PARALLEL BLOCK LANCZOS FOR

SOLVING LARGE BINARY

SYSTEMS

by

MICHAEL PETERSON, B.S.

A THESIS

IN

MATHEMATICS

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Christopher Monico
Chairperson of the Committee

Philip Smith

Edward Allen

Accepted

John Borrelli
Dean of the Graduate School

August, 2006

TABLE OF CONTENTS

ABSTRACT iii

CHAPTER

I. INTRODUCTION 1

1.1 Background and Motivation 1

1.2 Notations 3

II. EXISTING METHODS 4

2.1 Standard Lanczos over $\mathbb{R}$ 4

2.2 Montgomery's Adaptation of Lanczos over $\mathbb{F}_2$ 6

2.3 Coppersmith's Adaptation of Lanczos over $\mathbb{F}_2$ 7

III. ALTERNATE METHOD 8

3.1 Gram-Schmidt Lanczos over $\mathbb{R}$ 8

3.2 Application over $\mathbb{F}_2$ 10

3.3 Constructing the Subspaces 15

IV. THE ALGORITHM 19

4.1 Summary of the Algorithm 19

4.2 Simplifying Expensive Computations 20

4.3 Our Adaptation of Lanczos 22

4.4 Proof of the Algorithm 24

4.5 Cost of Lanczos 27

V. PARALLELIZATION 28

5.1 Matrix Storage 28

5.2 Operations M and M^{T} 30

5.3 Two Struct Types 32

5.4 Initialization Code 33

5.5 Master Function 33

5.6 Slave Functions 34

VI. EFFICIENT IMPLEMENTATION 35
6.1 Comparing $M^{\mathsf{T}}M$ with MM^{T} 35
6.1.1 When A is set to be $M^{\mathsf{T}}M$ 35
6.1.2 When A is set to be MM^{T} 36
6.2 Computation Tricks 38
VII. CONCLUSION 41
BIBLIOGRAPHY 42

ABSTRACT

The Lanczos algorithm is very useful in solving a large, sparse linear system $A\boldsymbol{x} = \boldsymbol{y}$ and then finding vectors in the kernel of A. Such a system may arise naturally through number factoring algorithms for example. In this thesis, we present a variant of the binary Lanczos algorithm that directly finds vectors in the kernel of A without first solving the system. The number factoring algorithms ultimately require us to find such vectors. Our adaptation combines ideas of Peter Montgomery and Don Coppersmith. We also discuss implementation issues, including parallelization.

CHAPTER 1
INTRODUCTION

Combining ideas of Peter Montgomery and Don Coppersmith, we wish to develop a variant of the binary Lanczos algorithm. In Chapter II of this thesis, we describe briefly the existing methods. In Chapter III, we will describe a geometrically motivated version of the Lanczos algorithm over $\mathbb{F}_2$ which will be presented precisely in the fourth chapter. Finally, we discuss in Chapters V and VI implementation and efficiency issues followed by a short summary in Chapter VII.

1.1 Background and Motivation

The goal of most Lanczos algorithms is to solve a large system of equations [9]. Such a system can be represented as $A\boldsymbol{x} = \boldsymbol{y}$, where A is a large symmetric $n \times n$ matrix. However, if the system is represented by $B\boldsymbol{x} = \boldsymbol{y}$, where B is a non-symmetric matrix, we must somehow obtain a symmetric matrix since this is required by Lanczos. The standard procedure is to set $A = B^{\mathsf{T}}B$. This construction gives us a symmetric matrix A. However, it may be preferable in a parallel environment to use $A = BB^{\mathsf{T}}$, a possibility we will explore later in the thesis.

Large binary linear systems of this type naturally arise from several different situations. One common way is in factoring numbers using the quadratic sieve [3] or the general number field sieve (GNFS) [11, 14]. These algorithms produce a sparse linear system with n on the order of 10^6 or 10^7 over $\mathbb{F}_2$ which can be represented as a large, mostly sparse matrix. It follows a very predictable pattern in terms of where sparse rows and dense rows occur. This will be discussed in detail later in the parallelization section.

Another place that large binary systems commonly appear is in the linearization of nonlinear systems over $\mathbb{F}_2$ [1, 15]. To attempt to solve a nonlinear system over $\mathbb{F}_2$, an under-determined multivariate system of linear equations is formed by introducing

new variables to replace nonlinear terms. More equations are obviously needed to be able to find a solution to the system. After attaining these equations by various methods, a large matrix A can then represent the system. We now wish to solve $A\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$, which is an ideal candidate for an implementation of the Lanczos algorithm.

If A is small to moderate in size and invertible, one may compute A^{-1} explicitly to solve $A\boldsymbol{x} = \boldsymbol{y}$. Even if A is non-invertible, Gaussian elimination can be used to solve the system, if such solution exists. If A is sparse, we can apply structured Gaussian elimination [2, 8] to change A into a dense matrix with one-third as many rows and columns.

However, Gaussian elimination (or structured Gaussian elimination) is a poor choice in some situations for two reasons: memory and runtime. For example, with $n = 500,000$, a matrix A having .1% nonzero entries would require about 1 GB of storage if we only store addresses of nonzero entries. For an arbitrary matrix, it would take about 32 GB to store the entire matrix entry-by-entry. For an invertible $n \times n$ matrix A, Gaussian elimination takes about $\frac{n^3}{3}$ additions and the same amount of multiplications. The matrix A, with $n = 500,000$, would require about 8.3 $\times 10^{16}$ total operations. For a large enough matrix, such as this one, Gaussian elimination is infeasible because there are too many operations to perform.

Thankfully, Gaussian elimination is not the only way to solve $A\boldsymbol{x} = \boldsymbol{y}$. The Lanczos algorithm is also capable of finding the vectors $\boldsymbol{x}$ that solve this, and traditionally finds such vectors. The methods by Montgomery and Coppersmith solve $A\boldsymbol{x} = \boldsymbol{y}$ for vectors $\boldsymbol{x}$. They let $\boldsymbol{y}_1$ be one such vector. Then $A\boldsymbol{x} = \boldsymbol{y} = A\boldsymbol{y}_1$, from which we see that $A(\boldsymbol{x} - \boldsymbol{y}_1) = 0$. They can find vectors in $\ker(A)$ in this fashion. On the other hand, we do not attempt to solve $A\boldsymbol{x} = \boldsymbol{y}$. Our adaptation directly finds vectors in the kernel of A, i.e.

$$A\boldsymbol{x} = 0.$$

It is likely that our method of finding vectors in the kernel may be modified to also

solve the linear system problem of $A\boldsymbol{x} = \boldsymbol{y}$, although it is not explored further in this thesis.

Parallelization may allow us to solve much larger problems as we can divide the work among many machines. The structure of the Block Lanczos algorithm suggests an efficient parallel implementation. If parallelization allows larger problems to be solved, then one specific application would be the factorization of larger integers. This has close ties to cryptography and the security of RSA keys composed of products of large primes. This algorithm with parallel implementation should facilitate the factorization of such integers.

1.2 Notations

The following is a short list of notations used through the remainder of the thesis.

- A will denote a symmetric $n \times n$ matrix over a field K.
- I_k denotes the $k \times k$ identity matrix.
- If V is a $k \times n$ matrix, then $\langle V \rangle$ denotes the subspace of K^k generated by the column vectors of V, i.e. $\langle V \rangle = \text{Colsp}(V)$.
- $\mathcal{W}$ is a subspace of K^n.
- Two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are defined to be orthogonal if $\boldsymbol{u}^{\mathsf{T}}\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{v} = 0$.
- Two matrices U and V are defined to be orthogonal if each vector $\boldsymbol{u}$ of U is orthogonal to each vector $\boldsymbol{v}$ of V.
- m represents the index of the last non-zero vector. This means that $m + 1$ will be the index of the first zero vector.

CHAPTER 2
EXISTING METHODS

Peter Montgomery [12] and Don Coppersmith [4] have each given versions of the Lanczos algorithm over $\mathbb{F}_2$ with their own modifications and improvements. Our eventual goal is to make a slight improvement over these versions. The following is a broad overview of the widely-used Lanczos method over $\mathbb{R}$ and adaptations over $\mathbb{F}_2$.

2.1 Standard Lanczos over $\mathbb{R}$

Montgomery gives a description [12] for standard Lanczos over $\mathbb{R}$ similar to the following. Let A be a symmetric $n \times n$ matrix over $\mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}^n$. To solve $A\boldsymbol{x} = \boldsymbol{y}$, the Lanczos algorithm computes the following sequence of vectors:

$$\begin{aligned} \boldsymbol{w}_0 &= \boldsymbol{y}, \\ \boldsymbol{w}_1 &= A\boldsymbol{w}_0 - c_{1,0}\boldsymbol{w}_0, \\ \boldsymbol{w}_i &= A\boldsymbol{w}_{i-1} - \sum_{j=0}^{i-1} c_{i,j}\boldsymbol{w}_j, \qquad (2.1) \\ \text{where} \quad c_{ij} &= \frac{(A\boldsymbol{w}_j)^\top (A\boldsymbol{w}_{i-1})}{\boldsymbol{w}_j^\top A\boldsymbol{w}_j}. \end{aligned}$$

$$(2.2)$$

The $\boldsymbol{w}_i$'s are computed until some $\boldsymbol{w}_i = 0$. Let $m+1$ denote this first i. For $i < j$, by inducting on j and using the symmetry of A, the reader can verify that

$$\boldsymbol{w}_j^\top A\boldsymbol{w}_i = 0 \text{ for } i \neq j. \qquad (2.3)$$

The proof follows similarly to the end of the proof of Proposition 3.2.3 given later in Chapter 3. Now we will show that the computation of vectors in the sequence may be simplified greatly.

Lemma 2.1.1 *For the sequence of w_j's above and $i < j - 2$, $c_{ij} = 0$.*

Proof. From Equations 2.1 and 2.3,

$$\begin{aligned}(A\boldsymbol{w}_j)^\mathsf{T}(A\boldsymbol{w}_{i-1}) &= \left(\boldsymbol{w}_{j+1} + \sum_{k=0}^{j} c_{j+1,k}\boldsymbol{w}_k\right)^\mathsf{T} A\boldsymbol{w}_{i-1} \\ &= \boldsymbol{w}_{j+1}^\mathsf{T} A\boldsymbol{w}_{i-1} + \sum_{k=0}^{j}(c_{j+1,k}\boldsymbol{w}_k)^\mathsf{T} A\boldsymbol{w}_{i-1} = 0.\end{aligned}$$

■

Hence, Equation 2.1 simplifies to

$$\boldsymbol{w}_i = A\boldsymbol{w}_i - c_{i,i-1}\boldsymbol{w}_{i-1} - c_{i,i-2}\boldsymbol{w}_{i-2} \text{ for } i \geq 2. \tag{2.4}$$

While computing the $\boldsymbol{w}_i$'s, we simultaneously compute

$$\boldsymbol{x} = \sum_{j=0}^{m-1} \frac{\boldsymbol{w}_j^\mathsf{T}\boldsymbol{y}}{\boldsymbol{w}_j^\mathsf{T} A\boldsymbol{w}_j}\boldsymbol{w}_j.$$

Now, by Equation 2.1,

$$A\boldsymbol{x} - \boldsymbol{y} \in \langle A\boldsymbol{w}_0, A\boldsymbol{w}_1, \ldots, A\boldsymbol{w}_m, \boldsymbol{y}\rangle \subseteq \langle \boldsymbol{w}_0, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\rangle. \tag{2.5}$$

By construction, for $0 \leq j \leq m-1$, $\boldsymbol{w}_j^\mathsf{T} A\boldsymbol{x} = \boldsymbol{w}_j^\mathsf{T}\boldsymbol{y}$. Hence,

$$(A\boldsymbol{x} - y)^\mathsf{T}(A\boldsymbol{x} - y) = 0 \Rightarrow A\boldsymbol{x} = y.$$

We can also see that $\text{Proj}(A\boldsymbol{x} - \boldsymbol{y}, \boldsymbol{w}_j) = 0$, since

$$\begin{aligned}\text{Proj}(A\boldsymbol{x} - \boldsymbol{y}, \boldsymbol{w}_j) &= \boldsymbol{w}_j(\boldsymbol{w}_j^\mathsf{T}\boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\mathsf{T}(A\boldsymbol{x} - \boldsymbol{y}) \\ &= \boldsymbol{w}_j(\boldsymbol{w}_j^\mathsf{T}\boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\mathsf{T}(A\sum_{j=0}^{m-1}\frac{\boldsymbol{w}_j^\mathsf{T}\boldsymbol{y}}{\boldsymbol{w}_j^\mathsf{T} A\boldsymbol{w}_j}\boldsymbol{w}_j - \boldsymbol{y}) \\ &= 0 - \boldsymbol{w}_j(\boldsymbol{w}_j^\mathsf{T}\boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\mathsf{T}\boldsymbol{y} \\ &= 0.\end{aligned}$$

Most authors work around avoiding the problem vectors by embedding $\mathbb{F}_2$ into another field $\mathbb{F}_{2^n}$. Now, instead of a given vector having a $\frac{1}{2}$ chance of being self-orthogonal, it now has a one in 2^n chance. The drawback to this embedding is that computations become much more complex in the larger field. We now look at specifics of Montgomery's and Coppersmith's adaptations over $\mathbb{F}_2$.

2.2 Montgomery's Adaptation of Lanczos over $\mathbb{F}_2$

For a symmetric $n \times n$ matrix A over a field K, let $\boldsymbol{w}_j$ be as in the previous section, with $\boldsymbol{w}_i \neq 0$ for $0 \leq i < m$ and $\boldsymbol{w}_{m+1} = 0$. When $K = \mathbb{R}$, the vectors from 2.4 satisfy the following:

$$\begin{aligned} &\boldsymbol{w}_i^T A \boldsymbol{w}_i \neq 0 \qquad && (0 \leq i < m), \\ &\boldsymbol{w}_j^T A \boldsymbol{w}_i = 0 \qquad && (i \neq j), \\ &A\mathcal{W} \subseteq \mathcal{W}, \quad && \text{where } \mathcal{W} = \langle \boldsymbol{w}_0, \boldsymbol{w}_1, \cdots, \boldsymbol{w}_m \rangle. \end{aligned} \tag{2.6}$$

Montgomery defines two vectors to be A-orthogonal if the second condition of 2.6 is met. We may generalize these three conditions to replace vectors $\boldsymbol{w}_i$ with sequences of subspaces. The Block Lanczos algorithm [7] modifies 2.6 to produce a pairwise A-orthogonal sequence of subspaces $\{\mathcal{W}_i\}_{i=0}^m$ of K^n. Now, instead of the requirement that no vector can be A-orthogonal to itself, no vector may be A-orthogonal to all of $\mathcal{W}_i$.

Montgomery defines a subspace $\mathcal{W}$ of $\mathbb{R}^n$ as A-*invertible* if has a basis of column vectors W such that $W^{\mathsf{T}}AW$ is invertible [12]. He constructs a sequence of these $\mathcal{W}$ subspaces such that:

$$\begin{aligned} &\mathcal{W}_i \qquad && \text{is } A\text{-invertible}, \\ &\mathcal{W}_j^{\mathsf{T}} A \mathcal{W}_i = 0 \qquad && (i \neq j)\ , \\ &A\mathcal{W} \subseteq \mathcal{W}, \quad && \text{where } \mathcal{W} = \mathcal{W}_0 + \mathcal{W}_1 + \cdots + \mathcal{W}_m, \end{aligned} \tag{2.7}$$

in which, $\mathcal{W}_k$ represents the subspace of K^n spanned by all of the columns of W_k.

Eventually, with very high probability there will be a $\mathcal{W}_i$ that is not A-invertible, in other words, $\mathcal{W}_i^{\mathsf{T}} A \mathcal{W}_i = 0$. The problem with this is that it is necessary to divide by $\mathcal{W}_i^{\mathsf{T}} A \mathcal{W}_i$ very often in computing projections for the algorithm. Peter Montgomery modifies the Lanczos algorithm to produce a sequence of orthogonal A-invertible subspaces of $(\mathbb{F}_2)^n$. Each subspace has dimension close to N, the computer word size. He applies A to N binary vectors at once, using the machine's bitwise operators. Comparing with the work required to apply A to one vector, in this way, we will get

$N-1$ applications for free (at the cost of just the one). Our efficiency increases N-fold. In the process, he corrects a subspace $\mathcal{W}_i$ which is not A-invertible by restricting to a subspace of $\mathcal{W}_i$ which does satisfy the conditions in 2.7 (the complement of this subspace will then be included in $\mathcal{W}_{i+1}$).

2.3 Coppersmith's Adaptation of Lanczos over $\mathbb{F}_2$

Coppersmith has more of a geometric motivation than Montgomery. His sequence of vectors are pairwise orthogonal, instead of being pairwise A-orthogonal. He appeals to a solution similar to the *look-ahead Lanczos method* [13], which allows work in $\mathbb{F}_2$. Previous authors have embedded $\mathbb{F}_2 \hookrightarrow \mathbb{F}_{2^n}$ to work around the problem of self-orthogonal vectors. This method computes the subspace $\mathcal{W}_i$ by using $A\mathcal{W}_{i-2}, \mathcal{W}_{i-1}, \mathcal{W}_{i-2}, \ldots, \mathcal{W}_{i-k}$ where k is associated to the number of self-orthogonal rows.

There are a few differences between the two authors in correcting subspaces which do not satisfy the conditions in 2.7. Coppersmith creates a sequence of subspaces $\mathcal{R}_i$ to hold any vectors that break the orthogonality condition of each $\mathcal{W}_i$. After separating off these vectors at each step, the remaining vectors will have all of the necessary properties. These vectors that he puts into $\mathcal{R}_i$ at each step are the problem vectors, which are the vectors in $\mathcal{W}_i$ that are orthogonal to all of $\mathcal{W}_i$. We will use these vectors to help with constructing future $\mathcal{W}_i$'s as they are not only orthogonal to the current $\mathcal{W}_i$, but also to all previous $\mathcal{W}_i$'s by construction. Coppersmith has many more inner products to compute than Montgomery, making his adaptation less efficient.

The main purpose of Coppersmith's paper is to introduce a 'block-version' of the Lanczos algorithm [4], which allows 32 matrix-vector operations for the cost of one. This is commonly referred to as the 'Block Lanczos algorithm' and was developed earlier, but independent of, the block algorithm of Montgomery.

CHAPTER 3
ALTERNATE METHOD

In this chapter, we first present a geometrically motivated Lanczos variant over $\mathbb{R}$. We then discuss and solve the difficulties with adapting this algorithm for use over $\mathbb{F}_2$.

3.1 Gram-Schmidt Lanczos over $\mathbb{R}$

For vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$, we denote the projection of $\boldsymbol{u}$ onto $\boldsymbol{v}$ by

$$\mathrm{Proj}(\boldsymbol{u}; \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\boldsymbol{v} \cdot \boldsymbol{v}} \boldsymbol{v}.$$

The goal is to find vectors in the kernel of some symmetric, $N \times N$ matrix A. We use the Gram-Schmidt process with the initial sequence of vectors $A^1\boldsymbol{y}, A^2\boldsymbol{y}, A^3\boldsymbol{y}, \ldots$ to create the following sequence:

$$\begin{aligned}
\boldsymbol{w}_0 &= A\boldsymbol{y} \\
\boldsymbol{w}_1 &= A\boldsymbol{w}_0 - \mathrm{Proj}(A\boldsymbol{w}_0; \boldsymbol{w}_0) \\
&\vdots \\
\boldsymbol{w}_{j+1} &= A\boldsymbol{w}_j - \sum_{i=0}^{j} \mathrm{Proj}(A\boldsymbol{w}_j; \boldsymbol{w}_i).
\end{aligned}$$

According to our use of the Gram-Schmidt process the collection of vectors $\{\boldsymbol{w}_0, \ldots, \boldsymbol{w}_k\}$ is orthogonal and has the same span as the sequence we started with: $\{A\boldsymbol{y}, A^2\boldsymbol{y}, \ldots, A^k\boldsymbol{y}\}$. It follows that $\boldsymbol{w}_{m+1} = 0$ for some m, so let $m+1$ be the least such positive integer.

Notice also that the sequence spans an A-invariant subspace. That is,

$$\boldsymbol{z} \in \mathrm{span}\{\boldsymbol{w}_0, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\} \Rightarrow A\boldsymbol{z} \in \mathrm{span}\{\boldsymbol{w}_0, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\}. \tag{3.1}$$

The crucial observation is that the computation of $\boldsymbol{w}_{n+1}$ may be greatly simplified. Since $\boldsymbol{w}_i \cdot \boldsymbol{w}_j = 0$ for $i \neq j$, we have for $i \leq n-2$ that

$$\begin{aligned} A\boldsymbol{w}_n \cdot \boldsymbol{w}_i &= \boldsymbol{w}_n^{\mathsf{T}} A^{\mathsf{T}} \boldsymbol{w}_i = \boldsymbol{w}_n^{\mathsf{T}} A \boldsymbol{w}_i \\ &= \boldsymbol{w}_n \cdot \left(\boldsymbol{w}_{i+1} + \sum_{j=0}^{i} \operatorname{Proj}(A\boldsymbol{w}_i; \boldsymbol{w}_j) \right) \\ &= \boldsymbol{w}_n \cdot \boldsymbol{w}_{i+1} + \sum_{j=0}^{i} \boldsymbol{w}_n \cdot \operatorname{Proj}(A\boldsymbol{w}_i; \boldsymbol{w}_j) \\ &= 0 + \sum_{j=0}^{i} \boldsymbol{w}_n \cdot (\alpha_j \boldsymbol{w}_j) \qquad \text{for some } \alpha_j \in \mathbb{R} \\ &= 0. \end{aligned}$$

It now follows that for all $i \leq n-2$, $\operatorname{Proj}(A\boldsymbol{w}_n; \boldsymbol{w}_i) = 0$, so that for each $n > 1$:

$$\boldsymbol{w}_{n+1} = A\boldsymbol{w}_n - \operatorname{Proj}(A\boldsymbol{w}_n; \boldsymbol{w}_n) - \operatorname{Proj}(A\boldsymbol{w}_n; \boldsymbol{w}_{n-1})$$

Finally, we set

$$\boldsymbol{x} = \sum_{i=0}^{m-1} \frac{\boldsymbol{w}_i \cdot \boldsymbol{y}}{\boldsymbol{w}_i \cdot \boldsymbol{w}_i} \boldsymbol{w}_i,$$

Theorem 3.1.1 *Let A be symmetric $N \times N$ and $\boldsymbol{w}_j$'s as before. If $A\boldsymbol{w}_i = \sum \boldsymbol{w}_j \boldsymbol{v}_j$ for some $\boldsymbol{v}_j$, then*

$$A(A\boldsymbol{x} + \boldsymbol{y}) = 0.$$

Proof. First, notice that $A\boldsymbol{x} + \boldsymbol{y}$ is in the sequence of $\boldsymbol{w}_j$'s by 2.5. Also, by 3.1, $A(A\boldsymbol{x} + \boldsymbol{y})$ must be in the sequence of $\boldsymbol{w}_j$'s since it is A-invariant. Thus, we may write

$$A(A\boldsymbol{x} + \boldsymbol{y}) = \sum \boldsymbol{w}_j \boldsymbol{z}_j \qquad \text{for some } \boldsymbol{z}_j$$

Now, notice

$$\begin{aligned}
\mathrm{Proj}(A\boldsymbol{x}+\boldsymbol{y};\boldsymbol{w}_j) &= \sum_{j=0}^{m-1} \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top(A\boldsymbol{x}-\boldsymbol{y}) \\
&= \sum_{j=0}^{m-1} \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \left(\sum_{k=0}^{m-1} \mathrm{Proj}(\boldsymbol{y};\boldsymbol{w}_i)\right) - \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \boldsymbol{y} \\
&= \sum_{j=0}^{m-1} \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top (\boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \boldsymbol{y}) - \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \boldsymbol{y} \\
&= \sum_{j=0}^{m-1} \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \boldsymbol{y} - \boldsymbol{w}_j(\boldsymbol{w}_j^\top \boldsymbol{w}_j)^{-1}\boldsymbol{w}_j^\top \boldsymbol{y} \\
&= 0.
\end{aligned}$$

It is easy to see now that for all j less than m,

$$\begin{aligned}
\mathrm{Proj}(A\boldsymbol{x}+\boldsymbol{y};\boldsymbol{w}_j) &= 0 \\
A\boldsymbol{x}+\boldsymbol{y} &= 0 \\
A\boldsymbol{x}+\boldsymbol{y} &\in \ker(A) \\
A(A\boldsymbol{x}+\boldsymbol{y}) &= 0.
\end{aligned}$$

■

3.2 Application over $\mathbb{F}_2$

We would like to modify the previous algorithm to work over the field $\mathbb{F}_2$. The obvious problem we encounter is that some vectors may be self-orthogonal (exactly half of the vectors in $(\mathbb{F}_2)^N$). This problem is detrimental unless dealt with since we find it necessary to divide by $\boldsymbol{w}_j^\top \boldsymbol{w}_j$ *many* times. Over $\mathbb{F}_2$, $\mathcal{W}$ has an orthogonal complement if and only if there does not exist an $\boldsymbol{x} \in \mathcal{W}$ such that

$$\boldsymbol{x} \cdot \boldsymbol{w} = \boldsymbol{0}, \text{ for all } \boldsymbol{w} \in \mathcal{W}.$$

If such $\boldsymbol{x}$ exists, then it would have to be in the orthogonal complement, since that consists of every vector that is orthogonal to each vector in $\mathcal{W}$. This leads us to a

contradiction because $\boldsymbol{x}$ cannot be in both $\mathcal{W}$ and the complement of $\mathcal{W}$. Obviously, since self-orthogonal vectors exist in $\mathbb{F}_2$, an orthogonal complement does not always exist for a given collection of vectors. A randomly chosen subspace has an orthogonal complement with probability of about $\frac{1}{2}$. An orthogonal complement is precisely what is needed in order to be able to project onto a subspace in a well-defined way. The ability to project onto the constructed sequence of subspaces is an essential ingredient of the Lanczos algorithm. The following is a simple example of a subspace with no orthogonal complement.

Example 3.2.1 *The subspace $\mathcal{A} \subset \mathbb{F}_2^3$ spanned by the vectors $(1,0,0)^\mathsf{T}$ and $(0,1,1)^\mathsf{T}$ does not have an orthogonal complement.*

The other two vectors in the subspace $\mathcal{A}$ are $(1,1,1)^\mathsf{T}$ and $(0,0,0)^\mathsf{T}$. There are $2^3 = 8$ distinct vectors in $\mathbb{F}_2^3$. In this case, we can exhaust the possibility of an orthogonal complement by brute force. None of the remaining vectors $((1,0,1)^\mathsf{T}, (0,0,1)^\mathsf{T}, (0,1,0)^\mathsf{T}, (1,1,0)^\mathsf{T})$ is orthogonal to all of the vectors in $\mathcal{A}$. To see this, we just need to find one vector in $\mathcal{A}$ that yields an inner product of 1 for each remaining vector.

$$\begin{aligned}
(1,0,1)(1,0,0)^\mathsf{T} &= 1 \\
(0,0,1)(1,1,1)^\mathsf{T} &= 1 \\
(0,1,0)(1,1,1)^\mathsf{T} &= 1 \\
(1,1,0)(1,0,0)^\mathsf{T} &= 1
\end{aligned}$$

Hence, the complement of $\mathcal{A}$ is not orthogonal to $\mathcal{A}$. The root problem in this example is that the vector $(0,1,1)^\mathsf{T}$ is self-orthogonal. The following proposition describes (1) how to identify subspaces that have an orthogonal complement and (2) how to project onto these subspaces.

Proposition 3.2.2 *Let $\mathcal{W}$ be a subspace of $\mathbb{F}_2^N$. $\mathcal{W}$ has a basis of column vectors W so that $\mathcal{W} = \mathrm{Colsp}(W)$ with $W^\mathsf{T} W$ invertible if and only if each $\boldsymbol{u} \in \mathbb{F}_2^N$ can*

be uniquely *written as* $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{v}$ *with* $\boldsymbol{w} \in \mathcal{W}$ *and* $\mathcal{W}^\mathsf{T}\boldsymbol{v} = 0$. *Furthermore, this property is independent of the choice of basis for* $\mathcal{W}$.

Proof. Let $\boldsymbol{u} \in \mathbb{F}_2^N$ and set

$$\begin{aligned} \boldsymbol{w} &= W(W^\mathsf{T}W)^{-1}W^\mathsf{T}\boldsymbol{u}, \quad \text{and} \\ \boldsymbol{v} &= \boldsymbol{u} - \boldsymbol{w}. \end{aligned}$$

Then $\boldsymbol{w} \in \mathrm{Colsp}(W) = \mathcal{W}$, $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{w}$ and

$$\begin{aligned} W^\mathsf{T}\boldsymbol{v} &= W^\mathsf{T}\boldsymbol{u} - W^\mathsf{T}\boldsymbol{w} \\ &= W^\mathsf{T}\boldsymbol{u} - W^\mathsf{T}W(W^\mathsf{T}W)^{-1}W^\mathsf{T}\boldsymbol{u} = 0, \end{aligned}$$

so that $\mathcal{W}^\mathsf{T}\boldsymbol{v} = 0$ as desired.

Let $\boldsymbol{w}'$ be another vector in $\mathrm{Colsp}(W)$ so that $W^\mathsf{T}(\boldsymbol{u} - \boldsymbol{w}') = 0$. Then $\boldsymbol{w}' = W\boldsymbol{\alpha}$ for some $\boldsymbol{\alpha} \in \mathbb{F}_2^N$. So we have $0 = W^\mathsf{T}(\boldsymbol{u} - \boldsymbol{w}') = W^\mathsf{T}\boldsymbol{u} - W^\mathsf{T}W\boldsymbol{\alpha}$ whence $W^\mathsf{T}W\boldsymbol{\alpha} = W^\mathsf{T}\boldsymbol{u}$. Since $W^\mathsf{T}W$ is invertible, it follows that $\boldsymbol{\alpha} = (W^\mathsf{T}W)^{-1}W^\mathsf{T}\boldsymbol{u}$. Left multiplying both sides by W, we find that

$$\boldsymbol{w}' = W\boldsymbol{\alpha} = W(W^\mathsf{T}W)^{-1}W^\mathsf{T}\boldsymbol{u} = \boldsymbol{w},$$

giving us the desired uniqueness property. Furthermore, the converse also holds. Finally, note that if U is another basis of column vectors for $\mathcal{W} = \mathrm{Colsp}(W) = \mathrm{Colsp}(U)$, then $U = WA$ for some invertible matrix A. It follows that

$$\begin{aligned} U^\mathsf{T}U &= A^\mathsf{T}W^\mathsf{T}WA \\ &= A^\mathsf{T}(W^\mathsf{T}W)A, \end{aligned}$$

which is invertible, so the result is independent of the choice of basis. ■

The property of a subspace having an orthogonal complement is a necessary and sufficient condition for projection onto the subspace to be possible. If W has an orthogonal complement, then we can project onto W. If $W^{\mathsf{T}}W$ is invertible, we define

$$\mathrm{Proj}(U;W) := W(W^{\mathsf{T}}W)^{-1}W^{\mathsf{T}}U. \tag{3.2}$$

Geometrically speaking, if we can project a vector onto a sequence of orthogonal subspaces, then we may see the parts of the vector that lie in each. These projections are a critical part of the Lanczos algorithm. The following proposition describes the necessary properties of each subspace and forms the basis for our algorithm over $\mathbb{F}_2$.

Proposition 3.2.3 *Let A be any $n\times n$ matrix. Suppose $W_0, W_1, \ldots, W_m$ is a sequence of matrices satisfying all of the following properties:*

1. *W_i is $n \times k_i$.*
2. *$W_i^{\mathsf{T}}W_i$ is invertible for $0 \le i < m$.*
3. *$W_i^{\mathsf{T}}W_j = 0$ for $i \neq j$.*
4. *$\langle W_0, W_1, \ldots, W_m\rangle$ is A-invariant.*

Then $\mathrm{rank}(W_0|W_1|\ldots|W_m) = \sum_{j=0}^{m} k_j$. *Furthermore, if $Y = \sum_{j=0}^{m} W_jU_j$ for some U_j, then $Y = \sum_{j=0}^{m} \mathrm{Proj}(Y;W_j)$. For any Y with $AY \in \langle W_0, W_1, \ldots, W_m\rangle$,*

$$X = \sum_{i=0}^{m} W_i(W_i^{\mathsf{T}}W_i)^{-1}W_i^{\mathsf{T}}Y$$

is a solution to $A(X+Y) = 0$. In other words, $X + Y \in \ker(A)$.

Proof. To see first that the $n \times \sum k_i$ matrix $(W_0|W_1|\ldots|W_m)$ has full rank, notice that any linear dependence on the columns of this matrix can be expressed as

$$W_0C_0 + W_1C_1 + \ldots + W_{m-1}C_{m-1} = 0,$$

for some square C_j matrices of dimension $k_j \times k_j$. It follows from the hypothesis that for each i, we have

$$\begin{aligned} 0 &= W_i^\mathsf{T}(W_0C_0 + W_1C_1 + \ldots + W_mC_m) \\ &= W_i^\mathsf{T}W_iC_i. \end{aligned}$$

Since $W_i^\mathsf{T}W_i$ is invertible, C_i must be zero, which implies that the columns are linearly independent, as desired.

Observe now that if $Y = \sum_{j=0}^{m} W_jU_j$, then for each j

$$\begin{aligned} \text{Proj}(Y; W_j) &= W_j(W_j^\mathsf{T}W_j)^{-1}W_j^\mathsf{T}Y \\ &= \sum_{i=0}^{m} W_j(W_j^\mathsf{T}W_j)^{-1}W_j^\mathsf{T}W_iU_i \\ &= W_j(W_j^\mathsf{T}W_j)^{-1}W_j^\mathsf{T}W_jU_j \\ &= W_jU_j, \end{aligned}$$

so that $\sum \text{Proj}(Y; W_j) = Y$, proving the second statement.

Now, for the third statement, set

$$U = X + Y.$$

We know that X is in $\text{span}\{W_0, \ldots, W_m\}$. Since the span is A-invariant, AX must also be in the span. $AY \in \text{span}\{W_0, \ldots, W_m\}$ by assumption. Therefore, $A(X + Y) = AU \in \text{span}\{W_0, \ldots, W_m\}$. Next, we wish to show that $\text{Proj}(A(X + Y); W_k) = \text{Proj}(AU; W_k) = 0$ for all k. This will lead us to the desired conclusion. We show that $W_k^\mathsf{T}U = 0$.

$$\begin{aligned} W_k^\mathsf{T}U &= \sum_{j=0}^{m-1} W_k^\mathsf{T}W_j(W_j^\mathsf{T}W_j)^{-1}W_j^\mathsf{T}Y - W_k^\mathsf{T}Y \\ &= W_k^\mathsf{T}Y - W_k^\mathsf{T}Y \\ &= 0. \end{aligned}$$

Now, using the symmetry of A, for each $1 \leq k \leq m$,

$$\begin{aligned}
\mathrm{Proj}(AU; W_k) &= W_k(W_k^{\mathsf{T}} W_k)^{-1} W_k^{\mathsf{T}} AU \\
&= W_k(W_k^{\mathsf{T}} W_k)^{-1} (AW_k)^{\mathsf{T}} U \\
&= W_k(W_k^{\mathsf{T}} W_k)^{-1} \left(\sum_{i=0}^{m} W_i V_{i,j} \right)^{\mathsf{T}} U \\
&= W_k(W_k^{\mathsf{T}} W_k)^{-1} \left(\sum_{i=0}^{m} V_{i,j}^{\mathsf{T}} W_i^{\mathsf{T}} \right) U \\
&= W_k(W_k^{\mathsf{T}} W_k)^{-1} \left(\sum_{i=0}^{m} V_{i,j}^{\mathsf{T}} (W_i^{\mathsf{T}} U) \right) \\
&= 0.
\end{aligned}$$

Since $\mathrm{Proj}(AU; W_k) = 0$, it follows from previous argument that $AU = 0$, hence $U = X + Y \in \ker(A)$. ■

3.3 Constructing the Subspaces

The difficulty in adapting Lanczos to $\mathbb{F}_2$ is precisely the problem of constructing a sequence of W_j's satisfying the hypotheses of Proposition 3.2.3. The natural thing to try would be

$$\begin{aligned}
W_0 &= AY, \\
W_{n+1} &= AW_n - \sum_{j=0}^{n} \mathrm{Proj}(AW_n; W_j).
\end{aligned}$$

Notice that for $j \leq n-2$,

$$\begin{aligned}
AW_n \cdot W_i &= W_n^{\mathsf{T}} AW_i \\
&= W_n^{\mathsf{T}} (W_{i+1} + \sum_{j=0}^{i} \mathrm{Proj}(AW_n; W_j) \\
&= W_n^{\mathsf{T}} W_{i+1} + W_n^{\mathsf{T}} \sum_{j=0}^{i} S_j W_j \text{ for some matrix } S_j \\
&= 0.
\end{aligned}$$

This implies that $\text{Proj}(AW_n; W_j) = 0$ for $j \leq n-2$, simplifying the recurrence to two terms. However, even if $W_i^{\mathsf{T}} W_i$ is invertible for $0 \leq i \leq n$, there is no guarantee that $W_{n+1}^{\mathsf{T}} W_{n+1}$ is invertible. Indeed, $W_{n+1}^{\mathsf{T}} W_{n+1}$ is only invertible with about a 50% chance. As a result, $\text{Proj}(-; W_{n+1})$ may not be defined. However, with this recursion as a starting point, we can build a sequence of W_j's satisfying the hypotheses of Proposition 3.2.3.

$W^{\mathsf{T}} W$ is invertible if and only if there does not exist an $\boldsymbol{x} \in W$ such that $\boldsymbol{x} \cdot \boldsymbol{w} = 0$ for all $\boldsymbol{w} \in W$. If $W^{\mathsf{T}} W$ is not invertible, we attempt to find a basis for such $\boldsymbol{x}$ vectors and remove it. The remaining vectors will be linearly independent, making the resulting W satisfy the invertibility condition. If this inner product is invertible, then we can project onto the subspace in a meaningful way. This inner product inverse is required in the definition of projection in 3.2. If $\text{rank}(W_{n+1}^{\mathsf{T}} W_{n+1}) = r$ and $r < n$, we wish to find an $r \times r$ submatrix with rank r. We will see by the following lemma that this submatrix is invertible.

We may find a maximum subspace that we can project onto by finding a maximal submatrix which is invertible. We claim that this can be found by taking a number of linearly independent rows of a given symmetric matrix equal to the rank of the matrix. The following lemma and proof are adapted from Montgomery [12].

Lemma 3.3.1 *Let R be a symmetric $n \times n$ matrix over a field K with rank r. Upon selecting any r linearly independent rows of R, the $r \times r$ submatrix of R with the same column indices is invertible.*

Proof. After simultaneous row and column permutations, we may assume that

$$R = \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline R_{21} & R_{22} \end{array}\right].$$

where R_{11} is the symmetric $n \times n$ matrix claimed to be invertible. Then R_{22} is also symmetric, and $R_{12} = R_{21}^{\mathsf{T}}$. Note that we just need to show that the matrix R_{11} has

rank r (i.e. full rank) to show it is invertible. The first r columns of R are linearly independent and the rest of the columns are dependent on these, so these first r columns must generate all of R. This implies that there must exist some $r \times (n-r)$ matrix T such that

$$\left[\begin{array}{c} R_{11} \\ \hline R_{21} \end{array}\right] T = \left[\begin{array}{c} R_{12} \\ \hline R_{22} \end{array}\right].$$

From this we find that

$$\begin{aligned} R_{12} &= R_{11}T \\ R_{21} &= R_{12}^{\mathsf{T}} = T^{\mathsf{T}} R_{11}^{\mathsf{T}} = T^{\mathsf{T}} R_{11} \\ R_{22} &= R_{21}T = R_{12}^{\mathsf{T}}T = T^{\mathsf{T}} R_{11}^{\mathsf{T}} T = T^{\mathsf{T}} R_{11} T. \end{aligned}$$

Now we can substitute for each component of R and rewrite the matrix:

$$\begin{aligned} R &= \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline R_{21} & R_{22} \end{array}\right] \\ &= \left[\begin{array}{c|c} R_{11} & R_{11}T \\ \hline T^{\mathsf{T}} R_{11} & T^{\mathsf{T}} R_{11} T \end{array}\right] \\ &= \left[\begin{array}{c|c} I_r & 0 \\ \hline T^{\mathsf{T}} & I_{n-r} \end{array}\right] \left[\begin{array}{c|c} R_{11} & 0 \\ \hline 0 & 0 \end{array}\right] \left[\begin{array}{c|c} I_r & T \\ \hline 0 & I_{n-r} \end{array}\right] \end{aligned}$$

Since the first and last matrices of the final line are invertible (lower triangular and upper triangular, respectively), $\text{rank}(R_{11}) = \text{rank}(R)$. Thus, $\text{rank}(R_{11}) = \text{rank}(R) = r$. ■

If the specified subspace is not invertible, we need to find and remove a basis for the "singular part." Once we remove this, the subspace will be invertible, hence allowing us to project onto it.

If $W_0, \ldots, W_n$ have been computed, attempt to compute W_{n+1} first by setting

$$E = AW_n - \sum_{j=0}^{n} \operatorname{Proj}(AW_n; W_j). \tag{3.3}$$

If E does not have full rank, we can immediately recover some vectors in $\ker(A)$. The vectors which we can recover are vectors $\boldsymbol{x} \in E$ such that $A\boldsymbol{x} = 0$. If E has full rank, then we cannot immediately recover any vectors in the kernel of A. According to Lemma 3.3.1, if $\operatorname{rank}(E^{\mathsf{T}}E) = r$, then we can take any r linearly independent rows of $E^{\mathsf{T}}E$ to find the invertible submatrix. If E has full rank but $E^{\mathsf{T}}E$ is not invertible, then let $P = [D_{n+1|E}]$, where D_{n+1} is comprised of columns removed from the previous iteration, and let $T = P^{\mathsf{T}}P$. Perform row operations on T and corresponding column operations on P by finding some matrix U such that UT is in reduced row echelon form. The first columns of PU will satisfy all of the necessary conditions from Lemma 3.2.3, and thus are set to be our desired W_{n+1}. The remaining columns are orthogonal to all columns of W and to all previous W_j. These columns comprise D_{n+2}, which is used in the next iteration.

We show how to simplify the recurrence later in the proof of the algorithm. We will see that it can be simplified to a three term recurrence.

CHAPTER 4
THE ALGORITHM

The following gives an overall description of the algorithm, followed by the algorithm itself, and then the proof that it corrects the problem subspaces that we encounter along the way.

4.1 Summary of the Algorithm

After initialization, we begin the repeated process of the algorithm. This begins with setting a matrix E to be A times the previous W. Half the time, this is a two-term recurrence, and half the time this is a three term recurrence. Next, we look at the concatenation of the columns of the previous W that were a basis for the vectors orthogonal to that entire W with this matrix E. Call this concatenation P and set $T = P^{\mathsf{T}}P$.

We now find some invertible matrix U so that UT is in reduced row echelon form, causing any zero rows to appear at the bottom. Because

$$UTU^{\mathsf{T}} = U(P^{\mathsf{T}}P)U^{\mathsf{T}} = (PU^{\mathsf{T}})^{\mathsf{T}}PU^{\mathsf{T}}$$

is obviously symmetric, if there are t zero rows at the bottom of UT, there are now also t zero columns at the right end of UTU^{T}. Also, note that the t zero rows at the bottom will remain zero rows in UTU^{T} since multiplying on the right by a matrix U^{T} is just taking linear combinations of columns of UT; thus, we are performing simultaneous row and column operations on T.

We may perceive this U matrix as a change of basis since it is finding (1) a basis for the vectors we may project onto (W) and (2) a basis for the vectors we cannot include in this W, i.e. the ones that are orthogonal to each vector in W. We will show that the upper-left submatrix (everything non-zero) is invertible and has rank equal to T. Note that the columns of PU^{T} (corresponding to the linear independent rows of UT) must be linearly independent and that the remaining rows (corresponding to

zero rows of UT) must be dependent on these first rows. The former columns are set to be W and the latter columns are the ones we will concatenate in the next iteration.

Since the linearly independent rows of UT are the same as the linearly independent rows of $UTU^{\top}$, we may just work with UT (only necessary to perform row operations rather than row *and* column operations). A few tricks and clever observations are used to get around having to compute some of the expensive matrix inner products.

4.2 Simplifying Expensive Computations

Now, we will describe a clever observation that removes the necessity of one of the expensive inner products. F_n originally is set to be $W_n^{\top} A W_{n+1}$, which would be one of the inner products we wish to avoid. First, we observe that $E_{n+1}^{\top} W_{n+1} = W_n^{\top} A W_{n+1}$:

$$\begin{aligned} E_{n+1}^{\top} W_{n+1} &= (AW_n + W_n J_n H_n + W_{n-1} J_{n-1} F_{n-1} + W_{n-2} J_{n-2} G_{n-2})^{\top} W_{n+1} \\ &= (AW_n)^{\top} W_{n+1} + (W_n J_n H_n)^{\top} W_{n+1} + (W_{n-1} J_{n-1} F_{n-1})^{\top} W_{n+1} \\ &\quad + (W_{n-2} J_{n-2} G_{n-2})^{\top} W_{n+1} \\ &= (W_n)^{\top} A W_{n+1} + H_n^{\top} J_n^{\top} W_n^{\top} W_{n+1} + F_{n-1}^{\top} J_{n-1}^{\top} W_{n-1}^{\top} W_{n+1} \\ &\quad + G_{n-2}^{\top} J_{n-2}^{\top} W_{n-2}^{\top} W_{n+1} \\ &= (W_n)^{\top} A W_{n+1}. \end{aligned}$$

Later, in the proof of the algorithm, we see that

$$(U_{n+1} T)^{\top} = \left[\begin{array}{c|c} D_{n+1}^{\top} W_{n+1} & D_{n+1}^{\top} D_{n+2} \\ \hline E_{n+1}^{\top} W_{n+1} & E_{n+1}^{\top} D_{n+2} \end{array} \right]. \tag{4.1}$$

Since $F_n = W_n^{\top} A W_{n+1} = E_{n+1}^{\top} W_{n+1}$, we can avoid computing the inner product and simply read F_n directly off of $(U_{n+1} T)^{\top}$; it is the lower-left submatrix.

Another expensive vector inner product is G_n, which is originally set to be $W_{n-2}^{\top} A W_n$. We would like to find a shortcut so that we can obtain G_n from something already computed or something easier to compute (or possibly a combination of both).

Notice first that

$$
\begin{aligned}
W_{n-2}^{\mathsf{T}} A W_n &= (W_n^{\mathsf{T}} A W_{n-2})^{\mathsf{T}} \\
&= (W_n^{\mathsf{T}}(E_{n-1} - \sum_{j=0}^{n-1} \mathrm{Proj}(AW_{n-2}; W_j)))^{\mathsf{T}} \\
&= (W_n^{\mathsf{T}}(E_{n-1} - \sum_{j=0}^{n-1} W_j (W_j^{\mathsf{T}} W_j)^{-1} W_j^{\mathsf{T}} A W_{n-2}))^{\mathsf{T}} \\
&= (W_n^{\mathsf{T}} E_{n-1})^{\mathsf{T}} \\
&= E_{n-1}^{\mathsf{T}} W_n.
\end{aligned}
$$

Also, let $k_{n-1} = \dim(W_{n-1})$, and notice that

$$
\begin{aligned}
0 &= W_n^{\mathsf{T}} W_{n-1} && (4.2) \\
&= W_n^{\mathsf{T}} [D_{n-1} | E_{n-1}] U_{n-1}^{\mathsf{T}} S_{n-1} && (4.3) \\
&= [W_n^{\mathsf{T}} D_{n-1} | W_n^{\mathsf{T}} E_{n-1}] U_{n-1}^{\mathsf{T}} S_{n-1} && (4.4)
\end{aligned}
$$

where S_{n-1} is a matrix that selects the first k_{n-1} columns from the matrix preceding it. Taking the transpose, we see that the first k_{n-1} rows of $U_{n-1} \begin{bmatrix} D_{n-1}^{\mathsf{T}} W_n \\ E_{n-1}^{\mathsf{T}} W_n \end{bmatrix}$ are zero. Since U_{n-1} is invertible, $E_{n-1}^{\mathsf{T}} W_n = 0$. Thus, if $k_{n-1} = 64$, $W_{n-2}^{\mathsf{T}} A W_n = E_{n-1}^{\mathsf{T}} W_n = 0$.

We now proceed to look at the case when k_{n-1} is not 64. Notice

$$
\begin{aligned}
E_{n-1}^{\mathsf{T}} W_n &= (W_n^{\mathsf{T}} E_{n-1})^{\mathsf{T}} \\
&= (W_n^{\mathsf{T}} [D_n | W_{n-1}] (U_{n-1}^{\mathsf{T}})^{-1} S'_{n-2})^{\mathsf{T}} \\
&= ([\ W_n^{\mathsf{T}} D_n |\ 0\ ] (U_{n-1}^{\mathsf{T}})^{-1} S'_{n-2})^{\mathsf{T}} \\
&= (S'_{n-2})^{\mathsf{T}} (U_{n-1})^{-1} \begin{bmatrix} D_n^{\mathsf{T}} W_n \\ 0 \end{bmatrix}.
\end{aligned}
$$

Thus, $W_{n-2}^{\mathsf{T}} A W_n$ is equal to the first k_{n-2} rows of $U_{n-1}^{-1} \begin{bmatrix} D_n^{\mathsf{T}} W_n \\ 0 \end{bmatrix}$.

In this algorithm, we make it a requirement that $\langle D_n \rangle \in \langle W_n \rangle$. The following describes how we will ensure this. Note that we concatenate E_{n+1} to the end of D_{n+1}.

We want to find a U_{n+1} such that

$$U_{n+1}[D_{n+1}|E_{n+1}]^{\mathsf{T}}[D_{n+1}|E_{n+1}] = U_{n+1}\left[\begin{array}{c|c} 0 & D_{n+1}^{\mathsf{T}}E_{n+1} \\ \hline E_{n+1}^{\mathsf{T}}D_{n+1} & E_{n+1}^{\mathsf{T}}E_{n+1} \end{array}\right]$$

is in reduced row echelon form. We must be sure to not zero-out any top rows since they are from D_{n+1} and we *must* be sure to use all of D_{n+1}. We begin by looking for a one in the first column of the above matrix. This one will occur (with high probability) in one of the first several entries of the $E_{n+1}^{\mathsf{T}}D_{n+1}$ submatrix since its entries are fairly random. This ensures us that we will not switch the top row down to the bottom of the matrix to be cancelled out. We will pivot on this first one and continue to the second column to find the first one to be that column's pivot. Since D_{n+1} only consists of a few vectors, we can be sure that they will all be switched into the early-middle section of the matrix (and thus will not be cancelled out). Therefore, we can be sure with high probability that $\langle D_{n+1}\rangle \in \langle W_{n+1}\rangle$.

4.3 Our Adaptation of Lanczos

Algorithm 4.3.1 $\mathbb{F}_2$*-Lanczos Kernel*

Inputs: a square, symmetric matrix A and a matrix Y

Output: matrix X, which is in the kernel of A

1. *Initialize:*

$$\begin{aligned} n &\leftarrow -1 \\ W_{-1} &\leftarrow Y \\ k_{-1} &\leftarrow 64 \end{aligned}$$

All other variables are set to zero.

2. *Compute AW_n and $H_n \leftarrow W_n^{\mathsf{T}}AW_n$. Set*

$$E_{n+1} \leftarrow AW_n + W_nJ_nH_n + W_{n-1}J_{n-1}F_{n-1}.$$

If $k_{n-1} < 64$ *then do* $E_{n+1} \leftarrow E_{n+1} + W_{n-2}J_{n-2}G_{n-1}$.

3. *Compute* $T \leftarrow [D_{n+1}|E_{n+1}]^\mathsf{T}[D_{n+1}|E_{n+1}]$. *Find an invertible* U_{n+1} *so that* $U_{n+1}T$ *is in reduced row echelon form. We must carefully find this* U_{n+1} *since we make it a requirement that all of* D_{n+1} *is used in* W_{n+1} *with high probability. We set* $k_{n+1} = \mathrm{rank}(T)$. *Set* W_{n+1} *to be the first* k_{n+1} *columns of* $[D_{n+1}|E_{n+1}]U_{n+1}^\mathsf{T}$ *and* D_{n+2} *to be the remaining columns (or zero if there are no columns). If* $W_{n+1} = 0$, *then done. Use the fact that*

$$(U_{n+1}T)^\mathsf{T} = \left[\begin{array}{c|c} D_{n+1}^\mathsf{T}W_{n+1} & D_{n+1}^\mathsf{T}D_{n+2} \\ \hline E_{n+1}^\mathsf{T}W_{n+1} & E_{n+1}^\mathsf{T}D_{n+2} \end{array}\right], \tag{4.5}$$

to set $F_n \leftarrow E_{n+1}^\mathsf{T}W_{n+1}$ *as the lower left* $k_n \times k_{n+1}$ *submatrix. If* $k_{n+1} < 64$, *find and save* U_{n+1}^{-1}. *If* $k_n < 64$, *set* $G_n \leftarrow$ *(the first* k_{n-1} *rows of)* $U_n^{-1}\left[\frac{D_{n+1}^\mathsf{T}W_{n+1}}{0}\right]$. *Finally, compute* $U_{n+1}TU_{n+1}^\mathsf{T}$ *and use the fact that*

$$U_{n+1}TU_{n+1}^\mathsf{T} = \left[\begin{array}{c|c} W_{n+1}^\mathsf{T}W_{n+1} & W_{n+1}^\mathsf{T}D_{n+2} \\ \hline D_{n+2}^\mathsf{T}W_{n+1} & D_{n+2}^\mathsf{T}D_{n+2} \end{array}\right], \tag{4.6}$$

to set $J_{n+1} \leftarrow (W_{n+1}^\mathsf{T}W_{n+1})^{-1}$.

4. *Use the fact that* R_n, R_{n-1}, *and* R_{n-2} *are known from previous iterations to compute*

$$\begin{aligned} D_{n+1}^\mathsf{T}Y &= \text{the last } 64 - k_n \text{ rows of } U_n \left[\frac{D_n^\mathsf{T}Y}{E_n^\mathsf{T}Y}\right], \\ E_{n+1}^\mathsf{T}Y &= H_n^\mathsf{T}J_n^\mathsf{T}(R_n) + (F_{n-1}^\mathsf{T}J_{n-1}^\mathsf{T})(R_{n-1}) + (G_{n-2}^\mathsf{T}J_{n-2}^\mathsf{T})(R_{n-2}), \\ R_{n+1} = W_{n+1}^\mathsf{T}Y &= \begin{cases} \text{the first } k_{n+1} \text{ rows of } U_{n+1}\left[\frac{D_{n+1}^\mathsf{T}Y}{E_{n+1}^\mathsf{T}Y}\right], \text{ if } n \geq 0 \\ W_0^\mathsf{T}Y, \text{ if } n = -1. \end{cases} \end{aligned}$$

Set $X \leftarrow X + W_{n+1}J_{n+1}(R_{n+1})$.

5. *Increment* n *and go to Step 2.*

4.4 Proof of the Algorithm

We give a proof of our version of the algorithm to show that the assignments are precise. We also specify why D_{n+1} has a high probability of being completely used in W_{n+1} and use this fact to prove the three term recurrence.

First of all, we show that E from Equation 3.3 may be simplified to the three term recurrence given in the algorithm.

Similar to before, we wish to only subtract off a few projections at each step rather than the sum of all previous projections. The proof follows similarly to what we have already done. Notice that for $j \leq n-2$,

$$\begin{aligned} AW_n \cdot W_i &= W_n^{\mathsf{T}} AW_i \\ &= W_n^{\mathsf{T}} (E_{i+1} + \sum_{j=0}^{i} \text{Proj}(AW_n; W_j) \\ &= W_n^{\mathsf{T}} E_{i+1} + W_n^{\mathsf{T}} \sum_{j=0}^{i} W_j S_j \text{ for some matrix} S_j \\ &= W_n^{\mathsf{T}} E_{n-1}, \end{aligned}$$

which is zero only when $D_{n-1} = 0$, or in other words, $E_{n-1} = W_{n-1}$.

However, for $i \leq n-3$,

$$\begin{aligned} AW_n \cdot W_i &= W_n^{\mathsf{T}} AW_i \\ &= W_n^{\mathsf{T}} (E_{i+1} + \sum_{j=0}^{i} \text{Proj}(AW_n; W_j) \\ &= W_n^{\mathsf{T}} E_{i+1} + W_n^{\mathsf{T}} \sum_{j=0}^{i} W_j V_j \text{ for some matrix } V_j \\ &= W_n^{\mathsf{T}} [W_{i+1} | D_{i+2}] (U^{\mathsf{T}})^{-1} S \text{ where } S \text{ selects the first } k_i \text{ columns.} \end{aligned}$$

Now we need to show that $W_n^{\mathsf{T}} D_{i+2} = 0$. The largest index of D is $n-1$. Because of the requirement that $\langle D_{n-1} \rangle \subset \langle W_{n-1} \rangle$, and since $W_n^{\mathsf{T}} W_{n-1} = 0$, we see that $W_n^{\mathsf{T}} D_{n-1}$ must equal zero. Similarly, this holds true for all $i \leq n-3$ since each D is included in its respective W and W_n is orthogonal to each of these previous W.

However, if we try to increase the maximum of i by one, we cannot say that $W_n^\mathsf{T} D_n = 0$. Thus, we will have a three term recurrence, so

$$E = AW_n - \text{Proj}(AW_n; W_n) - \text{Proj}(AW_n; W_{n-1}) - \text{Proj}(AW_n; W_{n-2}). \tag{4.7}$$

Only half of the iterations will include the third recurring term. The other half will only have two recurring terms. Note that in Step 3 if $W_{n+1} = 0$, then we quit early. This is necessary since later in Step 3 we are required to find $(W_{n+1}^\mathsf{T} W_{n+1})^{-1}$.

To prove Equation 4.5, we notice that

$$\begin{aligned}
(U_{n+1}T)^\mathsf{T} &= (U_{n+1}[D_{n+1}|E_{n+1}]^\mathsf{T}[D_{n+1}|E_{n+1}])^\mathsf{T} \\
&= [D_{n+1}|E_{n+1}]^\mathsf{T}[D_{n+1}|E_{n+1}]U_{n+1}^\mathsf{T} \\
&= [D_{n+1}|E_{n+1}]^\mathsf{T}[W_{n+1}|D_{n+2}] \\
&= \left[\begin{array}{c|c} D_{n+1}^\mathsf{T} W_{n+1} & D_{n+1}^\mathsf{T} D_{n+2} \\ \hline E_{n+1}^\mathsf{T} W_{n+1} & E_{n+1}^\mathsf{T} D_{n+2} \end{array}\right],
\end{aligned}$$

and for proof of Equation 4.6,

$$\begin{aligned}
U_{n+1}TU_{n+1}^\mathsf{T} &= U_{n+1}[D_{n+1}|E_{n+1}]^\mathsf{T}[D_{n+1}|E_{n+1}]U_{n+1} \\
&= ([D_{n+1}|E_{n+1}]U_{n+1}^\mathsf{T})^\mathsf{T}([D_{n+1}|E_{n+1}]U_{n+1}) \\
&= [W_{n+1}|D_{n+2}]^\mathsf{T}[W_{n+1}|D_{n+2}] \\
&= \left[\begin{array}{c|c} W_{n+1}^\mathsf{T} W_{n+1} & W_{n+1}^\mathsf{T} D_{n+2} \\ \hline D_{n+2}^\mathsf{T} W_{n+1} & D_{n+2}^\mathsf{T} D_{n+2} \end{array}\right].
\end{aligned}$$

Proposition 4.4.1 *Suppose the sequence $W_0, W_1, \ldots, W_n$ has the same properties as those in Lemma 3.2.3. Also, suppose $D_n^\mathsf{T}[W_n|D_n] = 0$. Set*

$$R = U\left([D_n|E]^\mathsf{T}[D_n|E]\right)$$

for some invertible matrix U such that R is in reduced row echelon form.

Take W_{n+1} to be the columns of $[D_n|E]U^\mathsf{T}$ corresponding to nonzero rows of R, and take D_{n+1} to be the remaining columns – the ones corresponding to the zero rows of R. Then the following are true.

1. $W_{n+1}^{\mathsf{T}} W_{n+1}$ *is invertible.*

2. $D_n^{\mathsf{T}} W_{n+1} = 0$ *and* $D_n^{\mathsf{T}} D_n = 0$.

Proof. Let $r = \text{rank}(R)$. If we select any r linearly independent rows of R, then the $r \times r$ submatrix of R with the same column indices is invertible by Lemma 3.3.1.

R is in reduced row echelon form, so there must be r non-zero rows at the top of R since $r = \text{rank}(R)$. The remaining rows are completely zero and all appear together at the bottom. Let there be t of these rows. Since U is invertible, we may right-multiply R by U^{T}.

$$\begin{aligned} RU^{\mathsf{T}} &= U\left([D_n|E]^{\mathsf{T}}[D_n|E]\right)U^{\mathsf{T}} = \left([D_n|E]U^{\mathsf{T}}\right)^{\mathsf{T}}\left([D_n|E]U^{\mathsf{T}}\right) \\ &= \left[\begin{array}{c|c} W_{n+1}^{\mathsf{T}} W_{n+1} & W_{n+1}^{\mathsf{T}} D_n \\ \hline D_n^{\mathsf{T}} W_{n+1} & D_n^{\mathsf{T}} D_n \end{array}\right] \end{aligned}$$

Now, we can see that the $r \times r$ invertible submatrix $W_{n+1}^{\mathsf{T}} W_{n+1}$ appears in the upper-left corner.

Right-multiplying R by a square matrix yields some linear combination of the columns of R. Thus, the zero rows of R remain zero rows. The matrix RU^{T} will have t rows of zero at the bottom and t columns of zero together on the right end of the matrix since RU^{T} is symmetric. Thus,

$$\begin{aligned} D_n^{\mathsf{T}} W_{n+1} = (D_n^{\mathsf{T}} W_{n+1})^{\mathsf{T}} &= 0, \\ \text{and } D_n^{\mathsf{T}} D_n &= 0. \end{aligned}$$

■

4.5 Cost of Lanczos

We may find the expected number of iterations of the algorithm for an $N \times N$ matrix A by

$$\begin{aligned}\frac{N}{\text{expected } \dim(W_i)} &= \frac{N}{K - \text{ expected } \dim(D_i)} \\ &= \frac{N}{K - (1(\frac{1}{2}) + 2(\frac{1}{4}) + 4(\frac{1}{8}) + \cdots + k(\frac{1}{2^k}))} \\ &\geq \frac{N}{K - (\sum_{j=1}^{\infty} \frac{j}{2^j})} \\ &= \frac{N}{K-2}.\end{aligned}$$

Then the expected number of iterations is *at most* $\frac{N}{K-2}$.

At each iteration, we have three types of expensive operations. The most expensive type is applying A to a vector, which we have to do only once at each iteration: AW_n. The second most expensive operation is performing a vector inner product, which there are two of: $W_n^{\mathsf{T}} A W_n$ and $[D_n|E_n]^{\mathsf{T}}[D_n|E_n]$. The third most expensive operation is multiplying a vector times a small square matrix. There are 4.5 of these:

1. W_n multiplied by the product of $J_n H_n$,
2. W_{n-1} multiplied by the product of $J_{n-1} F_{n-1}$,
3. W_{n-2} multiplied by the product of $J_{n-2} G_{n-2}$ (occurs half of the time),
4. $[D_{n+1}|E_{n+1}]U_{n+1}^{\mathsf{T}}$,
5. W_{n+1} multiplied by the product of $J_{n+1} R_{n+1}$.

All other computations can be absorbed into the constant.

CHAPTER 5
PARALLELIZATION

Implementation of the Block Lanczos algorithm is an optimal candidate for parallelization. Since the size of the matrix A is so large, the main bottleneck in the algorithm is the calculation of AX. However, this is easily parallelized by assigning a different portion of the matrix to each available node, in such a way that the whole matrix is accounted for. These portions of the matrix are collections of columns which are multiples of 64 for convenience. The final portion of the matrix obviously may not have 64 columns exactly; in this case, we may use many different options such as padding the matrix to give it a complete set of 64 columns. Each slave node then computes its portion of AX and sends its result back to the master, which will combine these partial results together to find AX.

First, we discuss the best strategy for storing the matrix. Then we will describe the parallelization, i.e. initializations and functionality of the master node and slave nodes.

5.1 Matrix Storage

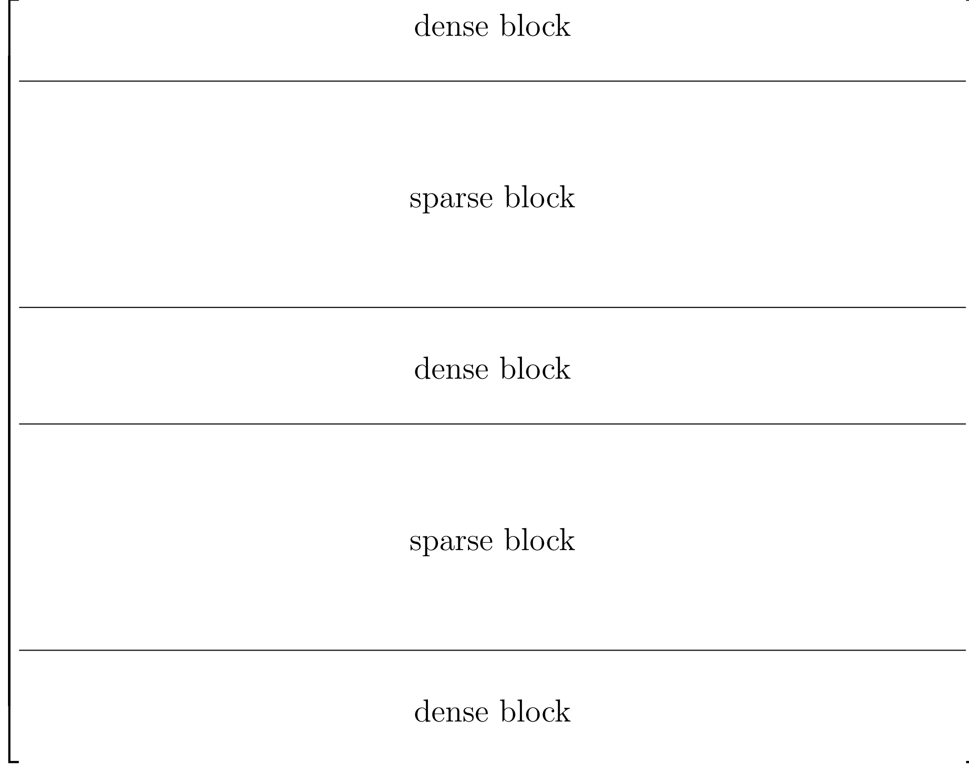
Recall that the typical matrices which we wish to apply to this algorithm are very large and mostly sparse. We can take advantage of the latter and store the matrix in a way that is much more clever than just explicitly storing every entry in the matrix. Storing each entry is already infeasible for a matrix with n of size 500,000, since we would need about 32 GB of RAM to store it. Note that this requirement is much too large to be fulfilled by the RAM of today's typical machine. Also recall that our typical n may be two to twenty times larger than this, increasing the RAM requirement substantially.

First of all, the matrix corresponding to the system of equations that we get from the number field sieve follows a very predictable pattern. The structure of this

particular matrix is described to give the reader an excellent example of the Lanczos algorithm in practice.

The matrix will be stored by collections of rows; each collection may form a dense block or a sparse block. The number field sieve (much like the quadratic sieve) uses three factor bases (rational, algebraic, and quadratic characters) in sieving as part of the process of factoring a large number. Dense rows of the matrix correspond to the smaller primes, and sparse rows correspond to larger primes. These first few rows are called dense since they have relatively many nonzero entries. To find the point where the rows change from dense to sparse, we find the first row with entries having a probability of less than $\frac{1}{64}$ (assuming 64 is the computer word size) of being a one. Once this occurs, it will be more worthwhile to store the locations of these entries rather than storing all the particular entries. This will be the first sparse row, and hence the first row of our sparse block corresponding to this factor base.

The dense sections of the matrix result from the beginning of the algebraic factor base, the beginning of the rational factor base, and all of the quadratic character base. The sparse sections of the matrix result from the remaining parts of the algebraic and rational factor bases. A typical NFS-produced matrix might look like:



where there are far more sparse rows than dense rows.

5.2 Operations M and M^{T}

The GNFS (General Number Field Sieve) has a final step which requires finding some vectors in the kernel of a matrix M of the form described in the previous section. The Block Lanczos algorithm is very efficient at finding them. The bottleneck is computing MX and $M^{\mathsf{T}}Y$ for the fixed matrix M, and many varying vectors $\boldsymbol{x}$, $\boldsymbol{y}$, etc. Our task is to multiply a matrix M of size $r \times c$ by a matrix X of size $r \times 64$ as well as multiplying the transpose of this matrix (i.e. M^{T}) by $\boldsymbol{x}$. To facilitate this, assuming we have N-number of processors, we will partition M and $\boldsymbol{x}$ in the following way:

$$M = \left(\begin{array}{c|c|c|c} \\ M_1 & M_2 & \cdots & M_N \\ \\ \end{array} \right)$$

$$\boldsymbol{x} = \left(\begin{array}{c} \boldsymbol{x}_1 \\ \hline \boldsymbol{x}_2 \\ \hline \vdots \\ \hline \boldsymbol{x}_N \end{array} \right)$$

Each M_j is of size $r \times s$, where s is the number c divided by the total number of nodes. Then each $M_j \boldsymbol{x}_j$ is of size $r \times 64$. Now, observe that:

$$M^\mathsf{T} = \left(\begin{array}{c} M_1^\mathsf{T} \\ \hline M_2^\mathsf{T} \\ \hline \vdots \\ \hline M_N^\mathsf{T} \end{array} \right).$$

Then each M_j^T is of size $s \times r$ and each $M_j^\mathsf{T} X$ is of size $s \times 64$. Thus, our matrix multiplication operation will be the following:

$$
\begin{aligned}
MX &= \left(\begin{array}{c|c|c|c} M_1 & M_2 & \cdots & M_N \end{array}\right)\left(\begin{array}{c} X_1 \\ \hline X_2 \\ \hline \vdots \\ \hline X_N \end{array}\right) \\
&= M_1X_1 \oplus M_2X_2 \oplus \cdots \oplus M_NX_N.
\end{aligned}
$$

Our matrix transpose multiplication operation will be the following:

$$
M^{\mathsf{T}}X = \left(\begin{array}{c} M_1^{\mathsf{T}} \\ \hline M_2^{\mathsf{T}} \\ \hline \vdots \\ \hline M_N^{\mathsf{T}} \end{array}\right) X = \left(\begin{array}{c} M_1^{\mathsf{T}}X \\ \hline M_2^{\mathsf{T}}X \\ \hline \vdots \\ \hline M_N^{\mathsf{T}}X \end{array}\right) \tag{5.1}
$$

5.3 Two Struct Types

The first type we will use to store the data each node has. The second type will store the actual data, which we will pass to each node via MPI's `Send` command.

The first struct type will hold information for the master node. It keeps all information together in one place.

- Number of slave nodes
- A copy of what each slave node has
- Other necessary information that may be needed

The second struct type will hold information specific to a particular slave node.

- Beginning and ending column positions of its data
- Quality/rating/dependability of the particular node
- Other necessary information that may be needed

5.4 Initialization Code

Secondly, the initialization code run from the master node will do the following:

- Retrieve the total number of available processors
- Assign a subset of columns to each node
- Send part of the matrix (a range of columns) to each slave node

For example, assuming we have three slave nodes, we will send M_1, M_2, and M_3 from M to the first, second, and third nodes, respectively:

$$M = \left(\begin{array}{c|c|c} & & \\ M_1 & M_2 & M_3 \\ & & \end{array} \right)$$

5.5 Master Function

The ‘Master’ function will do the following:

- Send to each client:
    1. Function needed for computation (either $M\boldsymbol{x}$, $M^{\top}\boldsymbol{x}$, or $MM^{\top}\boldsymbol{x}$)
    2. The data (portion of the matrix)
- Receive the partial results from the slave nodes
- Reassemble the data

The possible commands (which we will define in the code) are `MULT`, `TMULT`, or `MMT` to compute $M\boldsymbol{x}$, $M^{\top}\boldsymbol{x}$, or $MM^{\top}\boldsymbol{x}$ respectively. Each slave node gets its own portion of the matrix. Reassembling the data at the end is done in one of two ways:

1. Stacking matrices (from `TMULT` command)
2. Adding the matrices component-wise together via XOR (from `MULT` or `MMT` commands)

5.6 Slave Functions

The main loop in the slave functions will do the following:

- Receive the proper command:
    1. Function needed for computation (either $M\boldsymbol{x}$ or $M^{\top}\boldsymbol{x}$)
    2. The data $\boldsymbol{x}$ (entire $\boldsymbol{x}$ or a portion $\boldsymbol{x}_j$, depending on the operation)
- Perform the command
- Send the results back to the Master

In the next chapter, we look at ways to improve the parallelization and tricks to reduce the number of computations.

CHAPTER 6
EFFICIENT IMPLEMENTATION

There are several ways to make the implementation more efficient. One trick lies in how we define our square, symmetric matrix A. Other tricks enable us to greatly decrease our total number of required computations.

6.1 Comparing $M^{\mathsf{T}}M$ with MM^{T}

The Lanczos algorithm requires a square symmetric matrix A. The two ways to find such an A from our given matrix M are

1. $A = M^{\mathsf{T}}M$.
2. $A = MM^{\mathsf{T}}$.

Note that Lanczos never needs the intermediate steps of $M^{\mathsf{T}}\boldsymbol{x}$ or $M\boldsymbol{x}$. Thus, we are *only* interested in finding $A\boldsymbol{x}$. We will look closely at each possibility and explain why MM^{T} is the superior choice.

6.1.1 When A is set to be $M^{\mathsf{T}}M$

Traditionally, A is set to be $M^{\mathsf{T}}M$. Setting A in this way gives a larger square matrix than MM^{T} yields since M has more columns than rows. Thus, we will have more total operations to perform using $M^{\mathsf{T}}M$. Another drawback is that our storage requirements will be greater when dealing with a larger matrix. Also, obviously our final result $A\boldsymbol{x}$ will be larger when starting with $M^{\mathsf{T}}M$ rather than using MM^{T}.

We will have to use the two commands `MULT` and `TMULT` in order to find $A\boldsymbol{x}$. The command `MULT`, which performs $M\boldsymbol{x}$, will be performed first. Assuming we have three nodes, node 1 will receive the vector $\boldsymbol{x}_1$ (corresponding to section M_1 of the matrix). Node 2 will receive $\boldsymbol{x}_2$ (corresponding to section M_2 of the matrix). Node 3 will receive $\boldsymbol{x}_3$ (corresponding to section M_3 of the matrix), where $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ are portions of $\boldsymbol{x}$ in the following way:

$$\boldsymbol{x} = \left(\begin{array}{c} \boldsymbol{x}_1 \\ \hline \boldsymbol{x}_2 \\ \hline \boldsymbol{x}_3 \end{array} \right).$$

Each slave computes its particular result and sends it back to the master. The master then reassembles the data according to Equation 5.1. By passing to each slave node only its required part of $\boldsymbol{x}$, we will avoid sending unnecessary data to each of the slave nodes. Thus, the bandwidth of information we are sending will be less (than if we sent the full vector $\boldsymbol{x}$), which allows for faster results from the slave nodes. In Equation 5.1, the master adds the results together (each living in mod 2) via XOR because this is the method of binary addition. This computes the product $M\boldsymbol{x}$. Next, the master sends $M\boldsymbol{x}$ to each node and the TMULT command. The entire vector $M\boldsymbol{x}$ must be sent to each node since each row of $M^{\top}$ will need to be multiplied by the full vector. Note that node j still has its portion M_j of M, so it can just perform a transpose operation to find $M_j^{\top}$. Thus each node does not need any additional data from M. Each node computes its respective result and sends it back to the master. The master finally is able to compute $M^{\top}M\boldsymbol{x}$ by Equation 5.1, which gives us $A\boldsymbol{x}$.

We now show how to recover vectors in the kernel of M given vectors in the kernel of A. Since $0 = A\boldsymbol{x} = M^{\top}M\boldsymbol{x}$, we can see that either $\boldsymbol{x} \in \ker(M)$ or $M\boldsymbol{x} \in \ker(M^{\top})$. From the latter, we need to make a few observations to find vectors in the kernel of M. Since M has many more columns than rows, $M^{\top}$ has small rank with high probability. $M\boldsymbol{x}$ is mostly zero with the nonzero columns being linearly dependent. We can find linear combinations of the columns to find vectors in $\ker(M)$.

6.1.2 When A is set to be $MM^{\top}$

To find $M^{\top}M$ we must find the outer product, while finding $MM^{\top}$ requires the inner product. This inner product yields a smaller matrix since A has more columns

than rows. We would save a lot of bandwidth when sending the blocks of A to the various nodes. Also, the runtime would be shorter since we require fewer operations at each step. If A is set to be MM^T, we claim that we can cut the number of total communications in half. Multiplying $MM^\mathsf{T}\boldsymbol{x}$, we find that

$$\begin{aligned} MM^\mathsf{T}\boldsymbol{x} &= \left(\begin{array}{c|c|c|c} M_1 & M_2 & \cdots & M_N \end{array}\right)\left(\begin{array}{c} M_1^\mathsf{T} \\ \hline M_2^\mathsf{T} \\ \hline \vdots \\ \hline M_N^\mathsf{T} \end{array}\right)\boldsymbol{x} \\ &= \left(\begin{array}{c|c|c|c} M_1 & M_2 & \cdots & M_N \end{array}\right)\left(\begin{array}{c} M_1^\mathsf{T}\boldsymbol{x} \\ \hline M_2^\mathsf{T}\boldsymbol{x} \\ \hline \vdots \\ \hline M_N^\mathsf{T}\boldsymbol{x} \end{array}\right) \\ &= M_1M_1^\mathsf{T}\boldsymbol{x} \oplus M_2M_2^\mathsf{T}\boldsymbol{x} \oplus \cdots \oplus M_NM_N^\mathsf{T}\boldsymbol{x}. \end{aligned}$$

Thus, each node only needs to receive the full vector $\boldsymbol{x}$ and a portion of the matrix A. For example, node j will only need to receive $\boldsymbol{x}$ and M_j. It will first find M_j^T, then compute $M_jM_j^\mathsf{T}\boldsymbol{x}$ by first applying M_j^T to $\boldsymbol{x}$ and then applying M_j to this result. Note that we do not want to compute and apply $M_jM_j^\mathsf{T}$ since it will be much less sparse than M_j. It will be much more efficient to use M_j^T and M_j at each step because of their sparseness. The slave finally sends the result of $M_jM_j^\mathsf{T}\boldsymbol{x}$ back to the master. The master will collect all of these matrices and add them together to find $A\boldsymbol{x}$.

This method is much more efficient in terms of bandwidth usage, operations, and runtime. In the former method, the master had to send to each node twice and receive from each node twice. In this method, the master only has to send and receive once

to each node. The number of total communications between master and slave (post initialization) is exactly one-half of the alternate method. As a side note, we may save even more communications by using MPI's `Broadcast` function, which can send data to all slave nodes with one command. Also there is no intermediate step for the master of formulating and resending a new vector.

Now we show how to recover vectors in the kernel of M given vectors in the kernel of A. Since $0 = A\boldsymbol{x} = MM^{\mathsf{T}}\boldsymbol{x}$, we can see that either $M^{\mathsf{T}}\boldsymbol{x} \in \ker(M)$ or $\boldsymbol{x} \in \ker(M^{\mathsf{T}})$. The former automatically gives us vectors in $\ker(M)$. For the latter, we need to make a few observations to find vectors in the kernel of M. Since M has many more columns than rows, M^{T} has small rank with high probability. Certain linear combinations of these $\boldsymbol{x}$ vectors will yield vectors in $\ker(M)$.

However, it could happen that MM^{T} is invertible, in which case we could not set $A = MM^{\mathsf{T}}$. This problem requires further investigation. An alternate method that would save the same bandwidth as this description of using MM^{T} would be to use $M^{\mathsf{T}}M$, but partition by rows (instead of columns as described above). It is difficult to ensure an even workload to each node because of the distribution of dense and sparse rows. In general, it appears that setting $A = MM^{\mathsf{T}}$ is the best choice (provided MM^{T} is not invertible).

6.2 Computation Tricks

Coppersmith gives a method ([5] for saving much work in computing the vector inner products. Say we wish to multiply

$$V^{\mathsf{T}}W = \left(\begin{array}{c|c|c|c} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_n \end{array}\right)\left(\begin{array}{c} \boldsymbol{w}_1 \\ \hline \boldsymbol{w}_2 \\ \hline \vdots \\ \hline \boldsymbol{w}_n \end{array}\right) \tag{6.1}$$

where V and W are each of size $n \times k$ (k much smaller than n). We then make the following assignments:

$$\begin{aligned} C_{\boldsymbol{v}_1} &= \boldsymbol{w}_1, \\ C_{\boldsymbol{v}_2} &= \boldsymbol{w}_2, \\ C_{\boldsymbol{v}_i} &= \boldsymbol{w}_i. \end{aligned}$$

If any of the $\boldsymbol{v}_i$'s are the same, we add their respective $\boldsymbol{w}_i$ values together to get the value for that $C_{\boldsymbol{v}_i}$.

Once we have finalized the variables by making a complete pass through V and W, we will find the solution to $V^{\mathsf{T}}W$ with the following. The first row of the solution matrix will be comprised of the sum of all variables with a one in the first position of its subscript. The second row will be found by the summation of all variables with a one in the *second* position of its subscript. This process continues, and the last row is found my summing all of the variables with a one in the last position of its subscript. Note that any variables with subscripts in the given range, for which we did not encounter a value in V^{T} will be zero. This process can be easily illustrated with a simple example.

Example 6.2.1 *The following inner product can be computed in a way different than straight matrix multiplication. This method will provide great speedups in runtime when dealing with inner products such as those we encounter with the Lanczos algorithm.*

Find the inner product of

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

We perform the following assignments/additions in this order:

$$\begin{aligned} C_5 &= C_{101} = [1\ \ 0\ \ 1] \\ C_1 &= C_{001} = [0\ \ 0\ \ 1] \\ C_5 &= C_{101} = C_{101} + [0\ \ 1\ \ 0] = [1\ \ 1\ \ 1] \\ C_6 &= C_{110} = [1\ \ 1\ \ 1] \\ C_2 &= C_{010} = [0\ \ 1\ \ 1] \\ C_6 &= C_{110} = C_{110} + [1\ \ 1\ \ 0] = [0\ \ 0\ \ 1] \\ C_0 &= C_{000} = [0\ \ 1\ \ 1] \end{aligned}$$

The solution matrix P is equal to

$$\begin{bmatrix} C_{100} + C_{101} + C_{110} + C_{111} \\ C_{010} + C_{011} + C_{110} + C_{111} \\ C_{001} + C_{011} + C_{101} + C_{111} \end{bmatrix} = \begin{bmatrix} (000) + (111) + (001) + (000) \\ (011) + (000) + (001) + (000) \\ (001) + (000) + (111) + (000) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

For this example, the reader may verify with matrix multiplication that this is the correct product.

He claims that ideas similar to those in the Fast Fourier Transform may be used to simplify this process. The trick is in finding where to 'save' additions since the same variables are added together in many different places throughout a given computation.

CHAPTER 7
CONCLUSION

We gave a geometrically motivated version of the Lanczos algorithm. Our adaptation shows improvements in computational requirements over Coppersmith. This algorithm directly finds vectors in the kernel of A without first solving the system. It is possible that one may find a way to take these vectors in $\ker(A)$ and find solutions to the system $A\boldsymbol{x} = \boldsymbol{y}$. Our hope is that this thesis will provide easier comprehension of the Lanczos algorithm as well as a description for parallel implementation, which may be used to solve much larger problems.

BIBLIOGRAPHY

[1] Adams, W. P. and Sherali, H. D. "Linearization strategies for a class of zero-one mixed integer programming problems", Operations Research 38 (1990), 217-226.

[2] Bender, E. A. and Canfield, E. R. "An approximate probabilistic model for structured Gaussian elimination", Journal of Algorithms 31 (1999), no. 2, 271–290.

[3] Buhler, J. P., Lenstra Jr., H. W., and Pomerance, C., "Factoring integers with the number field sieve", The Development of the Number Field Sieve (Berlin) (A. K. Lenstra and H. W. Lenstra, Jr., eds.), Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin (1993), 50-94.

[4] Coppersmith, D., "Solving linear equations over GF(2): Block Lanczos algorithm", Linear Algebra Applications 192 (1993), 33–60.

[5] Coppersmith, D., "Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm", Mathematics of Computation 62 (1994), no. 205, 333-350.

[6] Coppersmith, D., Odlyzko, A. M., and Schroeppel, R., "Discrete logarithms in GF(p)", Algorithmica 1 (1986), 1-15.

[7] Cullum, J. K. and Willoughby, R. A., "Lanczos algorithms for large symmetric eigenvalue computations", Vol. I Theory, Birkhauser, Boston, (1985).

[8] Knuth, D. E., "The art of computer programming, volume 2: seminumerical algorithms". Addison-Wesley, Reading. Third edition, (1998).

[9] Lanczos, C., "An iterative method for the solution of the eigenvalue problem of linear differential and integral operators", Journal of Research National Bureau of Standards Sec. B 45 (1950), 255-282.

[10] Lanczos, C., *Applied Analysis*, Prentice-Hall, Englewood Cliffs, NJ (1956).

[11] Lenstra, A. and Lenstra, H. (eds), “The development of the number field sieve”, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin (1993).

[12] Montgomery, P., “A block Lanczos algorithm for finding dependencies over GF (2)”, Advances in Cryptology - Eurocrypt ’95, Lecture Notes in Computer Science 921, (1995).

[13] Parlett, B. N., Taylor, D. R., and Liu, Z. A., “A look-ahead Lanczos algorithm for unsymmetric matrices”, Math. Comp. 44 (1985), 105-124.

[14] Pomerance, C., “The quadratic sieve factoring algorithm”, Advances in Cryptology, Proceedings of EUROCRYPT 84 (New York) (T. Beth, N. Cot, and I. Ingemarsson, eds.), Lecture Notes in Computer Science 209, Springer-Verlag, 169-182.

[15] Van den Bergh, M., “Linearisations of binary and ternary forms”, J. Algebra, 109 (1987), 172-183.

[16] Wiedemann, D. H., “Solving sparse linear equations over finite fields”, IEEE Trans. Inform. Theory 32 (1986), no. 1, 54-62.

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

__________Michael J. Peterson_____________________ __06-29-2006______

Student Signature Date

Disagree (Permission is not granted.)

____ _________________

Student Signature Date