November 2014

Enhanced Capabilities of the Spike Algorithm and a New Spike-OpenMP Solver

Braegan S. Spring
University of Massachusetts Amherst

ENHANCED CAPABILITIES OF THE SPIKE ALGORITHM AND A NEW SPIKE-OPENMP SOLVER

A Thesis Presented

by

BRAEGAN SPRING

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2014

Electrical and Computer Engineering

ENHANCED CAPABILITIES OF THE SPIKE ALGORITHM AND A NEW SPIKE-OPENMP SOLVER

A Thesis Presented

by

BRAEGAN SPRING

Approved as to style and content by:

Eric Polizzi, Chair

Zlatan Aksamija, Member

Hans Johnston, Member

Christopher V. Hollot , Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank Dr. Zlatan Aksamija & Dr. Hans Johnston for assisting me in this project, Dr. Polizzi for guiding me through it, and Dr. Anderson for his help beforehand.

ABSTRACT

ENHANCED CAPABILITIES OF THE SPIKE ALGORITHM AND A NEW SPIKE-OPENMP SOLVER

SEPTEMBER 2014

BRAEGAN SPRING

B.Sc., UNIVERSITY MASSACHUSETTS AMHERST

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Eric Polizzi

SPIKE is a parallel algorithm to solve block tridiagonal matrices. In this work, two useful improvements to the algorithm are proposed. A flexible threading strategy is developed, to overcome limitations of the recursive reduced system method. Allocating multiple threads to some tasks created by the SPIKE algorithm removes the previous restriction that recursive SPIKE may only use a number of threads equal to a power of two. Additionally, a method of solving transpose problems is shown. This method matches the performance of the non-transpose solve while reusing the original factorization.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

Linear systems are a basic tool, frequently used to to express our understanding of the natural and engineering world. Because of this, high quality linear algebra software is one of the cornerstones of computational science. Two well known examples of such software are BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage). These set 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. However, taking advantage of these improvements can be a burden for programmers due to the added complexity involved in parallelizing an algorithm. As a result, it is a often helpful to produce parallelized implementations of common subroutines. This allows users to take advantage of the capabilities of their hardware without redesigning their algorithms. Because of the popularity of linear systems, linear algebra subroutines are a good candidate for this task.

In this work we will discuss enhancements to and an implementation of the SPIKE algorithm. The end result is easy to use linear system solver for banded systems. This solver duplicates the features of the LAPACK banded solver, and uses the well supported OpenMP* threading library, for ease of use and installation.

1.2 The SPIKE Algorithm

1.2.1 Overview

The SPIKE algorithm is a domain decomposition method for solving block tridiagonal matrices. It can be traced back to work done by A. Sameh and associates on banded system in the late seventies. [2, 7] Since then, it has been developed and adapted for a number of special cases, such as diagonally dominant [3] and positive definite matrices [4]

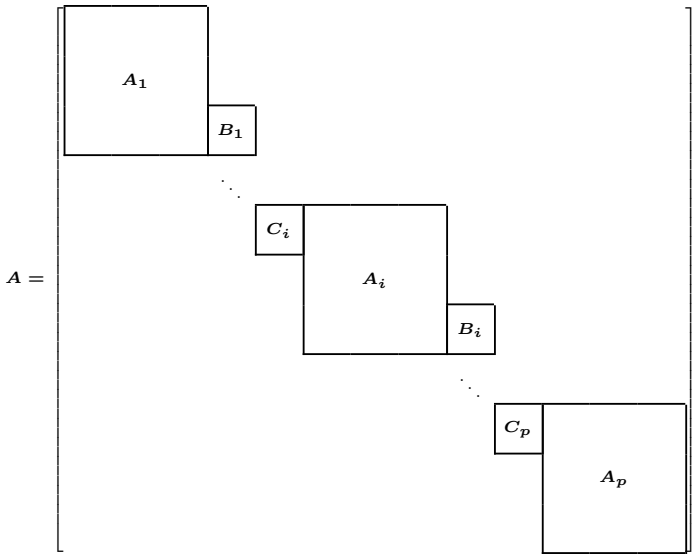
SPIKE is a flexible algorithm, and can be tuned for large scale distributed or consumer level multi-core systems. Parallelism is extracted by decoupling the relatively large blocks along the diagonal, solving them independently, and then reconstructing the system via the use of smaller reduced systems. There are a number of versions of the SPIKE algorithm, which handle the specifics of those steps in different ways. [6] We will begin by briefly discussing a simple version at a high level.

The overall problem to be solved is

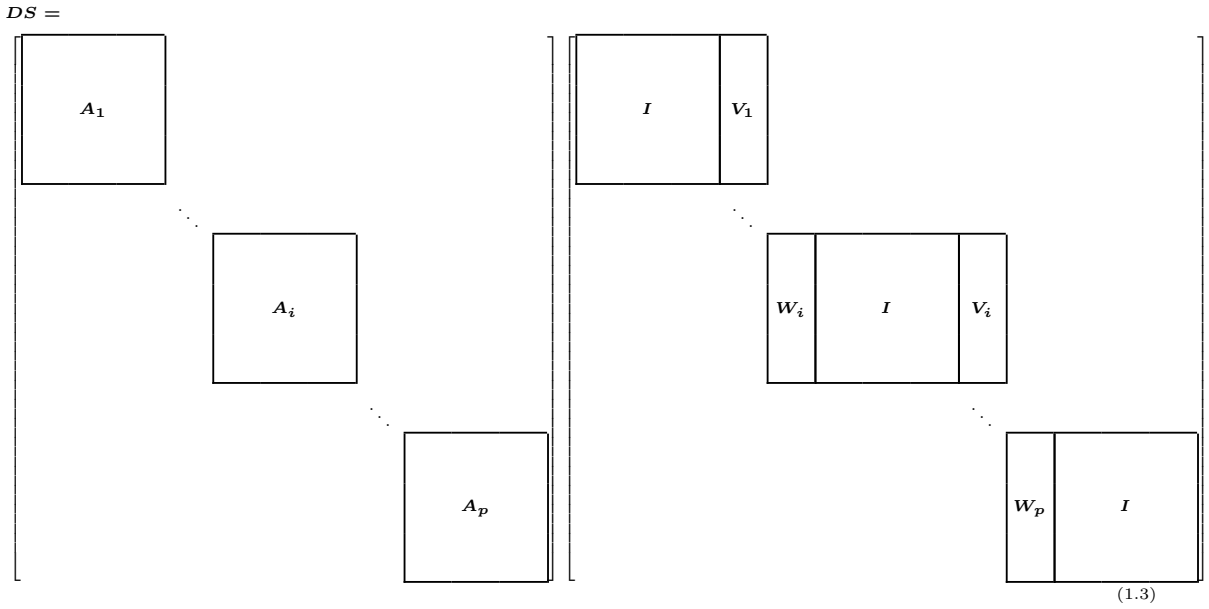
$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f} \text{ for } \boldsymbol{x} \tag{1.1}$$

Where $\boldsymbol{A}$ and $\boldsymbol{f}$ are known. In general for SPIKE, $\boldsymbol{A}$ must be block tridiagonal. However, for this work we will consider only a diagonal matrix with constant bandwidth. $\boldsymbol{A}$ is a $n \times n$ diagonal banded matrix, with upper and lower bandwidths k_u and k_l respectively. $\boldsymbol{x}$ and $\boldsymbol{f}$ are vectors with n elements each. It is possible that we would like to solve for multiple vectors, and we use the notation n_{rhs}(number of right hand sides) to represent the number of $\boldsymbol{x}$ or $\boldsymbol{f}$ vectors.

The first step of the SPIKE algorithm is the factorization stage. In this stage, we separate the matrix $\boldsymbol{A}$ into the matrices $\boldsymbol{D}$ and $\boldsymbol{S}$. This is the SPIKE factorization.



(1.2)



(1.3)

The $\boldsymbol{A_i}$ sub-matrices are the aforementioned large blocks along the diagonal of $\boldsymbol{A}$. These blocks are not necessarily all the same size. Instead, the size of a given sub-matrix is $n_i \times n_i$. The $\boldsymbol{V_i}$ and $\boldsymbol{W_i}$ matrices may be thought of as the coupling between the $\boldsymbol{A_i}$ matrices and their neighbors. They are formed as follows.

$$\boldsymbol{A_i V_i} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_i} \end{bmatrix} \rightarrow \boldsymbol{A_i^{-1}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_i} \end{bmatrix} = \boldsymbol{V_i} \tag{1.4a}$$

$$\boldsymbol{A_i W_i} = \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix} \rightarrow \boldsymbol{A_i^{-1}} \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{W_i} \tag{1.4b}$$

The $\boldsymbol{B_i}$ and $\boldsymbol{C_i}$ matrices contain the elements of $\boldsymbol{A}$ along the diagonal that are outside of $\boldsymbol{A_i}$. They are $k_u \times k_u$ and $k_l \times k_l$ respectively. Therefore, the $\boldsymbol{V_i}$ and $\boldsymbol{W_i}$ matrices are $n_i \times k_u$ and $n_i \times k_l$.

After the factorization in to $\boldsymbol{D}$ and $\boldsymbol{S}$, we enter the solve stage. We obtain two sub-problems:

$$\boldsymbol{Ax} = \boldsymbol{DSx} = \boldsymbol{f} \tag{1.5a}$$

$$\boldsymbol{Dy} = \boldsymbol{f} \tag{1.5b}$$

$$\boldsymbol{Sx} = \boldsymbol{y} \tag{1.5c}$$

The vectors $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{f}$ may be broken in to segments $\boldsymbol{x_i}$, $\boldsymbol{y_i}$ and $\boldsymbol{f_i}$ of height n_i. Because the blocks of $\boldsymbol{D}$ are uncoupled, 1.5b may be performed in parallel, as

$$\mathbf{y_i} = \mathbf{A_i^{-1}f_i} \text{ for all } i \in 1...p \tag{1.6}$$

This is all that is required for the $\boldsymbol{D}$ stage. The next subproblem is the $\boldsymbol{S}$ stage. Because of the structure of the $\boldsymbol{S}$ matrix, most of the work for this stage can actually be performed by multiplication. For this stage, we will have to further segment the $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors. We will require the tips of these vectors to be separated from the main body: $\boldsymbol{x_i} = \begin{bmatrix} \boldsymbol{x_{it}} \\ \tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ \boldsymbol{x_{ib}} \end{bmatrix}$. The height of the vectors $\boldsymbol{x_{it}}$ and $\boldsymbol{x_{ib}}$ are k_u and k_l respectively. The vector $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ takes the remainder of the elements, so it has a height of $n_i - (k_l + k_u)$ elements. Similarly, the $\boldsymbol{V_i}$ and $\boldsymbol{W_i}$ matrices can be segmented to $\boldsymbol{V_i} = \begin{bmatrix} \boldsymbol{V_{it}} \\ \tilde{\boldsymbol{V}}_{\boldsymbol{i}} \\ \boldsymbol{V_{ib}} \end{bmatrix}$ and $\boldsymbol{W_i} = \begin{bmatrix} \boldsymbol{W_{it}} \\ \tilde{\boldsymbol{W}}_{\boldsymbol{i}} \\ \boldsymbol{W_{ib}} \end{bmatrix}$. Thus, the $\boldsymbol{S}$ matrix becomes:

$$
\boldsymbol{Sx = y}
$$

$$
= \begin{bmatrix}
\boldsymbol{I} & \begin{matrix}\boldsymbol{V_{1t}}\\ \boldsymbol{\tilde{V}_1}\\ \boldsymbol{V_{1b}}\end{matrix} & & \\
& \begin{matrix}\boldsymbol{W_{it}}\\ \boldsymbol{\tilde{W}_i}\\ \boldsymbol{W_{ib}}\end{matrix} & \boldsymbol{I} & \begin{matrix}\boldsymbol{V_{it}}\\ \boldsymbol{\tilde{V}_i}\\ \boldsymbol{V_{ib}}\end{matrix} \\
& & \begin{matrix}\boldsymbol{W_{pt}}\\ \boldsymbol{\tilde{W}_p}\\ \boldsymbol{W_{pb}}\end{matrix} & \boldsymbol{I}
\end{bmatrix}
\begin{bmatrix}
\boldsymbol{x_{1t}}\\ \boldsymbol{\tilde{x}_i}\\ \boldsymbol{x_{1b}}\\ \vdots\\ \boldsymbol{x_{it}}\\ \boldsymbol{\tilde{x}_i}\\ \boldsymbol{x_{ib}}\\ \vdots\\ \boldsymbol{x_{pt}}\\ \boldsymbol{\tilde{x}_p}\\ \boldsymbol{x_{pb}}
\end{bmatrix}
=
\begin{bmatrix}
\boldsymbol{y_{1t}}\\ \boldsymbol{\tilde{y}_i}\\ \boldsymbol{y_{1b}}\\ \vdots\\ \boldsymbol{y_{it}}\\ \boldsymbol{\tilde{y}_i}\\ \boldsymbol{y_{ib}}\\ \vdots\\ \boldsymbol{y_{pt}}\\ \boldsymbol{\tilde{y}_p}\\ \boldsymbol{y_{pb}}
\end{bmatrix}
\tag{1.7}
$$

After performing this multiplication, we obtain a simpler set of equations.

$$
\begin{bmatrix}\boldsymbol{y_{1t}}\\ \boldsymbol{\tilde{y}_1}\\ \boldsymbol{y_{1b}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{x_{1t}}\\ \boldsymbol{\tilde{x}_1}\\ \boldsymbol{x_{1b}}\end{bmatrix} + \begin{bmatrix}\boldsymbol{V_{1t}}\\ \boldsymbol{\tilde{V}_1}\\ \boldsymbol{V_{1b}}\end{bmatrix} \boldsymbol{x_{2t}} \tag{1.8a}
$$

$$
\begin{bmatrix}\boldsymbol{y_{it}}\\ \boldsymbol{\tilde{y}_i}\\ \boldsymbol{y_{ib}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{x_{it}}\\ \boldsymbol{\tilde{x}_i}\\ \boldsymbol{x_{ib}}\end{bmatrix} + \begin{bmatrix}\boldsymbol{V_{it}}\\ \boldsymbol{\tilde{V}_i}\\ \boldsymbol{V_{ib}}\end{bmatrix} \boldsymbol{x_{i+1t}} + \begin{bmatrix}\boldsymbol{W_{it}}\\ \boldsymbol{\tilde{W}_i}\\ \boldsymbol{W_{ib}}\end{bmatrix} \boldsymbol{x_{i-1b}} \tag{1.8b}
$$

$$
\begin{bmatrix}\boldsymbol{y_{pt}}\\ \boldsymbol{\tilde{y}_p}\\ \boldsymbol{y_{pb}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{x_{pt}}\\ \boldsymbol{\tilde{x}_p}\\ \boldsymbol{x_{pb}}\end{bmatrix} + \begin{bmatrix}\boldsymbol{W_{pt}}\\ \boldsymbol{\tilde{W}_p}\\ \boldsymbol{W_{pb}}\end{bmatrix} \boldsymbol{x_{p-1b}} \tag{1.8c}
$$

Notice that the top and bottoms tips of the $\boldsymbol{x_i}$ and $\boldsymbol{y_i}$ vectors are independent of the middle sections. We can use this fact to extract a reduced system. This reduced system will give us $\boldsymbol{x_{it}}$ and $\boldsymbol{x_{ib}}$, which we can then use to retrieve $\boldsymbol{\tilde{x}_i}$.

The removal of the middle sections middle sections of the equations in 1.8 results in the following reduced system.

$$\begin{bmatrix} \boldsymbol{y_{1t}} \\ \boldsymbol{y_{1b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x_{1t}} \\ \boldsymbol{x_{1b}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{V_{1t}} \\ \boldsymbol{V_{1b}} \end{bmatrix} \boldsymbol{x_{2t}} \tag{1.9a}$$

$$\begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{y_{ib}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x_{it}} \\ \boldsymbol{x_{ib}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{V_{it}} \\ \boldsymbol{V_{ib}} \end{bmatrix} \boldsymbol{x_{(i+1)t}} + \begin{bmatrix} \boldsymbol{W_{it}} \\ \boldsymbol{W_{ib}} \end{bmatrix} \boldsymbol{x_{i-1b}} \tag{1.9b}$$

$$\begin{bmatrix} \boldsymbol{y_{pt}} \\ \boldsymbol{y_{pb}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x_{pt}} \\ \boldsymbol{x_{pb}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{W_{pt}} \\ \boldsymbol{W_{pb}} \end{bmatrix} \boldsymbol{x_{p-1b}} \tag{1.9c}$$

$$\begin{bmatrix} \boldsymbol{y_{1t}} \\ \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{y_{2b}} \\ \vdots \\ \boldsymbol{y_{i-1t}} \\ \boldsymbol{y_{i-1b}} \\ \boldsymbol{y_{it}} \\ \boldsymbol{y_{ib}} \\ \vdots \\ \boldsymbol{y_{p-1t}} \\ \boldsymbol{y_{p-1b}} \\ \boldsymbol{y_{pt}} \\ \boldsymbol{y_{pb}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V_{1t}} & & & & & & & & & & & \\ & \boldsymbol{I} & \boldsymbol{V_{1b}} & & & & & & & & & & & \\ \boldsymbol{W_{2t}} & & \boldsymbol{I} & & & & & & & & & & & \\ \boldsymbol{W_{2b}} & & & \boldsymbol{I} & & & & & & & & & & \\ & & & & \ddots & & & & & & & & & \\ & & & & & \boldsymbol{I} & & \boldsymbol{V_{i-1t}} & & & & & & \\ & & & & & & \boldsymbol{I} & \boldsymbol{V_{i-1b}} & & & & & & \\ & & & & & \boldsymbol{W_{it}} & & \boldsymbol{I} & & & & & & \\ & & & & & \boldsymbol{W_{ib}} & & & \boldsymbol{I} & & & & & \\ & & & & & & & & & \ddots & & & & \\ & & & & & & & & & & \boldsymbol{I} & & \boldsymbol{V_{p-1b}} & \\ & & & & & & & & & & & \boldsymbol{I} & \boldsymbol{V_{p-1b}} & \\ & & & & & & & & & & \boldsymbol{W_{pt}} & & \boldsymbol{I} & \\ & & & & & & & & & & \boldsymbol{W_{pb}} & & & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_{1t}} \\ \boldsymbol{x_{1b}} \\ \boldsymbol{x_{2t}} \\ \boldsymbol{x_{2b}} \\ \vdots \\ \boldsymbol{x_{i-1t}} \\ \boldsymbol{x_{i-1b}} \\ \boldsymbol{x_{it}} \\ \boldsymbol{x_{ib}} \\ \vdots \\ \boldsymbol{x_{p-1t}} \\ \boldsymbol{x_{p-1b}} \\ \boldsymbol{x_{pt}} \\ \boldsymbol{x_{pb}} \end{bmatrix} \tag{1.10}$$

This matrix may hypothetically be solved in a direct manner, resulting in values for the $\boldsymbol{x_{it}}$ and $\boldsymbol{x_{ib}}$. These may, in turn, be used to retrieve the rest of the values for $\boldsymbol{x}$ from 1.8. After this step, the problem has been solved.

$$\tilde{\boldsymbol{x}}_{\boldsymbol{1}} = \tilde{\boldsymbol{y}}_{\boldsymbol{1}} - \tilde{\boldsymbol{V}}_{\boldsymbol{1}} \boldsymbol{x_{2t}} \tag{1.11a}$$

$$\tilde{\boldsymbol{x}}_{\boldsymbol{i}} = \tilde{\boldsymbol{y}}_{\boldsymbol{i}} - \left(\tilde{\boldsymbol{V}}_{\boldsymbol{i}} \boldsymbol{x_{i+1t}} + \tilde{\boldsymbol{W}}_{\boldsymbol{i}} \boldsymbol{x_{i-1t}} \right) \tag{1.11b}$$

$$\tilde{\boldsymbol{x}}_{\boldsymbol{p}} = \tilde{\boldsymbol{y}}_{\boldsymbol{p}} - \tilde{\boldsymbol{W}}_{\boldsymbol{p}} \boldsymbol{x_{p-1t}} \tag{1.11c}$$

There are a number of areas for improvement in this method. The first relates to the reduced system. The reduced system in this problem is $2(k_l + k_u)p$ elements in size. This is much smaller than the original $\boldsymbol{A}$ matrix, but as the number of threads is increased, the size of this reduced system is increased. Because the reduced system is not solved in parallel, eventually it would become large enough to cause slowdown.

The next problem relates to the $\boldsymbol{V_i}$ and $\boldsymbol{W_i}$ matrices. These matrices are fairly large. On a shared memory machine, as the number of cores increases the system bandwidth to memory is not necessarily increased. As a result, the scheme discussed in this section is likely to become starved for memory bandwidth prematurely. This is particularly noticeable in the solve stage of the problem, as there is less work to do per element. Therefore, it would be an improvement if we could avoid working with $\boldsymbol{V}$ and $\boldsymbol{W}$ explicitly in the solve stage.

Finally, and relatedly, we would like to reduce the total number of solves operations required. When creating the $\boldsymbol{S}$ matrix, we must perform a solve for each $\boldsymbol{V_i} = \boldsymbol{A_i^{-1}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_i} \end{bmatrix}$, $\boldsymbol{W_i} = \boldsymbol{A_i^{-1}} \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix}$. We must also perform the solves for the blocks of the $\boldsymbol{D}$ matrix, $\boldsymbol{y_i} = \boldsymbol{A_i^{-1}} \boldsymbol{f_i}$. We would like to minimize the total number of large solves as much as possible.

1.2.2 Recursive implicit SPIKE with LU/UL factorization

This section describes the recursive, implicit SPIKE algorithm with LU/UL factorization. Each of those terms will be explained shortly. This is the base algorithm upon which the improvements described later in this paper have been made. We will begin by discussing the implicit treatment of the $\boldsymbol{V}$ and $\boldsymbol{W}$ spikes.

1.2.2.1 Implicit V and W matrices

Let us reconsider the state of the problem immediately after solving the reduced system. With the reduced system solved, we now have $\boldsymbol{x_{it}}$ and $\boldsymbol{x_{ib}}$. We also have, from the $\boldsymbol{D}$ stage

$$\boldsymbol{y_i} = \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{\tilde{y}_i} \\ \boldsymbol{y_{ib}} \end{bmatrix} = \boldsymbol{A_i^{-1} f_i} = \boldsymbol{A_i^{-1}} \begin{bmatrix} \boldsymbol{f_{it}} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{f_{ib}} \end{bmatrix} \tag{1.12}$$

Our original goal in the $\boldsymbol{S}$ stage was to perform the operations in 1.8, reproduced in compact form below.

$$\boldsymbol{y_1} = \boldsymbol{x_1} + \boldsymbol{V_1 x_{2t}} \tag{1.13a}$$

$$\boldsymbol{y_i} = \boldsymbol{x_i} + \boldsymbol{V_i x_{i+1t}} + \boldsymbol{W_i x_{i-1b}} \tag{1.13b}$$

$$\boldsymbol{y_p} = \boldsymbol{x_p} + \boldsymbol{W_p x_{p-1b}} \tag{1.13c}$$

Now we put the $\boldsymbol{V}$ and $\boldsymbol{W}$ terms back in to their original form and rearrange the equation to isolate the unknowns.

$$\begin{bmatrix} \boldsymbol{x_{1t}} \\ \boldsymbol{\tilde{x}_1} \\ \boldsymbol{x_{1b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{1t}} \\ \boldsymbol{\tilde{y}_1} \\ \boldsymbol{y_{1b}} \end{bmatrix} - \boldsymbol{V_1 x_{2t}} = \boldsymbol{A_1^{-1}} \left(\begin{bmatrix} \boldsymbol{f_{1t}} \\ \boldsymbol{\tilde{f}_1} \\ \boldsymbol{f_{1b}} \end{bmatrix} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_1} \end{bmatrix} \boldsymbol{x_{2t}} \right) \tag{1.14a}$$

$$\begin{bmatrix} \boldsymbol{x_{it}} \\ \boldsymbol{\tilde{x}_i} \\ \boldsymbol{x_{ib}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{\tilde{y}_i} \\ \boldsymbol{y_{ib}} \end{bmatrix} - \boldsymbol{V_i x_{i+1t}} - \boldsymbol{W_i x_{i-1b}} = \boldsymbol{A_i^{-1}} \left(\begin{bmatrix} \boldsymbol{f_{it}} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{f_{ib}} \end{bmatrix} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_i} \end{bmatrix} \boldsymbol{x_{i+1t}} - \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix} \boldsymbol{x_{i-1b}} \right) \tag{1.14b}$$

$$\begin{bmatrix} \boldsymbol{x_{pt}} \\ \boldsymbol{\tilde{x}_p} \\ \boldsymbol{x_{pb}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{pt}} \\ \boldsymbol{\tilde{y}_p} \\ \boldsymbol{y_{pb}} \end{bmatrix} - \boldsymbol{W_p x_{p-1b}} = \boldsymbol{A_p^{-1}} \left(\begin{bmatrix} \boldsymbol{f_{pt}} \\ \boldsymbol{\tilde{f}_p} \\ \boldsymbol{f_{pb}} \end{bmatrix} - \begin{bmatrix} \boldsymbol{C_p} \\ \boldsymbol{0} \end{bmatrix} \boldsymbol{x_{p-1b}} \right) \tag{1.14c}$$

Because the matrices containing the $\boldsymbol{B}$ and $\boldsymbol{C}$ sub-matrices consist mainly of zeroes, we can replace the large multiplications with $\boldsymbol{V}$ and $\boldsymbol{W}$ by the much smaller ones shown in 1.14. Unfortunately, it is still necessary to generate the entire $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices. This is because the top and bottom tips of these matrices are required to construct the reduced system. However, these matrices are generated in the factorization stage, and need not be stored once the tips have been extracted. This is all that is required to avoid explicitly using the $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices in the solve stage.

1.2.2.2 LU/UL factorization

Until now, the method actually solving problems such as $\boldsymbol{A_1^{-1} f_i}$ has been left vague. These operations are performed in one of two ways. In some cases, the popular LU or UL factorization and solve have been used. Specifically, a banded primitive for LU and UL factorization and solve, build upon the BLAS triangular primitives. In other cases, a special purpose SPIKE 2×2 primitive has been used, which will be discussed later.

Note that we do not use a PLU factorization. That is, we do not permute in the case of a zero-pivot. Instead, diagonal boosting is performed. This may result in the loss of numerical accuracy should zero-pivots be encountered. However, the SPIKE algorithm relies heavily on the upper and lower triangular nature of the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices in some cases. Should diagonal boosting result in a loss of accuracy, the user will have to be warned that an approximate answer has been produced.

The LU or UL factorization results in a pair of matrices $\boldsymbol{L}$ and $\boldsymbol{U}$, which are lower and upper triangular respectively. To solve a problem involving these factorizations, we require one solve for the $\boldsymbol{L}$ matrix and another for the $\boldsymbol{U}$ matrix. We call these solves 'sweeps' to distinguish from a full solve. This mnemonic is meant to evoke the idea of moving up the $\boldsymbol{U}$ matrix, and down the $\boldsymbol{L}$ matrix, finding values of the vector we are sweeping over. We can use the triangular nature of $\boldsymbol{L}$ and $\boldsymbol{U}$ to perform some significant optimizations.

For partitions 1 and p a particularly large optimization can be made. In the factorization stage, we must generate the $\boldsymbol{V}$ and $\boldsymbol{W}$ spikes, to extract their tips. In the case of the first and last partitions, the operations to be performed are:

$$\boldsymbol{A_1^{-1}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_1} \end{bmatrix} \tag{1.15a}$$

$$\boldsymbol{A_p^{-1}} \begin{bmatrix} \boldsymbol{C_p} \\ \boldsymbol{0} \end{bmatrix} \tag{1.15b}$$

We will perform the LU factorization on $\boldsymbol{A_i}$ and the UL factorization on $\boldsymbol{A_p}$. Because the matrices containing $\boldsymbol{B_1}$ and $\boldsymbol{C_p}$ consist mostly of zeroes, these sweeps will be greatly shortened.

$$\boldsymbol{A_1^{-1}}\begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{B_1}\end{bmatrix} = \boldsymbol{U_1^{-1}L_1^{-1}}\begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{B_1}\end{bmatrix} \tag{1.16a}$$

$$\boldsymbol{A_p^{-1}}\begin{bmatrix}\boldsymbol{C_p}\\ \boldsymbol{0}\end{bmatrix} = \boldsymbol{L_p^{-1}U_p^{-1}}\begin{bmatrix}\boldsymbol{C_p}\\ \boldsymbol{0}\end{bmatrix} \tag{1.16b}$$

Because the top of $\begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{B_1}\end{bmatrix}$ consists of zeroes, we may begin the sweep associated with $\boldsymbol{L_1^{-1}}$ at the point where the zeroes end and $\boldsymbol{B_1}$ begins. This reduces the height of the sweep from n_1 to k_u. Similarly, the sweep up the matrix containing $\boldsymbol{C_p}$ can be reduced in height from n_p to k_l.

Next, we observe that we do not actually need the value of $\boldsymbol{V_{1t}}$ and $\boldsymbol{W_{pb}}$. Recalling the relevant pieces of the reduced system,

$$\begin{bmatrix}\boldsymbol{y_{1t}}\\ \boldsymbol{y_{1b}}\\ \boldsymbol{y_{2t}}\\ \boldsymbol{y_{2b}}\\ \vdots\\ \boldsymbol{y_{p-1t}}\\ \boldsymbol{y_{p-1b}}\\ \boldsymbol{y_{pt}}\\ \boldsymbol{y_{pb}}\end{bmatrix} = \begin{bmatrix}
\boldsymbol{I} & & \boldsymbol{V_{1t}} & & & & & & \\
 & \boldsymbol{I} & \boldsymbol{V_{1b}} & & & & & & \\
 & \boldsymbol{W_{2t}} & \boldsymbol{I} & & & & & & \\
 & \boldsymbol{W_{2b}} & & \boldsymbol{I} & & & & & \\
 & & & & \ddots & & & & \\
 & & & & & \boldsymbol{I} & & \boldsymbol{V_{p-1b}} & \\
 & & & & & & \boldsymbol{I} & \boldsymbol{V_{p-1b}} & \\
 & & & & & & \boldsymbol{W_{pt}} & \boldsymbol{I} & \\
 & & & & & & \boldsymbol{W_{pb}} & & \boldsymbol{I}
\end{bmatrix}\begin{bmatrix}\boldsymbol{x_{1t}}\\ \boldsymbol{x_{1b}}\\ \boldsymbol{x_{2t}}\\ \boldsymbol{x_{2b}}\\ \vdots\\ \boldsymbol{x_{p-1t}}\\ \boldsymbol{x_{p-1b}}\\ \boldsymbol{x_{pt}}\\ \boldsymbol{x_{pb}}\end{bmatrix} \tag{1.17}$$

The values for $\boldsymbol{V_{1t}}$ and $\boldsymbol{W_{pb}}$ are used to compute $\boldsymbol{x_{1t}}$ and $\boldsymbol{x_{pb}}$. These values are not needed to recover the rest of $\boldsymbol{x}$, because there are no corresponding matrices $\boldsymbol{W_1}$ and $\boldsymbol{V_p}$. The values for $\boldsymbol{x_{1t}}$ and $\boldsymbol{x_{pb}}$ can be retrieved along with $\tilde{\boldsymbol{x}}_\boldsymbol{1}$ and $\tilde{\boldsymbol{x}}_\boldsymbol{p}$. As a

result, we need only perform enough of the sweep associated with $\boldsymbol{U^{-1}}$ to extract the bottom tip of $\boldsymbol{V_{1b}}$, and enough of the $\boldsymbol{L_p^{-1}}$ sweep to obtain $\boldsymbol{W_{pt}}$. These are, then, reduced in size from n_1 and n_p to k_u and k_l.Effectively, we have gotten rid of the sweeps required in the factorization stage for the first and last partitions.

For the middle partitions, we need the whole of $\boldsymbol{V}$ and $\boldsymbol{W}$ in the factorization stage, but there is still some possibility for optimization. The middle partitions, $\boldsymbol{A_i}$ = $\boldsymbol{A_2}$ to $\boldsymbol{A_{p-1}}$ are all LU factorized. The sweep associated with $\boldsymbol{L_i^{-1}} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_i} \end{bmatrix}$ may be shortened. Similar to the $\boldsymbol{L_1^{-1}} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_1} \end{bmatrix}$ sweep, we may skip the beginning zeroes and shorten the sweep from n_i to k_u.

1.2.2.3 Recursive Reduced System

To mitigate the problem of the growing reduced system the recursive reduced system method is used. [5] This is based on the observation that the reduced system is, itself, banded. So, we may use SPIKE to solve it.

Rather than describe the most general case, we will look at a specific case – the four partition case – and extrapolate. The recursive method includes multiple stages of SPIKE, so a new index has been included, indicating the level of recursion. Also, the reduced $\boldsymbol{S}$ matrices will be called $\boldsymbol{\tilde{S}_j}$, where j indicates recursion level.

We would like to factorize this to $\boldsymbol{D}$ and $\boldsymbol{S}$ matrices.

$$\tilde{\boldsymbol{S}}_1 = \left[\begin{array}{cccc|cccc} \boldsymbol{I} & & \boldsymbol{V}_{1t1} & & & & & \\ & \boldsymbol{I} & \boldsymbol{V}_{1b1} & & & & & \\ & \boldsymbol{W}_{2t1} & \boldsymbol{I} & & \boldsymbol{V}_{2t1} & & & \\ & \boldsymbol{W}_{2b1} & & \boldsymbol{I} & \boldsymbol{V}_{2b1} & & & \\ \hline & & \boldsymbol{W}_{3t1} & & \boldsymbol{I} & & \boldsymbol{V}_{3t1} & \\ & & \boldsymbol{W}_{3b1} & & & \boldsymbol{I} & \boldsymbol{V}_{3b1} & \\ & & & & \boldsymbol{W}_{4t1} & \boldsymbol{I} & & \\ & & & & \boldsymbol{W}_{4b1} & & \boldsymbol{I} & \end{array}\right] = \boldsymbol{D}_1 \tilde{\boldsymbol{S}}_2 \qquad (1.18)$$

The areas enclosed in the dotted lines becomes the new $\boldsymbol{B}$ and $\boldsymbol{C}$ matrices. Notice that half of each dotted area is empty. The new $\boldsymbol{V}$ and $\boldsymbol{W}$ spikes will be formed in the usual manner.

$$\begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V}_{1t1} & \\ & \boldsymbol{I} & \boldsymbol{V}_{1b1} & \\ \boldsymbol{W}_{2t1} & \boldsymbol{I} & & \\ \boldsymbol{W}_{2b1} & & \boldsymbol{I} & \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{V}_{2t1} & \boldsymbol{0} \\ \boldsymbol{V}_{2b1} & \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{V}_{1t2} & \boldsymbol{0} \\ \boldsymbol{V}_{1b2} & \boldsymbol{0} \\ \boldsymbol{V}_{2t2} & \boldsymbol{0} \\ \boldsymbol{V}_{2b2} & \boldsymbol{0} \end{bmatrix} \qquad (1.19a)$$

$$\begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V}_{3t1} & \\ & \boldsymbol{I} & \boldsymbol{V}_{3b1} & \\ \boldsymbol{W}_{4t1} & \boldsymbol{I} & & \\ \boldsymbol{W}_{4b1} & & \boldsymbol{I} & \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{W}_{3t1} & \boldsymbol{0} \\ \boldsymbol{W}_{3b1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{W}_{3t2} & \boldsymbol{0} \\ \boldsymbol{W}_{3b2} & \boldsymbol{0} \\ \boldsymbol{W}_{4t2} & \boldsymbol{0} \\ \boldsymbol{W}_{4b2} & \boldsymbol{0} \end{bmatrix} \qquad (1.19b)$$

Zeroes have been explicitly included in this case, to shows an interesting trait of the $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices in the recursive scheme. With each level of recursion, the

$\boldsymbol{V}$ and $\boldsymbol{W}$ matrices grow taller, but not wider. This is helpful, because it limits the interdependence between elements as the recursive stages continue.

We obtain the equations for $\boldsymbol{D_1}$ and $\boldsymbol{\tilde{S}_2}$

$\boldsymbol{D_1\tilde{S}_2} =$

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
\boldsymbol{I} & & \boldsymbol{V_{1t1}} & & & & & & \boldsymbol{I} & & & & \boldsymbol{V_{1t2}} & & & \\
 & \boldsymbol{I} & \boldsymbol{V_{1b1}} & & & & & & & \boldsymbol{I} & & & \boldsymbol{V_{1t2}} & & & \\
 & \boldsymbol{W_{2t1}} & \boldsymbol{I} & & & & & & & & \boldsymbol{I} & & \boldsymbol{V_{2t2}} & & & \\
 & \boldsymbol{W_{2b1}} & & \boldsymbol{I} & & & & & & & & \boldsymbol{I} & \boldsymbol{V_{2b2}} & & & \\
\hline
 & & & & \boldsymbol{I} & & \boldsymbol{V_{3t1}} & & & & & \boldsymbol{W_{3t2}} & \boldsymbol{I} & & & \\
 & & & & & \boldsymbol{I} & \boldsymbol{V_{3b1}} & & & & & \boldsymbol{W_{3b2}} & & \boldsymbol{I} & & \\
 & & & & & \boldsymbol{W_{4t1}} & \boldsymbol{I} & & & & & \boldsymbol{W_{4t2}} & & & \boldsymbol{I} & \\
 & & & & & \boldsymbol{W_{4b1}} & & \boldsymbol{I} & & & & \boldsymbol{W_{4t2}} & & & & \boldsymbol{I}
\end{array}\right] \tag{1.20}$$

Because we have only four partitions, this is the bottom level of our recursion. So, we must solve the problem:

$$\boldsymbol{\tilde{S}_1 x_{red}} = \boldsymbol{D_1 \tilde{S}_2 x_{red}} = \boldsymbol{y_{red}} \tag{1.21a}$$

$$\boldsymbol{D_1 g} = \boldsymbol{y_{red}} \tag{1.21b}$$

$$\boldsymbol{\tilde{S}_2 x_{red}} = \boldsymbol{g} \tag{1.21c}$$

The first step is to solve the 1.21b The sub-matrices along the diagonal are decoupled, so we may solve the following in parallel.

$$\begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V_{1t1}} & \\ & \boldsymbol{I} & \boldsymbol{V_{1b1}} & \\ & \boldsymbol{W_{2t1}} & \boldsymbol{I} & \\ & \boldsymbol{W_{2b1}} & & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{g_{1t}} \\ \boldsymbol{g_{1b}} \\ \boldsymbol{g_{2t}} \\ \boldsymbol{g_{2b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{1t}} \\ \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{y_{2b}} \end{bmatrix} \tag{1.22a}$$

$$\begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V_{3t1}} & \\ & \boldsymbol{I} & \boldsymbol{V_{3b1}} & \\ & \boldsymbol{W_{4t1}} & \boldsymbol{I} & \\ & \boldsymbol{W_{4b1}} & & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{g_{3t}} \\ \boldsymbol{g_{3b}} \\ \boldsymbol{g_{4t}} \\ \boldsymbol{g_{4b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{3t}} \\ \boldsymbol{y_{3b}} \\ \boldsymbol{y_{4t}} \\ \boldsymbol{y_{4b}} \end{bmatrix} \tag{1.22b}$$

The innermost block of these matrices is actually the only part that needs to be solved, the other values can be retrieved with a multiplication and subtraction.

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{V_{1b1}} \\ \boldsymbol{W_{2t1}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{g_{1b}} \\ \boldsymbol{g_{2t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \end{bmatrix} \tag{1.23a}$$

$$\boldsymbol{g_{1t}} = \boldsymbol{y_{1t}} - \boldsymbol{V_{1t1}}\boldsymbol{g_{2t}} \tag{1.23b}$$

$$\boldsymbol{g_{2b}} = \boldsymbol{y_{2b}} - \boldsymbol{W_{2b1}}\boldsymbol{g_{1b}} \tag{1.23c}$$

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{V_{3b1}} \\ \boldsymbol{W_{4t1}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{g_{3b}} \\ \boldsymbol{g_{4t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{3b}} \\ \boldsymbol{y_{4t}} \end{bmatrix} \tag{1.23d}$$

$$\boldsymbol{g_{3t}} = \boldsymbol{y_{3t}} - \boldsymbol{V_{3t1}}\boldsymbol{g_{4t}} \tag{1.23e}$$

$$\boldsymbol{g_{4b}} = \boldsymbol{y_{4b}} - \boldsymbol{W_{4b1}}\boldsymbol{g_{3b}} \tag{1.23f}$$

This gives us the entirety of $\boldsymbol{g}$, now we retrieve $\boldsymbol{x_{red}}$ from the $\boldsymbol{D}$ matrix.

$$\left[\begin{array}{cccc|cccc} \boldsymbol{I} & & & & \boldsymbol{V_{1t2}} & & & \\ & \boldsymbol{I} & & & \boldsymbol{V_{1b2}} & & & \\ & & \boldsymbol{I} & & \boldsymbol{V_{2t2}} & & & \\ & & & \boldsymbol{I} & \boldsymbol{V_{2b2}} & & & \\ \hline & & & \boldsymbol{W_{3t2}} & \boldsymbol{I} & & & \\ & & & \boldsymbol{W_{3b2}} & & \boldsymbol{I} & & \\ & & & \boldsymbol{W_{4t2}} & & & \boldsymbol{I} & \\ & & & \boldsymbol{W_{4b2}} & & & & \boldsymbol{I} \end{array}\right] \left[\begin{array}{c} \boldsymbol{x_{1t}} \\ \boldsymbol{x_{1b}} \\ \boldsymbol{x_{2t}} \\ \boldsymbol{x_{2b}} \\ \boldsymbol{x_{3t}} \\ \boldsymbol{x_{3b}} \\ \boldsymbol{x_{4t}} \\ \boldsymbol{x_{4b}} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{g_{1t}} \\ \boldsymbol{g_{1b}} \\ \boldsymbol{g_{2t}} \\ \boldsymbol{g_{2b}} \\ \boldsymbol{g_{3t}} \\ \boldsymbol{g_{3b}} \\ \boldsymbol{g_{4t}} \\ \boldsymbol{g_{4b}} \end{array}\right] \tag{1.24}$$

Similarly to the previous problem, we can solve a small reduced system and extract the rest of the answer.

$$\left[\begin{array}{cc} \boldsymbol{I} & \boldsymbol{V_{2b2}} \\ \boldsymbol{W_{3t2}} & \boldsymbol{I} \end{array}\right] \left[\begin{array}{c} \boldsymbol{x_{2b}} \\ \boldsymbol{x_{3t}} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{g_{2b}} \\ \boldsymbol{g_{3t}} \end{array}\right] \tag{1.25a}$$

$$\boldsymbol{x_{1t}} = \boldsymbol{g_{1t}} - \boldsymbol{V_{1t2}}\boldsymbol{x_{3t}} \tag{1.25b}$$

$$\boldsymbol{x_{1b}} = \boldsymbol{g_{1b}} - \boldsymbol{V_{1b2}}\boldsymbol{x_{3t}} \tag{1.25c}$$

$$\boldsymbol{x_{2t}} = \boldsymbol{g_{2t}} - \boldsymbol{V_{2t2}}\boldsymbol{x_{3t}} \tag{1.25d}$$

$$\boldsymbol{x_{3b}} = \boldsymbol{g_{3b}} - \boldsymbol{W_{3b2}}\boldsymbol{x_{2b}} \tag{1.25e}$$

$$\boldsymbol{x_{4t}} = \boldsymbol{g_{4t}} - \boldsymbol{W_{4t2}}\boldsymbol{x_{2b}} \tag{1.25f}$$

$$\boldsymbol{x_{4b}} = \boldsymbol{g_{4b}} - \boldsymbol{W_{4b2}}\boldsymbol{x_{2b}} \tag{1.25g}$$

This completes the recursive scheme for four partitions. In the presence of more partitions, it would be necessary to perform the spike factorization repeatedly until the problem was reduced to the two by two block matrix. Then, the problem is solved from the bottom level up, as we did in the example.

The recursive method is extremely useful because it allows the for parallelism in the reduced system. However, it does have the limitation that the total number of

partitions must be a power of two. This is an unavoidable consequence of halving the number of partitions with each recursive level. We will discuss mitigating this problem in the next section.

At this point, the base algorithm has hopefully been sufficiently described. The rest of this work will discuss new improvements to the SPIKE algorithm.

CHAPTER 2

ENHANCED LOAD BALANCING

A limitation for recursive SPIKE is the requirement that the number of partitions used is 2^m for some integer m. The work discussed here is meant to be a general purpose implementation of SPIKE, so it makes sense to make as few assumptions about the users' hardware as possible. Additionally, as the number of threads increases, the amount distance between subsequent values of 2^m increases. If the general trend for increasing cores continues, the amount of compute power possibly wasted by not having the ability to address the in-between values increases. Finally, even in cases where a user may have exactly 2^m cores, they may wish to allocate some of them to non-SPIKE related tasks. For these reasons we wish to expand the possible choices for number of threads used.

The method of attaining this increased resource utilization is surprisingly straightforward. The recursive SPIKE algorithm implemented here benefits greatly from exploiting the LU/UL factorization for the first and last partitions. However, inner partitions do not gain as much from this optimization. As the LU/UL factorization does not help much on the inner partitions, we may instead use the SPIKE algorithm in cases where matrix solves are called for. These partitions may then be increased in size, until the load is balanced equally between threads.

A simplified SPIKE 2×2 primitive has been developed. The ability to use one or two threads on inner partitions allows the algorithm to access anywhere from 2^m to $2^m - 2$ threads. This leaves us with a maximum gap of one thread wasted, in the case when the total number of threads is $2^m - 1$, which is an acceptable limitatation.

2.1 Distribution of threads

For the SPIKE algorithm, there are two operations that make up the majority of the computational cost. The most costly are the factorizations of the sub-matrices in the **D** matrix. The second most costly operations are solve sweeps which involve the whole of these sub-matrices.

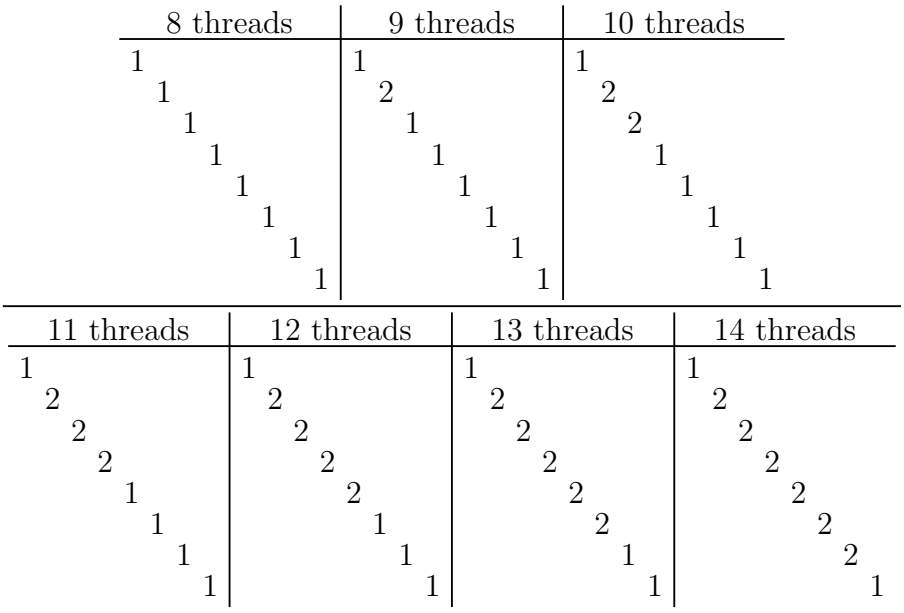
The factorizations occur in the factorization stage. On some partitions we will speed up these factorizations by using a simplified SPIKE 2×2 factorization instead of a normal LU factorization. Similarly, we will use SPIKE solve on these partitions. The degree to which this speeds up the work done in these partitions will be discussed in the next section.

In the first and last partitions, the LU/UL factorization is used to perform a great optimization. So, these partitions can not use the SPIKE factorization, and always use one thread. For the rest of the threads, we begin doubling the number of threads starting arbitrarily at the second topmost partition. For example, four, five, or size threads would be distributed as follows, with the number representing the number of threads that would be used for the partition associated with that location in the **D** matrix

4 threads	5 threads	6 threads
1	1	1
1	2	2
1	1	2
1	1	1

(2.1)

And eight to fourteen threads would follow the pattern:



(2.2)

With seven or fifteen threads, we ignore the final threads and use the scheme for six or fourteen threads respectively. Continuing with this pattern, when using $p = 2^m$ partitions (for some integer m), q single-thread partitions, and r double threaded partitions, may have t threads.

$$0 \leq r \leq p - 2 \tag{2.3a}$$

$$2 \leq q \leq p \tag{2.3b}$$

$$p = 2^m = q + r \tag{2.3c}$$

$$t = q + 2r = 2^m + r \tag{2.3d}$$

$$2^m \leq t \leq 2^m + p - 2 \tag{2.3e}$$

$$2^m \leq t \leq 2^{m+1} - 2 \tag{2.3f}$$

And so, we may use any number of threads, with the exception of $2^m - 1$

2.2 Partition ratios

Let us assume that we have a matrix $\mathbf{A}$ with a size of n and a one-half bandwidth of k_{lu}. Previously we have specified the upper and lower bandwidth as k_u and k_u,

but the distinction is not as important in this case. Instead we assume just that n is much greater than k_{lu}, to the point where we may approximate the half-bandwidth as equal for upper and lower bands.

Note that the LU solve primitive we use has the ability to work on a number of right hand sides. In the factorization stage of SPIKE, we perform solves on the **B** and **C** matrices – these are k_{lu} wide. In the solve stage, we perform these solves on the set of vectors supplied by the user, which are n_{rhs} wide. The idea of wideness is used below, where appropriate.

Additionally, we have a number of right hand side vectors which we are solving for, n_{rhs}

The costs incurred in each thread are as follows:

- First and last partitions
  - Factorization stage: 1×LU (or UL) factorization
  - Solve stage: 1× LU solve (width n_{rhs})
- Inner double threaded partitions
  - Factorization stage: 1×SPIKE factorization, 1×SPIKE solve (width $2k_{lu}$)
  - Solve stage: 2×SPIKE solve (width n_{rhs})
- Inner single threaded partitions
  - Factorization stage: 1×LU factorization, .5×LU solve (width $2k_{lu}$), .5×LU solve (width k_{lu})
  - Solve stage: 2× LU solve (width n_{rhs})

We will have three partition sizes, n_1, n_2, and n_3. Respectively, they are the sizes of the first/last partitions, the inner partitions on which the two threaded spike is used, and the inner partitions which receive the single threaded LU factorization.

The relationship between these sizes is defined in terms of the ratios: $R_{12} = \frac{n_1}{n_2}$ and $R_{13} = \frac{n_1}{n_3}$. For optimal load balancing, we would like to have each partition take the same amount of time to complete.

This SPIKE implementation uses a block LU factorization and solve, based on the BLAS implementation provided by the system. The factorization has a performance of approximately[1] $O(n \times k_{lu}^2)$, and the solve has a performance of approximately[2] $O(n \times k_{lu} \times width)$. Let us approximate these costs as $K_1 \times n \times k_{lu}^2$ and $K_2 \times n \times k_{lu} \times width$. In addition, let us call the ratio between K_2 and K_1 simply K, we will need it later. For the two-partition case, the block LU factorizations and solves are replaced by simplified SPIKE factorizations and solves, each using two threads. Because the SPIKE 2×2 primitive scales perfectly, one SPIKE factorization requires one half the time of an LU factorization, and one SPIKE solve takes one half the time of a full LU solve.

The total amount of work on each of the inner partitions is the same, with one exception. In the factorization stage, we perform the operation:

$$\boldsymbol{A_i^{-1}} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{C_i} \end{bmatrix} = \boldsymbol{U_i^{-1} L_i^{-1}} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{C_i} \end{bmatrix} = \boldsymbol{V_i} \tag{2.4}$$

Because the sweep associated with the $\mathbf{L}$ matrix can be shortened to k_{lu}, it is ignored. A $\boldsymbol{LU}$ solve requires an $\boldsymbol{L}$ and a $\boldsymbol{U}$ sweep, so the solve for $\boldsymbol{V}$ can only really be called a half solve. Combined, we will call this 1.5 solves.

---

[1] This is assuming that the system BLAS dense LU factorization performs as $O(k_{lu}^3)$. The primitive called by SPIKE is mainly driven by the system factorization. Roughly, the banded matrix is broken up in to blocks of size $k_{lu} \times k_{lu}$ along the diagonal. This results in a number of blocks equal to n/k_{lu}, and so we really end up with $O(n \times k_{lu}^{x-1})$, where x is dependent on the system factorization.

[2] Working under the same assumptions as in 1

Putting this all together, we obtain the following requirement for a balanced load:

$$K_1 n_1 k_{lu}^2 + K_2 n_1 k_{lu} n_{rhs} = \frac{K_1 n_2 k_{lu} + 2K_2 n_2 k_{lu}^2 + 2K_2 n_2 k_{lu} n_{rhs}}{2} = K_1 n_3 k_{lu}^2 + 1.5 K_2 n_3 k_{lu}^2 + 2K_2 n_3 k_{lu} n_{rhs} \tag{2.5a}$$

$$K_1 n_1 k_{lu} + K_2 n_1 n_{rhs} = \frac{K_1 n_2 k_{lu}}{2} + K_2 n_2 k_{lu} + K_2 n_2 n_{rhs} = K_1 n_3 k_{lu} + 1.5 K_2 n_3 k_{lu} + 2K_2 n_3 n_{rhs} \tag{2.5b}$$

$$(\frac{K_1 n_2}{2} + K_2 n_2) k_{lu} + K_2 n_2 n_{rhs} = (K_1 n_3 + 1.5 K_2 n_3) k_{lu} + 2K_2 n_3 n_{rhs} \tag{2.5c}$$

The relationship between k_{lu} and n_{rhs} is, naturally, defined by the problem. Let us start with the easier case. If n_{rhs} is much larger than k_{lu}, we may treat k_{lu} as zero. This will allow us to find a system non-specific value for the ratios.

$$K_2 n_1 n_{rhs} = K_2 n_2 n_{rhs} = 2K_2 n_3 n_{rhs} \tag{2.6a}$$

$$n_1 = n_2 = 2n_3 \tag{2.6b}$$

$$R_{12} = \frac{n_1}{n_2} = 1 \tag{2.6c}$$

$$R_{13} = \frac{n_1}{n_3} = 2 \tag{2.6d}$$

The other case is one in which we have very few right hand sides and very large bandwidth, such that n_{rhs} is approximated as zero.

$$K_1 n_1 k_{lu} = (\frac{K_1 n_2}{2} + K_2 n_2) k_{lu} = (K_1 n_3 + 1.5 K_2 n_3) k_{lu} \tag{2.7a}$$

$$K_1 n_1 = \frac{K_1 n_2}{2} + K_2 n_2 = K_1 n_3 + 1.5 K_2 n_3 \tag{2.7b}$$

$$R_{12} = \frac{n_1}{n_2} = \frac{1}{2} + \frac{K_2}{K_1} = \frac{1}{2} + K \tag{2.7c}$$

$$R_{13} = \frac{n_1}{n_3} = 1 + 1.5\frac{K_2}{K_1} = 1 + 1.5K \tag{2.7d}$$

The constant K depends on the system hardware and the underlying BLAS implementation. Due to the vagaries of hardware and software, it is unlikely that a universally good value for K exists. However, for a given system K may be easily found. Using the same approximations as above,

$$\text{factorization time} = K_1 \times n \times k_{lu}^2 \tag{2.8a}$$

$$\text{solve time} = K_2 \times n \times k_{lu} \times n_{rhs} \tag{2.8b}$$

$$K = \frac{K_2}{K_1} \tag{2.8c}$$

$$= \frac{\text{solve time}}{n \times k_{lu} \times n_{rhs}} \times \frac{n \times k_{lu}^2}{\text{factorization time}} \tag{2.8d}$$

$$\tag{2.8e}$$

So, we may calculate the value of K by performing a factorization and solve on a matrix and vector such that $n_{rhs} = k_{lu}$

$$K = \frac{\text{solve time}}{\text{factorization time}} \tag{2.9a}$$

This calculation could be performed when the SPIKE package is installed.

2.3 Partition sizes

Once the ratios between partition sizes have been decided upon, sizing the partitions is easier. The main requirement is that all of the partitions must add up to be equal in size to the matrix $\mathbf{A}$, n. Assume that there are x partitions of size n_2, y partitions of size n_3, and the first/last partitions, each of which is size n_1.

$$n = 2n_1 + xn_2 + yn_3 \tag{2.10}$$

$$= 2n_1 + \frac{xn_1}{R_{12}} + \frac{yn_1}{R_{13}} \tag{2.11}$$

$$\frac{nR_{12}R_{13}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_1 \tag{2.12}$$

$$\frac{nR_{13}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_2 \tag{2.13}$$

$$\frac{nR_{12}}{2R_{12}R_{13} + xR_{13} + yR_{12}} = n_3 \tag{2.14}$$

CHAPTER 3

TRANSPOSE SCHEME FOR SPIKE

An efficient transpose solve option, to solve $\boldsymbol{A^T x = f}$ for $\boldsymbol{x}$, is a standard feature of the BLAS and Lapack libraries. The alternative would be to explicitly transpose $\boldsymbol{A}$, and them perform a normal factorization and solve. This is wasteful, because it involves needlessly moving values around in memory. Additionally, because the factorization stage is more computationally expensive than the solve stage, some algorithms will benefit from the ability to use the same factorization for both transpose non-transpose problems.

3.1 Two partition case

We will begin with the simplified case for two partitions. Here we would like to perform the operation

$$\text{Solve } \boldsymbol{A^T x = f} \text{ for } \boldsymbol{x}$$

Where $\boldsymbol{A}$ is an n-by-n diagonal matrix with upper and lower half-bandwidth k_u and k_l, $\boldsymbol{x}$ and $\boldsymbol{f}$ are vectors. We would also like the computational cost of this operation to be minimal – roughly the same as for the non-transpose option. Primarily, we would like to limit this solve stage to performing one full solve (comprised of two solve sweeps) on each of the large sub-matrices, $\boldsymbol{A_1}$ and $\boldsymbol{A_p}$.

Given the factorization designed for non-transpose SPIKE:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{A_2} \end{bmatrix} = \tag{3.1}$$

$$\boldsymbol{DS} = \begin{bmatrix} \boldsymbol{L_1U_1} & \\ & \boldsymbol{U_2L_2} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \boldsymbol{V} \\ \boldsymbol{W} & \boldsymbol{I} \end{bmatrix} \tag{3.2}$$

The transpose of $\boldsymbol{A}$ can clearly be rewritten in terms of $\boldsymbol{D}$ and $\boldsymbol{S}$.

$$\boldsymbol{A^T} = \boldsymbol{S^T D^T} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{W^T} \\ \boldsymbol{V^T} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{U_1^T L_1^T} & \\ & \boldsymbol{L_2^T U_2^T} \end{bmatrix} \tag{3.3}$$

At this point we obtain two subproblems for the solve stage. These are analogous to the two subproblems in non-transpose SPIKE, although their order has been reversed.

$$\boldsymbol{S^T D^T x} = \boldsymbol{f} \tag{3.4a}$$

$$\boldsymbol{S^T y} = \boldsymbol{f} \tag{3.4b}$$

$$\boldsymbol{D^T x} = \boldsymbol{y} \tag{3.4c}$$

3.1.1 Transpose S matrix

The form of the $\boldsymbol{S^T}$ matrix limits the unknowns in the $\boldsymbol{y}$ vector to a small number of values, which are located at the center of $\boldsymbol{y}$. The eventual goal will be to use this fact to construct a reduced system and solve for just these values.

The $\boldsymbol{y}$ and $\boldsymbol{f}$ vectors can be broken up as follows:

$$\boldsymbol{S^T y} = \left[\begin{array}{cc|c} & & \mathbf{0} \\ \boldsymbol{I} & & \\ \hline & & \boldsymbol{W^T} \\ \hline & \boldsymbol{V^T} & \\ \hline \mathbf{0} & & \boldsymbol{I} \end{array}\right] \left[\begin{array}{c} \tilde{\boldsymbol{y}}_{\mathbf{1}} \\ \boldsymbol{y}_{\mathbf{1}\boldsymbol{b}} \\ \boldsymbol{y}_{\mathbf{2}\boldsymbol{t}} \\ \tilde{\boldsymbol{y}}_{\mathbf{2}} \end{array}\right] = \left[\begin{array}{c} \tilde{\boldsymbol{f}}_{\mathbf{1}} \\ \boldsymbol{f}_{\mathbf{1}\boldsymbol{b}} \\ \boldsymbol{f}_{\mathbf{2}\boldsymbol{t}} \\ \tilde{\boldsymbol{f}}_{\mathbf{2}} \end{array}\right] \tag{3.5}$$

For now let us assume that the segments are sized as follows: $\boldsymbol{y_{1b}}$, $\boldsymbol{y_{2t}}$, $\boldsymbol{f_{1b}}$ and $\boldsymbol{f_{2t}}$ are each a $\max(k_l, k_u)$ elements tall; $\tilde{\boldsymbol{y}}_{\mathbf{1}}$, $\tilde{\boldsymbol{y}}_{\mathbf{2}}$, $\tilde{\boldsymbol{f}}_{\mathbf{1}}$ and $\tilde{\boldsymbol{f}}_{\mathbf{2}}$ take up the remainder of the elements in the corresponding halves of the vector, so they are all $\frac{n}{2} - \max(k_l, k_u)$ elements tall.

$$\tilde{\boldsymbol{y}}_{\mathbf{1}} = \tilde{\boldsymbol{f}}_{\mathbf{1}} \tag{3.6a}$$

$$\tilde{\boldsymbol{y}}_{\mathbf{2}} = \tilde{\boldsymbol{f}}_{\mathbf{2}} \tag{3.6b}$$

$$\boldsymbol{y_{1b}} = \boldsymbol{f_{1b}} - \boldsymbol{W^T} \begin{bmatrix} \boldsymbol{y_{2t}} \\ \tilde{\boldsymbol{y}}_{\mathbf{2}} \end{bmatrix} = \boldsymbol{f_{1b}} - \left((\boldsymbol{U_2 L_2})^{-1} \begin{bmatrix} \boldsymbol{C} \\ \mathbf{0} \end{bmatrix} \right)^{\boldsymbol{T}} \begin{bmatrix} \boldsymbol{y_{2t}} \\ \tilde{\boldsymbol{f}}_{\mathbf{2}} \end{bmatrix} \tag{3.6c}$$

$$\boldsymbol{y_{2t}} = \boldsymbol{f_{2t}} - \boldsymbol{V^T} \begin{bmatrix} \tilde{\boldsymbol{y}}_{\mathbf{1}} \\ \boldsymbol{y_{1b}} \end{bmatrix} = \tilde{\boldsymbol{f}}_{\mathbf{2}} - \left((\boldsymbol{L_1 U_1})^{-1} \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B} \end{bmatrix} \right)^{\boldsymbol{T}} \begin{bmatrix} \tilde{\boldsymbol{f}}_{\mathbf{1}} \\ \boldsymbol{y_{1b}} \end{bmatrix} \tag{3.6d}$$

A large gain in performance is obtained by making two observations. First in 3.6c and 3.6d, in the $\mathbf{0}$ sub-matrices are much larger than the $\boldsymbol{C}$ and $\boldsymbol{B}$ sub-matrices. Secondly, the $\boldsymbol{U^{-1}}$ and $\boldsymbol{L^{-1}}$ matrices are upper and lower triangular. Because of this, some rearranging allows for the solve sweeps associated with $\boldsymbol{U_2}$ and $\boldsymbol{L_1}$ to be reduced in height from $\frac{n}{2}$ to $\max(k_u, k_l)$.

To simplify the notation, $\boldsymbol{U_{2t}^{-1}}$ will be used to represent the top-left $\max(k_l,k_u)$ by $\max(k_l,k_u)$ blocks of elements from $\boldsymbol{U_2^{-1}}$. Similarly, $\boldsymbol{L_{1b}^{-1}}$ will be used to represent bottom-right $\max(k_l,k_u)$ by $\max(k_l,k_u)$ blocks of elements from $\boldsymbol{L_1^{-1}}$.

$$\begin{aligned}\left((\boldsymbol{U_2L_2})^{-1}\begin{bmatrix}\boldsymbol{C}\\\boldsymbol{0}\end{bmatrix}\right)^{\boldsymbol{T}}\begin{bmatrix}\boldsymbol{y_{2t}}\\\tilde{\boldsymbol{f}}_{\boldsymbol{2}}\end{bmatrix} &= \left(\boldsymbol{U_2^{-1}}\begin{bmatrix}\boldsymbol{C}\\\boldsymbol{0}\end{bmatrix}\right)^{\boldsymbol{T}}\boldsymbol{L_2^{-1,T}}\begin{bmatrix}\boldsymbol{y_{2t}}\\\tilde{\boldsymbol{f}}_{\boldsymbol{2}}\end{bmatrix}\\ &= \begin{bmatrix}\boldsymbol{U_{2t}^{-1}C}\\\boldsymbol{0}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{L_2^{-1,T}}\begin{bmatrix}\boldsymbol{y_{2t}}\\\tilde{\boldsymbol{f}}_{\boldsymbol{2}}\end{bmatrix}\end{aligned} \tag{3.7a}$$

$$\begin{aligned}\left((\boldsymbol{L_1U_1})^{-1}\begin{bmatrix}\boldsymbol{0}\\\boldsymbol{B}\end{bmatrix}\right)^{\boldsymbol{T}}\begin{bmatrix}\tilde{\boldsymbol{f}}_{\boldsymbol{1}}\\\boldsymbol{y_{1b}}\end{bmatrix} &= \left(\boldsymbol{L_1^{-1}}\begin{bmatrix}\boldsymbol{0}\\\boldsymbol{B}\end{bmatrix}\right)^{\boldsymbol{T}}\boldsymbol{U_1^{-1,T}}\begin{bmatrix}\tilde{\boldsymbol{f}}_{\boldsymbol{1}}\\\boldsymbol{y_{1b}}\end{bmatrix}\\ &= \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{L_{1b}^{-1}B}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{U_1^{-1,T}}\begin{bmatrix}\tilde{\boldsymbol{f}}_{\boldsymbol{1}}\\\boldsymbol{y_{1b}}\end{bmatrix}\end{aligned} \tag{3.7b}$$

These can then be placed back in to the systems from 3.6

$$\begin{aligned}\boldsymbol{y_{1b}} &= \boldsymbol{f_{1b}} - \begin{bmatrix}\boldsymbol{U_{2t}^{-1}C}\\\boldsymbol{0}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{L_2^{-1,T}}\begin{bmatrix}\boldsymbol{y_{2t}}\\\tilde{\boldsymbol{f}}_{\boldsymbol{2}}\end{bmatrix}\\ &= \boldsymbol{f_{1b}} - \left(\boldsymbol{L_2^{-1}}\begin{bmatrix}\boldsymbol{U_{2t}^{-1}C}\\\boldsymbol{0}\end{bmatrix}\right)^{\boldsymbol{T}}\begin{bmatrix}\boldsymbol{y_{2t}}\\\boldsymbol{0}\end{bmatrix} - \begin{bmatrix}\boldsymbol{U_{2t}^{-1}C}\\\boldsymbol{0}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{L_2^{-1,T}}\begin{bmatrix}\boldsymbol{0}\\\tilde{\boldsymbol{f}}_{\boldsymbol{2}}\end{bmatrix}\end{aligned} \tag{3.8a}$$

$$\begin{aligned}\boldsymbol{y_{2t}} &= \boldsymbol{f_{2t}} - \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{L_{1b}^{-1}B}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{U_1^{-1,T}}\begin{bmatrix}\tilde{\boldsymbol{f}}_{\boldsymbol{1}}\\\boldsymbol{y_{1b}}\end{bmatrix}\\ &= \boldsymbol{f_{2t}} - \left(\boldsymbol{U_1^{-1}}\begin{bmatrix}\boldsymbol{0}\\\boldsymbol{L_{1b}^{-1}B}\end{bmatrix}\right)^{\boldsymbol{T}}\begin{bmatrix}\boldsymbol{0}\\\boldsymbol{y_{1b}}\end{bmatrix} - \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{L_{1b}^{-1}B}\end{bmatrix}^{\boldsymbol{T}}\boldsymbol{U_1^{-1,T}}\begin{bmatrix}\tilde{\boldsymbol{f}}_{\boldsymbol{1}}\\\boldsymbol{0}\end{bmatrix}\end{aligned} \tag{3.8b}$$

The coefficient of $\begin{bmatrix} \boldsymbol{y_{2t}} \\ \boldsymbol{0} \end{bmatrix}$ in equation 3.8a is the transpose of $\boldsymbol{W_t}$ from the non-transpose case, which has been created in the factorization stage. Similarly, the coefficient of $\begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{y_{1b}} \end{bmatrix}$ in 3.8b is the transpose of $\boldsymbol{V_t}$. This leads to the reduced system below.

$$\left[\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{W^T} \\ \hline \boldsymbol{V^T} & \boldsymbol{I} \end{array}\right] \begin{bmatrix} \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f_{1b}} - \begin{bmatrix} \boldsymbol{U_{2t}^{-1}C} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{L_2^{-1,T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_2} \end{bmatrix} \\ \boldsymbol{f_{2t}} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{L_{1b}^{-1}B} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{U_1^{-1,T}} \begin{bmatrix} \boldsymbol{\tilde{f}_1} \\ \boldsymbol{0} \end{bmatrix} \end{bmatrix} \tag{3.9}$$

The right hand side of 3.9 requires solve sweeps of $\boldsymbol{L_2^{-1,T}}$ and $\boldsymbol{U_1^{-1,T}}$ (performed in parallel). These are large solves, of height $\boldsymbol{n/2}$, in fact they are the bulk of the work involved in solving the $\boldsymbol{S^T}$ matrix. Fortunately, this work can be reused in the next stage.

After the right hand side is constructed, all that remains to be done is to solve this reduced system. Since it is only of size $2\times\max(kl, ku)$ by $2\times\max(kl, ku)$, solving this system is not too costly. For the more general case, where the number of partitions is increased, a recursive scheme has been found.

3.1.2 Transpose D matrix

The $\boldsymbol{D}$ matrix is simpler to transpose than $\boldsymbol{S}$ because $\boldsymbol{D}$ is block diagonal. Recalling 3.3 the problem to be solved in this stage is:

$$\boldsymbol{D^T x} = \left[\begin{array}{cc|cc} \multicolumn{2}{c|}{\multirow{2}{*}{$\boldsymbol{U_1^T L_1^T}$}} & & \\ & & \multicolumn{2}{c}{\multirow{2}{*}{}} \\ \hline & & \multicolumn{2}{c}{\multirow{2}{*}{$\boldsymbol{L_2^T U_2^T}$}} \\ & & & \end{array}\right] \begin{bmatrix} \boldsymbol{x_{1t}} \\ \boldsymbol{x_{1b}} \\ \boldsymbol{x_{2t}} \\ \boldsymbol{x_{2b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tilde{y}_1} \\ \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{\tilde{y}_2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tilde{f}_1} \\ \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{\tilde{f}_2} \end{bmatrix} \tag{3.10}$$

The system 3.10 in is trivially broken up in to two which may be solved in parallel. Also, work done on $\tilde{\boldsymbol{f}}_{\boldsymbol{1}}$ and $\tilde{\boldsymbol{f}}_{\boldsymbol{2}}$ may be recovered at this point, by splitting the $\boldsymbol{y}$ vector in to large parts which have been solved already, and small parts which have not. Finally, because $\boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}}$ is lower triangular, and $\boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}}$ is upper, optimizations similar to those in 3.7 can be used to reduce the sweeps over sections $\boldsymbol{y}$ associated with these sub-matrices from a height of $\frac{n}{2}$ to $\max(kl, ku)$. The notation for subsections of the $\boldsymbol{U}$ and $\boldsymbol{L}$ matrices from those equations is repeated here.

$$\begin{aligned}
\begin{bmatrix} \boldsymbol{x_{1t}} \\ \boldsymbol{x_{1b}} \end{bmatrix} &= \boldsymbol{L}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \tilde{\boldsymbol{f}}_{\boldsymbol{1}} \\ \boldsymbol{y_{1b}} \end{bmatrix} = \boldsymbol{L}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \left(\boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \tilde{\boldsymbol{f}}_{\boldsymbol{1}} \\ \boldsymbol{0} \end{bmatrix} + \boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{y_{1b}} \end{bmatrix} \right) \\
&= \boldsymbol{L}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \left(\boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \tilde{\boldsymbol{f}}_{\boldsymbol{1}} \\ \boldsymbol{0} \end{bmatrix} + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \boldsymbol{y_{1b}} \end{bmatrix} \right) \qquad (3.11a) \\
\begin{bmatrix} \boldsymbol{x_{2t}} \\ \boldsymbol{x_{2b}} \end{bmatrix} &= \boldsymbol{U}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{y_{2t}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{2}} \end{bmatrix} = \boldsymbol{U}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \left(\boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{2}} \end{bmatrix} + \boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{y_{2t}} \\ \boldsymbol{0} \end{bmatrix} \right) \\
&= \boldsymbol{U}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \left(\boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{2}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{L}_{\boldsymbol{2t}}^{-\boldsymbol{1},\boldsymbol{T}} \boldsymbol{y_{2t}} \\ \boldsymbol{0} \end{bmatrix} \right) \qquad (3.11b)
\end{aligned}$$

Where $\boldsymbol{U}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \tilde{\boldsymbol{f}}_{\boldsymbol{1}} \\ \boldsymbol{0} \end{bmatrix}$ and $\boldsymbol{L}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}} \begin{bmatrix} \boldsymbol{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{2}} \end{bmatrix}$ were performed in the $\boldsymbol{S}$ stage. This means that the only large sweeps required at this point are the outermost ones associated with $\boldsymbol{L}_{\boldsymbol{1}}^{-\boldsymbol{1},\boldsymbol{T}}$ in 3.11a and $\boldsymbol{U}_{\boldsymbol{2}}^{-\boldsymbol{1},\boldsymbol{T}}$ in 3.11b.

3.1.3 Summary

The two partition transpose spike scheme described above has essentially the same positive aspects as the non-transpose scheme. Specifically, it contains the same number of $\frac{n}{2}$ height sweeps – two per partition. These sweeps make up the majority of the work for the solve when the matrix size is large and the bandwidth is small.

Because a normal BLAS solve requires two sweeps of height n, this scheme will give almost perfect $2\times$ scaling in these conditions.

3.2 Multi partition case

In order to increase the number of threads that may be used by SPIKE, the number of partitions is increased. This results in an increase in the size of the reduced system, as it did in the case of the non-transpose version. To resolve this issue, a recursive scheme is used to solve the reduced system. The recursive scheme for the reduced system is conceptually isolated from the rest of the problem – that is, we can abstract it away as just a matrix solve problem. So, we will begin by discussing the scheme to deal with the increased number of partitions, then go on to discuss the specifics of the recursive reduced system method.

3.2.1 New partitions

In this case, we have a matrix of the form:

(3.12)

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} & \boldsymbol{B_1} & & & \\ & \ddots & & & \\ & \boldsymbol{C_i} & \boldsymbol{A_i} & \boldsymbol{B_i} & \\ & & & \ddots & \\ & & & \boldsymbol{C_p} & \boldsymbol{A_p} \end{bmatrix} \tag{3.13}$$

Where p is some power of two equal to the number of partitions and i is some integer between 1 and p. The requirement that p is a power of two comes from the recursive reduced system scheme. The $\boldsymbol{A_i}$ is not necessarily the same size for all i, but they are all square.

Partitions 1 and p are similar to those in the two partition case. The inside partitions, however, are different in that they have both a $\boldsymbol{C_i}$ and $\boldsymbol{B_i}$ attached. Because of this, some of the optimizations available to partitions on the ends will no longer be available to them.

The overall method in this section is again to perform:

$$\boldsymbol{S^T D^T x = f} \tag{3.14a}$$

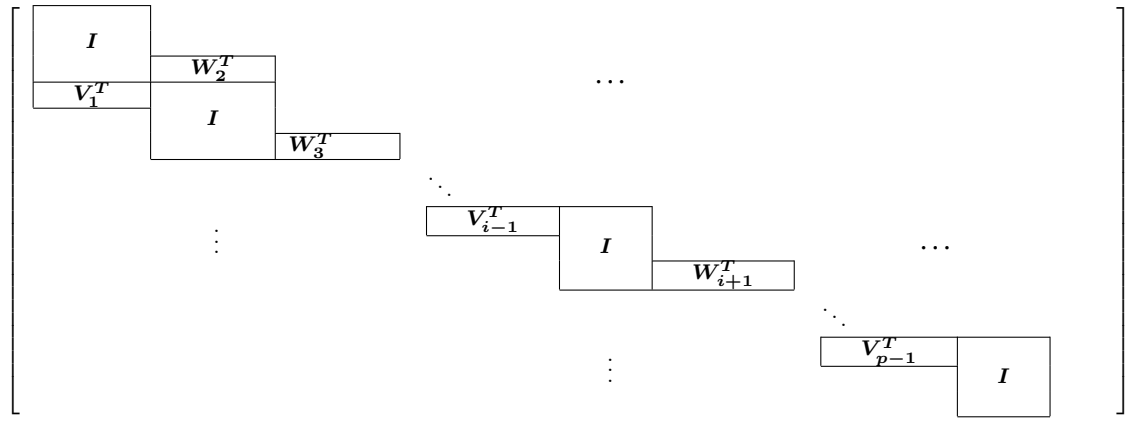
$$\boldsymbol{S^T y = f} \tag{3.14b}$$

$$\boldsymbol{D^T x = y} \tag{3.14c}$$

3.2.1.1 Transpose S matrix

The transpose $\boldsymbol{S}$ matrix has the form

$\boldsymbol{S^T} =$ (3.15)



The above graphic is not perfectly to scale. Specifically, the $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices are the same size as they have always been, k_u or k_l elements wide, and a height equal

to that of the partition with which they are associated. As such, the $\boldsymbol{V^T}$ matrices are k_u elements high, and the $\boldsymbol{W^T}$ matrices are k_l elements high.

The $\boldsymbol{S}$ matrix can be broken in to subproblems of the form (following the notation of 3.6):

$$\boldsymbol{y_{1t}} = \boldsymbol{f_{1t}} \tag{3.16a}$$

$$\tilde{\boldsymbol{y}}_{\boldsymbol{1}} = \tilde{\boldsymbol{f}}_{\boldsymbol{1}} \tag{3.16b}$$

$$\boldsymbol{y_{1b}} = \boldsymbol{f_{1b}} - \boldsymbol{W_2^T y_2} \qquad = \boldsymbol{f_{1b}} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_2} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_2^{-1,T}} \begin{bmatrix} \boldsymbol{y_{2t}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{2}} \\ \boldsymbol{y_{2b}} \end{bmatrix} \tag{3.16c}$$

$$\boldsymbol{y_{it}} = \boldsymbol{f_{it}} - \boldsymbol{V_{i-1}^T y_{i-1}} \qquad = \boldsymbol{f_{it}} - \begin{bmatrix} \boldsymbol{C_{i-1}} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_{i-1}^{-1,T}} \begin{bmatrix} \boldsymbol{y_{i-1t}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i-1}} \\ \boldsymbol{y_{i-1b}} \end{bmatrix} \tag{3.16d}$$

$$\tilde{\boldsymbol{y}}_{\boldsymbol{i}} = \tilde{\boldsymbol{f}}_{\boldsymbol{i}}$$

$$\boldsymbol{y_{ib}} = \boldsymbol{f_{ib}} - \boldsymbol{W_{i+1}^T y_{i+1}} \qquad = \boldsymbol{f_{ib}} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_{i+1}} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_{i+1}^{-1,T}} \begin{bmatrix} \boldsymbol{y_{i+1t}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i+1}} \\ \boldsymbol{y_{i+1b}} \end{bmatrix} \tag{3.16e}$$

$$\boldsymbol{y_{pt}} = \boldsymbol{f_{pt}} - \boldsymbol{V_{p-1}^T y_{p-1}} \qquad = \boldsymbol{f_{pt}} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{B_{p-1}} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_{p-1}^{-1,T}} \begin{bmatrix} \boldsymbol{y_{p-1t}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{p-1}} \\ \boldsymbol{y_{p-1b}} \end{bmatrix} \tag{3.16f}$$

$$\tilde{\boldsymbol{y}}_{\boldsymbol{p}} = \tilde{\boldsymbol{f}}_{\boldsymbol{p}}$$

$$\boldsymbol{y_{pb}} = \boldsymbol{f_{pb}} \tag{3.16g}$$

Note that, unlike in the two partitions case, the $\boldsymbol{A}$ matrices are not explicitly broken up into $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices. In the two partition case, the matrix was written in a factorized form to allow us to exploit the upper and lower triangular nature of $\boldsymbol{U}$ and $\boldsymbol{L}$ to reduce the number of full $\frac{n}{2}$ height sweeps. Those optimizations were dependent on knowing all of the values for a given partition of $\boldsymbol{y}$ past a certain point. As a result, in the many partition case these optimizations can only be applied to the sweeps over the first and last partitions of $\boldsymbol{y}$ which are not shown explicitly here. However, they are performed in exactly the same way as they were in the two partition case.

For the rest of the partitions ($\mathbf{1} < \boldsymbol{i} < \boldsymbol{p}$) we may break apart the known and unknown values to obtain the following:

$$\begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_i} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{y_{it}} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i}} \\ \boldsymbol{y_{ib}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_i} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \left(\begin{bmatrix} \boldsymbol{y_{it}} \\ \mathbf{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i}} \\ \mathbf{0} \end{bmatrix} \right) \tag{3.17a}$$

$$= \boldsymbol{W_i^T} \begin{bmatrix} \boldsymbol{y_{it}} \\ \mathbf{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_i} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \mathbf{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i}} \\ \mathbf{0} \end{bmatrix} \tag{3.17b}$$

$$= \begin{bmatrix} \boldsymbol{W_{it}^T} & \mathbf{0} & \boldsymbol{W_{ib}^T} \end{bmatrix} \begin{bmatrix} \boldsymbol{y_{it}} \\ \mathbf{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \boldsymbol{B_i} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \mathbf{0} \\ \tilde{\boldsymbol{f}}_{\boldsymbol{i}} \\ \mathbf{0} \end{bmatrix} \tag{3.17c}$$

$$\begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{y_{ib}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \left(\begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{0} \end{bmatrix} \right) \tag{3.18a}$$

$$= \boldsymbol{V_i^T} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{0} \end{bmatrix} \tag{3.18b}$$

$$= \begin{bmatrix} \boldsymbol{V_{it}^T} & \boldsymbol{0} & \boldsymbol{V_{ib}^T} \end{bmatrix} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{C_i} \\ \boldsymbol{0} \end{bmatrix}^{\boldsymbol{T}} \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{0} \end{bmatrix} \tag{3.18c}$$

This arrangement of the sub-matrices represents the implicit spike method. The $\boldsymbol{W}$ and $\boldsymbol{V}$ matrices are generated in the factorization stage, but only the tips are saved. However, for the set of multiplications involving $\boldsymbol{y}$ in 3.17c and 3.18c only the tips of the $\boldsymbol{W}$ and $\boldsymbol{V}$ are used. The middle values of these matrices are simply multiplied by zeros, so their value is unimportant.

The sweeps associated with $\boldsymbol{A_i^{-1,T}}$ over the $\boldsymbol{\tilde{f}_i}$ are unavailable. These are a part of building the modified right hand side for the reduced system. They do require two large $\frac{n}{2}$ sweeps each, and we cannot reuse them perfectly for the $\boldsymbol{D}$ stage, unlike in the two partition case. However, by performing the operations in this order, rather than explicitly generating the $\boldsymbol{W}$ and $\boldsymbol{V}$ spikes, we attain a substantial space savings. This is because the post-sweeps values for $\boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{0} \end{bmatrix}$ are stored back in to the $\boldsymbol{f_i}$ array. Because the $\boldsymbol{C_i}$ and $\boldsymbol{B_i}$ sub-matrices are only $k_l \times k_l$ and $k_u \times k_u$, we only need that much additional storage from these operations. This can lead to significant speedup, because the solve stage is likely to be memory bound (naturally this is dependent on the physical system and the specifics of the problem being solved). At this point we can build the reduced system:

$$
\begin{bmatrix}
\boldsymbol{I} & \boldsymbol{W_{2t}^T} & \boldsymbol{W_{2b}^T} & \ddots & & & & & & & \\
 & \ddots & & & & & & & & & \\
 & & \boldsymbol{V_{i-1b}^T} & \boldsymbol{V_{i-1t}^T} & \boldsymbol{I} & & & & & & \\
 & & & & & & & & \ddots & & \\
 & & & & & \boldsymbol{I} & \boldsymbol{W_{i+1t}^T} & \boldsymbol{W_{i+1b}^T} & & & \\
 & & & \ddots & & & & & & & \\
 & & & & & & & & \boldsymbol{V_{pb}^T} & \boldsymbol{V_{pt}^T} & \boldsymbol{I}
\end{bmatrix}
\begin{bmatrix}
\boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{y_{2b}} \\ \vdots \\ \boldsymbol{y_{i-1t}} \\ \boldsymbol{y_{i-1b}} \\ \boldsymbol{y_{it}} \\ \boldsymbol{y_{ib}} \\ \boldsymbol{y_{i+1t}} \\ \boldsymbol{y_{i+1b}} \\ \vdots \\ \boldsymbol{y_{p-1t}} \\ \boldsymbol{y_{p-1b}} \\ \boldsymbol{y_{pt}}
\end{bmatrix}
= \tag{3.19}
$$

$$
\begin{bmatrix}
\boldsymbol{g_{1b}} \\ \boldsymbol{g_{2t}} \\ \boldsymbol{g_{2b}} \\ \vdots \\ \boldsymbol{g_{i-1t}} \\ \boldsymbol{g_{i-1b}} \\ \boldsymbol{g_{it}} \\ \boldsymbol{g_{ib}} \\ \boldsymbol{g_{i+1t}} \\ \boldsymbol{g_{i+1b}} \\ \vdots \\ \boldsymbol{g_{p-1t}} \\ \boldsymbol{g_{p-1b}} \\ \boldsymbol{g_{pt}}
\end{bmatrix} \tag{3.20}
$$

The $\boldsymbol{g}$ vector is the modified right hand side, set up from combining 3.17c and 3.18c back in to 3.16. Explicitly,

$$\boldsymbol{g_{p-1b} = f_{p-1b} - \left(L_p^{-1} \begin{bmatrix} 0 \\ B_p \end{bmatrix} \right)^T U_p^{-1,T} \begin{bmatrix} 0 \\ \tilde{f}_p \\ f_{pb} \end{bmatrix}} \tag{3.21a}$$

$$\boldsymbol{g_{2t} = f_{2t} - \left(U_1^{-1} \begin{bmatrix} C_1 \\ 0 \end{bmatrix} \right)^T L_1^{-1,T} \begin{bmatrix} f_{1t} \\ \tilde{f}_1 \\ 0 \end{bmatrix}} \tag{3.21b}$$

$$\boldsymbol{g_{ib} = f_{ib} - \begin{bmatrix} 0 \\ B_{i+1} \end{bmatrix}^T A_{i+1}^{-1,T} \begin{bmatrix} 0 \\ \tilde{f}_{i+1} \\ 0 \end{bmatrix} \qquad 1 \leq i < p-1} \tag{3.21c}$$

$$\boldsymbol{g_{it} = f_{it} - \begin{bmatrix} C_{i-1} \\ 0 \end{bmatrix}^T A_{i-1}^{-1,T} \begin{bmatrix} 0 \\ \tilde{f}_{i-1} \\ 0 \end{bmatrix} \qquad 2 < i \leq p} \tag{3.21d}$$

The optimizations from the two partition case can naturally be applied to reduce the size of the $\boldsymbol{U_1^{-1}}$ and $\boldsymbol{L_p^{-1}}$ sweeps in 3.21a and 3.21b. Also, it should be noted that for a given sub-matrix $\boldsymbol{A_i}$, the solve is only performed once here – in the implementation, a thread is generally linked to a given sub-matrix of A. So, the $\boldsymbol{A_i}$ sweeps and the multiplications by $\boldsymbol{B_i}$ and $\boldsymbol{C_i}$ are performed in a thread, and then the result is sent to the threads that require it. This avoids repeating the work done for the matrix solves.

With that in mind, the reduced system can be solved at this point. The solution of the reduced system will be discussed in the next section. One could conceptually invert the matrix in 3.19, although the recursive method discussed later is much more efficient. After the reduced system has been solved, we know all of the values in the $\boldsymbol{y}$ vector, and so we will go on to the $\boldsymbol{D}$ stage.

3.2.1.2 Transpose D matrix

$$\boldsymbol{D^T x = y} \tag{3.22}$$

$$\begin{bmatrix} \boxed{\boldsymbol{U_1^T L_1^T}} & & & & \\ & \boxed{\boldsymbol{A_2^T}} & & & \\ & & \ddots & & \\ & & & \boxed{\boldsymbol{A_i^T}} & \\ & & & & \ddots \\ & & & & \boxed{\boldsymbol{L_p^T U_P^T}} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_i} \\ \vdots \\ \boldsymbol{x_p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \\ \vdots \\ \boldsymbol{y_i} \\ \vdots \\ \boldsymbol{y_p} \end{bmatrix}$$

For partitions $\boldsymbol{x_1}$ and $\boldsymbol{x_p}$, this stage is the same as it was in the two partition case. For the other partitions, we are in a slightly interesting situation. We know that the inner portions of $\boldsymbol{y_i}$, $\boldsymbol{\tilde{y}_i}$ should be equal to the inner portions of $\boldsymbol{f_i}$. So we would like to perform the following operation for all $1 < i < p$.

$$\boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{y_{ib}} \end{bmatrix} \tag{3.23}$$

But, because we have already performed $\boldsymbol{A}$ sweeps over the modified versions of the $\boldsymbol{f}$ in the previous stage, we no longer have $\boldsymbol{\tilde{f}_i}$ Instead, we must perform sweeps on a vector built from the tips of $\boldsymbol{y}$.

$$\boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{y_{ib}} \end{bmatrix} = \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{y_{it}} \\ \boldsymbol{0} \\ \boldsymbol{y_{ib}} \end{bmatrix} + \boldsymbol{A_i^{-1,T}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{\tilde{f}_i} \\ \boldsymbol{0} \end{bmatrix} \tag{3.24}$$

This incurs the cost of two large sweeps (bringing us up to a total of four for the middle partitions), and a large addition. There does not seem to be a way to avoid these sweeps, which somewhat intuitively bring us up to the same number of sweeps as the non-transpose case.

After this addition is performed, the problem is complete. The presence of the large addition would seem to give this algorithm a disadvantage compared to the non-transpose case, but preliminary tests have shown the transpose case running in the same or less time than the non-transpose case, for most problem sizes.

3.2.2 Recursive reduced system solve

As mentioned in the previous section, increasing the number of threads used by the SPIKE algorithm requires increasing the number of partitions. This means that the number of interfaces between these partitions is increased, which can result in an impractically large reduced system. The recursive method of solving the reduced system takes advantage of the fact that the reduced system is itself block diagonal, conceptually using the SPIKE algorithm to solve it. Another advantage of the recursive reduced system is that it is largely constructed in the factorization stage. In fact, the $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices have been created for the non-transpose case, for each level of recursion, in the factorization stage. The transpose case has been designed to take advantage of this fact.

Because the recursive method involves repeatedly recasting the problem in terms of new spike matrices, it can be difficult to follow. So, let us begin with the four partition case.

The reduced spike matrix will be designated as $\tilde{\boldsymbol{S}}_{\mathbf{1}}$. The transpose is shown below. The new indices indicate the recursion level. They begin at two because there will be two levels, and the spikes visible now are the bottom level.

$$\tilde{S}_1^T y_{red} = g = \begin{bmatrix} I & & & & & & & \\ & I & W_{2t1}^T & W_{2b1}^T & & & & \\ V_{1t1}^T & V_{1b1}^T & I & & & & & \\ & & & I & W_{3t1}^T & W_{3b1}^T & & \\ & & V_{2t1}^T & V_{2b1}^T & I & & & \\ & & & & & I & W_{4t1}^T & W_{4b1}^T \\ & & & & V_{3t1}^T & V_{3b1}^T & I & \\ & & & & & & & I \end{bmatrix} \begin{bmatrix} y_{1t} \\ y_{1b} \\ y_{2t} \\ y_{2b} \\ y_{3t} \\ y_{3b} \\ y_{4t} \\ y_{4b} \end{bmatrix} \tag{3.25}$$

This matrix is clearly block diagonal, and so we apply transpose SPIKE to it.

$$\tilde{S}_1^T = \left[\begin{array}{cccc|cccc} I & & & & & & & \\ & I & W_{2t1}^T & W_{2b1}^T & & & & \\ V_{1t1}^T & V_{1b1}^T & I & & 0 & 0 & & \\ & & & I & W_{3t1}^T & W_{3b1}^T & & \\ \hline & & V_{2t1}^T & V_{2b1}^T & I & & & \\ & & 0 & 0 & & I & W_{4t1}^T & W_{4b1}^T \\ & & & & V_{3t1}^T & V_{3b1}^T & I & \\ & & & & & & & I \end{array}\right] = \tag{3.26}$$

$$\left[\begin{array}{cccc|cc} I & & & & & \\ & I & & & & \\ & & I & & & \\ & & & I & W_{3t2}^T & W_{3b2}^T \quad W_{4t2}^T \quad W_{4b2}^T \\ \hline V_{1t2}^T & V_{1b2}^T & V_{2t2}^T & V_{2b2}^T & I & \\ & & & & & I \\ & & & & & & I \\ & & & & & & & I \end{array}\right] \left[\begin{array}{cccc|cccc} I & & & & & & & \\ & I & W_{2t1}^T & W_{2b1}^T & & & & \\ V_{1t1}^T & V_{1b1}^T & I & & & & & \\ & & & I & & & & \\ \hline & & & & I & & & \\ & & & & & I & W_{4t1}^T & W_{4b1}^T \\ & & & & V_{3t1}^T & V_{3b1}^T & I & \\ & & & & & & & I \end{array}\right] \tag{3.27}$$

$$= \tilde{S}_2^T D_2^T \tag{3.28}$$

The $\boldsymbol{V}$ and $\boldsymbol{W}$ blocks enclosed in dashed lines have become the new $\boldsymbol{B}$ and $\boldsymbol{C}$ partitions, and a level is of SPIKE is created. As noted earlier, the new $\boldsymbol{V}$ and $\boldsymbol{W}$ spikes have already been created for the non-transpose case in the factorization stage, so we can just use those matrices, transposed. Also, notice that the new matrices, $\boldsymbol{V_{12}^T}$, $\boldsymbol{V_{22}^T}$, $\boldsymbol{W_{32}^T}$ and $\boldsymbol{W_{42}^T}$ are actually each $2\max(k_l, k_u) \times \max(k_l, k_u)$. This is because the rows of zeroes in the boxed blocks in 3.26. For example,

$$\begin{bmatrix} \boldsymbol{I} & & \boldsymbol{V_{1t1}} & \\ & \boldsymbol{I} & \boldsymbol{V_{1b1}} & \\ & \boldsymbol{W_{2t1}} & \boldsymbol{I} & \\ & \boldsymbol{W_{2b1}} & & \boldsymbol{I} \end{bmatrix}^{-1} \left[\begin{array}{cc} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{V_{2t1}} & \boldsymbol{0} \\ \boldsymbol{V_{2b1}} & \boldsymbol{0} \end{array}\right] = \left[\begin{array}{cc} \boldsymbol{V_{1t2}} & \boldsymbol{0} \\ \boldsymbol{V_{1b2}} & \boldsymbol{0} \\ \hline \boldsymbol{V_{2t2}} & \boldsymbol{0} \\ \boldsymbol{V_{2b2}} & \boldsymbol{0} \end{array}\right] \tag{3.29}$$

The new spike retains the column of zeros (which becomes the bottom row of zeroes when transposed). This is convenient, as it limits the inter-element dependencies as we work our way through recursive steps. As a result, most of the work to solve these matrices can be done via modifying the right hand side with matrix multiplication, rather than performing large matrix solves.

This can be broken up in the conventional subproblems. The $\boldsymbol{z}$ vector will be used to keep track of intermediary stages in the recursive process.

$$\boldsymbol{\tilde{S}_2^T D_2^T y_{red}} = \boldsymbol{g} \tag{3.30}$$

$$\boldsymbol{\tilde{S}_2^T z} = \boldsymbol{g} \tag{3.31}$$

$$\boldsymbol{D_2^T y_{red}} = \boldsymbol{z} \tag{3.32}$$

Following the usual pattern, we will begin with the $\boldsymbol{\tilde{S}_2^T}$ matrix.

$$\left[\begin{array}{cccc|cccc} \boldsymbol{I} & & & & & & & \\ & \boldsymbol{I} & & & & & & \\ & & \boldsymbol{I} & & & & & \\ & & & \boldsymbol{I} & \boldsymbol{W_{3t2}^T} & \boldsymbol{W_{3b2}^T} & \boldsymbol{W_{4t2}^T} & \boldsymbol{W_{4b2}^T} \\ \hline \boldsymbol{V_{1t2}} & \boldsymbol{V_{1b2}} & \boldsymbol{V_{2t2}^T} & \boldsymbol{V_{2b2}^T} & \boldsymbol{I} & & & \\ & & & & & \boldsymbol{I} & & \\ & & & & & & \boldsymbol{I} & \\ & & & & & & & \boldsymbol{I} \end{array}\right] \left[\begin{array}{c} \boldsymbol{z_{1t}} \\ \boldsymbol{z_{1b}} \\ \boldsymbol{z_{2t}} \\ \boldsymbol{z_{2b}} \\ \boldsymbol{z_{3t}} \\ \boldsymbol{z_{3b}} \\ \boldsymbol{z_{4t}} \\ \boldsymbol{z_{4b}} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{g_{1t}} \\ \boldsymbol{g_{1b}} \\ \boldsymbol{g_{2t}} \\ \boldsymbol{g_{2b}} \\ \boldsymbol{g_{3t}} \\ \boldsymbol{g_{3b}} \\ \boldsymbol{g_{4t}} \\ \boldsymbol{g_{4b}} \end{array}\right] \tag{3.33}$$

All of $\boldsymbol{z}$ is known and equal to the corresponding value of $\boldsymbol{g}$, with the exception of $\boldsymbol{z_{2b}}$ and $\boldsymbol{z_{3t}}$. These can be then be multiplied by the corresponding sections of the $\boldsymbol{V}$ and $\boldsymbol{W}$ matrices. This results in a reduced system of size $(2\times\max(k_l, k_u))^2$.

$$\left[\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{W_{3t2}^T} \\ \hline \boldsymbol{V_{2b2}^T} & \boldsymbol{I} \end{array}\right] \begin{bmatrix} \boldsymbol{z_{2b}} \\ \boldsymbol{z_{3t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{g_{2b}} \\ \boldsymbol{g_{3t}} \end{bmatrix} - \begin{bmatrix} \boldsymbol{W_{3b2}^T} & \boldsymbol{W_{4t2}^T} & \boldsymbol{W_{4b2}^T} \\ \boldsymbol{W_{1t2}^T} & \boldsymbol{W_{1b2}^T} & \boldsymbol{W_{2t2}^T} \end{bmatrix} \begin{bmatrix} \boldsymbol{z_{3b}} \\ \boldsymbol{z_{4t}} \\ \boldsymbol{z_{4b}} \end{bmatrix} \begin{bmatrix} \boldsymbol{z_{1t}} \\ \boldsymbol{z_{1b}} \\ \boldsymbol{z_{2t}} \end{bmatrix} \tag{3.34}$$

After this is solved, we move on to the $\boldsymbol{D}$ stage of this recursive level. Looking back at 3.27, now that the $\boldsymbol{S}$ stage is solved the $\boldsymbol{D}$ matrix is effectively decoupled. So, we have two independent problems, which can be distributed to different cores.

$$\begin{bmatrix} \boldsymbol{I} & & & \\ & \boldsymbol{I} & \boldsymbol{W_{4t1}^T} & \boldsymbol{W_{4b1}^T} \\ \boldsymbol{V_{3t1}^T} & \boldsymbol{V_{3b1}^T} & \boldsymbol{I} & \\ & & & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{z_{3t}} \\ \boldsymbol{z_{3b}} \\ \boldsymbol{z_{4t}} \\ \boldsymbol{z_{4b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{3t}} \\ \boldsymbol{y_{3b}} \\ \boldsymbol{y_{4t}} \\ \boldsymbol{y_{4b}} \end{bmatrix} \tag{3.35a}$$

$$\begin{bmatrix} \boldsymbol{I} & & & \\ & \boldsymbol{I} & \boldsymbol{W_{2t1}^T} & \boldsymbol{W_{2b1}^T} \\ \boldsymbol{V_{1t1}^T} & \boldsymbol{V_{1b1}^T} & \boldsymbol{I} & \\ & & & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{z_{1t}} \\ \boldsymbol{z_{1b}} \\ \boldsymbol{z_{2t}} \\ \boldsymbol{z_{2b}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y_{1t}} \\ \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \\ \boldsymbol{y_{2b}} \end{bmatrix} \tag{3.35b}$$

We are now at the bottom level of the recursion – the sub-blocks of this $\boldsymbol{D}$ matrix are already set up as if they were each an $\boldsymbol{S}$ matrix. So, we may simply solve these to find the final version of $\boldsymbol{y}$,

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{W}_{4t1}^{\boldsymbol{T}} \\ \boldsymbol{V}_{3b1}^{\boldsymbol{T}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{y_{3b}} \\ \boldsymbol{y_{4t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{z_{3b}} & -\boldsymbol{V}_{3t1}^{\boldsymbol{T}} \boldsymbol{z_{3t}} \\ \boldsymbol{z_{4t}} & -\boldsymbol{W}_{4b1}^{\boldsymbol{T}} \boldsymbol{z_{4b1}} \end{bmatrix} \tag{3.36a}$$

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{W}_{2t1}^{\boldsymbol{T}} \\ \boldsymbol{V}_{1b1}^{\boldsymbol{T}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{y_{1b}} \\ \boldsymbol{y_{2t}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{z_{1b}} & -\boldsymbol{V}_{1t1}^{\boldsymbol{T}} \boldsymbol{z_{1t}} \\ \boldsymbol{z_{2t}} & -\boldsymbol{W}_{2b1}^{\boldsymbol{T}} \boldsymbol{z_{2b1}} \end{bmatrix} \tag{3.36b}$$

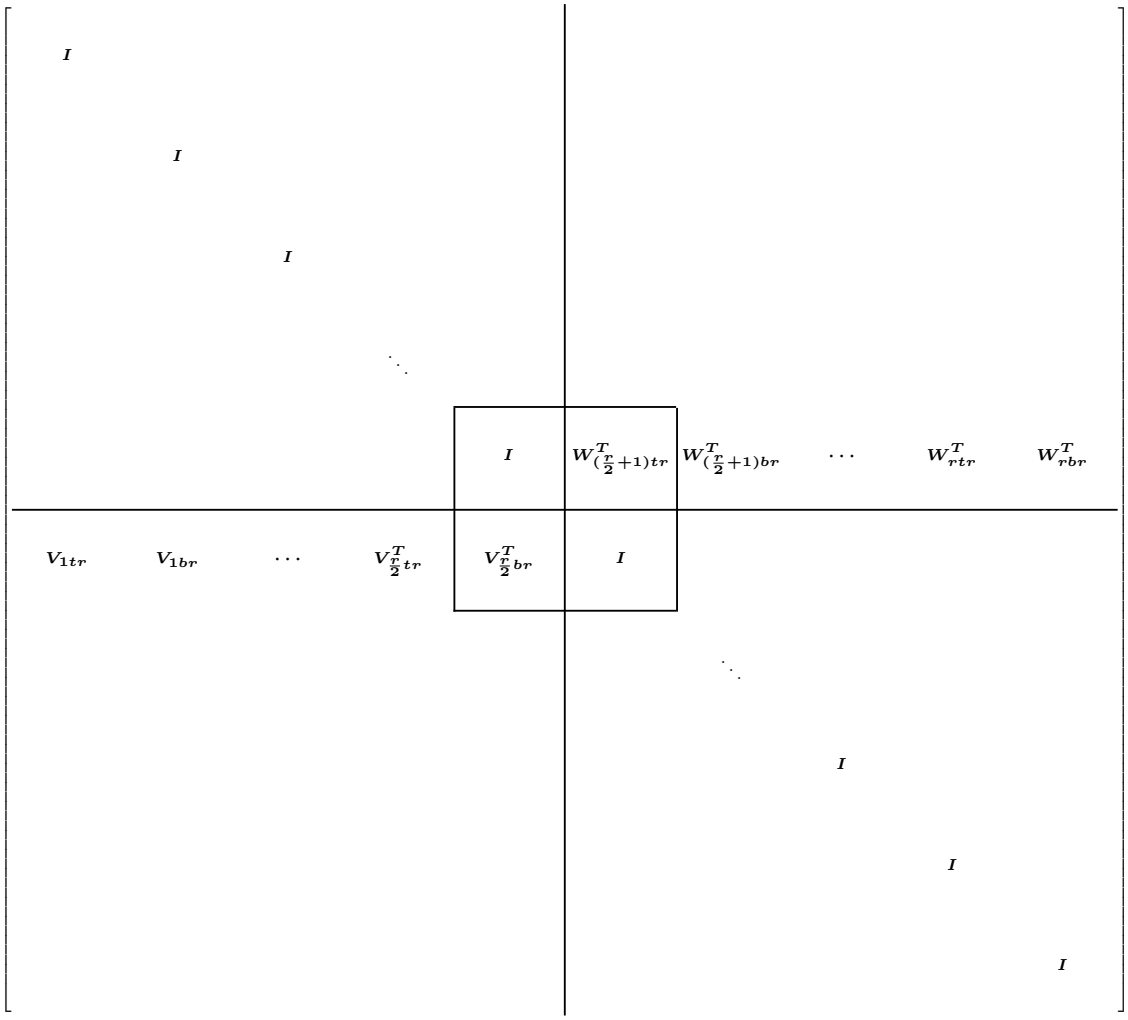
After solving these problems, the recursive scheme is completed. Because the matrix solves are performed in parallel for a given level, the critical path length in this case is two matrix solves.

To expand this process to a larger number of partitions, we would just continue performing the transpose SPIKE factorization. This results in a sequence of $\boldsymbol{D}$ matrices. The matrices used for this process may seem to be numbered in reverse – this is to retain compatibility with the non-transpose version. Each recursive stage works on twice as many partitions as the previous. So, if the total number of partitions is p, the total of stages is r=$\log_2(p)$, including the outermost $\boldsymbol{S}$ stage.

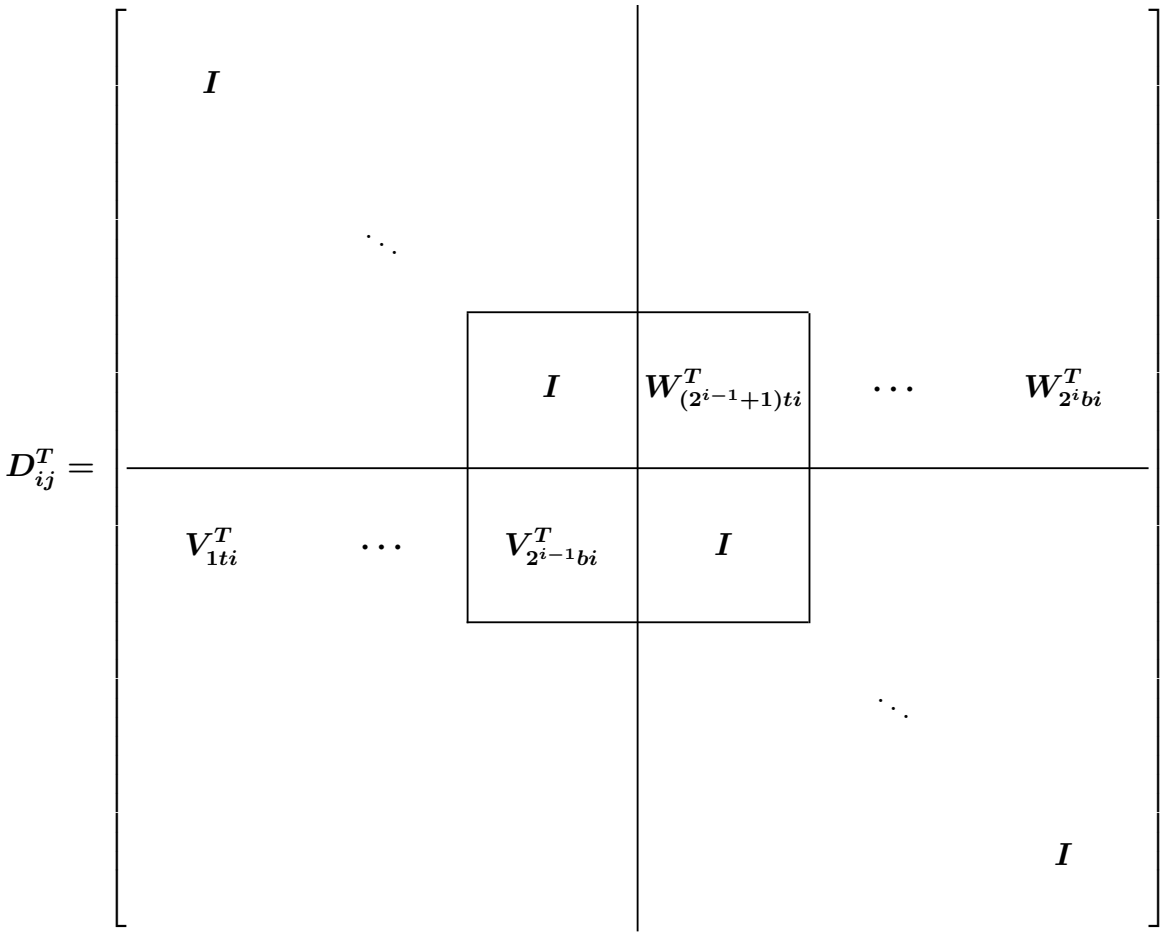
$$\tilde{\mathbf{S}}_{\mathbf{1}}^{\mathbf{T}} = \tilde{\mathbf{S}}_{r}^{\mathbf{T}} \Pi_{i=r-1}^{1} \mathbf{D}_{\mathbf{i}}^{\mathbf{T}} \tag{3.37}$$

The shape of these matrices is shown below. The process to solve these matrices is a natural extension of the four partition example given above, so it will not be shown here. $\boldsymbol{D_{i,j}}$, used below, is $2^i\max(k_l,k_u)\times 2^i\max(k_l,k_u)$ elements in size.

$$\tilde{\mathbf{S}}_r^{\mathbf{t}} = \tag{3.38}$$



$$\boldsymbol{D}_i^{\boldsymbol{T}} = \begin{bmatrix} \boldsymbol{D}_{i1}^{\boldsymbol{T}} & & & \\ & \ddots & & \\ & & \boldsymbol{D}_{ij}^{\boldsymbol{T}} & \\ & & & \ddots \\ & & & \boldsymbol{D}_{i2^{r-i}}^{\boldsymbol{T}} \end{bmatrix} \tag{3.39}$$



(3.40)

This concludes the description of the transpose recursive reduced system solve. In preliminary tests the recursive reduced system solve has not consumed a significant amount of the run time. However, this has not specifically been measured, it is inferred from the fact that other portions of the program combine to make up the vast majority of the run time in most cases.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 Advantages and Disadvantages of OpenMP

Overall, OpenMP is a useful tool for parallel programming. First and foremost, it is well supported and portable. The OpenMP API is supported by majority of major compilers for Fortran and C. Because support is provided by the compiler, an end user is not required to locate an obscure external library.

OpenMP is also fairly user friendly. The API has been designed to allow the user to add parallelism after the base program is working. For example a do-loop may be run in parallel, assuming no dependencies between iterations, with a single call to the API. This will create a create a task for each iteration of the loop, each of which may be completed by a different thread.

Additionally, the fork/join model is fairly intuitive and well known. In general, the expectation is that a single master thread will work sequentially until a parallel section is encountered. At this point the master thread will spawn a set of child threads, which will complete the tasks in the parallel section.

An added benefit of the fork/join model is the assumption that the calling code will be single-threaded. One goal of this project was the creation of a version of SPIKE that is easy to use. The fork/join model allows the user to write single threaded code, and offload the responsibility for managing threads to SPIKE.

Unfortunately, the main limitation encountered in OpenMP for this project was related to the fork/join model. In particular, the problem is nested parallelism. The child threads in a parallel region are able to encounter further parallel regions and

spawn their own set of child threads. This feature appeared to be a straightforward way to implement the flexible threading enhancement for the SPIKE algorithm, described in section two. Unfortunately, this feature is limited. An implementation may comply with the OpenMP standard by creating a thread team of size one when a nested parallel section is encountered. As a result, using nested parallelism may result in the use of no additional threads, which prevents the desired increase in parallelism.

The alternative to nested parallelism is to create all necessary threads at once. This allows us to ensure that the threads are created an distributed appropriately, but partitions upon which two threads are used require special handling, as the SPIKE 2×2 primitive requires some communication between threads. Explicit communication and waiting between pairs of threads is not a strength of OpenMP. It appears that the common method of synchronization is the use of OpenMP barriers, which cause all threads to wait, or the simple termination of the parallel section.

4.2 Point to Point communication in OpenMP

The issue of point to point communication in OpenMP is covered well in [1], which develops a general purpose function to indicate dependencies between OpenMP threads. Our requirements are only for synchronization between two threads. Additionally, our synchronizations take place infrequently, and between blocks of code significant computational cost. As a result, our code implements a simplified and specialized version of their method.

The basis of this implementation is a series of spin-locks, which monitor a counter in a shared section of memory. This counter is held in a small three element array, which also contains flags to indicate ownership of chunks of work. Ownership of these chunks of work is distributed explicitly before the parallel section is encountered. A simple example will make this scheme clear. Let us assume we have only two threads,

with thread numbers 1 and 2. Let us also assume that we have some blocks of code, called A,B,C, and D. A and B can be performed in parallel. C and D can also be performed in parallel, but require the results of A and B.

```
keys(1) = 0 !Counter to indicate how much work has been completed
keys(2) = 1 !Indicates ownership of some chunks by thread 1
keys(3) = 2 !Indicates ownership of some chunks by thread 2
!$OMP PARALLEL
  If keys(2) .eq. omp_get_thread_number() then
    Perform chunk A ! Thread 1 will do this work
    !$OMP ATOMIC
    keys(1) = keys(1) + 1
  End If
  If keys(3) .eq. omp_get_thread_number() then
    Perform chunk B ! Thread 2 will do this work
    !$OMP ATOMIC
    keys(1) = keys(1) + 1
  End If
  Do While( keys(1) .lt. 2 )
    !$OMP FLUSH
  End Do
  If keys(2) .eq. omp_get_thread_number() then
    Perform chunk C ! Thread 1 will do this work
  End If
  If keys(3) .eq. omp_get_thread_number() then
    Perform chunk D ! Thread 2 will do this work
  End If
```

!$OMP END PARALLEL

The ATOMIC command prevents a possible write collision, while the FLUSH command indicates that the thread should look to memory and ensure that its context accurately reflects the global context. Without the FLUSH commands, the spin-lock loops could continue spinning indefinitely, as they would not be notified of the change in the key array.

This appears to be a simple and efficient way to synchronise a pair of OpenMP threads. The use of spin-locks causes the threads to stay awake, preventing them from being retasked by the OS. For SPIKE, this is acceptable because the run times of the pairs of tasks are nearly identical. In the presence of load imbalance, a slightly more complicated scheme would likely be necessary to allow for the reuse of waiting threads.

CHAPTER 5

RESULTS

The following benchmarks were possible thanks to the kindness of Dr. Sameh, and his associates at Purdue, whom generously allowed us to use their cluster, Golub. Golub is a large shared memeory machine with the following characteristics and software:

- 8×Intel® Xeon® E7-8870: 10 cores @ 2.40 GHz with 30MB cache
- Intel® fortran 12.0.4
- Intel® MKL 10.3.4
- OpenMP 3.0

The Xeon® E7-8870 uses hyperthreading, which causes each core to appear as two threads. For HPC applications this ability is frequently detrimental. For these experiments, hyperthreads have been avoided using OpenMP core affinity settings. For Intel® compilers this functionality is accessed using the following Linux environment variable.

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

The substring 'compact' instructs the OpenMP runtime to link cores to threads in such a way that neighboring OpenMP threads are located as 'closely' as possible. The substring '1,0' defines cores inside the same socket to be 'very close,' and hyperthreads for the same core to be 'very far away.' A more detailed description may be found online, at the following address:

https://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/fortran/lin/optaps/common/optaps\openmp\thread\affinity.htm

By using the above affinity setting, and limiting ourselves to 80 threads at most, we will ensure that hyperthreads are always seen as too far away to use. Unfortunately, thread affinity settings are a non-standard feature in OpenMP, and so this environment variable would need to be modified for alternative compilers.

5.1 Partition size ratios

In chapter two, we discuss the relative size of the partitions into which the matrix $\boldsymbol{A}$ is to be broken. The relationship between the partition sizes is defined in terms of ratios, and these ratios are largely dependent the specifics of the underlying BLAS implementation.

Variables are defined as in chapter two – k_{lu} is the half-bandwidth of the $\boldsymbol{A}$ matrix, and it is assumed that the upper and lower bandwidths are equal. The following equations define the ratios for cases in which the bandwidth is much greater than the number of vectors in the solution, n_{rhs}. K is the hardware dependent tuning variable.

$$R_{12} = \frac{n_1}{n_2} = \frac{1}{2} + \frac{K_2}{K_1} = \frac{1}{2} + K \tag{5.1a}$$

$$R_{13} = \frac{n_1}{n_3} = 1 + 1.5\frac{K_2}{K_1} = 1 + 1.5K \tag{5.1b}$$

Using the simplified version of the big-O run-times for the factorization and solve, we will attempt to find K. This is possible by performing a factorization and solve on a matrix for which the bandwidth is equal to the number of vectors in the solution; $k_{lu} = n_{rhs}$.

$$\text{factorization time} = K_1 \times n \times k_{lu}^2 \tag{5.2a}$$

$$\text{solve time} = K_2 \times n \times k_{lu} \times n_{rhs} \tag{5.2b}$$

$$K = \frac{K_2}{K_1} \tag{5.2c}$$

$$= \frac{\text{solve time}}{n \times k_{lu} \times n_{rhs}} \times \frac{n \times k_{lu}^2}{\text{factorization time}} \tag{5.2d}$$

$$= \frac{\text{solve time}}{\text{factorization time}} \tag{5.2e}$$

To begin, a problem of size $N = 640,000$ $k_{lu} = n_{rhs} = 256$ was run. This resulted in a tuning variable value $K = \frac{22.8}{13.5} = 1.7$. The partition ratios associated with this value are $R_{12} = 2.2$ and $R_{13} = 3.5$

To check the accuracy of this method, a search of plausible ratio values was performed. This search was performed using for 16, 23, and 30 OpenMP threads. These values were chosen to cover the different cases for the partition schemes. 16 threads ensures that all partitions are given one thread. 30 threads ensures that all partitions, with the exception of the first and last, are given two threads. 23 is directly in the middle, with half of the inner partitions allocated one thread, and the other half allocated two.

An alternative selection would have been 32,47,and 62 threads. However, as the number of partitions is increased, the problem becomes less sensitive to changes in the ratios. The selected numbers of threads make the benefit clearly visible.

For the factorization stage, matrices of size $N = 1,000,000$ with bandwidths of $2 \times k_{lu} = 320$ and 160 were used. In figure 5.1 we see the results for 16 threads. As one would suspect, the run times are not modified by the ratio R_{12}, because there are no two-thread partitions. The computed ratio is directly inside the band of good values. In figure 5.2 we can see that the run times are insensitive to R_{13} for 30 threads. In this case we are slightly off of the optimal value when bandwidth is 160, but still within roughly .04s, or 6.5% of the run time of the optimal case. Finally, in figure

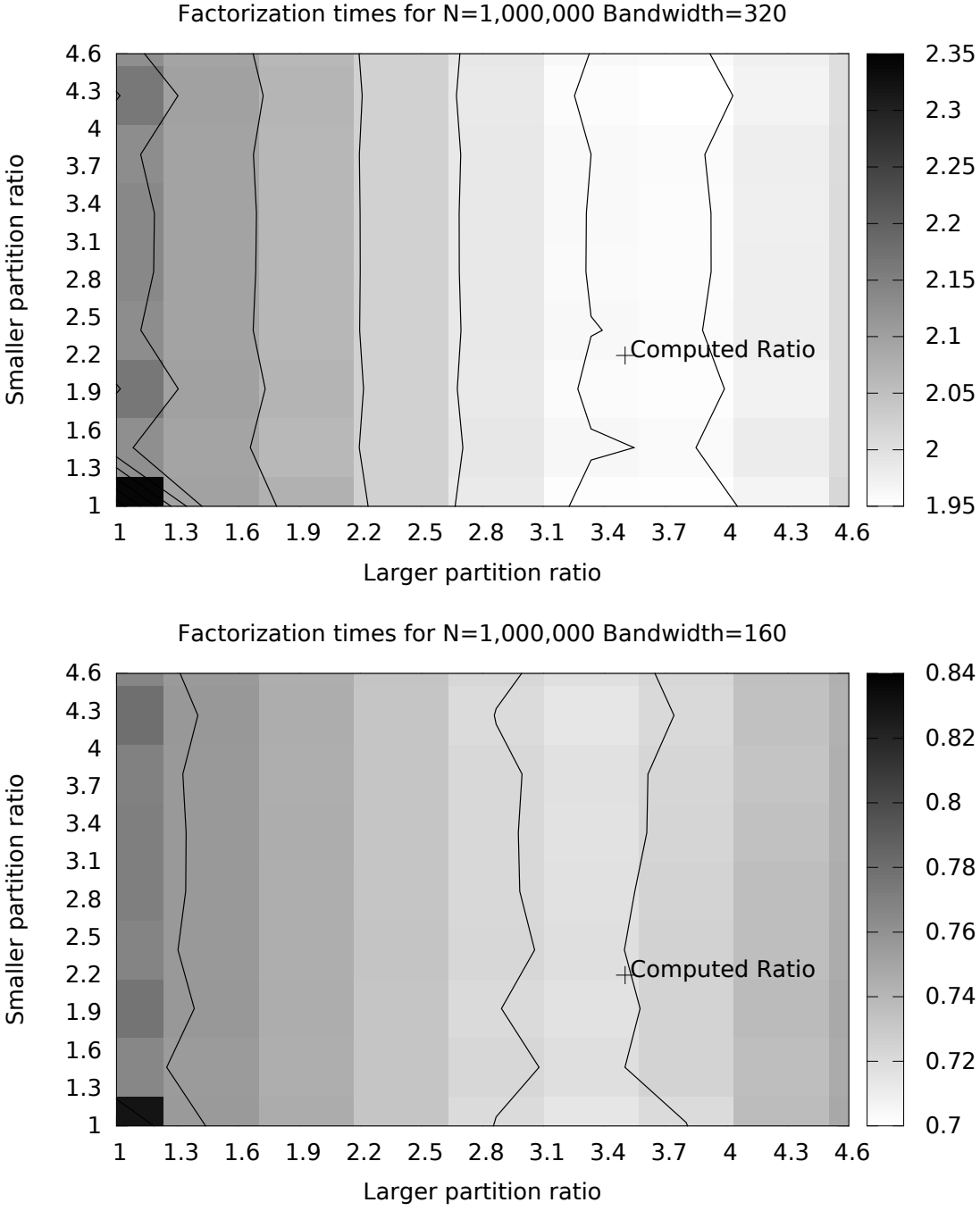


Figure 5.1. Partition size map for 16 threads, contours represents .04s

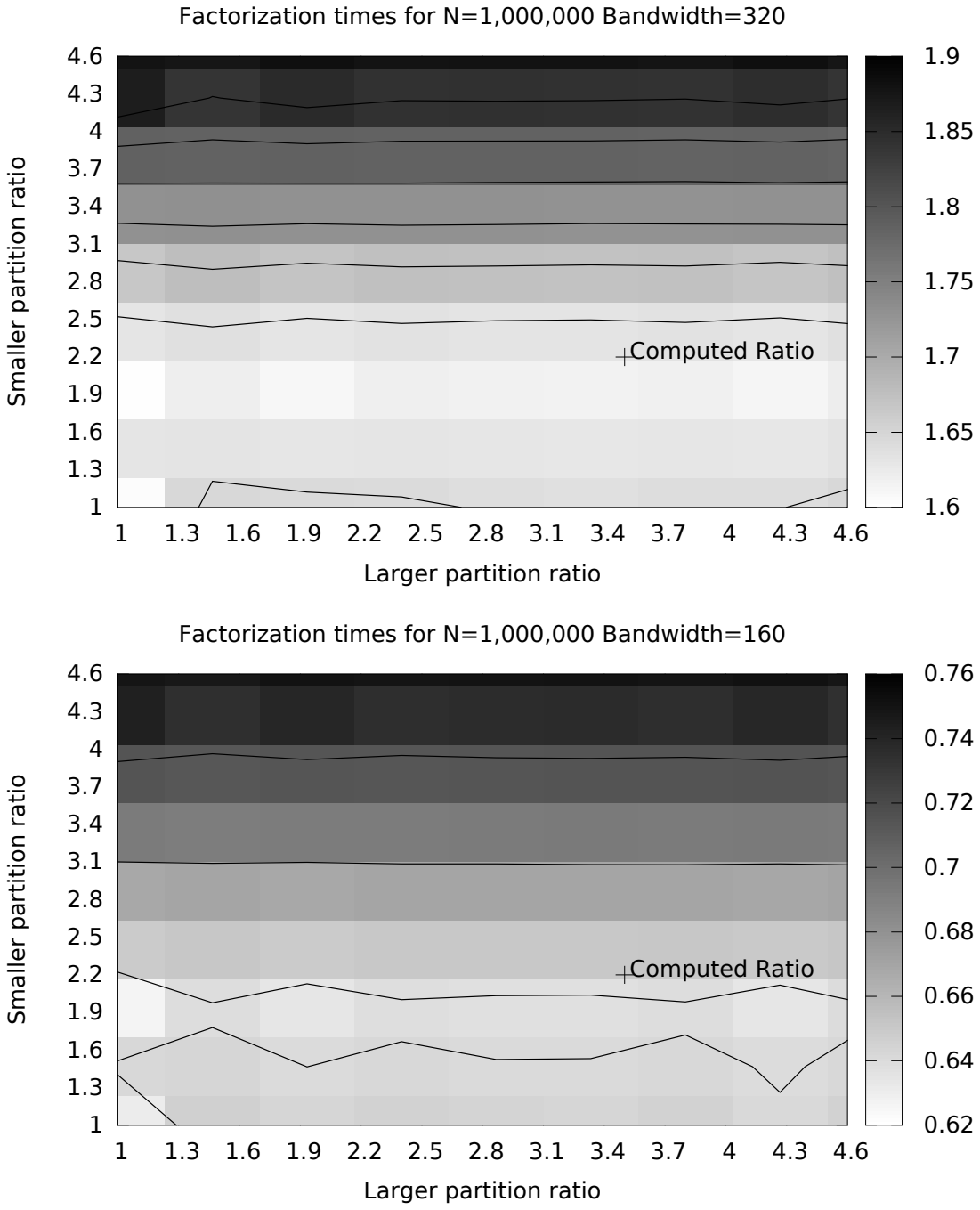


Figure 5.2. Partition size map for 30 threads, contours represents .04s

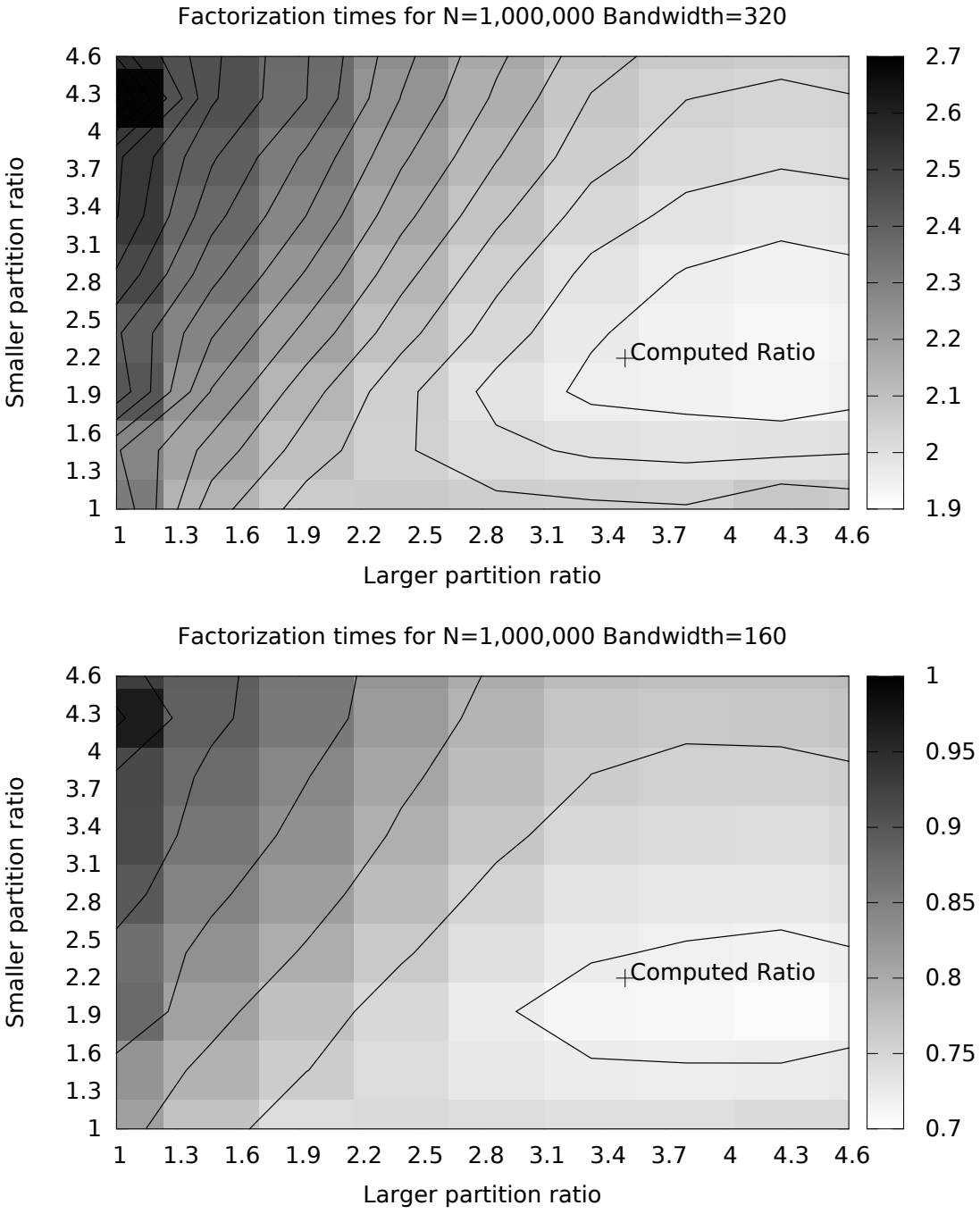


Figure 5.3. Partition size map for 23 threads, contours represents .04s

5.3, we see the case in which the run time depends on both ratios. As expected, the computed ratios are in the area defining the fastest set of runs. For the solve stage, the expected optimal ratios are $R_{12} = 1$ and $R_{13} = 2$, independent of the tuning variable K. This means that the value for R_{12} was at the minimum edge of the ratios checked. Despite this, in figures 5.4, 5.5 and 5.6 show the computed ratios are consistent with the ratios found through searching.

As a result, we may come to the conclusion that the tuning ratios may be found by performing a single computation, rather than searching the total space of possible ratios. This will ease the process of installing this implementation of the SPIKE algorithm considerably.

5.2 Scaling

We now would like to measure the scalability of the enhanced load balancing scheme. The overall goal is to obtain additional speedup when the number of threads used is not a power of two. We will use the tuning ratios obtained in the previous section.

In this section, we will compare the performance of this implementation of SPIKE to two other solvers. The baseline is the banded Lapack solver, provided by Intel® MKL 10.3.4. We will also compare against an older version of SPIKE, implemented in MPI. This older version does not include the flexible threading features, or the LU/UL factorization strategy.

In 5.7 and 5.8 we see the factorization stage of these solvers compared. The new SPIKE-OpenMP implementation outperforms the MKL solver in most cases, with the exception of a small number of threads, for larger bandwidths. The SPIKE-OpenMP and SPIKE-MPI implementations are more competitive in performance. For smaller numbers of threads, SPIKE-OpenMP has a clear advantage, owing to

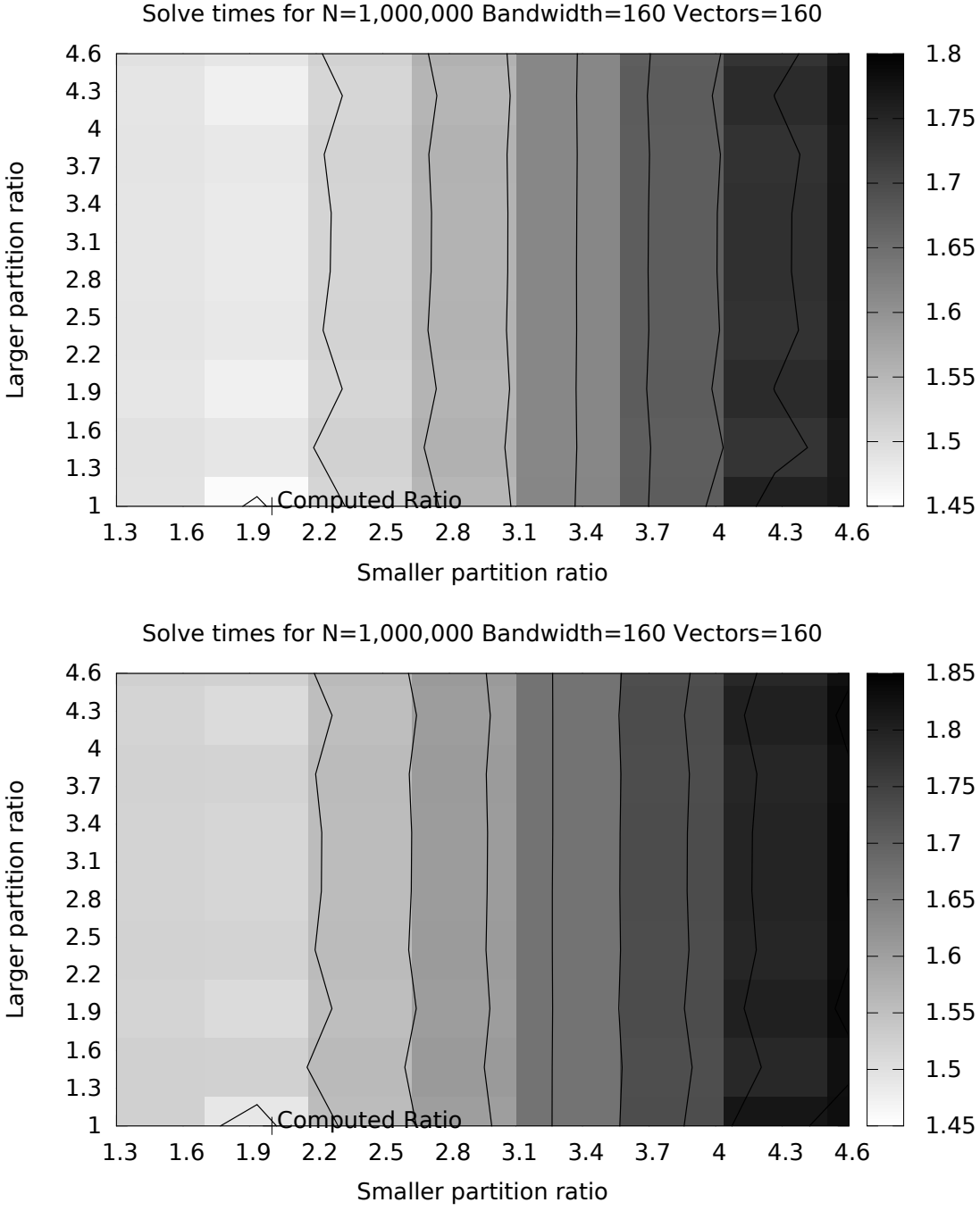


Figure 5.4. Partition size map for 16 threads, contours represents .04s

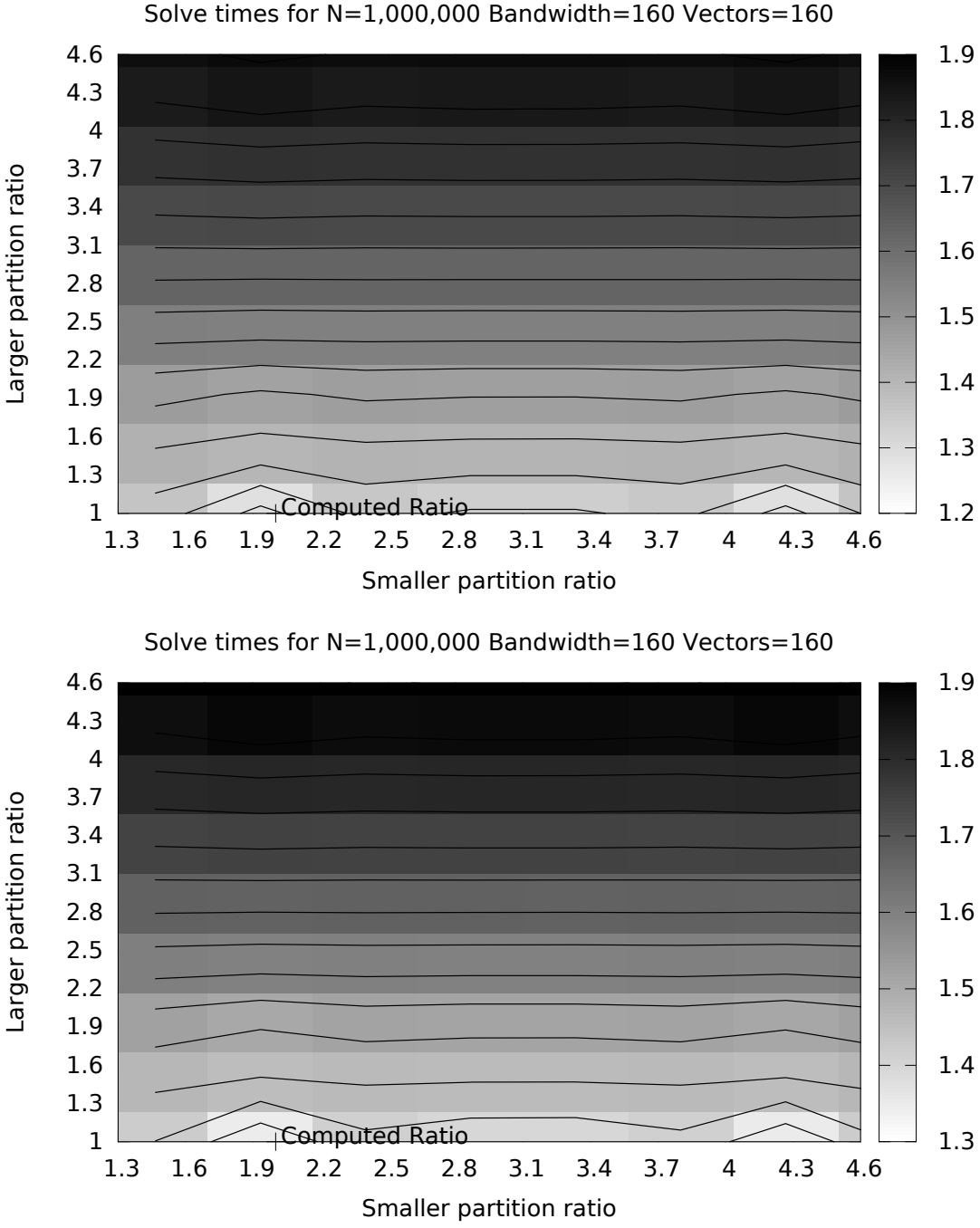


Figure 5.5. Partition size map for 30 threads, contours represents .04s

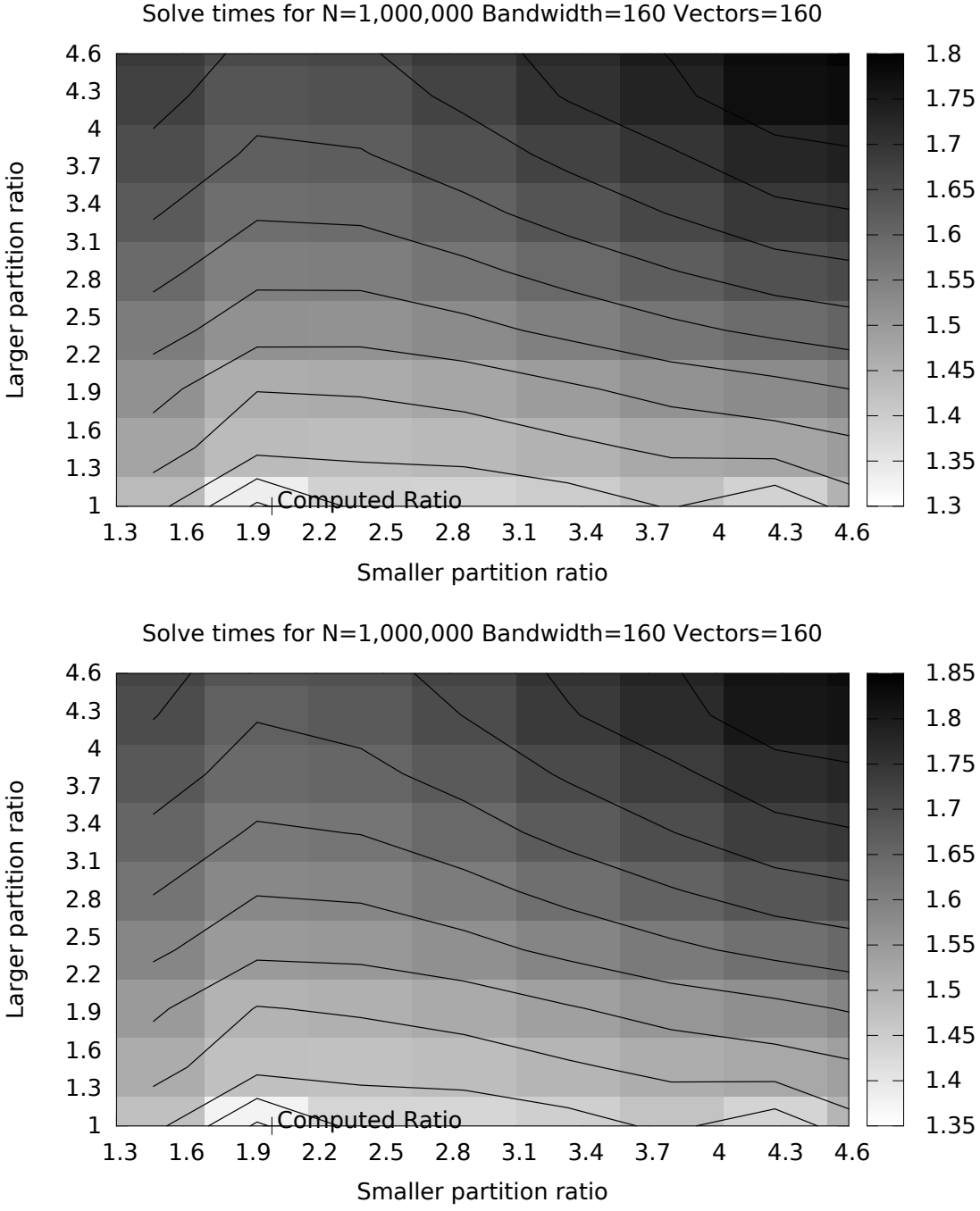


Figure 5.6. Partition size map for 23 threads, contours represents .04s

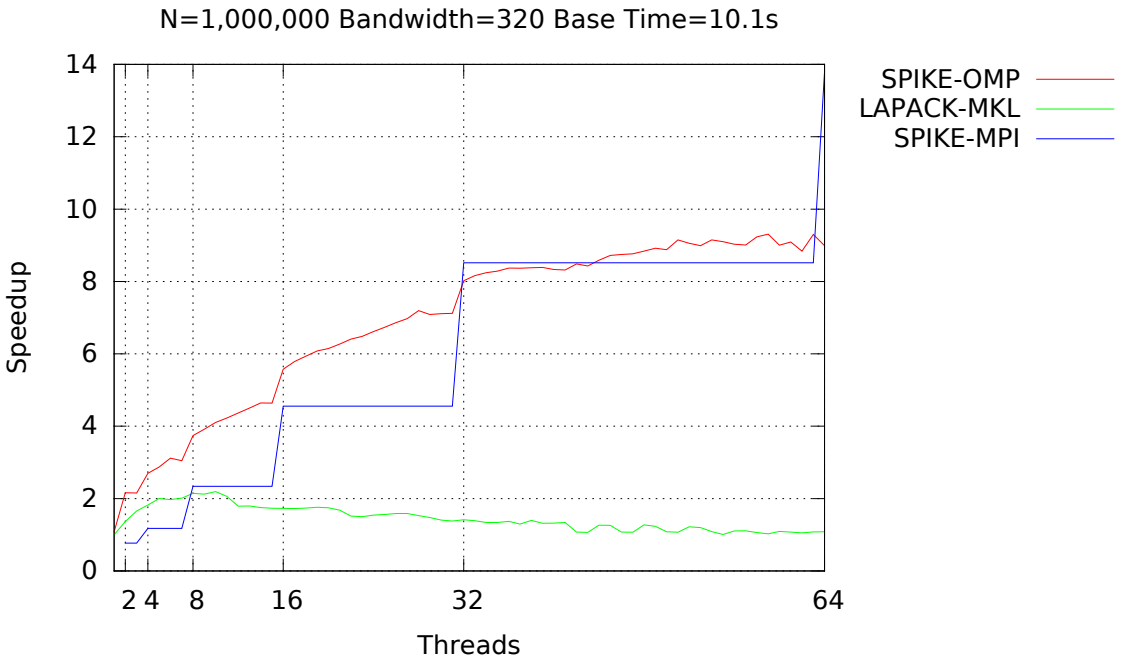


Figure 5.7. Comparison of factorization stage scalability for bandwidth 320, with matrix size 1M

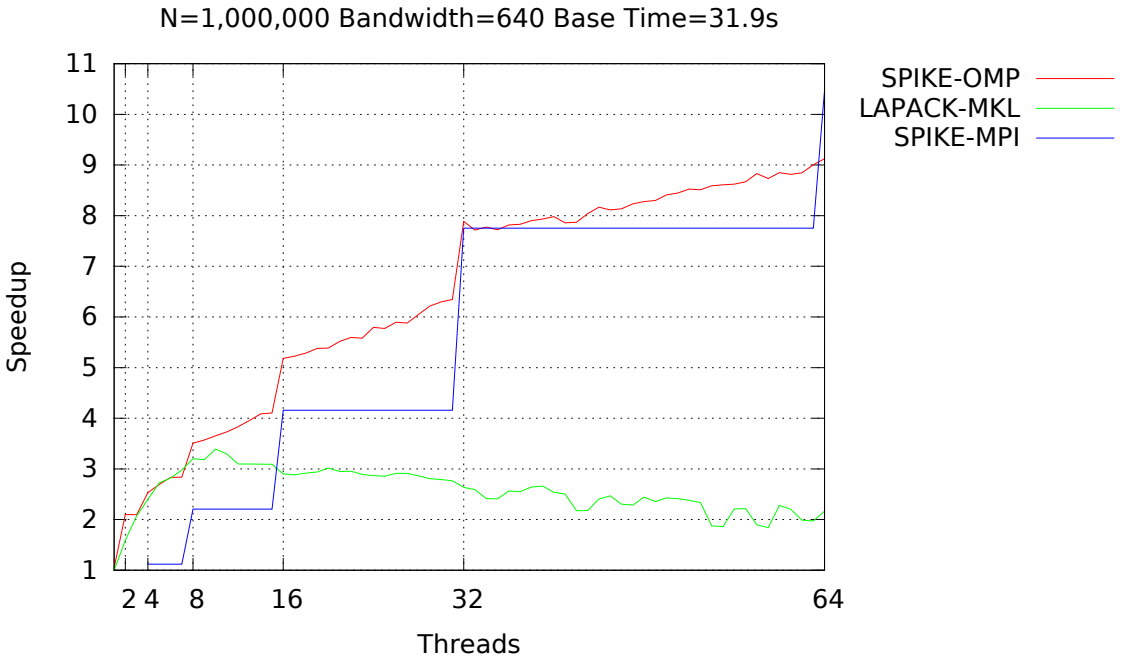


Figure 5.8. Comparison of factorization stage scalability for bandwidth 640, with matrix size 1M

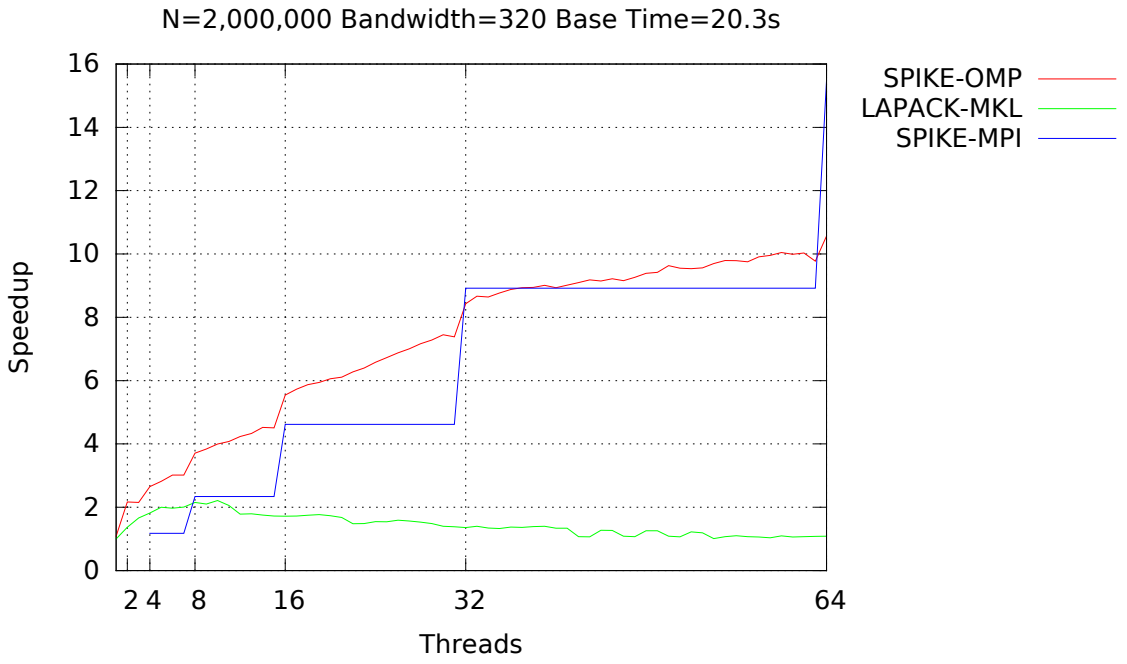


Figure 5.9. Comparison of factorization stage scalability for bandwidth 320, with matrix size 2M

the enhancements that have been made to the algorithm. As the number of threads increases, the older MPI implementation overcomes the OpenMP implementation.

This is likely a result of the explicit nature of communication in MPI. The assumption for SPIKE-OpenMP is that the code will be called from a single threaded environment, so the benchmarking code for SPIKE-OpenMP creates the matrix $\boldsymbol{A}$ with just one thread. When the OpenMP section is opened, the proper sections of the matrix $\boldsymbol{A}$ are unlikely to be resident in the proper cache for most cores. This is not a problem for the MPI code, which distributes the matrix outside of the benchmarking code. Conceptually, the assumption is that an OpenMP program will originate as a single-threaded code, to which parallelism is added. MPI, on the other hand, requires that the code be designed from the ground up with parallelism in mind. So, these benchmarks reflect the most likely real world use case for their respective paradigms.

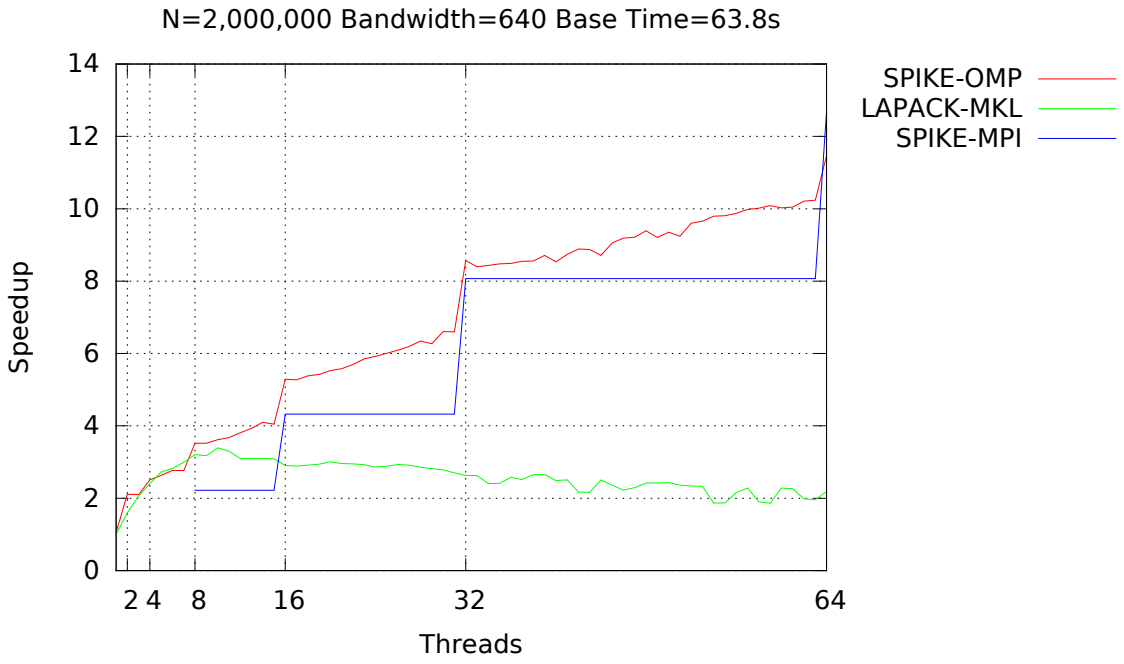


Figure 5.10. Comparison of factorization stage scalability for bandwidth 640, with matrix size 2M

In figures 5.9 and 5.10, the matrix size is increased to 2,000,000. The results are largely the same. For a small number of threads, the SPIKE-MPI failed to perform the factorization. This is likely a consequence of the MPI runtime limiting the amount of memory allocated per thread. Decreasing the number of threads increases the amount of memory that must be allocated to each thread, as a larger portion of the matrix must be held by each thread.

The most interesting measurement is this set is shown in 5.10. It appears that SPIKE-OpenMP maintains an advantage over SPIKE-MPI in with this extremely large matrix in until 64 threads are reached. This is consistent with the hypothesis that SPIKE-MPI is obtaining an advantage from explicit communication resulting in better data locality. As the matrix increases in size, the percentage of the matrix that is likely to begin inside the proper cache is reduced for SPIKE-MPI.

5.3 Solve Stage

In the solve stage, we look at the performance of the normal and transpose solve. We desire the solve times to be uniform between the two. For the first set of runs, we will use the solve-stage tuned partition size ratios for SPIKE-OpenMP.

In figure 5.11 we see a substantial advantage for the new SPIKE-OpenMP. Given that the advantage is retained over the SPIKE-MPI implementation for numbers of threads equal to a power of two, it is likely that a much of the advantage comes from having the optimal partition size ratios.

It is also worthwhile to recall that, as mentioned in section 1.2.2, the SPIKE uses an in-house banded primitive to perform the factorizations and solves on the diagonal blocks of $\boldsymbol{A}$. The banded primitive is configured to use boosting, rather than pivoting, in the presence of zero-pivots. Additionally, the banded primitive is designed to improve cache locality. This is done by tiling the matrix and performing all of the work for a given tile of the matrix on each of the vectors in the solution before moving along the diagonal to the next tile. For example, the banded primitive performs the factorization and solve a matrix of size N=1,000,000 and Bandwidth=160, with 160 vectors, in 3.37 and 9.18 seconds, respectively. This is a substantial savings over the native Lapack-MKL solver, which takes 3.8 and 20.1 seconds, respectively, to perform the same operations. As a result, the appearance of superlinear scalability in the SPIKE-OpenMP solver is illusionary.

For this problem size, the solve stage takes up the majority of the running time. However, there is still a significant contribution from the factorization stage. The combined scalability, for the factorization and solve stages, has been shown in 5.12. From this measurement, it is apparent that a slight price is paid in the factorization stage by the SPIKE-OpenMP implementation. This is the result of using the solve tuned factorization ratios. Despite this, the SPIKE-OpenMP implementation shows strong performance across all numbers of threads.

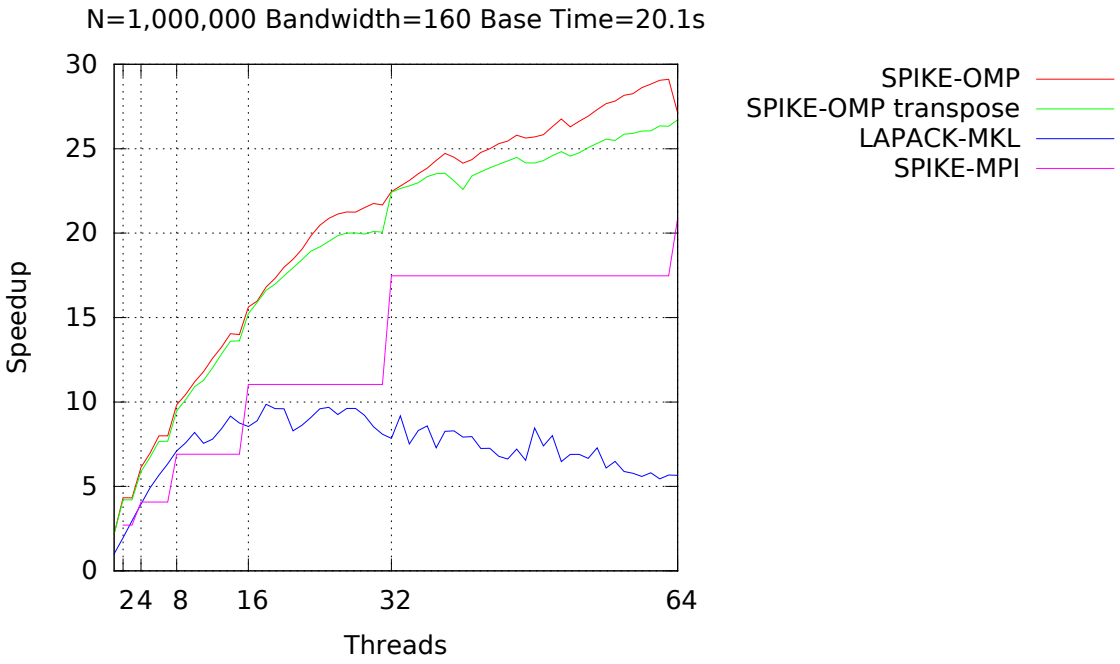


Figure 5.11. Comparison of solve stage scalability for bandwidth 160, with matrix size 1M, and 160 vectors, using solve tuned ratios

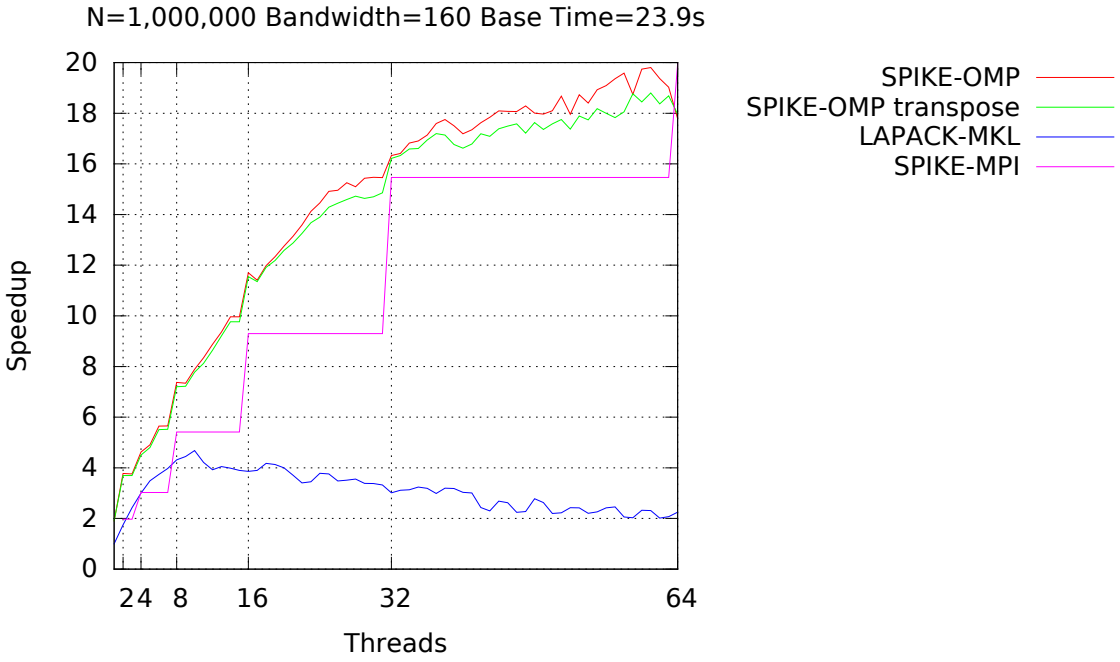


Figure 5.12. Comparison of combined scalability for bandwidth 160, with matrix size 1M, and 160 vectors, using solve tuned ratios

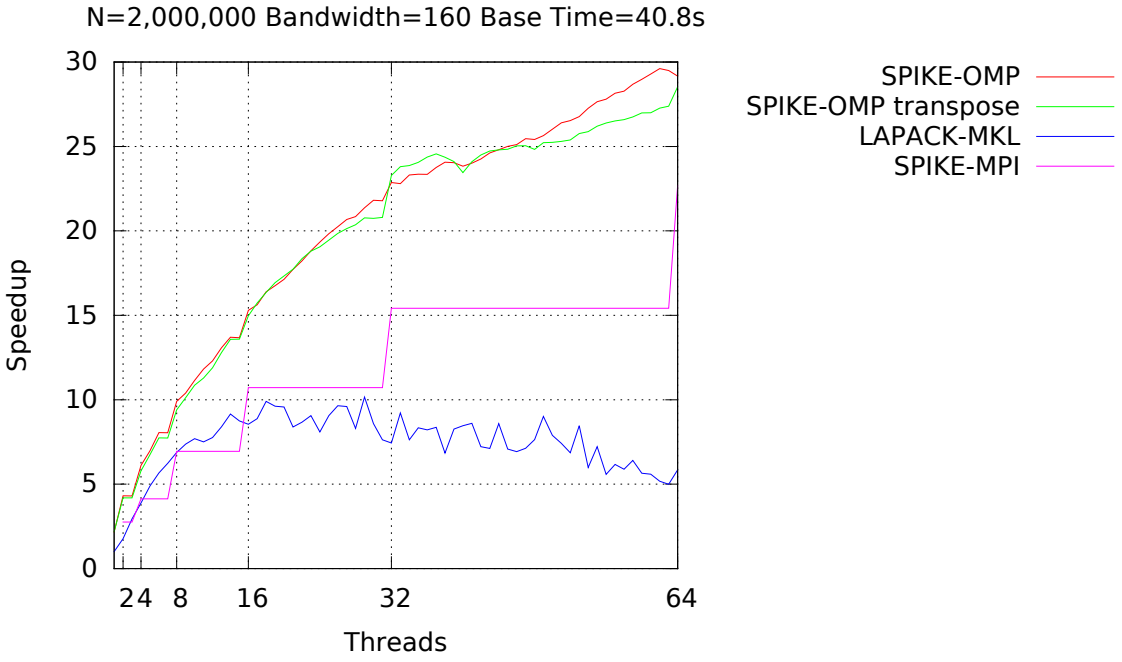


Figure 5.13. Comparison of solve stage scalability for bandwidth 160, with matrix size 2M, and 160 vectors, using solve tuned ratios

Finally, in figures 5.13 and 5.14 we see effect of doubling the matrix size. The scalability in these measurements is qualitatively the same as in the previous set, with only a slight increase in the advantage gained by SPIKE-OpenMP. Additionally, all cases the transpose solve appears to closely mimic the performance of the non-transpose solve.

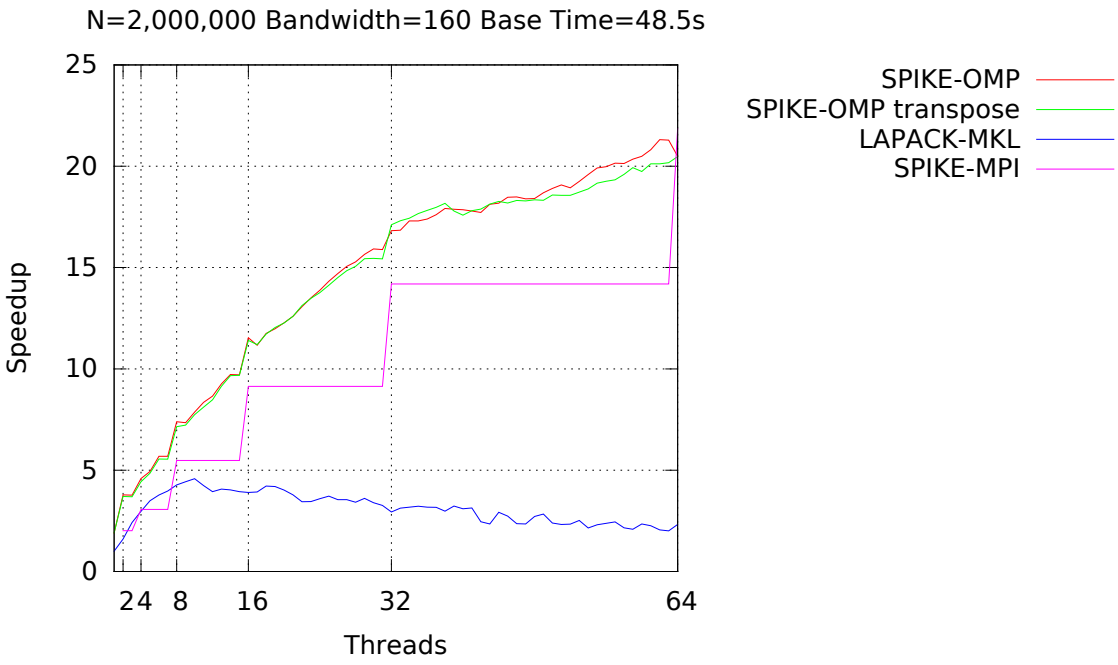


Figure 5.14. Comparison of combined scalability for bandwidth 160, with matrix size 2M, and 160 vectors, using solve tuned ratios

CHAPTER 6

CONCLUSION

Significant progress has been made in the direction of an easy to use, high performance implementation of the SPIKE algorithm. The ability to solve transpose problems has been shown, which brings SPIKE to feature parity with the LAPACK banded solve. With these enhancements SPIKE may soon have the capability to become a drop-in replacement for the standard LAPACK solver. Additionally, a more flexible threading strategy has been developed. Combined, these features will allow for greatly improved utilization of computational resources, with little effort on the part of the user.

BIBLIOGRAPHY

[1] Bull, M. J., and Ball, C. Point-to-point synchronisation on shared memory architectures.

[2] Chen, S. C., Kuck, D. J., and Sameh, A. H. Practical parallel band triangular system solvers. *ACM Trans. Math. Softw. 4*, 3 (Sept. 1978), 270–277.

[3] Dongarra, Jack J., and Sameh, Ahmed H. On some parallel banded system solvers. *Parallel Comput. 1*, 3-4 (Dec. 1984), 223–235.

[4] Lawrie, D H., and Sameh, A H. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw. 10*, 2 (May 1984), 185–195.

[5] Mendiratta, Karan, and Polizzi, Eric. A threaded "spike" algorithm for solving general banded systems. *Parallel Computing 37*, 12 (2011), 733 – 741. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[6] Polizzi, Eric, and Sameh, Ahmed. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids 36*, 1 (2007), 113 – 120. Challenges and Advances in Flow Simulation and Modeling.

[7] Sameh, A. H., and Kuck, D. J. On stable parallel linear system solvers. *J. ACM 25*, 1 (Jan. 1978), 81–91.