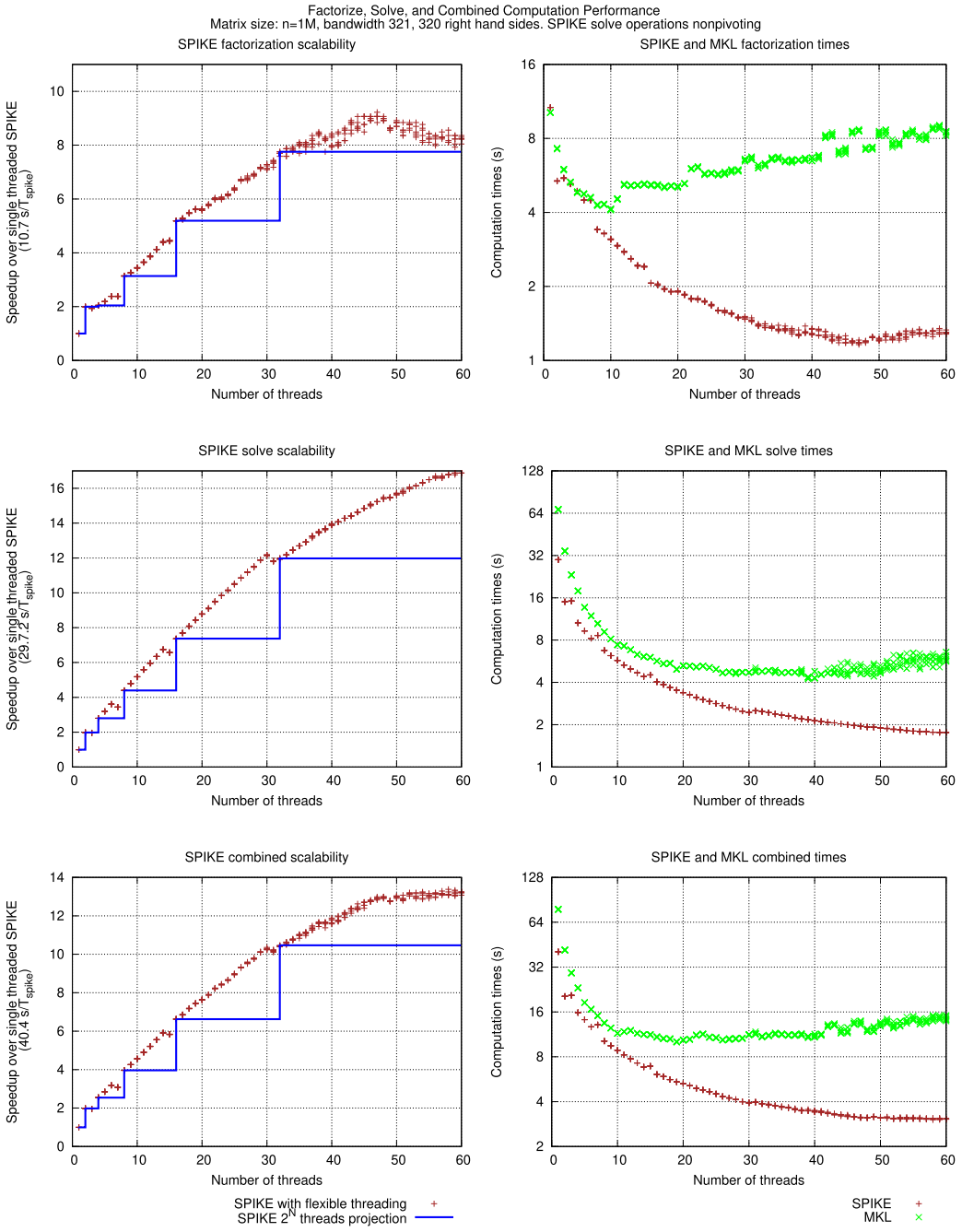



Fig. 5. Scalability and computation time for 320 right-hand sides.

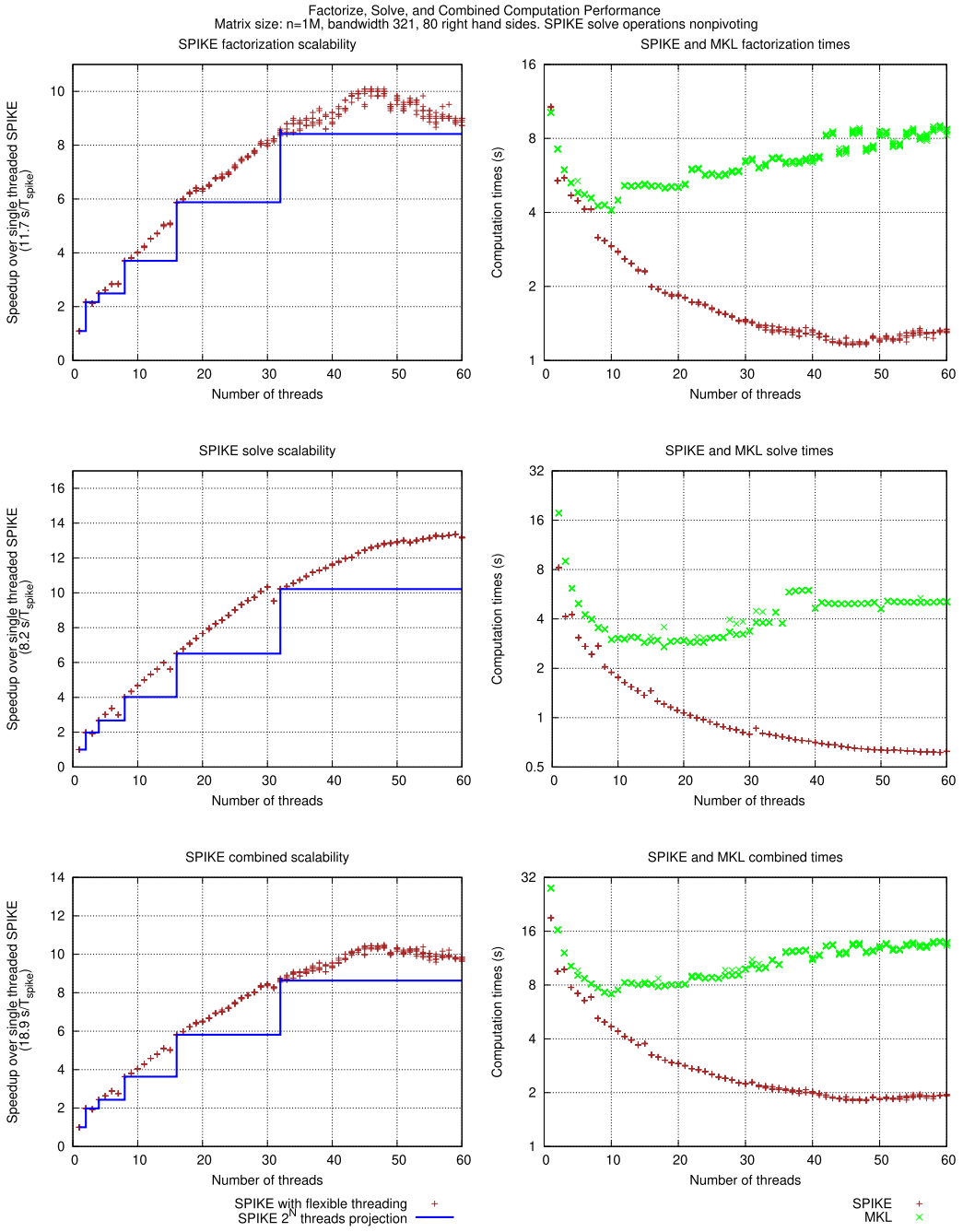



Fig. 6. Scalability and computation time for 80 right-hand sides.

Table 3. Performance Gain from Using Best Measured Partition Ratios $\left|\frac{t_{calculated}}{t_{measured}} - 1\right|$

| Solution Vectors | 40 | 80 | 160 | 320 |
|---|---|---|---|---|
| Factorize | 2.44% | 2.15% | 2.36% | 2.44% |
| Solve | 1.43% | 1.05% | 0 | 0 |
| Combined | 1.22% | 0.04% | 1.21% | 1.22% |

Table 4. Partition Ratios Used for Figures 5 and 6

| Solution Vectors | 80 | 320 |
|---|---|---|
| R13 | 2.7 | 2.3 |
| R12 | 1.35 | 1.15 |

combined stages) were taken using the calculated partition ratios given in Equations (47) and (49) and summarized in Table 4.

Scalability is measured relative to the computation time of the single-threaded non-pivoting solver used on the individual partitions. Overall, the combined factorization/solve stages appear to scale well until around 45 cores are used. As the number of cores increases beyond that point, performance stalls and would eventually degrade. We note that for larger matrices, the scalability breaking point could go well beyond the 45 cores. The trade-off used to determine the partition ratios can be seen by comparing the scaling of each set of benchmarks. As the number of solution vectors decreases, the partition size ratios move to favor the factorization stage of the computation. This can be observed in the increased scaling of the factorization stage, and the decrease in the solve stage scaling. We note that the optimal ratios for the factorization stage given in Equation (54) are equal to $R_{13} = 3$ and $R_{12} = 1.5$ for the measured value of $K$ on our software/hardware set-up. The ratios provided in Table 4 will progressively reach these values with the number of solution vectors decreasing. In turn, the optimal ratio for the solve stage Equation (55) give the values $R_{13} = 2$ and $R_{12} = 1$, which are close to the values reported in Table 4 with large number of right-hand sides. Overall for these particular numerical experiments, the solve stage has noticeably superior scalability to the factorization stage.

The scalability measurements also show the benefit of the flexible threading scheme. This is one of the most important results presented here, since the standard recursive SPIKE scheme is limited by the use of power of two number of threads. The line labeled "SPIKE $2^N$ threads projection" shown the effects of limiting the number of threads used to powers of two by extending the performance measured at these points. Naturally, the performance gap is most dramatic soon before the number of threads is increased to the next power of two. For example, looking at Figure 5, at 30 threads the overall computation scaling increases from roughly 6× to roughly 9×, as a result of the increased overall utilization of resources.

Finally, overall computation time is generally superior to MKL. We note that the two solvers are close in time until 10 threads are reached, at which point SPIKE begins pulling away. This is particularly apparent in the factorization stage. In contrast to the SPIKE *DS* factorization, parallelism performance for the inherently recursive serial *LU* approach used by MKL mainly relies on BLAS, which quickly reaches its limits. However, MKL parallelizes well over solution vectors, and so when their number increases, MKL remain moderately closer in performance to SPIKE. We note that the base solver used for SPIKE provides performance advantage, as it is non-pivoting. To minimize the effects of pivoting for MKL, all the test matrices in the numerical experiments were

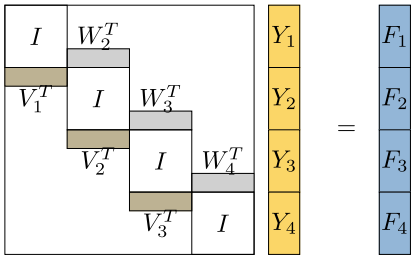



Fig. 7. Four partition transpose S-matrix.

chosen diagonally dominant (both solvers producing relative residuals of $10^{-13}$ or below). However, SPIKE recursive is applicable to non-diagonally dominant systems as well. In most cases, a zero-pivot may never been found even for matrices with large condition numbers. The latter, however, could affect the relative residual and a SPIKE pivoting strategy will be presented in Section 5 to address this issue.

## 4 TRANSPOSE SOLVE OPTION FOR RECURSIVE SPIKE

A transpose solve option is a standard feature for LAPACK subroutines. This option allows transpose problems to be solved without explicitly transposing the matrix in memory. Transpose solve retrieves $X$ for the following problem:

$$A^T X = F,$$

where $A$, $X$, and $F$ are defined as in the previous sections: An $n \times n$ banded matrix with half-bandwidth $k$, and two $n \times n_{rhs}$ collections of vectors, respectively.

Similarly to the standard LAPACK solver, the transpose solve option reuses the factorization from the non-transpose case. That is, once a matrix has been factorized it may be used for either transpose or non-transpose solve operations. Because the factorization stage has the potential to be much more time-consuming than the solve stage, this feature can result in great time savings. For SPIKE, this means we reuse the $D$ and $S$ matrices and the reduced system from the previous section. The transpose problem may be written as follows:

$$A^T X = (DS)^T X = S^T D^T X = F, \tag{60}$$

$$S^T Y = F, \tag{61}$$

$$D^T X = Y. \tag{62}$$

This presents two sub-problems. As in the non-transpose case, partitions of the $D$ matrix are uncoupled, and so the $D^T$ stage can be parallelized in a familiar, straightforward manner. For the $S^T$ matrix, a new algorithm will need to be designed, because this matrix is structurally different from the $S$ matrix. In particular, a transpose formulation of the recursive method for solving the reduced system solver is required. Ultimately near performance parity with the non-transpose solver will be achieved by matching the count of these operations. This will guide the development of the algorithm.

### 4.1 Transpose S Stage

The first sub problem to solve is $S^T Y = F$. This problem can be visualized using the four-partition example in Figure 7. A reduced system can be extracted from this matrix, by exploiting the fact that many of the elements of the $Y$ vector are not affected by the solve operation, and therefore

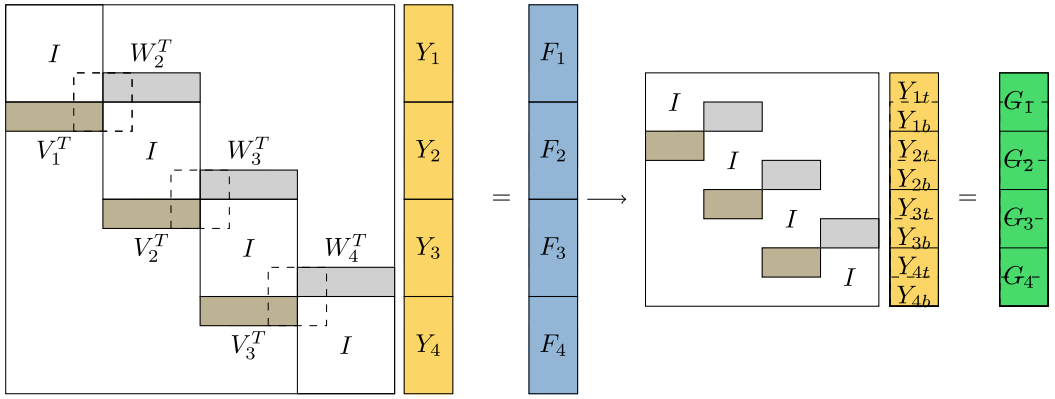



Fig. 8. Reduced transpose system extraction for four partitions.

are simply equal to the corresponding elements of $F$. This can be seen if the $V_i^T$ and $W_i^T$ spikes, and the $Y_i$ and $F_i$ vectors are partitioned in the following manner:

$$V_i^T = \begin{bmatrix} V_{it}^T, & \tilde{V}_i^T, & V_{ib}^T \end{bmatrix}; \quad W_i^T = \begin{bmatrix} W_{it}^T, \tilde{W}_i^T, W_{ib}^T \end{bmatrix}, \tag{63}$$

$$Y_i = \begin{bmatrix} Y_{it}^T, & \tilde{Y}_i^T, & Y_{ib}^T \end{bmatrix}^T; \quad F_i = \begin{bmatrix} F_{it}^T, \tilde{F}_i^T, F_{ib}^T \end{bmatrix}^T. \tag{64}$$

When viewing a given horizontal slice of the $S^T$ matrix, shown in Figure 7, it is visually clear that $\tilde{F}_i = \tilde{Y}_i$. Indeed, we obtain

$$F_i = \begin{bmatrix} F_{it} \\ \tilde{F}_i \\ F_{ib} \end{bmatrix} = \begin{bmatrix} V_{i-1}^T \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} Y_{i-1t} \\ \tilde{Y}_{i-1} \\ Y_{i-1b} \end{bmatrix} + \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} Y_{it} \\ \tilde{Y}_i \\ Y_{ib} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ W_{i+1}^T \end{bmatrix} \begin{bmatrix} Y_{i+1t} \\ \tilde{Y}_{i+1} \\ Y_{i+1b} \end{bmatrix}. \tag{65}$$

If $Y_{it}$ and $Y_{ib}$ are given a height of $k$ rows each, and $\tilde{Y}_i$ is given the remaining elements, then this equation can be rewritten as follows:

$$\begin{aligned} F_{it} &= Y_{it} + V_{i-1}^T \begin{bmatrix} Y_{i-1t} \\ \tilde{Y}_{i-1} \\ Y_{i-1b} \end{bmatrix} = Y_{it} + V_{i-1}^T \begin{bmatrix} 0 \\ \tilde{Y}_{i-1} \\ 0 \end{bmatrix} + V_{i-1}^T \begin{bmatrix} Y_{i-1t} \\ 0 \\ 0 \end{bmatrix} + V_{i-1}^T \begin{bmatrix} 0 \\ 0 \\ Y_{i-1b} \end{bmatrix}, \\ \tilde{F}_i &= \tilde{Y}_i, \\ F_{ib} &= Y_{ib} + W_{i+1}^T \begin{bmatrix} Y_{i+1t} \\ \tilde{Y}_{i+1} \\ Y_{i+1b} \end{bmatrix} = Y_{ib} + W_{i+1}^T \begin{bmatrix} 0 \\ \tilde{Y}_{i+1} \\ 0 \end{bmatrix} + W_{i+1}^T \begin{bmatrix} Y_{i+1t} \\ 0 \\ 0 \end{bmatrix} + W_{i+1}^T \begin{bmatrix} 0 \\ 0 \\ Y_{i+1b} \end{bmatrix}. \end{aligned} \tag{66}$$

The solve for $Y_{it}$ and $Y_{ib}$ must now be modified to adjust for the presence of the known values in $\tilde{Y}_i$. It is then possible to extract a reduced system as depicted in Figure 8, and where the modified right-hand side $G_i$ is given by

$$\begin{aligned} i > 1, \quad G_{it} = F_{it} - V_{i-1}^T \begin{bmatrix} 0 \\ \tilde{F}_{i-1} \\ 0 \end{bmatrix} &= Y_{it} + V_{i-1}^T \begin{bmatrix} Y_{i-1t} \\ 0 \\ 0 \end{bmatrix} + V_{i-1}^T \begin{bmatrix} 0 \\ 0 \\ Y_{i-1b} \end{bmatrix} \\ &= Y_{it} + V_{i-1t}^T Y_{i-1t} + V_{i-1b}^T Y_{i-1b}, \end{aligned} \tag{67}$$

$$i < p-1, \quad G_{ib} = F_{ib} - W_{i+1}^T \begin{bmatrix} 0 \\ \tilde{F}_{i+1} \\ 0 \end{bmatrix} = Y_{ib} + W_{i+1}^T \begin{bmatrix} Y_{i+1t} \\ 0 \\ 0 \end{bmatrix} + W_{i+1}^T \begin{bmatrix} 0 \\ 0 \\ Y_{i+1b} \end{bmatrix} \\ = Y_{ib} + W_{i+1t}^T Y_{i+1t} + W_{i+1b}^T Y_{i+1b}. \tag{68}$$

At this point it should be noted that for this implementation, once the factorization stage is complete, the middle values of $V_i^T$ and $W_i^T$ are no longer available, so they cannot then be used to construct the components of $G_i$ in Equations (67) and (68). Using Equation (5) for the spikes $V_i$ and $W_i$, $G_i$ can be rewritten as

$$i > 1, \quad G_{it} = F_{it} - \left(A_{i-1}^{-1} C_{i-1}\right)^T \begin{bmatrix} 0 \\ \tilde{F}_{i-1} \\ 0 \end{bmatrix} = F_{it} - \begin{bmatrix} \hat{C}_{i-1}^T & 0 & \dots \end{bmatrix} A_{i-1}^{-T} \begin{bmatrix} 0 \\ \tilde{F}_{i-1} \\ 0 \end{bmatrix}, \tag{69}$$

$$i < p-1, \quad G_{ib} = F_{ib} - \left(A_{i+1}^{-1} B_{i+1}\right)^T \begin{bmatrix} 0 \\ \tilde{F}_{i+1} \\ 0 \end{bmatrix} = F_{ib} - \begin{bmatrix} \dots & 0 & \hat{B}_{i+1}^T \end{bmatrix} A_{i+1}^{-T} \begin{bmatrix} 0 \\ \tilde{F}_{i+1} \\ 0 \end{bmatrix}. \tag{70}$$

Overall, this approach is preferable to using the $V_i$ and $W_i$ matrices for two reasons.

First, as it can be seen in Figure 8, the top tip of $Y_1$ and the bottom tip of $Y_p$ make it through this transpose S-stage unchanged (respectively, $Y_{1t} = F_{1t}$ and $Y_{pb} = F_{pb}$). Therefore, the spikes $V_1$ and $W_p$ do not need to be formed during the factorization stage leading to the load-balancing optimization presented in Section 3.3 (i.e., the first and last partition can be chosen bigger in size).

Second, $G_{i+1t}$ and $G_{i-1b}$ both require the same solve operation over the modified $F_i$ vectors,

$$A_i^{-T} \begin{bmatrix} 0 \\ \tilde{F}_i \\ 0 \end{bmatrix}. \tag{71}$$

Therefore, creating the $G$ vector in this manner incurs the cost of one large solve operation and two small multiplications per partition (since $B_{i+1}$ and $C_{i-1}$ are mostly comprised of zeroes). This is likely to be less expensive than the cost of performing two large multiplications (if $V_i$ and $W_i$ were available).

Once the reduced system and $G$ vector have been constructed, all that remains in the S stage is to solve it. Notably, this reduced system matrix is simply the transpose of the reduced system matrix used in non-transpose SPIKE given in Equation (20) for four partitions. In Section 4.3 a recursive method for solving the transpose reduced system will be presented.

### 4.2 Transpose D Stage

Because the partitions of the $D$ matrix are completely decoupled, performing this stage is much simpler than the S stage as illustrated in Figure 9. The overall goal is to obtain $X$ in $D^T X = Y$. In the S stage, it was shown that $\tilde{Y}_i = \tilde{F}_i$. Therefore, once the solutions of the reduced system $Y_{it}$ and $Y_{ib}$ are known, the whole solution $X_i$ is simply retrieved as follows:

$$X_i = A_i^{-T} \begin{bmatrix} Y_{it} \\ \tilde{Y}_i \\ Y_{ib} \end{bmatrix} = A_i^{-T} \begin{bmatrix} Y_{it} \\ \tilde{F}_i \\ Y_{ib} \end{bmatrix}. \tag{72}$$

This concludes the description of the basic transpose SPIKE solver.

Similarly to the non-transpose case, optimizations are possible for transpose SPIKE to achieve the same computational costs reported in Table 2 for the total number of solve sweeps depending

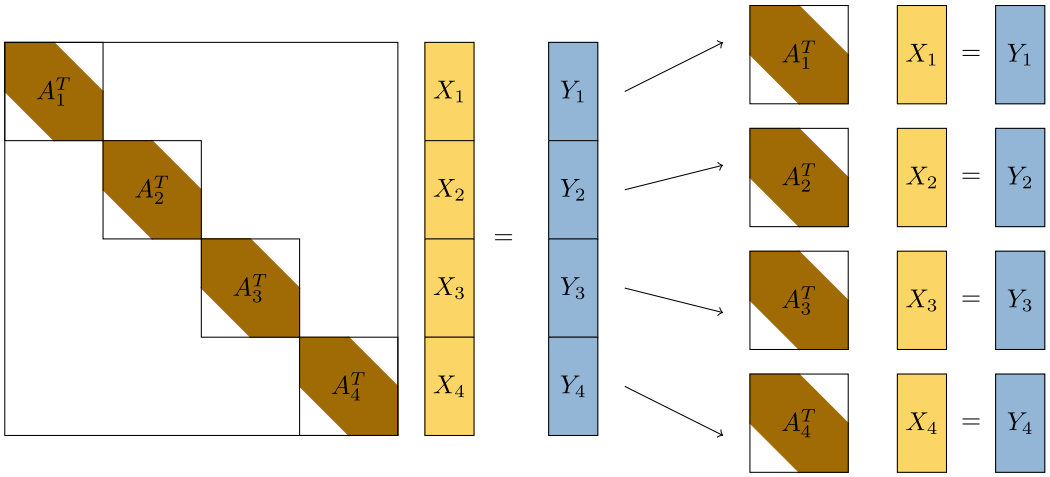



Fig. 9. Transpose D stage.

of the type of partition [Spring 2014]. The fact that it is not necessary to generate the full $W$ spike for the first partition and $V$ spike for the last partition, allows for the creation of a $2 \times 2$ "transpose" kernel, which can be used for developing a flexible threading strategy applied to transpose SPIKE similar to the one presented in Section 3.

### 4.3 Recursive Scheme for Transpose Reduced System

In Section 2.2, a description of the recursive method of solving the reduced system was described. Because the reduced system of transpose SPIKE is simply the transpose of the original reduced system, it suffers from the same problem: increasing the number of partitions increases the size of the reduced system. Therefore, a recursive method for solving the reduced system is also required for the transpose case.

For the transpose reduced system, we aim at reusing the recursive factorization performed for the non-transpose case. The result from a second level of SPIKE DS factorization applied to the original reduced system was given in Equation (21) (using half the number of partitions):

$$S^{[1]} = D^{[1]} S^{[2]}, \tag{73}$$

and this process can be repeated on the new generated spike matrix until only two partitions are left, i.e.,

$$S^{[i]} = D^{[i]} S^{[i+1]}. \tag{74}$$

With each step of this recursion, the number of partition is divided by two and the size of the partitions doubles. If $p$ is the number of partitions into which the original matrix was broken, then the process can be repeated in $r = \log_2(p)$ times [Polizzi and Sameh 2006]. It becomes

$$S^{[1]} = \left(\Pi_{i=1}^{r-1} D^{[i]}\right) S^{[r]}, \tag{75}$$

where $S^{[r]}$ has only two partitions left. For the transpose case, we have $S^T Y^{[1]} = G$ (see Figure 8), so we may perform the transpose operation on the series of products above:

$$S^T = {S^{[r]}}^T \left(\Pi_{i=r-1}^{1} {D^{[i]}}^T\right). \tag{76}$$

This could be thought of as performing the original, non transpose, reduced system solve, but with the solve stages in reverse. The operation to be performed is

$$Y^{[1]} = S^{-T}G = \left(\Pi_{i=1}^{r-1} D^{[i]^{-T}}\right) S^{[r]^{-T}} G. \tag{77}$$

The full process of solving the reduced system using four partitions, is shown in Figures 10 and 11 where non-transpose and transpose cases are detailed side-by-side.

### 4.4 Transpose Solver Performance

Figure 12 shows the solve stage, as well as overall, scaling compared to the single-threaded non-pivoting non-transpose solver. This base solver was chosen to make a one-to-one comparison with the non-transpose solver. Because the factorization is reused for both the transpose and non-transpose problem, factorization time is not shown.

The transpose option has little effect on performance. There is a very slight performance loss in the overall case, and a more noticeable one when just looking at the solve stage. However, in either case, the loss of performance generally occurs well past the point where diminishing returns have already set in, and does not appear to degrade overall performance significantly.

## 5 AN EFFICIENT PIVOTING SCHEME

The standard LAPACK libraries use partial pivoting to increase the numerical stability of the solve operation [Higham 2002]. Partial pivoting operates by exchanging rows when the pivot element is selected, placing the greatest element in the column on the diagonal. This decreases the loss of accuracy caused by rounding, and reduces the chances of selecting zero as the pivot element.

As originally described by Polizzi and Sameh [2006], the recursive SPIKE algorithm is using non-pivoting factorization schemes together with a diagonal boosting strategy. With diagonal boosting, a small value is added to near zero-pivots when they are discovered, resulting in an approximate factorization. As a result, an alternative version of SPIKE that uses partial pivoting factorizations may be desired to improve numerical stability. A partial pivoting SPIKE solver also allows better one to one comparisons with the LAPACK LU solver, but it should noted that the partial-pivoting SPIKE scheme is more constrained due to the requirement that pivots are selected from within each diagonal block.

### 5.1 Pivoting LU Factorization

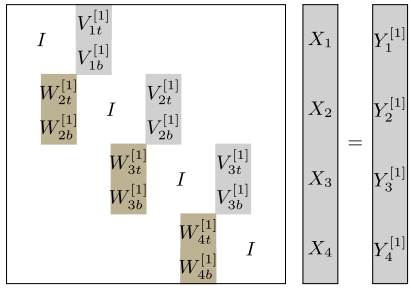
The algorithm implemented for the LAPACK LU factorization is essentially similar to the Doolittle algorithm. In particular, the L and U matrices are crafted column-by-column, progressing from left to right along the diagonal [Du Croz et al. 1990]. As a result, the only legitimate selections for pivot rows are those below the diagonal. In addition, the row selected must have a non-zero value, restricting the choices to those within the band—essentially partial pivoting can pull a row "upwards" at most $k$ elements when applied to a banded matrix. Because the partial pivoting matrix, $P_i$, produces the same action when applied to a matrix or vector, we can exploit this restriction when performing solve operations using the submatrices $A_i$. This will allow us to continue using the optimizations described in Section 2.3. The related operations are performed for Equations (31) and (33).

First, looking at Equation (31), the original equation was

$$V_i = A_i^{-1} B_i = U_i^{-1} L_i^{-1} \begin{bmatrix} 0 & 0 \\ \hat{B}_i & 0 \end{bmatrix}. \tag{78}$$

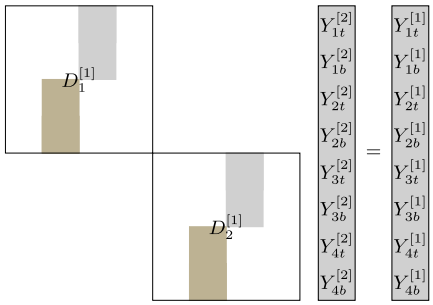
Non-Transpose

The overall problem to be solved is find $X$ in $S^{[1]}X = Y^{[1]}$ for a given $Y^{[1]}$.

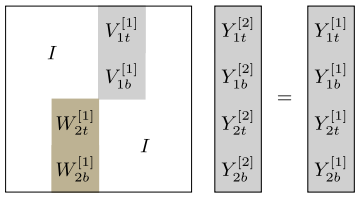


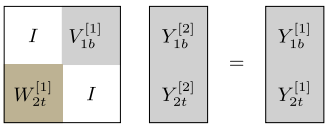
$$S^{[1]}X = D^{[1]}S^{[2]}X = Y^{[1]}$$
$$D^{[1]}Y^{[2]} = Y^{[1]}$$
$$S^{[2]}X = Y^{[2]}$$

The $D$ stage comes first.



Each of the $D_i^{[1]}$ partitions contains a $2 \times 2$ spike matrix.

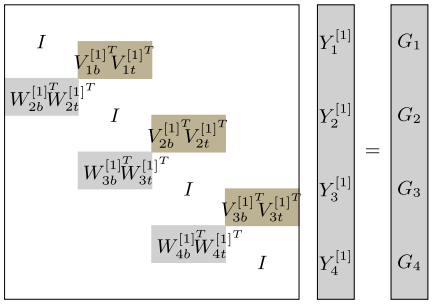




$$Y_{1t}^{[2]} = Y_{1t}^{[1]} - V_{1t}^{[1]}Y_{2t}^{[2]}$$
$$Y_{2b}^{[2]} = Y_{2b}^{[1]} - W_{2b}^{[1]}Y_{1b}^{[2]}$$

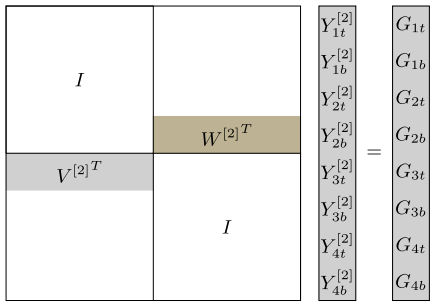
Transpose

The overall problem to be solved is find $Y^{[1]}$ in ${S^{[1]}}^T Y^{[1]} = G$ for a given $G$.



$${S^{[1]}}^T Y^{[1]} = {S^{[2]}}^T {D^{[1]}}^T Y^{[1]} = G$$
$${S^{[2]}}^T Y^{[2]} = G$$
$${D^{[1]}}^T Y^{[1]} = Y^{[2]}$$

The $S$ stage comes first.



$$Y_{1t}^{[2]} = G_{1t} \quad ; \quad Y_{1b}^{[2]} = G_{1b}$$
$$Y_{2t}^{[2]} = G_{2t} \quad ; \quad Y_{3b}^{[2]} = G_{3b}$$
$$Y_{4t}^{[2]} = G_{4t} \quad ; \quad Y_{4b}^{[2]} = G_{4b}$$

Use the $\hat{G}$ vector to exploit the known values in $Y^{[2]}$.

$$\hat{G}_{2b} = G_{2b} - \left(W_{3b}^{[2]\,T} G_{3b} + W_{4t}^{[2]\,T} G_{4t} + W_{4b}^{[2]\,T} G_{4b}\right)$$

$$\hat{G}_{3t} = G_{3t} - \left(V_{1t}^{[2]\,T} G_{1t} + V_{1b}^{[2]\,T} G_{1b} + V_{2t}^{[2]\,T} G_{2t}\right)$$

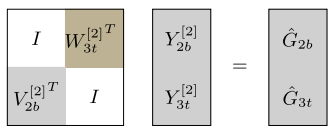
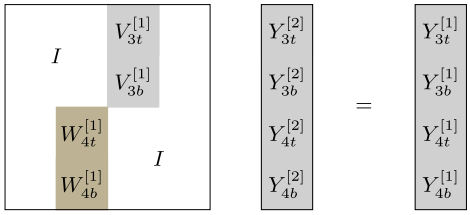
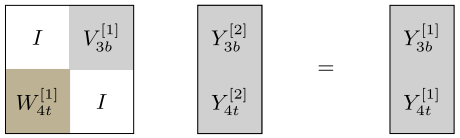


Fig. 10. Recursive SPIKE four partition reduced system solve, part 1.

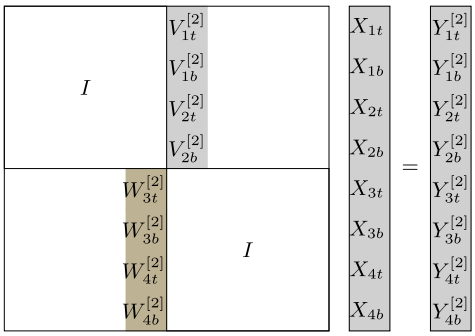
Non-Transpose
(D-stage continued from previous page)

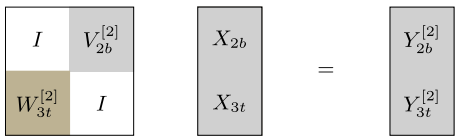




$$Y_{3t}^{[2]} = Y_{3t}^{[1]} - V_{3t}^{[1]} Y_{4t}^{[2]}$$
$$Y_{4b}^{[2]} = Y_{4b}^{[1]} - W_{4b}^{[1]} Y_{3b}^{[2]}$$

Next, the $S$ stage.

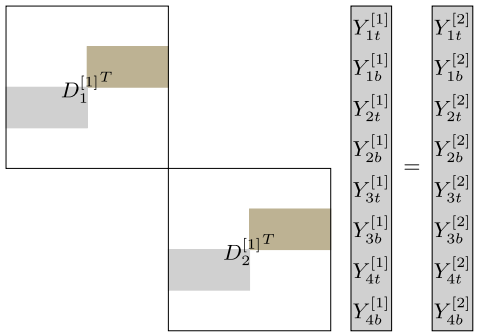




$$X_{1t} = Y_{1t}^{[2]} - V_{1t}^{[2]} X_{3t}$$
$$X_{1b} = Y_{1b}^{[2]} - V_{1t}^{[2]} X_{3t}$$
$$X_{2t} = Y_{2t}^{[2]} - V_{2t}^{[2]} X_{3t}$$
$$X_{3b} = Y_{3b}^{[2]} - W_{3b}^{[2]} X_{2b}$$
$$X_{4t} = Y_{4t}^{[2]} - W_{4t}^{[2]} X_{2b}$$
$$X_{4b} = Y_{4b}^{[2]} - W_{4b}^{[2]} X_{2b}$$

At this point, we have recovered the entire $X$ vector, and so the reduced system is solved.

Transpose
Next, the transposed $D$ stage.

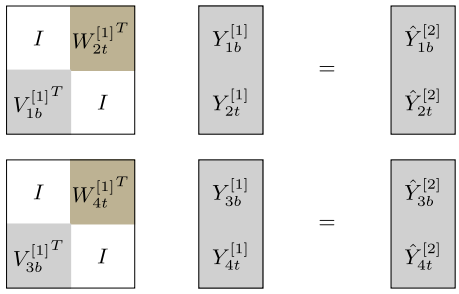


The problems to be solved for each of the submatrices are very similar to the previous stage.

$$Y_{1t}^{[1]} = Y_{1t}^{[2]}$$
$$Y_{2b}^{[1]} = Y_{2b}^{[2]}$$
$$Y_{3t}^{[1]} = Y_{3t}^{[2]}$$
$$Y_{4b}^{[1]} = Y_{4b}^{[2]}$$

Construct the $\hat{Y}^{[2]}$ vectors and use them find the remaining values in $Y^{[1]}$.

$$\hat{Y}_{1b}^{[2]} = Y_{1b}^{[2]} - {W_{2b}^{[1]}}^T Y_{2b}^{[2]}$$
$$\hat{Y}_{2t}^{[2]} = Y_{2t}^{[2]} - {V_{1t}^{[1]}}^T Y_{1t}^{[2]}$$
$$\hat{Y}_{3b}^{[2]} = Y_{3b}^{[2]} - {W_{4b}^{[1]}}^T Y_{4b}^{[2]}$$
$$\hat{Y}_{4t}^{[2]} = Y_{4t}^{[2]} - {V_{3t}^{[1]}}^T Y_{3t}^{[2]}$$



At this point we have found the entire $Y^{[1]}$ vector, and so the reduced system is solved.

Fig. 11. Recursive SPIKE four partition reduced system solve, part 2.

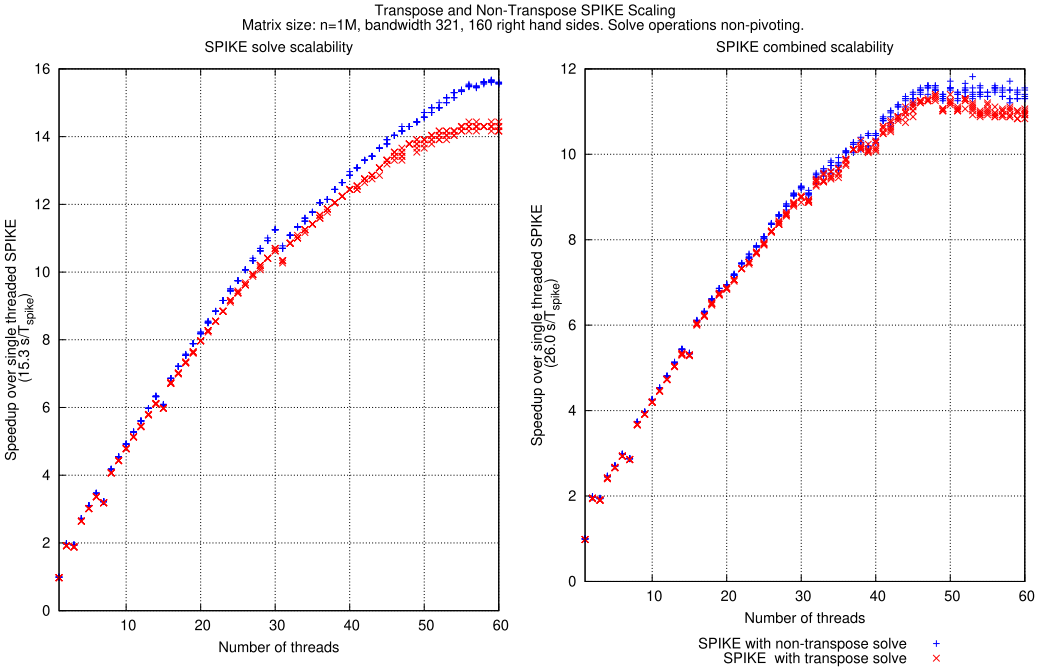



Fig. 12. Computation time comparisons.

With $A_i$ instead factorized as $P_iA_i = L_iU_i$, the permutation matrix must now be inserted as follows:

$$V_i = A_i^{-1}B_i = U_i^{-1}L_i^{-1}P_i \begin{bmatrix} 0 & 0 \\ \hat{B}_i & 0 \end{bmatrix}. \tag{79}$$

When performing solve operation with $L$, we may simply break up the zero-matrices as follows:

$$L_i^{-1}P_i \begin{bmatrix} 0 & 0 \\ \hat{B}_i & 0 \end{bmatrix} = L_i^{-1}P_i \begin{bmatrix} 0 & 0 \\ \hat{0} & \hat{0} \\ \hat{B}_i & 0 \end{bmatrix}, \tag{80}$$

where $\hat{0}$ is a matrix with $k$ rows. Now, we may begin the L-sweep at the top of $\hat{0}$, and any pivoted rows of $B$ will still be involved in the solve operation. From here on, the operations may continue as in non-pivoting SPIKE.

## 5.2 Pivoting UL

There is no UL factorization specified in LAPACK. However, an efficient UL factorization and solve is necessary to reduce the number of solve sweeps used in the last SPIKE partition, as shown in Section 2.3. Specifically, we require the ability to obtain the topmost elements of $W_p$ without using any large sweeps, and limit the contamination caused by the reduced system to the topmost elements of $Y_p$.

Implementing a pivoting UL factorization with performance comparable to, for example, Intel MKL is clearly beyond the scope of this project. Instead, we use a permutation $Q$ with ones on the anti-diagonal, to effectively obtain a UL factorization using the native LAPACK LU factorization. This permutation has the property that pre-multiplying some matrix by $Q$ reverses the order of the rows of that matrix, and post-multiplying a matrix by $Q$ reverses the order of the columns. It

is also orthogonal and symmetric. Thus, a given linear system solve problem using arbitrary $A$, $X$, and $F$ may be rewritten as follows:

$$AX = F = QQAQQX = Q(QAQ)QX, \tag{81}$$

and therefore,

$$X = Q(QAQ)^{-1}QF. \tag{82}$$

Because both the rows and columns of $QAQ$ have been reversed, this matrix is still banded. So, it may still be operated upon using the standard pivoting LU factorization. In addition, the topmost elements of $F$ become the bottom most elements of $QF$. As a result, the successive permutations and triangular solves can be performed from right to left, as follows:

$$X = Q\left(U^{-1}\left(L^{-1}\left(P(QF)\right)\right)\right). \tag{83}$$

Thus, the structure of the collections of vectors used for the final partition is essentially the same as that of the vectors used in the first partition. $QW_p$ has the same structure as $V_1$. And so, we may reuse the same optimizations for the final partition as were used for the first.

### 5.3 Numerical Experiments

*5.3.1 Performance.* The purpose of pivoting SPIKE is to reduce the accuracy loss associated with using a non-pivoting solver, while retaining some of the performance advantage over a pivoting one. So, the relevant metrics are the computation time, scaling, and the residual produced. In comparison with a non-pivoting solver, the use of pivoting has two noticeable performance impacts. First, during the factorization, the pivot element is selected by scanning through the column and locating the element with the greatest magnitude. Second, when the matrix is not diagonally dominant, there is a cost associated with performing the swapping operation.

For the sake of these comparisons, it is useful to vary both the number of threads and the diagonal dominance of the matrix. As a slight extension to the concept of a diagonally dominant matrix, let us define $DD$, the "degree of diagonal dominance," as follows:

$$DD = \min_{i \in 1 \ldots n} \left( \frac{|A_{ii}|}{\sum_{j \neq i} |A_{ji}|} \right). \tag{84}$$

A diagonally dominant matrix would have $DD \geq 1$. To generate matrices with a desired value for $DD$, the following procedure has been used: Each element within the non-zero band of the matrix has been filled with random values using the LAPACK DLARNV command. Then, the columns are summed and multiplied by the desired value for $DD$ and the result is placed on the diagonal.

Figure 13 shows the overall performance comparisons for non-pivoting SPIKE, pivoting SPIKE (partition ratios computed via the method discussed in Section 3.3), and MKL. Note that computation time is plotted on a log scale to retain the visibility of performance changes for large numbers of threads. The hardware and software used for these runs were detailed in Section 3.4. Two matrix configurations are used, one in which the matrix is diagonally dominant ($DD = 1.5$), and one in which it is not ($DD = 10^{-3}$). Non-pivoting SPIKE clearly demonstrates the best performance. Pivoting SPIKE and MKL perform well in different conditions, with MKL obtaining a noticeable advantage for low numbers of threads—the additional cost of not having a dedicated and optimal pivoting UL factorization is a likely cause of this issue (involving also an additional permutation in the solve stage). SPIKE improves in performance as the number of threads increases. In particular, the MKL factorization stage does not scale well beyond 10 threads on this machine, most likely because at this point the computation begins to access additional processor packages. These experiments appear to indicate that even coupled with the previously described partial pivoting, SPIKE maintains performance scalability.

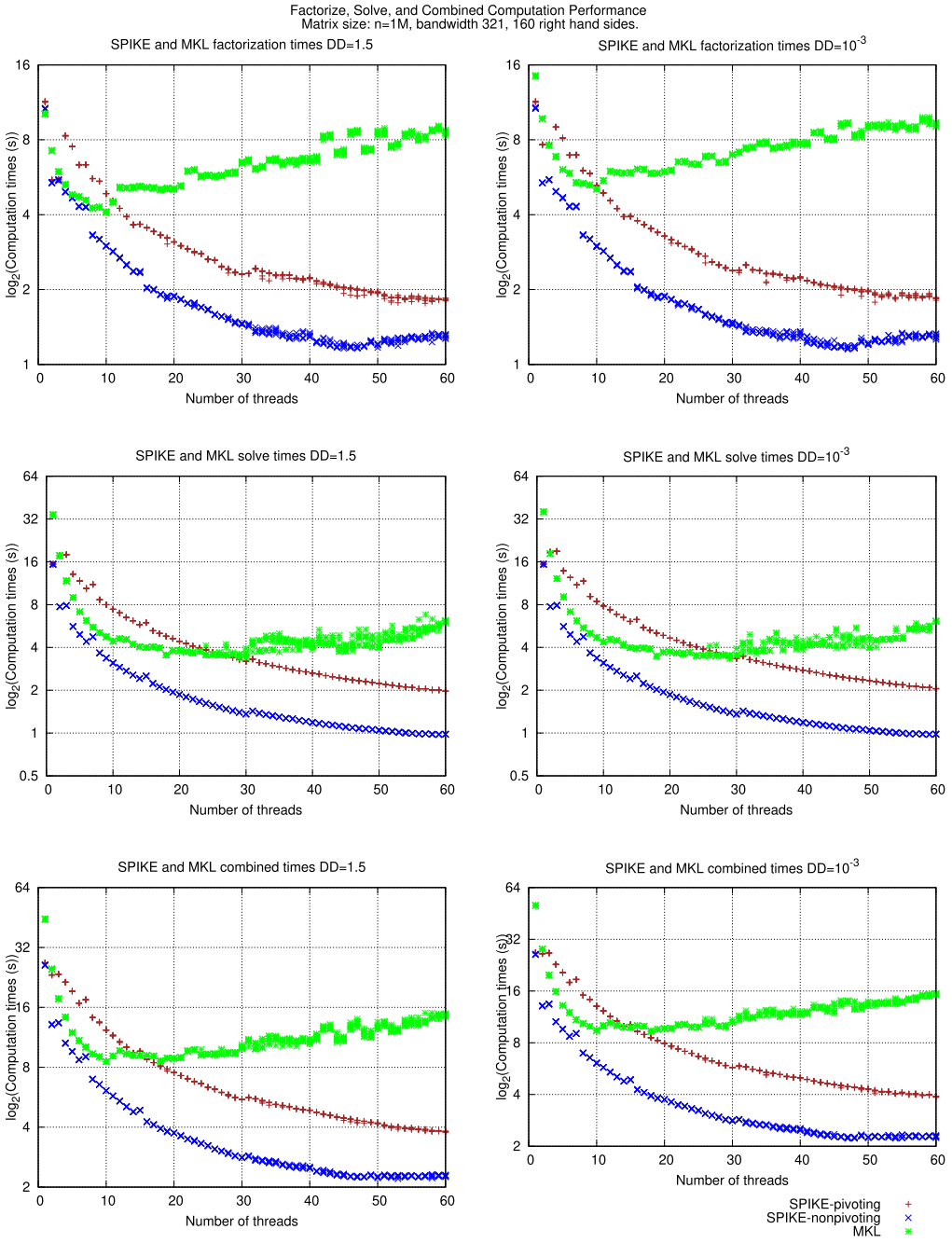



Fig. 13. Computation time comparisons.

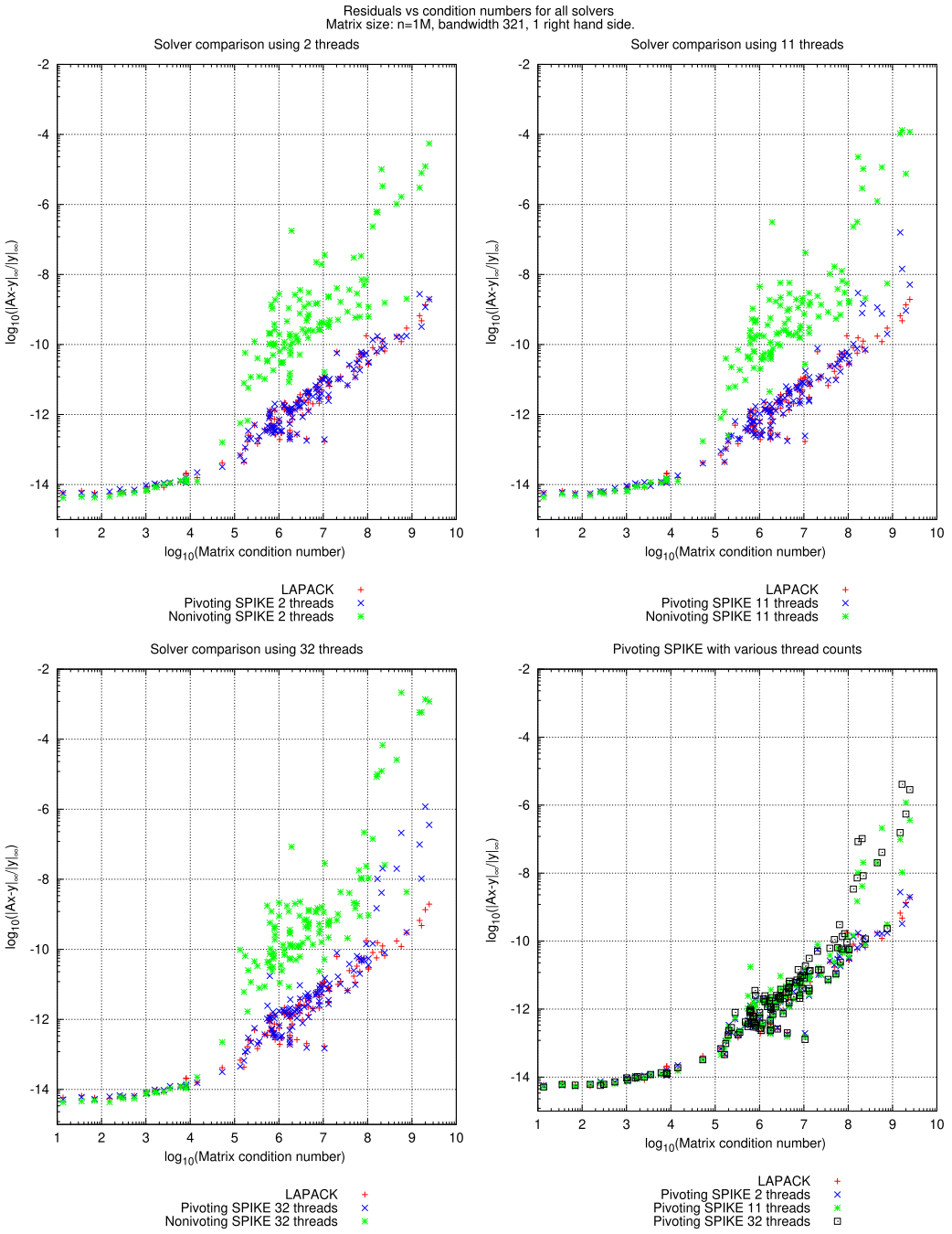



Fig. 14. Condition number and residual relationship.

*5.3.2 Accuracy.* Figure 14 shows the numerical accuracy advantages of pivoting SPIKE, by comparing the residual produced to the condition number. Matrices are produced in the same manner as the preceding section, and the condition number is estimated by the LAPACK function DGBCON. Using this setup, the condition number is highly correlated to the degree of diagonal dominance. All computations are performed in double precision.

The top-left, top-right, and bottom-left quadrants of the figure compare the three solvers. In the top-left quadrant it can be seen that, with two-partitions, pivoting SPIKE produces residuals indistinguishable from LAPACK. Results for non-pivoting SPIKE are also comparable for condition numbers less than $10^5$. The residuals start increasing after this point for all solvers, with a noticeable much higher increase for non-pivoting SPIKE. In the top-right and bottom-left quadrants, we see some loss of accuracy for the pivoting SPIKE, particularly as the condition number becomes very large.

The bottom-right quadrant shows a comparison of pivoting solvers for all thread counts used. Viewing this chart, it becomes apparent that there are three relevant ranges for the computation. For condition numbers in the range of 1 to $10^5$, all of the solvers perform well. For condition number in the range from $10^5$ to $10^8$, the residuals produced by the pivoting solvers are essentially identical. Finally, for condition numbers greater than $10^8$ there is some loss of accuracy for pivoting SPIKE based on the number of partitions used. These limitations could be explained by the facts that pivoting in SPIKE does not take place across partitions, and the reduced system may also inherit the poor conditioning of the original system.

## 6 CONCLUSION

A feature-complete recursive SPIKE algorithm has been presented. Three enhancements for SPIKE have been shown, achieving near feature-parity with the standard LAPACK banded linear system solver. In particular, both the transpose solve option and the partial pivoting option, provide standard capabilities found in LAPACK solvers. Transpose solve operation allows improved algorithmic flexibility and efficiency by eliminating the need for an additional transpose factorization. Pivoting operation provides a convenient middle-ground between the numerical accuracy of the standard LAPACK solver and the extreme scalability of the standard SPIKE algorithm.

All algorithms have been implemented with a flexible threading scheme that allows the effective utilization of any number of threads, overcoming a previous known limitation of the recursive SPIKE scheme. In addition, the per-partition performance has been characterized, resulting in a simple load-balancing equation controlled by a single machine specific parameter. With the addition of these features and demonstrated performance advantages, it is our hope that the new SPIKE-OpenMP library [SPIKE-library 2018] may be considered a drop-in replacement for the standard LAPACK banded factorize and solve operations.

## REFERENCES

E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'90)*. IEEE Computer Society Press, Los Alamitos, CA, 2–11. Retrieved from http://dl.acm.org/citation.cfm?id=110382.110385.

M. W. Berry and A. H. Sameh. 1988. Multiprocessor schemes for solving block tridiagonal linear systems. *Int. J. Supercomput. Appl.* 2, 3 (1988), 37–57. DOI: https://doi.org/10.1177/109434208800200304

L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu. 2012. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking,*

*Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 27, 11 pages. Retrieved from http://dl.acm.org/citation.cfm?id=2388996.2389033.

S. C. Chen, D. J. Kuck, and A. H. Sameh. 1978. Practical parallel band triangular system solvers. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 270–277. DOI: https://doi.org/10.1145/355791.355797

E. Cuthill and J. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference (ACM'69)*. ACM, New York, NY, 157–172. DOI: https://doi.org/10.1145/800195.805928

J. J. Dongarra and A. H. Sameh. 1984. On some parallel banded system solvers. *Parallel Comput.* 1, 3–4 (Dec. 1984), 223–235. DOI: https://doi.org/10.1016/S0167-8191(84)90165-0

J. Du Croz, P. Mayes, and G. Radicati. 1990. *Factorizations of Band Matrices Using Level 3 BLAS*. Technical Report 21. LAPACK Working Note. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn21.pdf.

V. Eijkhout and R. van de Geijn. 2012. The spike factorization as a domain decomposition method: Equivalent and variant approaches. In *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied (Eds.). Springer, London, 157–170.

FEAST-library. 2009-2020. FEAST Eigenvalue Solver. Retrieved from http://www.feast-solver.org/.

K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, E. Polizzi, Y. Saad, F. Saied, and D. Sorensen. 2012. Parallel numerical computing from Illiac IV to exascale—The contributions of Ahmed H. Sameh. In *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied (Eds.). Springer, London, 1–44. DOI: https://doi.org/10.1007/978-1-4471-2437-5_1

E. Gallopoulos, P. Bernard, and A. H. Sameh. 2016. *Parallelism in Matrix Computations*. Springer.

N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Chapter 9, 157–193. DOI: https://doi.org/10.1137/1.9780898718027.ch9

J. Kestyn, E. Polizzi, and P. T. P. Tang. 2016. Feast eigensolver for non-hermitian problems. *SIAM J. Sci. Comput.* 38, 5 (2016). DOI: https://doi.org/10.1137/15M1026572

D. H. Lawrie and A. H. Sameh. 1984. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.* 10, 2 (May 1984), 185–195. DOI: https://doi.org/10.1145/399.401

A. Li, A. Seidl, R. Serban, and D. Negrut. 2014. *SPIKE::GPU—A SPIKE-based Preconditioned GPU Solver for Sparse Linear Systems*. Technical Report.

M. Manguoglu, M. Koyutürk, A. H. Sameh, and A. Grama. 2010. Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM J. Sci. Comput.* 32, 3 (2010), 1201–1216.

M. Manguoglu, F. Saied, A. H. Sameh, and A. Grama. 2011. Performance models for the Spike banded linear system solver. *Sci. Program.* 19, 1 (2011), 13–25.

M. Manguoglu, A. H. Sameh, and O. Schenk. 2009. PSPIKE: A parallel hybrid sparse linear system solver. In *Proceedings of the International Conference on Parallel Computing (Euro-Par'09)*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.). Springer, Berlin, 797–808.

K. Mendiratta and E. Polizzi. 2011. A threaded "SPIKE" algorithm for solving general banded systems. *Parallel Comput.* 37, 12 (2011), 733–741. DOI: https://doi.org/10.1016/j.parco.2011.09.003

C. Mikkelsen and M. Manguoglu. 2009. Analysis of the truncated SPIKE algorithm. *SIAM J. Matrix Anal. Appl.* 30, 4 (2009), 1500–1519. DOI: https://doi.org/10.1137/080719571

M. Naumov, M. Manguoglu, and A. H. Sameh. 2010. A tearing-based hybrid parallel sparse linear system solver. *J. Comput. Appl. Math.* 234, 10 (2010), 3025–3038.

E. Polizzi. 2009. Density-matrix-based algorithm for solving eigenvalue problems. *Phys. Rev. B* 79 (Mar. 2009), 115112. Issue 11. DOI: https://doi.org/10.1103/PhysRevB.79.115112

E. Polizzi. 2011. SPIKE. In *Encyclopedia of Parallel Computing*, D. Padua (Ed.). Springer U.S., 1912–1920. DOI: https://doi.org/10.1007/978-0-387-09766-4_88

E. Polizzi. 2020. FEAST eigenvalue solver v4.0 user guide. Retrieved from arxiv:cs.MS/2002.04807.

E. Polizzi and N. Ben Abdallah. 2005. Subband decomposition approach for the simulation of quantum electron transport in nanostructures. *J. Comput. Phys.* 202, 1 (Jan. 2005), 150–180. DOI: https://doi.org/10.1016/j.jcp.2004.07.003

E. Polizzi and A. Sameh. 2006. A parallel hybrid banded system solver: The SPIKE algorithm. *Parallel Comput.* 32, 2 (2006), 177–194. DOI: https://doi.org/10.1016/j.parco.2005.07.005

E. Polizzi and A. Sameh. 2007. SPIKE: A parallel environment for solving banded linear systems. *Comput. Fluids* 36, 1 (2007), 113–120. DOI: https://doi.org/10.1016/j.compfluid.2005.07.005

A. H. Sameh and D. J. Kuck. 1978. On stable parallel linear system solvers. *J. ACM* 25, 1 (Jan. 1978), 81–91. DOI: https://doi.org/10.1145/322047.322054

A. H. Sameh and V. Sarin. 1999. Hybrid parallel linear system solvers. *Int. J. Comput. Fluid Dynam.* 12, 3–4 (1999), 213–223. DOI: https://doi.org/10.1080/10618569908940826

SPIKE-library. 2018. SPIKE shared-memory solver, v1.0. Retrieved from http://www.spike-solver.org/.

SPIKE-MPI-library. 2011. Intel Adaptive Spike-based Solver. Retrieved from https://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/.

B. S. Spring. 2014. *Enhanced Capabilities of the Spike Algorithm and a New Spike-OpenMP Solver.* Master's thesis. University of Massachusetts, Amherst. Retrieved from http://scholarworks.umass.edu/masters_theses_2/116.

Ioannis E. Venetis, Alexandros Kouris, Alexandros Sobczyk, Efstratios Gallopoulos, and Ahmed H. Sameh. 2015. A direct tridiagonal solver based on Givens rotations for GPU architectures. *Parallel Comput.* 49 (2015), 101–116.

Received November 2018; revised July 2020; accepted July 2020