Computing eigenpairs of a matrix corresponding to a specific geometry in the complex plane is an important topic in real time signal processing, pattern recognition, spectral analysis, systems theory, radar, sonar, and geophysics. We have studied the matrix sign and matrix sector function iterations to extract the eigenpairs belonging to various geometries without resorting to computationally expensive eigenanalysis methods. We propose a parallelization of an existing matrix sign function algorithm, which was implemented on a Meiko CS-2 multiprocessor. We obtain a fast and stable algorithm for computing the matrix sector functions using Halley’s generalized iteration formula for solving nonlinear equations. We propose a parallel iterative algorithm to compute the principal $n$th root of a positive definite matrix using Gauss-Legendre integration formula. Furthermore computing functions of square matrices is also an important topic in linear algebra, engineering, and applied mathematics. A parallelization of Parlett’s algorithm for computing arbitrary functions of upper triangular matrices is introduced. We propose a block-recursive and a parallel algorithm for fast and efficient computation of functions of triangular matrices. The parallel complexity and cache efficiency of these algorithms for computers with two levels of memory are also analyzed.
Parallel and High Performance
Matrix Function Computations

by
Bertan Bakkaloğlu

A DISSERTATION
submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Completed February 26, 1996
Commencement June 1996
I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.
ACKNOWLEDGEMENT

I would like to thank Dr. Ç. K. Koç for his patience and support throughout my graduate studies.

I would also thank Dr. L. Flick, Dr. S. L. Lu, Dr. R. Schreier and Dr. M. Shor for being in my committee.

Finally, with utmost respect and gratitude I would like to thank my family for their superior guidance and constant support in my education.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Parallel Matrix Sign Iterations using PVM</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>The Parallel Newton Iteration</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>The Partial Fraction Expansion Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>The Parallel Continued Fraction Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Implementation Results and Conclusions</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Theory and Applications of the Matrix Sector Function</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Definition of Matrix Sector Functions</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Applications of Matrix Sector Functions</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Halley's Method for the Matrix Sector Function</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Perturbation Analysis</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Application Examples</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>Numerical Experiments</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>A Parallel Algorithm for Computing the nth Roots of Positive Definite Matrices</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallel Algorithm for the Matrix nth Root</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Review of the Gauss-Legendre Quadrature Formula</td>
<td>32</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Parallel Iteration</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Error Analysis</td>
<td>37</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (CONTINUED)

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Implementation Results</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>A Parallelization of Parlett's Algorithm for Functions of Triangular Matrices</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Computing Matrix Functions</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Parlett's Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Parallelization of Parlett's Algorithm</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Analysis of Efficiency and Implementation Results</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusions</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>A Divide-and-Conquer Algorithm for Functions of Triangular Matrices</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Derivation of the Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Computational Complexity</td>
<td>63</td>
</tr>
<tr>
<td>6.4</td>
<td>Numerical Experiments</td>
<td>66</td>
</tr>
<tr>
<td>6.5</td>
<td>Cache Performance of the Algorithm</td>
<td>68</td>
</tr>
<tr>
<td>7</td>
<td>A Parallel Algorithm for Functions of Triangular Matrices</td>
<td>70</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>70</td>
</tr>
<tr>
<td>7.2</td>
<td>Derivation of the Algorithm</td>
<td>71</td>
</tr>
<tr>
<td>7.3</td>
<td>The New Parallel Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>7.4</td>
<td>An Example</td>
<td>76</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>79</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (CONTINUED)

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1.</td>
<td>The sector regions and angles for ( n = 2, 3, 4 ).</td>
</tr>
<tr>
<td>3.2.</td>
<td>Basin of attraction for various algorithms.</td>
</tr>
<tr>
<td>3.3.</td>
<td>Total and decomposed system impulse responses.</td>
</tr>
<tr>
<td>5.1.</td>
<td>Data dependency to compute ( f_{36} ).</td>
</tr>
<tr>
<td>5.2.</td>
<td>Experimental efficiencies as a function of the matrix size.</td>
</tr>
<tr>
<td>5.3.</td>
<td>Theoretical efficiencies as a function of number of processors.</td>
</tr>
<tr>
<td>5.4.</td>
<td>Efficiencies as a function of number of processors.</td>
</tr>
<tr>
<td>6.1.</td>
<td>Computation of the square-root of an ( 8 \times 8 ) matrix.</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>The parallel times for the algorithms (in seconds).</td>
<td>10</td>
</tr>
<tr>
<td>2.2.</td>
<td>The number of iterations for the algorithms.</td>
<td>11</td>
</tr>
<tr>
<td>3.1.</td>
<td>The eigenvalues of $A$ and their properties.</td>
<td>24</td>
</tr>
<tr>
<td>3.2.</td>
<td>The eigenvalues of $S_4(A)$ after the convergence.</td>
<td>28</td>
</tr>
<tr>
<td>4.1.</td>
<td>Node processes for the iteration.</td>
<td>36</td>
</tr>
<tr>
<td>4.2.</td>
<td>Number of iteration steps for sequential algorithms.</td>
<td>40</td>
</tr>
<tr>
<td>4.3.</td>
<td>Number of iteration steps for the parallel algorithm.</td>
<td>41</td>
</tr>
<tr>
<td>4.4.</td>
<td>Convergence time for the sequential and parallel algorithms.</td>
<td>42</td>
</tr>
<tr>
<td>5.1.</td>
<td>Parlett’s algorithm for computing functions of triangular matrices.</td>
<td>45</td>
</tr>
<tr>
<td>5.2.</td>
<td>The parallel version of Parlett’s algorithm.</td>
<td>49</td>
</tr>
<tr>
<td>5.3.</td>
<td>The distribution of the elements for $n = 16$ and $P = 4$.</td>
<td>50</td>
</tr>
<tr>
<td>6.1.</td>
<td>Matlab routine for the recursive function evaluation algorithm.</td>
<td>59</td>
</tr>
<tr>
<td>6.2.</td>
<td>Matlab routine to solve the Sylvester’s equation.</td>
<td>60</td>
</tr>
<tr>
<td>6.3.</td>
<td>Non-recursive matrix function evaluation.</td>
<td>62</td>
</tr>
<tr>
<td>6.4.</td>
<td>Timing and speedup values for the algorithms.</td>
<td>65</td>
</tr>
<tr>
<td>6.5.</td>
<td>Error values for some matrix functions and eigenvalue separations.</td>
<td>67</td>
</tr>
<tr>
<td>6.6.</td>
<td>Relative error for the square-root and cube-root functions.</td>
<td>68</td>
</tr>
</tbody>
</table>
Computing the invariant subspace of a matrix corresponding to a specific geometry in the complex plane is an important topic in real time signal processing, systems theory, pattern recognition, spectral analysis, radar, sonar, and geophysics. In the first chapter we propose a parallel method to compute the matrix sign function which can be utilized to compute the positive and negative invariant subspace of a given matrix. The matrix sign function has several applications in system theory and matrix analysis, including solution of algebraic Riccati and matrix Lyapunov equations, system decomposition, model reduction, separation of eigenpairs, condition theory and most recently for the numerical solution of M/G/1 and G/M/1 type Markov chains. In Chapter 2, we propose a generalization of the matrix sign function which can be used to determine the number of eigenvalues of a matrix in a specific sector of the complex plane, and to extract the eigenpairs belonging to this sector without explicitly computing the eigenvalues. It is known that Newton's method, which can be used for computing the matrix sign function, is not globally convergent for the matrix sector function. The only existing algorithm for computing the matrix sector function is based on the continued fraction expansion approximation to the principal $n$th root of an arbitrary complex matrix. We analyze the existing algorithms and introduce a new algorithm, based on Halley's generalized iteration formula for solving nonlinear equations.

Chapter 3 introduces a parallel iterative algorithm to compute the principal $n$th root of a positive definite matrix without prior knowledge of the eigenvalues. A review of the Gauss-Legendre quadrature formula is given. We derive the summation expression for the iterative $n$th root algorithm and analyze the error properties of this
iteration. We give numerical examples and their timings to demonstrate the speedup with respect to the Hoskins-Walton algorithm, which is the fastest sequential algorithm.

Computing functions of square matrices is an important topic in linear algebra, engineering, and applied mathematics. There are several methods for this task: Jordan decomposition, Schur decomposition, and approximation methods, e.g., Taylor expansion and rational Padé approximations. The approximation methods may not be suitable for arbitrary functions, since specific properties of the function are exploited. The Jordan and Schur decomposition methods are more general in the sense that an arbitrary function of a given square matrix can be computed using these algorithms. In Chapter 4 we give a brief introduction to matrix functions and propose a parallelization of Parlett's algorithm for computing functions of triangular matrices. In Chapter 5 we introduce a divide-and-conquer algorithm which has the same order of complexity as the existing one, but has better performance on computers with two levels of memory due to its block structure. The new algorithm requires approximately the same number of arithmetic operations as Parlett's algorithm. However, it has better performance on computers with two levels of memory due to its block structure and thus less memory-cache traffic requirements. The parallelization of this algorithm is proposed in Chapter 6. This is the first algorithm to date requiring polylogarithmic time to compute an arbitrary function of a triangular matrix.
Chapter 2
Parallel Matrix Sign Iterations using PVM

2.1 Introduction

The matrix sign function has several applications in system theory and matrix analysis, including solution of algebraic Riccati and matrix Lyapunov equations [39], system decomposition, model reduction [38, 48, 42], separation of eigenpairs [32], condition theory [35] and most recently for the numerical solution of M/G/1 and G/M/1 type Markov chains [1]. The matrix sign function maps the stable and unstable eigenvalues of a given matrix to $-1$ and $1$, respectively, while preserving the eigenvectors of the original matrix. This property of the matrix sign function is useful for studying the eigenstructures of matrices without explicitly computing the eigenvalues. The sign of a complex scalar $\lambda$ is defined over $\text{Re}(\lambda) \neq 0$ by

$$
\text{sign}(\lambda) = \begin{cases} 
1 & \text{if } \text{Re}(\lambda) > 0 , \\
-1 & \text{if } \text{Re}(\lambda) < 0 .
\end{cases}
$$

This definition can be extended to a matrix $A \in \mathbb{C}^{n \times n}$ whose eigenvalues do not lie on the imaginary axis. Let $M$ take $A$ to its Jordan form $J$ as

$$
A = MJM^{-1} .
$$

(2.1)

Let $J$ be defined as

$$
J = \begin{bmatrix} J_+ & 0 \\
0 & J_- \end{bmatrix} = J_+ \oplus J_- ,
$$

where $J_+ \in \mathbb{C}^{n_1 \times n_1}$ and $J_- \in \mathbb{C}^{n_2 \times n_2}$ are the Jordan blocks with $\text{Re}(\sigma(A)) > 0$ and $\text{Re}(\sigma(A)) < 0$, respectively, and $n = n_1 + n_2$. Applying the sign function to both sides of Equation 2.1 we obtain

$$
\text{sign}(A) = M \text{sign}(J) M^{-1} .
$$
The matrix sign of the Jordan blocks determine the sign of $A$ as follows:

$$\text{sign}(A) = M \begin{bmatrix} \text{sign}(J_+) & 0 \\ 0 & \text{sign}(J_-) \end{bmatrix} M^{-1} = M \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} M^{-1}.$$  \hspace{1cm} (2.2)

Equation 2.2 shows that the Jordan blocks corresponding to positive (negative) eigenvalues are mapped to positive (negative) identity matrices, whose dimensions are the same as the number of positive (negative) eigenvalues. It follows that $S = \text{sign}(A)$ is a diagonalizable matrix which commutes with $A$ and is a square root of the identity, i.e.,

$$S^2 = I \quad \text{and} \quad AS = SA.$$  \hspace{1cm} (2.3)

Equation 2.3 is a quadratic equation in $S$, and can be solved by Newton’s method [47]. Another definition, which is based on integral representation, is given in [47]. It uses the integral formula

$$A^+ = \frac{1}{2\pi j} \int_C (zI - A)^{-1} dz,$$

where $C$ is a simple closed contour in $C^+$, containing the eigenvalues of $A$ with positive real part. Using the equality $A^+ = (\text{sign}(A) + I)/2$, we obtain an integral expression for $\text{sign}(A)$ as follows:

$$\text{sign}(A) = \frac{2A}{\pi} \int_0^\infty (y^2I + A^2)^{-1} dy.$$  

The parallel computation of the matrix sign function has recently received attention in order to deal with large matrices [26, 43, 12]. In the following we give brief descriptions of the previously proposed parallel algorithms along with the new parallel algorithm. The analysis of these parallel algorithms are performed by counting the number of arithmetic operations and communication steps per iteration. The number of iterations needed for the convergence is a function of several factors, e.g., the size and condition of the matrix, certain properties of the algorithm, etc. The implementation results on an 8-processor Meiko CS-2 multiprocessor using the PVM software are summarized in Section 5.
2.2 The Parallel Newton Iteration

This algorithm is a slightly modified version of the symmetric pivoting algorithm of [26]. The original algorithm has been applied to solution of the algebraic Riccati equation, where the iteration is carried out on a matrix pencil with Hamiltonian-like structure. The structure of the pencil has the property that multiplying with an anti-diagonal identity matrix would convert it to a symmetric matrix. In this chapter we have implemented a modified version of this algorithm for finding the matrix sign function of a general nonsymmetric matrix. The algorithm is based on parallel factorization of the iteration matrix \( S_k \in \mathcal{R}^{m \times m} \) at each step of the algorithm. We obtain the inverse of the symmetric matrix \( S_k \) in parallel and compute its determinant \( d_k \) in order to calculate the scaling factor \( \gamma_k \). The algorithm starts with \( S[0] = A \), and proceeds using the iteration

\[
S[k + 1] = \frac{1}{2} \left( \frac{1}{\gamma_k} S[k] + \gamma_k S^{-1}[k] \right). \tag{2.4}
\]

The determinantal scaling factor [36] is given as

\[
\gamma_k = \left| \det S[k] \right|^{1/m}.
\]

The inverse of the iteration matrix \( S[k] \) is computed and the scaling factor \( \gamma_k \) is obtained during the parallel \( LU \) decomposition. The iteration matrix \( S[k] \) is distributed among \( p \) processors in a column-wrapped fashion so that each processor works on an array of \( n \times n/p \) elements. The parallel \( LU \) decomposition requires \( O(n^3/p) \) arithmetic operations and \( O(n) \) communication steps. The scaling and matrix addition requires \( O(n^2/p) \) arithmetic steps. Finally, each processor sends its portion of the iteration matrix to the rest of the processors in order to calculate \( S[k + 1] \). This operation is called a multi-node broadcast operation. The details of the multi-node broadcast operation for various parallel architectures can be found in [7]. We denote the communication time of the multi-node broadcast operation of a single matrix element by \( B \). Since \( n^2/p \) matrix elements are being broadcast, the update operation
requires \( O(n^2B/p) \) communication steps. Thus, a single Newton iteration requires \( O(n^3/p) \) arithmetic operations with a communication penalty of \( O(n^2B/p) \).

### 2.3 The Partial Fraction Expansion Algorithm

The most common method for finding \( \text{sign}(A) \) is Newton's method which is globally convergent for matrices with nonzero eigenvalues. This method can also be extended to globally convergent rational iterations of arbitrary order. Howland [32] derived a closed form formula for a desired degree rational iteration from the following error relation

\[
\frac{s_{k+1} - 1}{s_{k+1} + 1} = \left(\frac{s_k - 1}{s_k + 1}\right)^p
\]

from which we solve for \( s_{k+1} \) as

\[
s_{k+1} = \frac{(s_k + 1)^p + (s_k - 1)^p}{(s_k + 1)^p - (s_k - 1)^p}.
\]

This is the principal Padé iteration of order \( p \) [33]. Recently it is noted in [37] that this rational approximation can be represented as

\[
s_{k+1} = \tanh(p \arctanh s_k),
\]

where

\[
\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.
\]

It is also noted in [37] that the above function for even \( p \) can be expanded in partial fractions as

\[
\tanh(p \arctanh(s)) = \frac{1}{p} \sum_{i=0}^{p-1} \frac{1}{\sin^2(\frac{(2i+1)\pi}{4p}) + \cos^2(\frac{(2i+1)\pi}{4p})s^2}
\]

Using the partial fraction expansion of 2.5, a parallel iteration [43] for the matrix sign function is obtained. The algorithm starts with \( S[0] = A \), and uses the iteration formula

\[
S[k+1] = S[k] \sum_{i=1}^{p} \frac{1}{p} (\alpha_i^2 I + \beta_i^2 S^2[k])^{-1},
\]

(2.6)
where \( p \) is the number of processors and
\[
\alpha_i = \sin \left( \frac{(2i - 1)\pi}{4n} \right) \quad \text{and} \quad \beta_i = \cos \left( \frac{(2i - 1)\pi}{4n} \right).
\]

Each step of the parallel sign function iteration starts with parallel squaring of the iteration matrix \( S[k] \) which requires \( O(n^3/p) \) parallel arithmetic steps. After \( S^2[k] \) is computed, it is distributed among the processors, and each processor sequentially computes \((\alpha_i^2 I + \beta_i^2 S^2[k])^{-1}\). This step requires \( O(n^2/p) \) communication steps and \( O(n^3) \) arithmetic operations. The summation of 2.6 is obtained using a binary tree which requires \( O(n^2 \log p) \) communication and arithmetic steps. After the sum is obtained, it is multiplied by \( S[k] \) in parallel to obtain \( S[k + 1] \). This step requires \( O(n^3/p) \) arithmetic operations. Thus, a single iteration step requires approximately \( O(n^3/p) \) arithmetic operations and \( O(n^2 \log p) \) communication steps.

### 2.4 The Parallel Continued Fraction Algorithm

The proposed algorithm is based on parallelization of the continued fraction algorithm [13]. This algorithm employs the inverse square-root of a matrix using the continued fraction expansion. The iterative algorithm for computing the inverse of the principal square root of the complex matrix \( A \in \mathbb{C}^{n \times n} \) is stated as follows:

\[
\begin{bmatrix}
  P_1 \\
  Q_1
\end{bmatrix} =
\begin{bmatrix}
  I \\
  I
\end{bmatrix},
\]

\[
\begin{bmatrix}
  P_j \\
  Q_j
\end{bmatrix} =
\begin{bmatrix}
  I & I \\
  A & I
\end{bmatrix}
\begin{bmatrix}
  P_{j-1} \\
  Q_{j-1}
\end{bmatrix},
\]

\[
\lim_{j \to \infty} P_j Q_j^{-1} = (\sqrt{A})^{-1},
\]

where \( P_j, Q_j \in \mathbb{C}^{n \times n} \). By replacing the block element \( A \) with \( A^2 \) in the iteration matrix of 2.7, we obtain an iterative algorithm for computing the inverse square root of the square of a complex matrix, which in turn can be used for the computation of the matrix sign function due to the following alternative definition of the matrix
sign function:
\[
\text{sign}(A) = A(\sqrt{A^2})^{-1} = A^{-1}(\sqrt{A^2}) .
\]

The continued fraction based matrix sign function algorithm starts with \( S[0] = A \), and uses the iteration:
\[
S[k + 1] = S[k] \left( \sqrt{S^2[k]} \right)^{-1} .
\]

For each value of \( k \), we compute an approximation for the inverse of the principal square root of \( S^2[k] \). This is achieved by iterating the continued fraction algorithm \( r \) times for \( j = 1, 2, \ldots, r \). The matrix sign function algorithm starts with \( S[0] = A \) and iteratively computes the sign function of \( A \) by going through a series of baby-steps and giant-steps, corresponding to the computation of an \( r \)-step approximation for the inverse of the principal square root and the computation of the new value of \( S \), respectively. Thus, the iteration for the inverse square root is modified for computing the matrix sign function as follows:

\[
\begin{align*}
\text{Start:} & \quad S[0] &= A , \\
\text{Baby-Step:} & \quad P_1[k] = I , \\
& \quad Q_1[k] = I , \\
& \quad j = 2, 3, \ldots, r : \quad P_j[k] = P_{j-1}[k] + Q_{j-1}[k] , \\
& \quad Q_j[k] = S^2[k]P_{j-1}[k] + Q_{j-1}[k] , \\
\text{Giant-Step:} & \quad S[k + 1] = S[k]P_r[k]Q_r^{-1}[k] .
\end{align*}
\]

A baby-step corresponds to the computation of \( P_r[k] \) and \( Q_r[k] \) using \( S[k] \) and the initial values \( P_1[k] = Q_1[k] = I \). There are no matrix inversions during this phase. The number of iterations during the baby-step phase is equal to \( r \), which is predetermined. We start with \( P_1[k] = I \) and \( Q_1[k] = I \) and compute \( P_j[k] \) and \( Q_j[k] \) for \( j = 2, 3, \ldots, r \), using the continued fraction based iterative algorithm for the inverse square root. A giant-step, on the other hand, corresponds to the computation of \( S[k + 1] \), using \( S[k] \), \( P_r[k] \), and \( Q_r[k] \). There is a single matrix inversion during
the giant-step. The number of giant-step iterations is a function of the convergence properties of the algorithm, the input matrix, as well as the baby-step length \( r \).

- Distribute \( P_{j-1}[k] \) and \( Q_{j-1}[k] \) among processors in a column wrapped fashion.
- Compute \( S^2[k] \) using parallel matrix multiply.
- Compute \( S^2[k]P_{j-1}[k] \) using parallel matrix multiply.
- Add the corresponding columns of \( P_{j-1}[k] \) and \( Q_{j-1}[k] \) to obtain \( P_j[k] \).
- Add \( S^2[k]P_{j-1}[k] \) and \( Q_{j-1}[k] \) in a similar fashion to obtain \( Q_j[k] \).
- Solve in parallel for \( S[k+1] \) in \( Q_r[k]S[k+1] = S[k]P_r[k] \).

The proposed algorithm achieves data parallelism by distributing the matrices among the processors and obtaining the LU decomposition and matrix products in parallel. With \( p < n \) processors, computing \( S^2[k] \) and \( S^2[k]P_{j-1}[k] \) and updating the new iterate on each processor requires \( O(n^3/p) \) arithmetic steps and \( O(n^2B/p) \) communication steps, where \( B \) is the communication overhead of a multi-node broadcast operation. Adding the corresponding rectangular arrays \( P_{j-1}[k] \), \( S^2[k]P_{j-1}[k] \) and \( Q_{j-1}[k] \) to obtain \( P_j[k] \) and \( Q_j[k] \) requires \( n^2/p \) arithmetic steps and \( O(n^2B/p) \) communication steps. Finally parallel solution of the linear system requires \( O(n^3/p) \) parallel arithmetic operations. Therefore each giant-step requires approximately \( O(rn^3/p) \) arithmetic steps and \( O(rn^2B/p) \) communication steps where \( r \) is the baby-step length.

2.5 Implementation Results and Conclusions

We have implemented these three parallel matrix sign function algorithms on an 8-processor partition of a Meiko CS-2 multiprocessor, in which each node is a Sparc processor equipped with 256 MBytes of memory. The algorithms are implemented using the PVM software. In our experiments, we have computed the sign functions
Table 2.1. The parallel times for the algorithms (in seconds).

<table>
<thead>
<tr>
<th>Size</th>
<th>$p = 2$</th>
<th></th>
<th></th>
<th>$p = 4$</th>
<th></th>
<th></th>
<th>$p = 8$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>PFE</td>
<td>CF</td>
<td>N</td>
<td>PFE</td>
<td>CF</td>
<td>N</td>
<td>PFE</td>
<td>CF</td>
</tr>
<tr>
<td>128</td>
<td>0.221</td>
<td>0.070</td>
<td>0.052</td>
<td>0.184</td>
<td>0.058</td>
<td>0.043</td>
<td>0.153</td>
<td>0.047</td>
<td>0.037</td>
</tr>
<tr>
<td>256</td>
<td>0.504</td>
<td>0.186</td>
<td>0.161</td>
<td>0.387</td>
<td>0.143</td>
<td>0.124</td>
<td>0.297</td>
<td>0.109</td>
<td>0.094</td>
</tr>
<tr>
<td>384</td>
<td>0.985</td>
<td>0.422</td>
<td>0.312</td>
<td>0.703</td>
<td>0.301</td>
<td>0.230</td>
<td>0.502</td>
<td>0.212</td>
<td>0.168</td>
</tr>
<tr>
<td>512</td>
<td>1.463</td>
<td>0.700</td>
<td>0.601</td>
<td>0.975</td>
<td>0.464</td>
<td>0.401</td>
<td>0.650</td>
<td>0.314</td>
<td>0.273</td>
</tr>
<tr>
<td>640</td>
<td>2.242</td>
<td>1.192</td>
<td>1.024</td>
<td>1.401</td>
<td>0.745</td>
<td>0.640</td>
<td>0.875</td>
<td>0.481</td>
<td>0.407</td>
</tr>
<tr>
<td>768</td>
<td>3.924</td>
<td>2.086</td>
<td>1.792</td>
<td>2.451</td>
<td>1.301</td>
<td>1.120</td>
<td>1.529</td>
<td>0.843</td>
<td>0.712</td>
</tr>
<tr>
<td>896</td>
<td>7.065</td>
<td>3.754</td>
<td>3.224</td>
<td>4.420</td>
<td>2.340</td>
<td>2.017</td>
<td>2.750</td>
<td>1.517</td>
<td>1.288</td>
</tr>
<tr>
<td>1024</td>
<td>12.993</td>
<td>6.941</td>
<td>5.967</td>
<td>8.146</td>
<td>4.325</td>
<td>3.729</td>
<td>5.100</td>
<td>2.807</td>
<td>2.379</td>
</tr>
</tbody>
</table>

of matrices of dimensions ranging from 128 to 1024. The matrices are generated randomly with geometrically distributed eigenvalues. In Table 1, we give the parallel times for the three algorithms as a function of time for $p = 2, 4,$ and 8.

The number of iteration steps for the the partial fraction algorithm is a function of the number of processors (the order of the summation) and the size of the matrix. For the continued fraction algorithm the number of the giant-steps depends on $r$. A thorough analysis of the dependency of the number of giant-steps on $r$ is given in [13]. For this implementation we selected $r$ to be 4. In Table 2, we tabulate the number of iterations for the algorithms as a function of the matrix size and the number processors.

Our implementation results show that Newton's method is the slowest of all three algorithms, mainly because of the high number of iterations. For example, for matrix size 1024, Newton's method requires nearly twice the time required by the partial fraction expansion algorithm. Furthermore, comparing the other two algorithms to
Table 2.2. The number of iterations for the algorithms.

<table>
<thead>
<tr>
<th>Size</th>
<th>$p = 2$</th>
<th></th>
<th>$p = 4$</th>
<th></th>
<th>$p = 8$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>PFE</td>
<td>CF</td>
<td>N</td>
<td>PFE</td>
<td>CF</td>
</tr>
<tr>
<td>128</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>13</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>256</td>
<td>13</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>384</td>
<td>13</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>512</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>14</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>640</td>
<td>14</td>
<td>8</td>
<td>7</td>
<td>14</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>768</td>
<td>14</td>
<td>8</td>
<td>7</td>
<td>14</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>896</td>
<td>14</td>
<td>8</td>
<td>7</td>
<td>14</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>1024</td>
<td>15</td>
<td>7</td>
<td>7</td>
<td>15</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

one another, we conclude that the continued fraction algorithm is slightly faster than the partial fraction expansion algorithm. Although the partial fraction algorithm requires the fewest number of iterations for $p = 4$ or $8$, its total time is slightly larger than the continued fraction algorithm. This is mainly due to the fact that the partial fraction expansion algorithm requires the summation of $n^3$ matrix elements distributed over $p$ processors, introducing a communication penalty of $O(n^3 \log p)$ at each step which considerably lengthens the total time.
The matrix n-sector function is a generalization of the matrix sign function, and can be used to determine the number of eigenvalues of a matrix in a specific sector of the complex plane, and to extract the eigenpairs belonging to this sector without explicitly computing the eigenvalues. It is known that Newton's method, which can be used for computing the matrix sign function, is not globally convergent for the matrix sector function. The only existing algorithm for computing the matrix sector function is based on the continued fraction expansion approximation to the principal nth root of an arbitrary complex matrix. We will analyze the existing algorithms and introduce a new algorithm, based on Halley's generalized iteration formula for solving nonlinear equations. It is shown that the iteration has good error propagation properties and high accuracy. Finally, we give two application examples, and summarize the results of our numerical experiments comparing Newton's, the continued fraction, and Halley's method.

3.1 Introduction

Fast computation of a restricted subset of eigenpairs of time varying matrices is an important topic in real-time signal processing and control applications. Approximation of a matrix by another of lower rank, or model reduction, is desired in many applications, e.g., in systems theory [49], data analysis, pattern recognition, spectral analysis, radar, sonar, and geophysics [19]. Fast sequential and parallel algorithms such as distributed QR decomposition, power methods, and Krylov-space based ap-
proximation algorithms for computing the eigenvalues and eigenvectors of unsymmetric matrices have been developed [9, 27, 55]. These algorithms can be used to compute all the eigenvalues of the matrix and then to extract the specified subset of the eigenvalue and eigenvector pairs. However, for these applications, methods which compute only a restricted subset of eigenpairs without resorting to computationally expensive eigenpair methods would be more useful and efficient. The matrix sign function and the matrix n-sector function can efficiently and reliably be used for this purpose [4, 3, 15, 11]. By obtaining the sector function of a matrix, we can easily determine the number of eigenvalues of a matrix in a specific sector of the complex plane, and extract the eigenpairs belonging to this sector without explicitly computing the eigenvalues and eigenvectors. By shifting the original matrix, or applying bilinear transformations, we can extend the sectors to various shapes and geometries. The matrix sign and sector function methods also have certain properties which make them more advantageous over the algorithms like QR decomposition, e.g., complex arithmetic is avoided for matrices with real entries.

The matrix sign function is a particular case of the matrix n-sector function for \( n = 2 \). Sequential and parallel algorithms for the matrix sign function have been developed [43, 35, 33, 26, 13], and its applications in systems theory and matrix analysis have been established [47, 10, 26]. However, the matrix sector function is a fairly new research topic. To the best of our knowledge, the only existing algorithm for computing the sector function of arbitrary complex matrices is the one given in [54]. Let a matrix \( A \in \mathbb{C}^{m \times m} \) have eigenspectrum \( \sigma(A) = \{\lambda_i, i = 1, \ldots, m\} \) where \( \lambda_i \neq 0 \) and \( \arg(\lambda_i) \neq \pi \). The principal nth root of \( A \) is defined as \( \sqrt[n]{A} \in \mathbb{C}^{m \times m} \) which satisfies \( (\sqrt[n]{A})^n = A \) and \( \arg(\sigma(\sqrt[n]{A})) \in (-\pi/n, \pi/n) \). It is known that Newton’s method can be used to compute the principal nth root of a positive definite matrix [31]. However, it has been pointed out in [50, 54] that Newton’s method fails to give the principal nth root of a general complex matrix, thus, cannot be used for computing the matrix sector function.
In this chapter we will introduce a fast and highly accurate algorithm. We first give the definitions of the matrix sector function and point out its applications. Then we introduce Halley's method, and give a perturbation analysis which shows that the first order errors in one step do not propagate to the next step. Finally, two application examples are given, and numerical experiments comparing Newton's, the continued fraction, and Halley's methods are summarized.

### 3.2 Definition of Matrix Sector Functions

The $n$-sector function of a scalar (matrix) is based on the principal $n$th root of the scalar (matrix). We begin with the definition of the sector function for a scalar. Let $\lambda \in \mathbb{C}$ be expressed by $\lambda = pe^{i\theta}$, where $p > 0$, $j = \sqrt{-1}$, $\theta \in [0, 2\pi)$, and $\theta \neq 2\pi(k + \frac{1}{2})/n$ for $k \in [0, n - 1]$. Assume that $\lambda$ lies within the sector $\Phi_k$ in $\mathbb{C}$ defined as the region bounded by the sector angles $2\pi(k - \frac{1}{2})/n$ and $2\pi(k + \frac{1}{2})/n$, where $k \in [0, n - 1]$. Then the scalar $n$-sector function of $\lambda$ is defined as

$$S_n(\lambda) = e^{2\pi k/n}.$$

Let $\sqrt[n]{\lambda}$ be the principal $n$th root of $\lambda \in \mathbb{C}$. As shown in [54], the scalar sector function of $\lambda$ can be expressed as

$$S_n(\lambda) = \frac{\lambda}{\sqrt[n]{\lambda^n}},$$

where $\lambda \neq 0$ and $\arg(\lambda) \neq 2\pi(k + \frac{1}{2})/n$ for $k \in [0, n - 1]$. Therefore, the scalar sector function maps a scalar in a specific sector to the bisector of the sector angles on the unit circle. The scalar sign function is a particular case of the $n$-sector function for $n = 2$, i.e., the complex plane is partitioned into 2 sectors: Complex numbers with positive and negative real parts are mapped to $+1$ and $-1$, respectively. Figure 3.1 shows the sector angles and the regions in the complex plane for $n = 2, 3, 4$.

We also define the $q$th $n$-sector function of the scalar $\lambda$ for $q \in [0, n - 1]$, denoted
Figure 3.1. The sector regions and angles for $n = 2, 3, 4$.

by $S_{n,q}(\lambda)$, as the transformation that takes $\lambda$ to 1 if $\lambda$ belongs to $\Phi_q$ and to zero otherwise:

$$S_{n,q}(\lambda) = \begin{cases} 
1 & \text{if } \lambda \in \Phi_q, \\
0 & \text{otherwise}.
\end{cases}$$

We can extend these definitions to complex square matrices as follows. Let $A \in \mathbb{C}^{m \times m}$ and $\sigma(A) = \{\lambda_i, i = 1, \ldots, m\}$ be its spectrum with not necessarily distinct eigenvalues $\lambda_i \neq 0$ and $\arg(\lambda_i) \neq 2\pi(k + \frac{1}{2})/n$ for $k \in [0, n - 1]$. Let $M \in \mathbb{C}^{m \times m}$ be the modal matrix that takes $A$ to its Jordan form as

$$A = M[J_1 \oplus J_2 \oplus \cdots \oplus J_k]M^{-1},$$

where $J_i \in \mathbb{C}^{r_i \times r_i}$ are the Jordan blocks corresponding to the $i$th eigenvalue with geometric multiplicity $r_i$, such that $\sum_{i=1}^{k} r_i = m$. Applying the matrix function definition of Giorgi [46], we can define the matrix sector function of $A$ as

$$S_n(A) = A\left(\sqrt[n]{A^n}\right)^{-1}$$

$$= M \left[ J_1 \left(\sqrt[n]{J_1^n}\right)^{-1} \oplus \cdots \oplus J_k \left(\sqrt[n]{J_k^n}\right)^{-1}\right] M^{-1},$$

where

$$J_i \left(\sqrt[n]{J_i^n}\right)^{-1} = S_n(\lambda_i)I_{r_i}.$$
Therefore, the definition of the sector function of a matrix becomes

\[ S_n(A) = M \left[ \bigoplus_{i=1}^{m} S_n(\lambda_i) \right] M^{-1} . \]

Following the definition of the scalar sector function, we see that the matrix sector function maps the eigenvalues of a given matrix to the bisector of the sector angles of the corresponding region onto the unit circle while preserving the eigenvectors. Similarly, the matrix \( q \)th \( n \)-sector function of \( A \), denoted by \( S_{n,q}(A) \), is defined as

\[ S_{n,q}(A) = M \left[ \bigoplus_{i=1}^{m} S_{n,q}(\lambda_i) \right] M^{-1} , \]

where \( S_{n,q}(\lambda) \) is the scalar \( q \)th \( n \)-sector function of \( \lambda \). The matrix \( q \)th \( n \)-sector function of \( A \) maps the eigenvalues of \( A \) in the sector \( \Phi_q \) to 1, and the remaining eigenvalues are mapped to zero. It can be easily proven (see, Theorem 4.2 in [50]) that the matrix \( q \)th \( n \)-sector function of \( A \) is equal to

\[ S_{n,q}(A) = \frac{1}{n} \sum_{i=1}^{n} \left[ S_n(A) \ e^{-j2\pi q/n} \right]^{i-1} \]

for \( q \in [0, n - 1] \), where \( S_n(A) \) is the \( n \)-sector function of \( A \).

### 3.3 Applications of Matrix Sector Functions

The matrix sector function can be utilized to block diagonalize a given matrix, without explicitly computing the eigenvalues and the corresponding eigenvectors. For \( A \in \mathbb{C}^{m \times m} \), with nonzero eigenvalues, we define the linearly independent column vectors \( \mu_i \) of \( S_{n,q}(A) \) as

\[ \mu_i = \text{ind}[S_{n,q}(A)] \in \mathbb{C}^{m \times m_i} \]

for \( q \in [0, n - 1] \), \( i \in [1, k] \), and \( m = \sum_{i=1}^{k} m_i \). For \( A \in \mathbb{C}^{m \times p} \), \( \text{ind}[A] \) is the set of linearly independent column vectors of \( A \). The block modal matrix, \( M \) which is defined as

\[ M = [\mu_1, \mu_2, \ldots, \mu_k] \in \mathbb{C}^{m \times m} , \]
can be used to block diagonalize the matrix $A$ as

$$D = M^{-1}AM = \text{diag}[A_1, A_2, \ldots, A_k],$$

where the block elements correspond to the eigenvalues in the specified sector of the complex plane. This strategy can be used to decompose a system into several smaller subsystems with similar transient characteristics. The location of the poles with respect to the sector angles determines the natural frequencies and the damping ratio of the system. Decoupling with respect to the given sectors would enable us to obtain a physical realization which is more precise and stable. This analysis can be performed from both state-space and matrix-fraction description points of view. Let a $q$-input, $p$-output system be described by

$$\dot{x}(t) = Ax(t) + Bu(t),$$
$$y(t) = Cx(t) + Du(t),$$

where $x(t) \in \mathbb{C}^{m \times 1}$, $u(t) \in \mathbb{C}^{q \times 1}$, and $y(t) \in \mathbb{C}^{p \times 1}$. Assuming the system is observable and controllable, we can define the left and right matrix fraction description of the system as

$$H_l(s) = C(sI - A)^{-1}B + D = D_l^{-1}(s)N_l(s) + D,$$
$$H_r(s) = C(sI - A)^{-1}B + D = N_r^{-1}(s)D_r(s) + D,$$

where $N_l(s)$, $D_l(s)$, $N_r(s)$, and $D_l(s)$ are polynomial matrices. Let $M$ be the block modal matrix which block diagonalizes $A$, obtained using the matrix sector functions $S_{n,q}(A)$. We have

$$A_d = M^{-1}AM = \text{diag}[\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_k] \text{ for } \hat{A}_i \in \mathbb{C}^{m_i \times m_i},$$
$$B_d = M^{-1}B = [\hat{B}_1^T, \hat{B}_2^T, \ldots, \hat{B}_k^T]^T \text{ for } \hat{B}_i \in \mathbb{C}^{m_i \times q},$$
$$C_d = CM = [\hat{C}_1, \hat{C}_2, \ldots, \hat{C}_k] \text{ for } \hat{C}_i \in \mathbb{C}^{p \times m_i}.$$

Thus, the system can be block decomposed into $k$ subsystems as

$$\dot{x}_d(t) = A_dx_d(t) + B_du(t),$$
$$y(t) = C_dx_d(t) + Du(t).$$
where \( x(t) = Mx_d(t) \). The input/output relationship after the decomposition is given as

\[
Y(s) = (C_d(sI - A_d)^{-1}B_d + D)U(s) .
\]

where the \( i \)th element of the transfer function matrix contains the \( m_i \) eigenvalues of the sector \( \Phi_i \).

### 3.4 Halley's Method for the Matrix Sector Function

It has been shown that Halley's generalized iteration formula for solving nonlinear equations is of third order, and its error-cubing variation converges faster than Newton's method [24, 20]. Halley's method can be derived by applying Newton's method to the function

\[
g(s) = \frac{f(s)}{\sqrt{f'(s)}} ,
\]

which is written as

\[
s_{k+1} = s_k - \frac{f(s_k)}{f'(s_k)} - \frac{f''(s_k)f(s_k)}{2f'(s_k)^2} .
\] (3.9)

We start with an alternative definition of the scalar sector function as the solution of the following equation

\[
f(s) = s^n - 1 = 0 .
\]

Solution of this equation with Halley iteration becomes

\[
s_{k+1} = s_k - \frac{2s_k(s_k^n - 1)}{(n+1)s_k^n + (n-1)} ,
\]

which reduces to

\[
s_{k+1} = s_k\frac{(n-1)s_k^n + (n+1)}{(n+1)s_k^n + (n-1)} .
\]

This iteration produces an order \([1,1]\) rational Padé approximant to

\[
f(s) = \frac{s}{\sqrt{1-z}} ,
\]
where \( z = 1 - s^n \). In the matrix case, \( f(S_n) \) can be defined as

\[
f(S_n) = S_n^n(A) - I = 0 .
\]

Let \( S_n[k] \) stand for the value of \( S_n(A) \) at step \( k \). Applying the iteration of Equation 3.9 to the above expression, we obtain Halley's method for matrix sector function as

\[
\begin{align*}
S_n[0] &= A , \\
S_n[k + 1] &= S_n[k] \times ((n - 1)S_n^n[k] + (n + 1)I) \times ((n + 1)S_n^n[k] - (n - 1)I)^{-1} \\
\lim_{k \to \infty} S_n[k] &= S_n(A) .
\end{align*}
\]

Now we give a convergence analysis of Halley’s method by checking the location of the eigenvalues of \( S_n(A) \) as the algorithm iterates starting from \( k = 0 \), i.e., \( S_n[0] = A \). We assume that \( S_n[k] \) has an eigenvalue \( \lambda_k \) in the sector \( \Phi_q \) at the \( k \)th step of the iteration, which can be expressed as

\[
\lambda_k = \rho_k e^{i(\psi_k + 2\pi q/n)} ,
\]

where \( \rho_k = |\lambda_k| \), \( q \in [0, n - 1] \), and \( |\psi_k| < \pi/n \). Here, \( \lambda_{k+1} \) can be given as

\[
\lambda_{k+1} = \rho_k e^{i(\psi_k + 2\pi q/n)} \frac{(n - 1)\rho_k^2 e^{i\psi_k} + (n + 1)}{(n + 1)\rho_k^2 e^{i\psi_k} + (n - 1)} .
\]

We expect

\[
\lim_{k \to \infty} \rho_k = 1 \quad \text{and} \quad \lim_{k \to \infty} \psi_k = 0 .
\]

Let \( \lim_{k \to \infty} \lambda_k \) exist and be finite. Denoting this limit by \( x \), from Equation 3.10 we obtain

\[
x = x \frac{(n - 1)x^n + (n + 1)}{(n + 1)x^n + (n - 1)}
\]

which reduces to

\[
x^{n+1} - x = 0 .
\]

Assuming \( x \neq 0 \), we find the solution of the limit equation as \( x = \sqrt[2]{1} \), i.e., an \( n \)th root of unity.
Let $z$ be a complex number in the sector $\Phi_0$ which contains the first real root of unity, +1. For the $n$ sector plane the following inequality should hold:

$$\left| \frac{(z - 1)}{(z - z_i)} \right| < 1 \text{ for } 1 \leq i \leq n - 1,$$

where $z_i = e^{i(2\pi i)/n}$, i.e. one of the $n$th roots of unity. In order to guarantee that the consecutive iterates do not pass the sector boundaries, this inequality should hold true at each step of the iteration. Let $s_k$ be a scalar at the $k$th step of the iteration, then the following equality should also be satisfied for all $k$ [37];

$$s_{k+1} - S_n(s_{k+1}) = s_{k+1} - S_n(s_k).$$

The relationship between two consecutive iterates can be obtained as follows

$$\frac{(s_{k+1} - 1)}{(s_{k+1} - z)} = \frac{(s_k - 1)^3}{(s_k - z)^3} \left( \frac{\sum_{j=1}^{n-1} \frac{(n - (2i - 1))s_k^{-j-1}}{\sum_{j=1}^{n-1} (s_{k+1}^{-j-1}z^{-j-1})} \right),$$

where $z$ is one of the $n$th roots of unity. This equation satisfies the convergence properties shown above, but contains a rational term, which for some $s_k$ may force the ratio to be greater than 1. In this case the iterate changes sectors and the iteration converges to an incorrect value. Consider $z_1 = e^{2\pi i/3}$, i.e. second root of unity. Taking $s_0 = -z_1 + 0.001 = 0.5010 - 0.8660j$ (which is in the sector $\Phi_0$) as our initial point the first step of the Halley iteration yields $s_1 = -0.4920 + 0.8660j$ and eventually the iteration converges to $z_1$ instead of 1. These inaccuracies mostly occur for points on or near the sector boundaries. Figure 3.2 gives a nice interpretation of the basin of attraction for $n = 3$.

### 3.5 Perturbation Analysis

In this section, we analyze Halley's iteration for matrix sector functions when the iterates are subject to perturbations from rounding errors at a given step $k$. Let $\tilde{S}_n[k] = S_n[k] + E[k]$, where $E[k]$ is the error at step $k$. The perturbed value of
Figure 3.2. Basin of attraction for various algorithms.
$S_n[k + 1]$ can be written as

$$\tilde{S}_n[k + 1] = (S_n[k] + E[k]) \times ((n - 1)(S_n^k + F[k]) + (n + 1)I) \times$$

$$(n + 1)(S_n^k + F[k]) + (n - 1)I^{-1},$$

where

$$F[k] = E[k]S_n^{k-1}[k] + S_n[k]E[k]S_n^{k-2}[k] + \cdots + S_n^{k-1}E[k].$$

Here, we have used the power expansion

$$(A + E)^n \approx A^n + EA^{n-1} + A^2E^{n-2} + \cdots + A^{n-1}E$$

by ignoring the terms involving more than one error term. Assuming

$$\|(n + 1)S_n^k + (n - 1)I\| > \|(n + 1)F[k]\|,$$

we utilize the perturbation formula in [56] and obtain

$$(A + E)^{-1} = A^{-1} - A^{-1}EA^{-1} + O(\|E\|^2).$$

Let

$$N[k] = (n - 1)S_n^k + (n + 1)I,$$

$$D[k] = (n + 1)S_n^k + (n - 1)I.$$

Ignoring error terms of degree two or more, we obtain

$$\tilde{S}_n[k + 1] = S_n[k]N[k]D^{-1}[k] - (n + 1)S_n[k]N[k]D^{-1}[k]F[k]D^{-1}[k] +$$

$$(n - 1)S_n[k]F[k]D^{-1}[k] + E[k]N[k]D^{-1}[k].$$

This gives the error expression as

$$E[k + 1] = \tilde{S}_n[k + 1] - S_n[k + 1]$$

$$= E[k]N[k]D^{-1}[k] + (n - 1)S_n[k]F[k]D^{-1}[k] - (n + 1)S_n[k + 1]F[k]D^{-1}[k].$$
Let $M$ be the modal matrix of $S_n[k]$ and $S_n[k + 1]$ such that

$$D[k] = M^{-1}S_n[k]M = \text{diag}(\lambda_1[k], \ldots, \lambda_m[k]),$$

$$D[k + 1] = M^{-1}S_n[k + 1]M = \text{diag}(\lambda_1[k + 1], \ldots, \lambda_m[k + 1]).$$

Let $\hat{E}[k] = M^{-1}E[k]M$ and $\hat{F}[k] = M^{-1}F[k]M$, then $\hat{F}[k]$ can be written element-wise as

$$\hat{F}_{ij}[k] = \left(\sum_{l=0}^{n-1} \lambda_i[k]^{n-1-l}[k]\right) \hat{E}_{ij}[k].$$

(3.11)

The matrix sector function identity suggests that

$$\lim_{k \to \infty} S_n^m[k] = I,$$

thus, the elementwise error iteration becomes

$$\hat{E}_{ij}[k + 1] = \hat{E}_{ij}[k] + \frac{(n - 1)}{2n} \lambda_i[k] \hat{F}_{ij}[k] - \frac{(n + 1)}{2n} \lambda_i[k + 1] \hat{F}_{ij}[k],$$

where $\lim_{k \to \infty} \lambda_i[k] = e^{j2\pi q/n}$. Therefore, the error expression becomes

$$\hat{E}_{ij}[k + 1] = \hat{E}_{ij}[k] - \frac{1}{n} e^{j2\pi q/n} \hat{F}_{ij}[k].$$

(3.12)

We have two cases to consider:

1. If $\lambda_i$ and $\lambda_j$ are in different sectors, Equation 3.11 gives $\hat{F}_{ij}[k] = 0$, and the error expression 3.12 becomes

$$\hat{E}_{ij}[k + 1] = \hat{E}_{ij}[k],$$

i.e., the error in the $k$th step is passed to, but not magnified in the $(k + 1)$st step.

2. If $\lambda_i$ and $\lambda_j$ are in the same sector, Equation 3.11 gives

$$\hat{F}_{ij}[k] = ne^{j2\pi q(n-1)/n} \hat{E}_{ij}[k].$$

Thus, the error expression 3.12 becomes

$$\hat{E}_{ij}[k + 1] = 0,$$

i.e., no first order error is propagated to the $(k + 1)$st step.
Table 3.1. The eigenvalues of $A$ and their properties.

<table>
<thead>
<tr>
<th>Eigenvalue</th>
<th>Damping ($\xi$)</th>
<th>$\omega_n$ (rad/sec)</th>
<th>Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.0073</td>
<td>1.0000</td>
<td>0.0073</td>
<td>$\Phi_2$</td>
</tr>
<tr>
<td>-0.0329 + 0.9467j</td>
<td>0.0348</td>
<td>0.9472</td>
<td>$\Phi_1$</td>
</tr>
<tr>
<td>-0.0329 - 0.9467j</td>
<td>0.0348</td>
<td>0.9472</td>
<td>$\Phi_3$</td>
</tr>
<tr>
<td>-0.5627</td>
<td>1.0000</td>
<td>0.5627</td>
<td>$\Phi_2$</td>
</tr>
</tbody>
</table>

Thus, we conclude that the first order errors in one step either have limited effect on the next step, or do not propagate to the next step at all.

3.6 Application Examples

Example 1

Here we analyze the state space matrix for a jet transport during cruise flight, taken from Matlab jetdemo (Version 4.1). The matrix $A$ is given as

$$
A = \begin{bmatrix}
-0.0558 & -0.9968 & 0.0802 & 0.0415 \\
0.5980 & -0.1150 & -0.0318 & 0 \\
-3.0500 & 0.3880 & -0.4650 & 0 \\
0 & 0.0805 & 1.0000 & 0
\end{bmatrix}.
$$

In Table 1 we give the eigenvalues and their damping coefficients, natural frequencies, and locations in the complex plane.

Terminating the iteration when $\|S_n[k] - S_n[k - 1]\| \leq 10^{-6}$ with a relative machine
precision of $\epsilon = 2.2204 \times 10^{-16}$, we obtain the $S_4(A)$ as

$$S_4(A) = \begin{bmatrix} -0.0445 & -1.1338 & 0.0653 & 0.0401 \\ 0.6226 & -0.3699 & -0.0916 & -0.0306 \\ -3.8290 & -0.6083 & -0.7567 & 0.0529 \\ 1.2161 & -4.3353 & 0.3667 & -0.8289 \end{bmatrix}.$$  

Since there are no eigenvalues in the first sector $S_{4,0}(A)$ is computed as a zero matrix. The other three partitioned matrix 4-sector functions, i.e., $S_{4,1}(A), S_{4,2}(A), S_{4,3}(A)$ are obtained as follows:

$$S_{4,1}(A) = \begin{bmatrix} 0.476 - 0.001j & -0.064 + 0.502j & -0.016 - 0.049j & -0.001 - 0.02j \\ -0.014 - 0.325j & 0.345 + 0.03j & -0.033 + 0.018j & -0.014 + 0.001j \\ -0.688 + 1.226j & -1.199 - 0.895j & 0.15 + 0.029j & 0.054 + 0.028j \\ 1.291 + 0.683j & -0.899 + 1.269j & 0.026 - 0.157j & 0.028 - 0.057j \end{bmatrix}.$$  

$$S_{4,2}(A) = \begin{bmatrix} 0.0474 & 0.1281 & 0.0331 & 0.0016 \\ 0.0278 & 0.3098 & 0.0663 & 0.0284 \\ 1.3766 & 2.3986 & 0.6988 & -0.1095 \\ -2.5828 & 1.7977 & -0.0524 & 0.9441 \end{bmatrix},$$  

$$S_{4,3}(A) = S_{4,1}^*(A).$$

We can extract the linearly independent column vectors by using the orthogonal projection algorithm to obtain the transformation matrix $M$ as

$$M = \begin{bmatrix} 0.4763 - 0.0014j & 0.4763 + 0.0014j & 0.0474 & 0.1281 \\ -0.0139 - 0.3252j & -0.0139 + 0.3252j & 0.0278 & 0.3098 \\ -0.6883 + 1.2262j & -0.6883 - 1.2262j & 1.3766 & 2.3986 \\ 1.2914 + 0.6834j & 1.2914 - 0.6834j & -2.5828 & 1.7977 \end{bmatrix},$$  

which transforms the system matrix to three subblocks as

$$D = M^{-1}AM = \begin{bmatrix} -0.0329 + 0.9467j & 0 & 0 & 0 \\ 0 & -0.0329 - 0.9467j & 0 & 0 \\ 0 & 0 & -0.5419 & -0.9578 \\ 0 & 0 & -0.0116 & -0.0280 \end{bmatrix}.$$
The matrix $D$ contains subblocks belonging to four sectors on the 4-sector plane. In fact, the first two diagonal elements are the complex eigenvalues of matrix $A$. The last block gives the real eigenvalues in $\Phi_2$. We have calculated the relative error in the computed solution $S_c$ with respect to the solution $S$ which is obtained by computing the eigenvalues of the matrix explicitly, i.e., $\|S_c - S\|/\|S\|$, for the methods mentioned. In our experiments, Halley’s method gave an error of $1.3965 \times 10^{-15}$ upon termination. The continued fraction algorithm converged with a slightly larger error of $1.7294 \times 10^{-15}$, and Newton’s method converged to an incorrect value. The absolute maximum off-diagonal element of matrix $D$ is found as $5.72 \times 10^{-5}$.

Example 2

In this example we show how the impulse response of a system can be decomposed into its oscillatory and damped exponential components by utilizing the matrix sector functions. We consider a linear time invariant system, represented by the matrices

$$A = \begin{bmatrix}
-5.1596 & 28.2055 & -15.6936 & -52.2117 & 46.9724 \\
3.4870 & -5.8394 & -9.6580 & 7.5337 & 0.1969 \\
12.3290 & -41.4465 & 3.2807 & 65.0112 & -50.2677
\end{bmatrix},$$

and

$$B = \begin{bmatrix}
12 & 6 & 12 & 13 & 16
\end{bmatrix}^T, \quad C = \begin{bmatrix}
0.1710 & -4.7202 & 4.8860 & 9.1554 & -9.3990
\end{bmatrix}.$$
\[ D = \text{diag}(7 + 7j, 7 - 7j, 70 + 70j, 70 - 70j) , \]
whose eigenvalues are on the sector angles \( \mp \pi/4 \) of the 4-sector plane. We shift the real part of this matrix by \( 1/\mu \) for \( \mu > 0 \) to obtain \( D(\mu) \) as
\[
D(\mu) = \text{diag}\left((7 + \frac{3}{\mu}) + 7j, (7 + \frac{1}{\mu}) - 7j, (70 + \frac{1}{\mu}) + 70j, (70 + \frac{1}{\mu}) - 70j \right).
\]
This changes the location of the eigenvalues of this matrix from the sector angles to \( \Phi_0 \) on the 4-sector plane. After this small shift, we form the following upper

**Figure 3.3.** Total and decomposed system impulse responses.

\( \Phi_1 \) and \( \Phi_3 \), where the damping ratio \( \xi \) is less than 0.707, and the eigenvalues of \( A_2 \) lies in \( \Phi_2 \), where \( \xi > 0.707 \). Figure 2 shows the total and decomposed system impulse responses, in which the solid and dashed lines correspond to the impulse response of subsystems \( A_1 \) and \( A_2 \), respectively.

### 3.7 Numerical Experiments

In this section, we analyze the accuracy of the three algorithms, namely, Newton’s method, the continued fraction method, and Halley’s method, according to the location of the eigenvalues with respect to the sector angles. We start with a diagonal matrix
\[ D = \text{diag}(7 + 7j, 7 - 7j, 70 + 70j, 70 - 70j) , \]
Table 3.2. The eigenvalues of $S_4(A)$ after the convergence.

<table>
<thead>
<tr>
<th>$\mu$</th>
<th>Newton</th>
<th>Continued Fraction</th>
<th>Halley</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$\sigma(S_4[k])$</td>
<td>$k$</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>$\pm j$</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>47</td>
<td>$\pm 1$</td>
<td>22</td>
</tr>
<tr>
<td>$10^2$</td>
<td>49</td>
<td>1, $\pm j$</td>
<td>26</td>
</tr>
<tr>
<td>$10^3$</td>
<td>54</td>
<td>$\pm j$</td>
<td>29</td>
</tr>
<tr>
<td>$10^4$</td>
<td>65</td>
<td>$\pm j$</td>
<td>32</td>
</tr>
<tr>
<td>$10^5$</td>
<td>71</td>
<td>1</td>
<td>35</td>
</tr>
</tbody>
</table>

triangular matrix

$$A(\mu) = D(\mu) + T$$

by adding a strictly upper triangular matrix $T$, with elements uniformly distributed over the interval $[0, 1]$. Now, an accurate sector function algorithm should produce $S_4(A(\mu))$ with all eigenvalues equal to 1, i.e.,

$$\sigma(S_4(A(\mu))) = \{1, 1, 1, 1\} \text{ for all } \mu > 0.$$

We have applied Newton's, the continued fraction, and Halley's methods to compute $S_4(A(\mu))$ for several values of $\mu$. The results are summarized in Table 2.

The iterative matrix sector algorithm may use more floating point operations than the QR algorithm, but it is easier to parallelize and contains simple matrix operations such as LU decomposition and matrix multiplication. Our experiments shows that only Halley's iteration converges accurately for all $\mu$. Furthermore, Halley's method computes $S_4(A)$ using fewer iterations than both Newton's and the continued fraction methods.
3.8 Conclusion

We have described an iterative algorithm for the computation of the matrix sector function, which is based on the solution of a nonlinear equation using Halley's method. The algorithm is fast and numerically stable, and gives accurate results even for matrices with ill-conditioned eigenstructures. We have discussed applications of matrix sector functions, and provided some examples supporting these applications. We are currently investigating the effects of scaling on the speed of convergence, and developing efficient methods for the computation of partitioned matrix sector functions from the matrix sector function.
Chapter 4
A Parallel Algorithm for Computing the \( n \)th Roots of Positive Definite Matrices

4.1 Introduction

Several computation methods for finding the \( n \)th roots of a matrix have been developed in \([31, 54, 30, 29, 8, 23]\). Björk and Hammarling \([8]\) proposed a fast and stable method for computing the square and cube root of a given matrix which is based on the Schur factorization \( A = QSQ^H \) and using a fast recursion similar to Parlett’s method \([44]\) to compute the upper triangular square root of \( S \). An extension of this algorithm has been developed by Higham \([30]\) to compute the real square roots of real matrices. Hoskins and Walton \([31]\) have proposed an accelerated iterative method for computing the \( n \)th roots of a positive definite matrix, and Denman \([23]\) has extended their method for general real matrices. Shieh et al. developed an algorithm to compute the principal \( n \)th roots of complex matrices \([51]\), and improved its convergence properties in \([54]\).

In this chapter, we will introduce a parallel iterative algorithm to compute the principal \( n \)th root of a positive definite matrix without prior knowledge of the eigenvalues \([16]\). Our algorithm is based on the Gauss-Legendre approximation of a definite integral. A similar approach have been proposed in \([43]\) to approximate matrix sign functions. After reviewing the Gauss-Legendre quadrature formula, we will derive the summation expression for the iterative \( n \)th root algorithm and analyze the error properties of this iteration. Finally we will give numerical examples and their timings to demonstrate the speed-up with respect to the Hoskins-Walton algorithm, which is the fastest sequential algorithm.
The principal $n$th root of a positive definite matrix can be defined in terms of its eigenvalue decomposition. Let $A \in \mathbb{R}^{q \times q}$ be a positive definite matrix and $\sigma(A) = \{\lambda_i, \; i = 1, \ldots, q\}$ be its spectrum with eigenvalues $\lambda_i \neq 0$ and $\arg(\lambda_i) \neq \pi$. Let $M \in \mathbb{C}^{q \times q}$ be the modal matrix that takes $A$ to its Jordan form as

$$A = M \text{ diag}(\lambda_1, \lambda_2, \ldots, \lambda_q) M^{-1}.$$  

Applying the matrix function definition of Giorgi [46], the principal $n$th root of a complex matrix can be defined as

$$\sqrt[n]{A} = M \text{ diag}(\sqrt[n]{\lambda_1}, \sqrt[n]{\lambda_2}, \ldots, \sqrt[n]{\lambda_q}) M^{-1},$$

where $n$ is a positive integer, and $\arg(\sqrt[n]{\lambda_i}) \in (-\pi/n, \pi/n)$.

### 4.2 Parallel Algorithm for the Matrix $n$th Root

Padé approximations of hypergeometric functions have been used to approximate several matrix valued functions (see [43, 33]). To approximate the principal $n$th roots of positive definite matrices, we will use the hypergeometric function

$$2F_1(\alpha, \beta, \gamma, z) = \sum_{i=0}^{\infty} \frac{(\alpha)_i (\beta)_i}{i! (\gamma)_i} z^i$$

where $\alpha, \beta, \gamma, z \in \mathbb{R}$, and

$$(\alpha)_i = \frac{\alpha(\alpha + 1) \cdots (\alpha + i - 1)}{i!} \quad \text{with} \quad (\alpha)_0 = 1.$$ 

For $\alpha = 1/n$, and $\beta = \gamma = 1$, the particular hypergeometric function

$$2F_1(1/n, 1, 1, z) = \frac{1}{\sqrt{1 - z}}$$

can be used to obtain the $n$th root of a given scalar.

As $0 < 1/n < 1$ for all positive integers $n$, the two coefficients of the power series expansion can be calculated by the following integral

$$\frac{(\alpha)_i (\beta)_i}{i! (\gamma)_i} = \int_0^1 x^i w(x) dx,$$
and the weighting function $w(x)$ is defined by (see [34] for proof)

$$w(x) = w(\alpha, \gamma, x) = \frac{\Gamma(\gamma)}{\Gamma(\alpha) \Gamma(\gamma - \alpha)} x^{\alpha-1} (1 - x)^{\gamma - \alpha - 1}, \quad (4.4)$$

where $\Gamma$ defines the Gamma function. For $\alpha = 1/n$, $\beta = \gamma = 1$, the weighting function becomes

$$w(1/n, 1, x) = \frac{\Gamma(1)}{\Gamma(1/n) \Gamma(1 - 1/n)} x^{(1/n)-1} (1 - x)^{-1/n}, \quad (4.5)$$

where

$$\Gamma(1/n) \Gamma(1 - 1/n) = \pi \csc(\pi/n).$$

Substituting $w(x)$ to the power series expansion of Equation 4.2, we obtain

$$\frac{1}{\sqrt{1 - z}} = \frac{1}{\pi \csc(\pi/n)} \sum_{i=0}^{\infty} \left( \int_0^1 x^i w(1/n, 1, x) dx \right) z^i$$

$$= \frac{1}{\pi \csc(\pi/n)} \int_0^1 \sum_{i=0}^{\infty} (xz)^i w(1/n, 1, x) dx .$$

Reducing the summation term inside the integration, we obtain

$$\frac{1}{\sqrt{1 - z}} = \frac{1}{\pi \csc(\pi/n)} \int_0^1 \frac{x^{(1/n)-1}}{(1 - x)^{1/n}(1 - xz)} dx. \quad (4.6)$$

Our starting point for the parallel nth root algorithm will be the Gauss-Legendre quadrature approximation to the derived integral formula.

4.2.1 Review of the Gauss-Legendre Quadrature Formula

Before deriving the quadrature expression for Equation 4.6 we will discuss the Gauss-Legendre approximation for definite integrals. Let $f$ and $g$ be two differentiable functions over the interval $(a, b)$, and $w(x)$ be a nonnegative weight function on the same interval. We assume that

$$\int_a^b |x|^n w(x) dx$$
is integrable and finite for all \( n \geq 0 \) and if

\[
\int_{a}^{b} w(x)g(x)dx = 0
\]

for some continuous function \( g(x) \), then the function \( g(x) \equiv 0 \) on \((a, b)\). We define the inner product of two continuous functions functions by

\[
(g, h) = \int_{a}^{b} w(x)g(x)h(x)dx \quad g, h \in (a, b) .
\]

(4.7)

According to the Gram-Schmidt Theorem, there exists a unique sequence of polynomials \( \{\varphi_n(x)|n \geq 0\} \) with degree(\( \varphi_n \)) = \( n \) for all \( n \) where

\[
(\varphi_n, \varphi_m) = 0 \quad \text{for all } n \neq m, n, m \geq 0 ,
\]

and \( (\varphi_n, \varphi_n) = 1 \) for all \( n \).

Let \( \{\varphi_n(x)|n \geq 0\} \) be an orthogonal family of polynomials on \((a, b)\) with weight function \( w(x) \geq 0 \). Then the polynomial \( \varphi_n(x) \) has exactly \( n \) distinct roots \( \{x_1, x_2, \ldots, x_n\} \) in the open interval \((a, b)\) (see [2] for proof). Defining \( A_n \) and \( B_n \) by

\[
\varphi_n(x) = A_n x^n + B_n x^{n-1} + \cdots ,
\]

we can write

\[
\varphi_n(x) = A_n (x - x_1)(x - x_2) \cdots (x - x_n) .
\]

Let

\[
a_n = A_{n+1}/A_n, \quad \gamma_n = (\varphi_n, \varphi_n) > 0 .
\]

Then the family of orthogonal polynomials on \((a, b)\) with weight function \( w(x) \geq 0 \) can be represented by the following triple recursion relation (see [53])

\[
\varphi_{n+1}(x) = (a_n x + b_n)\varphi_n(x) - c_n\varphi_{n-1}(x)
\]

(4.8)

with

\[
b_n = a_n \left( \frac{B_{n+1}}{A_{n+1}} - \frac{B_n}{A_n} \right) \quad \text{and} \quad c_n = \frac{A_{n+1}A_{n-1}}{A_n^2} \frac{\gamma_n}{\gamma_{n-1}} .
\]
For each $n \geq 1$, there is a unique numerical integration formula

$$
\int_a^b w(x)g(x)dx = \sum_{i=1}^{n} w_i g(x_i),
$$

(4.9)

which is exact for all polynomials of degree less than $2n - 1$. The explicit formula for the summation and the error is given by

$$
\int_a^b w(x)g(x)dx = \sum_{i=1}^{n} w_i g(x_i) + \frac{\gamma_n}{A_n^2(2n)!} g^{(2n)}(t) \quad a < t < b.
$$

The nodes $\{x_i\}$ are the zeros of $\varphi_n(x)$; and the weights $\{w_i\}$ are given by (see [52] for proof)

$$
w_i = \frac{-a_n \gamma_n}{\varphi'_n(x_i)\varphi_{n+1}(x_i)} \quad \text{for } i = 1, \ldots, n.
$$

To apply the Gauss-Legendre quadrature formula to the integral of Equation 4.6, we need to shift the interval of integration to $(-1,1)$. Changing variables we have

$$
x = \frac{\tilde{x} + 1}{2},
$$

and the integral takes the form

$$
\frac{1}{\sqrt{1-z}} = \frac{1}{\pi \csc(\pi/n)} \int_{-1}^{1} \frac{h(\tilde{x})}{(2 - (\tilde{x} + 1)z)}d\tilde{x},
$$

(4.10)

where

$$
h(\tilde{x}) = \frac{2(1 + \tilde{x})^{(1/n)-1}}{(1 - \tilde{x})^{1/n}}.
$$

(4.11)

This integral can be approximated by the following summation

$$
\int_{-1}^{1} \frac{h(\tilde{x})}{(2 - (\tilde{x} + 1)z)}d\tilde{x} = \sum_{i=1}^{m} w_i \frac{h(x_i)}{(2 - (x_i + 1)z)},
$$

with the nodes $\{x_i\}$ the zeroes of the degree $m$ Legendre polynomial $P_m(x)$ on $(-1,1)$ and the weights

$$
w_i = \frac{-2}{(m + 1)P'_m(x_i)P_{m+1}(x_i)} \quad \text{for } i = 1, 2, \ldots, m.
$$

(4.12)
4.2.2 Parallel Iteration

After obtaining the summation formula for the integral approximation, we design an iterative scheme to apply it for the matrix $n$th root. Since we have obtained an approximation for the $n$th root, we need to construct an iteration of type

\[
\begin{align*}
  s_0 &= 1, \\
  s_k + 1 &= s_k / \sqrt[n]{s_k^2 / \lambda}, \\
  \lim_{k \to \infty} s_k &= \sqrt[n]{\lambda}.
\end{align*}
\]

(4.13)

At each step of the iteration by substituting $z = 1 - s_k / \lambda$ in the quadrature formula we obtain the $n$th root approximation $1/ \sqrt[n]{s_k / \lambda}$ and the iteration becomes

\[
\begin{align*}
  s_0 &= 1, \\
  z_k &= 1 - s_k^n / \lambda, \\
  s_{k+1} &= \frac{1}{\pi \csc(\pi/n)} \frac{w_i h(x_i)}{s_k \sum_{i=1}^{m} (2 - (x_i + 1)z_k)} , \\
  \lim_{k \to \infty} s_k &= \sqrt[n]{\lambda}, \\
\end{align*}
\]

(4.14)

where $m$ is the number of processors. In the matrix case we have the following iteration

\[
\begin{align*}
  S_0 &= I, \\
  Z_k &= I - A^{-1} S_k^n, \\
  S_{k+1} &= \frac{1}{\pi \csc(\pi/n)} \frac{w_i h(x_i)(2I - (x_i + 1)Z_k)^{-1}}{S_k \sum_{i=1}^{m}} , \\
  \lim_{k \to \infty} S_k &= \sqrt[A]{A}.
\end{align*}
\]

(4.15)

At each step of the iteration, $n$th power of the iteration matrix $S_k$ is found using parallel matrix multiplication which has a computational complexity of $O([\log n]q^3/m)$,
Table 4.1. Node processes for the iteration.

<table>
<thead>
<tr>
<th>PARALLEL ITERATION $(A, n, m)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m =$ number of processors</td>
</tr>
<tr>
<td>$i =$ processor id</td>
</tr>
<tr>
<td>$S_0 = I$</td>
</tr>
<tr>
<td>$Z_0 = I - A^{-1}$</td>
</tr>
<tr>
<td>while $|Z_k| &lt; \epsilon$</td>
</tr>
<tr>
<td>compute $T_i = w_i h(x_i)(2I - (x_i + 1)Z_k)^{-1}$</td>
</tr>
<tr>
<td>send $T_i$ and compute their global sum $\Sigma_k$ among processors</td>
</tr>
<tr>
<td>compute $S_{k+1} = (1/\pi \csc(\pi/n))S_k \times \Sigma_k$ in parallel</td>
</tr>
<tr>
<td>compute $Z_{k+1} = I - A^{-1}S_{k+1}^n$ in parallel and broadcast</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>$S_k = \sqrt{A}$</td>
</tr>
</tbody>
</table>

and $Z_k$ is broadcast to all processors with a communication penalty of $O(q^3 \log m)$. Each processor multiplies $Z_k$ with its corresponding weight and computes its inverse, these operations have a computational complexity of $O(q^3)$. After completing scalar-matrix multiplication and inversion, all the components of the iteration on each processor is summed by using a binary tree, which has a communication overhead of $O(q^2 \log m)$. Since $\lim_{k \to \infty} S_k = \sqrt{A}$, the auxiliary iteration matrix $Z_k$ satisfies $\lim_{k \to \infty} Z_k = 0_q$, where $0_q$ is a $q \times q$ zero matrix, therefore convergence can be monitored by checking $\|Z_k\|$ at each step and the iteration can be terminated when $\|Z_k\| < \epsilon$. Table 4.1 gives the node processes for the iteration.
4.3 Error Analysis

We will analyze the error in the quadrature approximation, and the matrix iteration separately. Complete Gauss-Legendre integration including the error term approximation can be given as [2, 22]

\[ \int_{-1}^{1} f(x) \, dx = \sum_{i=1}^{m} w_i f(x_i) + E_m(f) , \]

where

\[ E_m(f) = \frac{2^{2m+1}(m!)^4}{(2m+1)((2m)!)^2} \frac{f^{(2m)}(t)}{(2m)!} = e_m \frac{f^{(2m)}(t)}{(2m)!} . \tag{4.16} \]

We will try to reduce the error term of Equation 4.16, and make it more understandable. Defining

\[ M_m = \max_{-1 < t < 1} |f^{(m)}(t)| \] \tag{4.17}

and combining it with the error term of Equation 4.16, we have

\[ |E_m(f)| \leq e_m M_{2m} . \tag{4.18} \]

At this point, we can approximate \( e_m \) with its asymptotic bound by substituting Stirling's formula

\[ n! \approx e^{-n}n^n\sqrt{2\pi n} \]

in Equation 4.16 and obtain

\[ e_m = \frac{\pi}{4^m} . \]

This reduces the error formula to

\[ |E_m(f)| \leq \frac{\pi}{4^m} M_{2m} . \tag{4.19} \]

In the scalar case of the definite integral for the matrix \( n \)th root, we have

\[ f(x, z) = \frac{2(1 + x)^{(1/n) - 1}}{(1 - x)^{1/n}(2 - (x + 1)z)} . \tag{4.20} \]

As seen above, the integrand has an apparent singularity at the upper limit of the integration. This singularity does not vanish for the higher derivatives of the function.
But the quadrature expression does not evaluate the integrand at the singularity. Integrals with several types of singularities, and their approximation error in the Gauss type integration has been analyzed in [22]. It has been shown that, for integrands of non-oscillatory behavior, endpoint singularities can be ignored to approximate the error term.

After analyzing the error properties of the scalar integration expression, we will formulate error expression for the matrix iteration. Let \( M \in \mathbb{R}^{q \times q} \) take the symmetric positive definite matrix \( S_k \) to its diagonal form \( D_k \) as follows

\[
D_k = M^{-1}S_kM,
\]

where \( D_k = \text{diag}(d_1[k], d_2[k], \ldots, d_q[k]) \). In a similar fashion, \( Z_k \) can be decomposed to its diagonal form by using the same modal matrix \( M \) as follows

\[
Z_k = M^{-1}(I - \Lambda^{-1}D_k^n)M,
\]

where \( \Lambda = M^{-1}AM, \Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_q) \) and

\[
(I - \Lambda^{-1}D_k^n) = \text{diag}(1 - \frac{d_1^n[k]}{\lambda_1}, 1 - \frac{d_2^n[k]}{\lambda_2}, \ldots, 1 - \frac{d_q^n[k]}{\lambda_q}).
\]  

(4.21)

Decomposing the matrix iteration with the same modal matrix \( M \), we obtain the matrix iteration on the singular values of \( S_k \) as follows

\[
d_j[k+1] = \frac{1}{\pi \csc(\pi/n)} d_j[k] \sum_{i=1}^{m} \frac{w_i h(x_i)}{(2 - (x_i + 1)(1 - d_j^n[k]/\lambda_j))}
\]

(4.22)

\[
d_j[k+1] = \sqrt{\lambda_j + \varepsilon_j^m[k]}.
\]  

(4.23)

The analysis of the scalar case suggests

\[
|\varepsilon_j^m[k]| \leq \frac{\pi}{4^m} M_{2m}^j M_{2m}^j[k] \text{ for } j = 1, \ldots, q,
\]

(4.24)

where

\[
M_{2m}^j[k] = \max \frac{f^{(2m)}(t, 1 - d_j^n[k]/\lambda_j)}{(2m)!} \text{ for } j = 1, \ldots, q \text{ and } -1 < t < 1.
\]

(4.25)
As seen from Equations 4.23 and 4.24, the error term at each step of the iteration depends on the order of summation $m$, and the eigenvalues of the original system $\lambda_j$. The implementation results supports this argument and as the number of processors hence order of summation increases, number of iteration steps decreases, and the algorithm converges more rapidly.

4.4 Implementation Results

We have implemented three parallel matrix sign function algorithms on a Meiko CS-2 multiprocessor with 16 processors, in which each node is a Sparc processor equipped with 256 MBytes of memory. In our experiments, we have computed the sign functions of matrices of dimensions ranging from 128 to 1024. The matrices are generated randomly with geometrically distributed eigenvalues with condition numbers $\kappa_2(A)$ up to $1 \times 10^3$. For the sequential code, we have implemented three most commonly used algorithms, Hoskins-Walton iteration, the continued fraction method and Newton's method. Hoskins-Walton algorithm was the fastest in terms of CPU time. We have implemented the parallel code for $p = 2, 4$ and 8. As seen from Table 2, increasing the number of processors decreased number of iteration steps for the parallel algorithm. The efficiency of the algorithm for $p = 2, 4$ and 8 with respect to Hoskins-Walton algorithm is given in Figure 1.
### Table 4.2. Number of iteration steps for sequential algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Matrix Size →</th>
<th>128</th>
<th>256</th>
<th>384</th>
<th>512</th>
<th>640</th>
<th>768</th>
<th>896</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Root</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continued Fraction</td>
<td></td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Hoskins-Walton</td>
<td></td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Newton</td>
<td></td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>23</td>
<td>26</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>32</td>
</tr>
</tbody>
</table>
Table 4.3. Number of iteration steps for the parallel algorithm.

<table>
<thead>
<tr>
<th>( P )</th>
<th>Matrix Size</th>
<th>128</th>
<th>256</th>
<th>384</th>
<th>512</th>
<th>640</th>
<th>768</th>
<th>896</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Root</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Table 4.4. Convergence time for the sequential and parallel algorithms.

<table>
<thead>
<tr>
<th>P</th>
<th>Matrix Size</th>
<th>Root</th>
<th>128</th>
<th>256</th>
<th>384</th>
<th>512</th>
<th>640</th>
<th>768</th>
<th>896</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>570</td>
<td>1004</td>
<td>1403</td>
<td>1757</td>
<td>2053</td>
<td>2293</td>
<td>2510</td>
<td>2704</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>795</td>
<td>1400</td>
<td>1957</td>
<td>2451</td>
<td>2864</td>
<td>3199</td>
<td>3501</td>
<td>3772</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1235</td>
<td>2174</td>
<td>3038</td>
<td>3804</td>
<td>4446</td>
<td>4965</td>
<td>5435</td>
<td>5855</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>1428</td>
<td>2513</td>
<td>3513</td>
<td>4398</td>
<td>5141</td>
<td>5741</td>
<td>6284</td>
<td>6769</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>498</td>
<td>796</td>
<td>1047</td>
<td>1255</td>
<td>1406</td>
<td>1470</td>
<td>1550</td>
<td>1670</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>641</td>
<td>1029</td>
<td>1359</td>
<td>1634</td>
<td>1812</td>
<td>1927</td>
<td>2059</td>
<td>2193</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>950</td>
<td>1530</td>
<td>2052</td>
<td>2470</td>
<td>2744</td>
<td>2920</td>
<td>3088</td>
<td>3217</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>1050</td>
<td>1721</td>
<td>2281</td>
<td>2783</td>
<td>3096</td>
<td>3337</td>
<td>3530</td>
<td>3678</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td>296</td>
<td>448</td>
<td>604</td>
<td>720</td>
<td>814</td>
<td>881</td>
<td>936</td>
<td>979</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>389</td>
<td>603</td>
<td>802</td>
<td>957</td>
<td>1068</td>
<td>1142</td>
<td>1215</td>
<td>1978</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>571</td>
<td>890</td>
<td>1186</td>
<td>1419</td>
<td>1587</td>
<td>1723</td>
<td>1836</td>
<td>1900</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>615</td>
<td>997</td>
<td>1330</td>
<td>1593</td>
<td>1785</td>
<td>1939</td>
<td>2040</td>
<td>2089</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td>150</td>
<td>267</td>
<td>350</td>
<td>422</td>
<td>466</td>
<td>494</td>
<td>506</td>
<td>520</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>220</td>
<td>350</td>
<td>470</td>
<td>567</td>
<td>628</td>
<td>644</td>
<td>673</td>
<td>703</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>321</td>
<td>512</td>
<td>690</td>
<td>819</td>
<td>911</td>
<td>940</td>
<td>999</td>
<td>1030</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>357</td>
<td>551</td>
<td>731</td>
<td>901</td>
<td>1020</td>
<td>1104</td>
<td>1138</td>
<td>1159</td>
</tr>
</tbody>
</table>
Chapter 5
A Parallelization of Parlett’s Algorithm for Functions of Triangular Matrices

5.1 Computing Matrix Functions

Computing functions of square matrices is an important topic in linear algebra, engineering, and applied mathematics. There are several methods for this task: Jordan decomposition, Schur decomposition, and approximation methods, e.g., Taylor expansion and rational Padé approximations. The approximation methods may not be suitable for arbitrary functions, since specific properties of the function are exploited. The Jordan and Schur decomposition methods are more general in the sense that an arbitrary function of a given square matrix can be computed using these algorithms.

Let $A$ be an $n \times n$ matrix with entries from the real or complex field, and $f(\cdot)$ be the function. The Jordan decomposition algorithm is used to obtain $A = MJM^{-1}$, and then $f(A)$ is computed using the formula $f(A) = Mf(J)M^{-1}$. However, there are some computational difficulties with the Jordan decomposition approach, unless $A$ can be diagonalized and has well-conditioned eigenvectors [28]. The Schur decomposition, on the other hand, is more stable, and can be used for computing arbitrary functions of matrices. Let $A = QTQ^H$ be the Schur decomposition of the full matrix $A$, then $f(A) = Qf(T)Q^H$, where $T$ is an upper triangular matrix. This way the computation of $f(A)$ for an arbitrary matrix $A$ is reduced to the computation of $f(T)$ for an upper triangular matrix $T$.

Let $F = f(T)$, and $f_{ij}$ and $t_{ij}$ be the elements in the $i$th row and $j$th columns of the upper triangular matrices $F$ and $T$, respectively. One approach in computing the entries $f_{ij}$ is to obtain explicit expressions for each $f_{ij}$ in terms of $t_{ij}$ for all possible
values of $i$ and $j$. However, these expressions become very complicated as we move away from the main diagonal, and do not allow cost-effective computation of the entries of $F$. Let $\lambda_i = t_{ii}$. It is shown in [21, 28] that $f_{ii} = f(\lambda_i)$ for $1 \leq i \leq n$ and $f_{ij} = 0$ for $1 \leq j < i \leq n$. Furthermore, for all $1 \leq i < j \leq n$, we have

$$f_{ij} = \sum_{(s_0, \ldots, s_k) \in S_{ij}} t_{s_0, s_1} t_{s_1, s_2} \cdots t_{s_k-1, s_k} f[\lambda_{s_0}, \ldots, \lambda_{s_k}] ,$$

(5.25)

where $S_{ij}$ is the set of distinct sequences of integers such that $i = s_0 < s_1 < \cdots < s_k = j$, $1 \leq k \leq j - i$, and $f[\lambda_{s_0}, \ldots, \lambda_{s_k}]$ is the $k$th order divided difference of $f$ at $\{\lambda_{s_0}, \ldots, \lambda_{s_k}\}$. Computing the upper triangular matrix function $F = f(T)$ using this method requires $O(2^n)$ arithmetic operations, which is computationally infeasible even for matrices of moderate size.

5.2 Parlett's Algorithm

A fast algorithm for computing $F = f(T)$ was proposed by Parlett [45]. Parlett's method is derived from the following commutativity result:

$$FT = TF .$$

(5.26)

Parlett shows that by expanding the matrix multiplication and solving for $f_{ij}$ in the above, we obtain the summation formula

$$f_{ij} = t_{ij} \frac{f_{jj} - f_{ii}}{t_{jj} - t_{ii}} + \frac{1}{t_{jj} - t_{ii}} \sum_{k=i+1}^{j-1} (t_{ik} f_{kj} - f_{ik} t_{kj}) ,$$

(5.27)

which requires that $t_{ii} \neq t_{jj}$ for all $i \neq j$. Parlett's algorithm starts with computing the main diagonal entries of $F$. Since the main diagonal entries $t_{ii}$ are the eigenvalues of $T$, $f_{ii}$ is calculated by applying $f$ to each $t_{ii}$, i.e., $f_{ii} = f(t_{ii})$. After computing the main diagonal entries, the algorithm computes the superdiagonals one at a time, using the summation expression 5.27. In this chapter we will introduce a parallelization of this algorithm. Parlett's algorithm is given in Table 5.1.
Table 5.1. Parlett’s algorithm for computing functions of triangular matrices.

<table>
<thead>
<tr>
<th>for i = 1 to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{ii} = f(t_{ii}) )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>for ( L = 1 ) to ( n - 1 )</td>
</tr>
<tr>
<td>for i = 1 to ( n - L )</td>
</tr>
<tr>
<td>( j = i + L )</td>
</tr>
<tr>
<td>( s = t_{ij}(f_{jj} - f_{ii}) )</td>
</tr>
<tr>
<td>for k = ( i + 1 ) to ( j - 1 )</td>
</tr>
<tr>
<td>( s = s + t_{ik}f_{kj} - f_{ik}t_{kj} )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>( f_{ij} = s/(t_{jj} - t_{ii}) )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Note that the number of terms in the summation becomes larger as the algorithm proceeds over the superdiagonals. As an example, the expressions for \( f_{3j} \) for \( j = 3, 4, 5, 6 \) are given below:

\[
\begin{align*}
    f_{33} & = f(t_{33}), \\
    f_{34} & = t_{34}\frac{f_{44} - f_{33}}{t_{44} - t_{33}}, \\
    f_{35} & = t_{35}\frac{f_{55} - f_{33}}{t_{55} - t_{33}} + \frac{t_{34}f_{45} - f_{34}t_{45}}{t_{55} - t_{33}}, \\
    f_{36} & = t_{36}\frac{f_{66} - f_{33}}{t_{66} - t_{33}} + \frac{(t_{34}f_{46} - f_{34}t_{46}) + (t_{35}f_{56} - f_{35}t_{56})}{t_{66} - t_{33}}.
\end{align*}
\]

The number of arithmetic operations required to compute an element of the \( L \)th superdiagonal is easily calculated as \( 4L \). For example, \( f_{36} \) belongs to the 3rd superdiagonal, and \( 4 \times 3 = 12 \) arithmetic operations (4 subtractions, 2 additions, 5 multiplications, 1 division) are needed to compute \( f_{36} \). Furthermore, any element of a superdiagonal is computed using the elements of \( F \) and \( T \) to the left of and beneath this element. As an example, we illustrate the data dependency for computing \( f_{36} \) in Figure 5.1.

Parlett's algorithm performs \( n \) function evaluations to obtain the main diagonal entries of \( F \). After the elements in the main diagonal are obtained, the summation formula 5.27 is used to compute the \((n - L)\) elements in the \( L \)th superdiagonal, each of which requires \( 4L \) arithmetic operations. Thus, the number of arithmetic operations for computing the strictly upper triangular portion of \( F \) is calculated as

\[
\sum_{L=1}^{n-1}(n - L)(4L) = \frac{2}{3} (n^3 - n).
\]

Thus, assuming a single scalar function evaluation requires \( K \) arithmetic operations, Parlett's algorithm requires a total of

\[
T_{a1} = Kn + \frac{2}{3} (n^3 - n)
\]

arithmetic operations to compute all elements of the upper triangular matrix \( F \).

However, we must remark that if \( T \) has close eigenvalues, this algorithm will give
Figure 5.1. Data dependency to compute $f_{36}$.

inaccurate results. Alternative methods for dealing with the repeated eigenvalue case can be found in [45, 28].

5.3 Parallelization of Parlett’s Algorithm

Parlett’s algorithm first computes the main diagonal elements of the matrix $F$ by performing $n$ independent scalar function evaluations. Provided that we have $n$ processors available, this step requires only $K$ parallel arithmetic operations. We can then obtain the remaining elements of the upper triangular matrix by computing each super diagonal in parallel. This parallelization does not destroy the order in which the elements are computed in the serial algorithm, and thus, the parallel algorithm we propose has the same error propagation and numerical stability characteristics as the serial algorithm.

The parallel algorithm has $n$ phases; a superdiagonal vector is computed at each phase using all the available processors. The number of processors $p$ is assumed to be less than or equal to the matrix size $n$. Note that as we proceed away from
the main diagonal, the length of the superdiagonal vector decreases and the number of arithmetic operations required per element increases. Thus, we perform an approximately equal amount of work at each phase, i.e., the parallel algorithm is well-balanced. We expect the parallel algorithm to be efficient provided that the communication penalty is not very high. In order to achieve a low communication penalty we distribute the upper triangular matrices $F$ and $T$ to all processors. This provides access to all elements of $F$ and $T$ by all processors at the beginning. In order to maintain this property, we broadcast the computed superdiagonal at the end of each phase. The processors then update their copy of the $F$ matrix, and thus, have the fresh elements at the beginning of every phase. We must note that, with this partitioning technique, we may not be able to fit very large matrices to the memory available in each processor. However, the resulting parallel algorithm is efficient, and matrices of dimension up to two thousand (with double-precision floating-point entries) can be dealt with using 16 MB of memory per processor.

As illustrated in Figure 2, in order to compute the element $f_{ij}$, we need to have access to $f_{ik}$ for $k = i, i + 1, \ldots, j - 1$ and $f_{kj}$ for $k = i + 1, \ldots, j$. Since the data dependency pattern becomes more complex and the length of the summation terms increases from phase to phase, the distribution of all elements of the matrices $F$ and $T$ to all processors seems justified. With this distribution, we achieve low communication penalty. Furthermore, the work distribution of the processors is easily handled. At each phase, each processor simply picks a starting and ending index in the superdiagonal to be computed during this phase. This processor is then responsible for computing the elements in this range.

In case we have fewer than $n$ processors, there is very little change in the structure of the parallel algorithm. At each phase, the processors compute the starting and the ending indices and perform the summations in this range according to the formula 5.27. At the end of the phase, the entire superdiagonal is broadcast to all $p$ processors, and the matrix $F$ is updated to get ready for the next phase. Figure 5.3 shows the
Table 5.2. The parallel version of Parlett's algorithm.

| broadcast the matrices $T$ and $F$
| for all $i$ that processor $P$ owns
| $f_{ii} = f(t_{ii})$
| end
| broadcast the main diagonal of $F$
| for $L = 1$ to $n - 1$
| for all $i$ that processor $P$ owns
| $j = i + L$
| $s = t_{ij}(f_{jj} - f_{ii})$
| for $k = i + 1$ to $j - 1$
| $s = s + t_{ik}f_{kj} - f_{ik}t_{kj}$
| end
| $f_{ij} = s/(t_{jj} - t_{ii})$
| end
| broadcast the $L$th superdiagonal of $F$
| end

distribution of the elements over the processors at each step of the algorithm for $n = 16$ and $p = 4$. The parallel version of Parlett's algorithm is given in Table 5.2

5.4 Analysis of Efficiency and Implementation Results

As seen from Figure 4, the parallel version of Parlett's algorithm first performs $[n/p]$ function evaluations. Assuming a single function evaluation requires $K$ arithmetic
**Table 5.3.** The distribution of the elements for $n = 16$ and $P = 4$.

<table>
<thead>
<tr>
<th>P</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td>1,5</td>
<td>1,6</td>
<td>1,7</td>
<td>1,8</td>
<td>1,9</td>
<td>1,10</td>
<td>1,11</td>
<td>1,12</td>
<td>1,13</td>
<td>1,14</td>
<td>1,15</td>
<td>1,16</td>
</tr>
<tr>
<td></td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td>2,5</td>
<td>2,6</td>
<td>2,7</td>
<td>2,8</td>
<td>2,9</td>
<td>2,10</td>
<td>2,11</td>
<td>2,12</td>
<td>2,13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3,3</td>
<td>3,4</td>
<td>3,5</td>
<td>3,6</td>
<td>3,7</td>
<td>3,8</td>
<td>3,9</td>
<td>3,10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,4</td>
<td>4,5</td>
<td>4,6</td>
<td>4,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5,5</td>
<td>5,6</td>
<td>5,7</td>
<td>5,8</td>
<td>5,9</td>
<td>5,10</td>
<td>5,11</td>
<td>5,12</td>
<td>4,12</td>
<td>4,13</td>
<td>4,14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6,6</td>
<td>6,7</td>
<td>6,8</td>
<td>6,9</td>
<td>7,10</td>
<td>6,10</td>
<td>6,11</td>
<td>6,12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7,7</td>
<td>7,8</td>
<td>7,9</td>
<td>7,10</td>
<td>6,10</td>
<td>6,11</td>
<td>6,12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8,8</td>
<td>8,9</td>
<td>8,10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9,9</td>
<td>9,10</td>
<td>9,11</td>
<td>9,12</td>
<td>8,13</td>
<td>8,14</td>
<td>7,14</td>
<td>6,14</td>
<td>6,15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10,10</td>
<td>10,11</td>
<td>10,12</td>
<td>9,12</td>
<td>8,12</td>
<td>8,14</td>
<td>7,14</td>
<td>6,14</td>
<td>6,15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11,11</td>
<td>11,12</td>
<td>11,13</td>
<td>10,13</td>
<td>9,13</td>
<td>9,14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12,12</td>
<td>12,13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13,13</td>
<td>13,14</td>
<td>12,14</td>
<td>11,14</td>
<td>10,14</td>
<td>10,15</td>
<td>9,15</td>
<td>8,15</td>
<td>7,16</td>
<td>6,16</td>
<td>5,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14,14</td>
<td>14,15</td>
<td>13,15</td>
<td>12,15</td>
<td>11,15</td>
<td>11,16</td>
<td>10,16</td>
<td>9,16</td>
<td>8,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15,15</td>
<td>15,16</td>
<td>14,16</td>
<td>13,16</td>
<td>12,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
steps, the parallel computation of the main diagonal requires

\[
K \left\lfloor \frac{n}{p} \right\rfloor \leq K \left( \frac{n}{p} + 1 \right)
\]  

(5.28)

arithmetic operations. When computing the \(L\)th superdiagonal, each processor computes \(\lfloor (n-L)/p \rfloor\) elements. We calculate the number of parallel arithmetic operations required to compute all superdiagonals as

\[
\sum_{L=1}^{n-1} \left(\frac{n-L}{p} + 1\right) (4L) = \frac{2}{3p} (n^3 - n) + 2n^2 - 2n.
\]  

(5.29)

Thus, the total arithmetic complexity of the parallel algorithm is found as

\[
T_{ap} = K \left( \frac{n}{p} + 1 \right) + \frac{2}{3p} (n^3 - n) + 2n^2 - 2n.
\]  

(5.30)

In order to calculate the communication penalty, we take a closer look at the communication steps of the algorithm. After a processor computes \(\lfloor (n-L)/p \rfloor\) elements of the \(L\)th superdiagonal, these elements are broadcast to all the other processors. This operation is called a multi-node broadcast operation. A naive method of accomplishing this task is to perform \(n\) sequentially-arranged single-node broadcast operations. A better strategy is to perform simultaneous broadcast operations in order to achieve maximum concurrency. Details of the multi-node broadcast operation on a hypercube computer can be found in [7]. Let \(T_b\) be the time required to perform a multi-node broadcast operation on \(p\) single-precision floating-point numbers residing on \(p\) processors. For example, on the hypercube architecture, we have \(T_b = 2p - 2\). Since during the \(L\)th phase \(\lfloor (n-L)/p \rfloor\) elements are to be broadcast, we calculate the total communication penalty of the parallel algorithm as

\[
T_c = \sum_{L=0}^{n-1} \left(\frac{n-L}{p} \right) T_b \leq \sum_{L=0}^{n-1} \left(\frac{n-L}{p} + 1\right) T_b = \left(\frac{n^2+n}{2p} + n\right) T_b.
\]  

(5.31)

The efficiency of the parallel algorithm can be estimated using the arithmetic and communication complexity values \(T_{a1}, T_{ap},\) and \(T_c\). Let \(\tau\) be the ratio of the time required to transfer a floating-point number to an adjacent node to the time required
to perform a floating-point operation. Using this value of $\tau$, we can calculate the estimated efficiency of the parallel algorithm as

$$E_{est} = \frac{T_{al}}{p(T_{ap} + \tau T_c)}.$$  \hspace{1cm} (5.32)

In order to compute the estimated efficiency we need to estimate the size of $K$, which is the number of arithmetic operations required to compute the scalar function $f(\cdot)$. Our experiments indicated that for most common functions, e.g., logarithm, square-root, trigonometric, and exponential functions, the value of $K$ is between 5 and 15. Thus we can take an average value $K = 10$.

We have implemented the parallel algorithm on a Meiko CS-2 multiprocessor with 16 processors, in which each node is a Sparc processor equipped with 256 MBytes of memory. In our experiments, we have computed the functions of matrices of dimensions ranging from 64 to 1536. In Figure 5.2, we give the actual (experimental) efficiency of the parallel algorithm as a function of the matrix dimension $n$, in which we have used the average of the timing values for computing matrix square-root, logarithm, and exponential functions. Expected (theoretical) efficiencies for the same
Figure 5.3. Theoretical efficiencies as a function of number of processors.

Figure 5.4. Efficiencies as a function of number of processors.
matrix sizes are given in Figure 5.3. For the theoretical efficiencies the model of equation 5.32 has been used. Our experiments showed that the ratio of the time required to transfer a floating-point number to an adjacent node to the time required to perform a floating-point operation, i.e. $\tau$ is approximately 35. In Figure 5.4 we give the experimental dependence of efficiency to the number of processors. Our experiments also indicate that the parallel algorithm obtains nearly constant efficiency (linear speedup) for $n > 400$.

5.5 Conclusions

We have presented a parallelization of Parlett’s algorithm for computing functions of upper triangular matrices. The parallel algorithm preserves the numerical properties of the serial algorithm, and is suitable for implementing on coarse-grain parallel computers. Our experiments on a 16 processor Meiko CS-2 have indicated that the parallel algorithm obtains nearly constant efficiency for matrices of size larger than 400.

The presented parallel algorithm computes an arbitrary function of an $n \times n$ upper triangular matrix in $O(n^2)$ time using $n$ processors. However, it is possible to compute the matrix function in $O(n \log n)$ time by parallel computation of the summation terms given by 5.27, which would require $O(n^2)$ processors [18]. It is an open question whether Parlett’s algorithm can be further parallelized, more specifically whether an $O(\log n)$-time parallel algorithm can be obtained, which uses Parlett’s summation 5.27. However, it is shown in [14] that the commutativity property and Bartels-Stewart algorithm for solving Sylvester’s equation yield a divide-and-conquer algorithm for computing functions of upper triangular matrices. The resulting algorithm requires approximately the same number of arithmetic operations as Parlett’s algorithm, and allows further parallelization. The parallel divide-and-conquer algorithm given in [17] requires $O(\log^3 n)$ time using $O(n^6)$ processors to compute an arbitrary function of an $n \times n$ upper triangular matrix. These two algorithms will
be analyzed in the next two chapters.
Chapter 6

A Divide-and-Conquer Algorithm for Functions of Triangular Matrices

6.1 Introduction

In the previous chapter we have analyzed the theory of matrix functions and introduced a new method to compute the functions of upper triangular matrices in parallel. In this chapter we propose a divide-and-conquer algorithm for computing arbitrary functions of upper triangular matrices, which requires approximately the same number of arithmetic operations as Parlett’s algorithm. However, the new algorithm has better performance on computers with two levels of memory due to its block structure and thus less memory-cache traffic requirements. The parallelization of this algorithm will be analyzed in the next chapter. The new algorithm also requires the eigenvalues of the input matrix be distinct, and computes the matrix function nearly as accurately as Parlett’s algorithm. Parlett has given an $O(n^3)$ algorithm for computing arbitrary functions of upper triangular matrices [45]. In fact, to the best of our knowledge, Parlett’s algorithm is the only algorithm known for performing this task. Parlett’s algorithm is derived using the Commutativity relationship of Equation 5.26 Parlett shows that by expanding the matrix multiplication and solving for $f_{ij}$ in the above, we obtain the following summation formula which is also given in Chapter 4

$$f_{ij} = t_{ij} \frac{f_{jj} - f_{ii}}{t_{jj} - t_{ii}} + \frac{1}{t_{jj} - t_{ii}} \sum_{k=i+1}^{j-1} (t_{ik}f_{kj} - f_{ikt_kj}) \quad (6.33)$$

Assuming a single scalar function evaluation requires $K$ arithmetic operations, evaluating the main diagonal requires $Kn$ arithmetic operations. The $L$th superdiagonal
contains \( n - L \) elements for \( L = 1, 2, \ldots, n - 1 \). Since the computation of each superdiagonal element requires \( 4L \) arithmetic operations, the number of arithmetic operations for computing \( F \) is found as

\[
Kn + \sum_{L=1}^{n-1} (n - L)(4L) = Kn + \frac{2}{3} (n^3 - n) .
\] (6.34)

We must remark that when \( T \) has repeated (or very close) eigenvalues, i.e., \( t_{ii} = t_{jj} \) (or \( t_{ii} \approx t_{jj} \) for some \( i \neq j \), Parlett's algorithm cannot be used (or will give inaccurate results). Alternative techniques for the repeated eigenvalue case are discussed in [45, 28].

In this chapter we provide a divide-and-conquer algorithm as an alternative to Parlett's algorithm. The new algorithm is of the same order of complexity as Parlett's algorithm, however, it seems to have some advantages. The algorithm is also derived from the commutativity relationship of Equation 5.26.

### 6.2 Derivation of the Algorithm

Let \( n = 2k \) and the matrices \( T \) and \( F \) be partitioned as

\[
T = \begin{bmatrix}
T_1 & T_2 \\
0 & T_3
\end{bmatrix}
\quad \text{and} \quad
F = \begin{bmatrix}
F_1 & F_2 \\
0 & F_3
\end{bmatrix},
\]

respectively. Here \( T_1, F_1 \in \mathbb{C}^{k \times k} \) and \( T_3, F_3 \in \mathbb{C}^{k \times k} \) are upper triangular, and \( T_2, F_2 \in \mathbb{C}^{k \times k} \) are full matrices. Here we use the commutativity relationship 5.26, and expand the matrix equation \( FT = TF \) in terms of the products of the matrix blocks as

\[
T_1 F_1 = F_1 T_1 ,
\]

\[
T_3 F_3 = F_3 T_3 ,
\]

\[
T_1 F_2 + T_2 F_3 = F_1 T_2 + F_2 T_3 .
\]

Since \( T_1 \) and \( T_3 \) are upper triangular, we have \( F_1 = f(T_1) \) and \( F_3 = f(T_3) \). Assuming \( F_1 \) and \( F_3 \) are already computed, we define \( C = F_1 T_2 - T_2 F_3 \), and proceed to solve
the matrix equation

\[ T_1 F_2 - F_2 T_3 = C \]  

(6.35)

in order to calculate \( F_2 \). This matrix equation is known as the Sylvester equation [28]. Let \( \lambda_i \) and \( \mu_i \) for \( i = 1, 2, \ldots, k \) be the distinct eigenvalues (diagonal elements) of \( T_1 \) and \( T_3 \). The Sylvester equation 6.35 has a unique solution \( F_2 \) if and only if \( \lambda_i \neq \mu_j \) for all \( i \) and \( j \). This unique solution can be found using Bartels-Stewart algorithm [5].

Thus, the divide-and-conquer matrix function evaluation algorithm first calls itself twice in order to compute the half-sized matrices \( F_1 = f(T_1) \) and \( F_3 = f(T_3) \), and then solves a Sylvester's equation using Bartels-Stewart algorithm in order to compute \( F_2 \). In Table 6.1, we give the recursive matrix function evaluation algorithm as a Matlab routine, which accepts the matrix \( T \) of size \( n \) (which is not required be a power of 2) and the function \( f(\cdot) \), and computes the upper triangular matrix \( F = f(T) \).

The subroutine \texttt{sylvester} in the above Matlab routine solves the Sylvester equation \( AX - XB = C \) using Bartels-Stewart algorithm, where \( A \in \mathbb{C}^{k \times k} \) and \( B \in \mathbb{C}^{m \times m} \) are upper triangular matrices and \( C \in \mathbb{C}^{k \times m} \) is a full matrix. Let \( C_i \) and \( X_i \) be the \( i \)th rows of the matrices \( C \) and \( X \), respectively. The Bartels-Stewart algorithm first solves the lower triangular system

\[
(a_{kk} I_m - B^T) X_k^T = C_k^T,
\]

(6.36)

and obtains \( X_k \), i.e., the last row of \( X \). The remaining rows of \( X \) are obtained by applying block back-substitution as

\[
(a_{ii} I_m - B^T) X_i^T = \left( C_i^T - \sum_{j=i+1}^{k} a_{ij} X_j^T \right)
\]

(6.37)

for \( i = k - 1, k - 2, \ldots, 1 \). Table 6.2 shows the Matlab routine for solving Sylvester's equation.

The new matrix function evaluation algorithm as given in Table 6.1 is a recursive algorithm, however, it can be 'unrolled' to obtain a non-recursive algorithm. Let \( n \)
Table 6.1. Matlab routine for the recursive function evaluation algorithm.

```
function f = tfun (t,fun)
[n,mm] = dim (t)
if n > 2
    f = feval (fun,t)
else
    m = floor (n/2); u = 1:m; v = m+1:n
    f1 = tfun (t(u,u),fun)
    f3 = tfun (t(v,v),fun)
    f2 = sylvester (t(u,u),t(v,v),f1*t(u,v)-t(u,v)*f3)
    f = [f1 f2;zeros(n-m,m) f3]
end
```
Table 6.2. Matlab routine to solve the Sylvester's equation.

```
function x = sylvester (a,b,c)

[k,kk] = dim (a)
[m,mm] = dim (b)
b = -b'
x = zeros(k,m)
x(k,:) = ((b+a(k,k)eye(m))/(c(k,:)))'
i = k-1:-1:1
for i = k-1:-1:1
    t = zeros(m,1)
    for j=i+1:k
        t = t + a(i,j)*(x(j,:))'
    end
    r = (c(i,:))' - t
    x(i,:) = ((b+a(i,i)*eye(m))/r)'
end
```

Figure 6.1. Computation of the square-root of an 8 × 8 matrix.

be a power of 2, i.e., \( n = 2^d \). The non-recursive algorithm first applies the function \( f \) to the main diagonal. It then goes through \( d \) steps for \( i = 1, 2, \ldots, d \). Prior to the \( i \)th step the evaluation of \( n/2^{i-1} \) matrix blocks (of dimension \( 2^{i-1} \times 2^{i-1} \)) in the main diagonal have been completed. During the \( i \)th step, the algorithm uses these \( n/2^{i-1} \) matrix blocks in pairs, and solves \( n/2^i \) Sylvester equations in order to obtain \( n/2^i \) matrix blocks (of dimension \( 2^i \times 2^i \)) required for the next step. The non-recursive algorithm is illustrated in Figure 6.1 for \( n = 8 \) and the square-root function.

We give the non-recursive algorithm in Table 6.3 as a Matlab routine. This routine accepts the upper triangular matrix \( T \) of any size and the function \( f(\cdot) \), and computes the upper triangular matrix \( F = f(T) \).

In the above routine, the function \( \text{mod}(a,b) \) returns the remainder of \( a \) divided by \( b \), and can be implemented in Matlab as

\[
\text{function } m = \text{mod}(a,b)
\]
Table 6.3. Non-recursive matrix function evaluation.

```matlab
function f = tfun(t,fun)
[n,mm] = size(t)
f = diag(feval(fun,diag(t)))
d = log(n)/log(2)
for i = 1:d
    s = 2^i
    for j = 0:n/s - 1
        u = j*s + 1 : j*s + s/2
        v = j*s + s/2 + 1 : (j + 1)*s
        f(u,v) = sylvester(t(u,u),t(v,v),f(u,u)*t(u,v)-t(u,v)*f(v,v))
    end
    if mod(n, s) = mod(n, 2*s)
        u = n - s - mod(n,s) + 1 : n - mod(n, s)
        v = n - mod(n, s) + 1 : n
        f(u,v) = sylvester(t(u,u),t(v,v),f(u,u)*t(u,v)-t(u,v)*f(v,v))
    end
end
end
```
$$m = a - \text{floor}(a/b) \times b;$$

### 6.3 Computational Complexity

In this section we analyze the computational complexity of the new algorithm for computing an arbitrary function of an \( n \times n \) triangular matrix \( T \). We will assume \( n = 2^d \) for simplicity of the analysis although the algorithm is suitable for any \( n \).

As seen in the Matlab routine given in Table 6.1, we apply the matrix function evaluation algorithm to each of the half-sized blocks \( F_1 = f(T_1) \) and \( F_3 = f(T_3) \) for a Sylvester matrix equation for \( F_2 \). Thus, the number of arithmetic operations required to compute \( F = f(T) \) for an \( n \times n \) matrix is given as

$$T(n) = 2T(n/2) + S(n/2) + U(n/2),$$

where \( S(k) \) is the number of arithmetic operations required to solve a Sylvester matrix equation of size \( k \), and \( U(k) \) is the number of arithmetic operations needed to compute the \( k \times k \) matrix \( C \) using \( C = F_1 T_2 - T_2 F_3 \), which is easily found to be \( U(k) = 2k^3 + k^2 \). When \( n = 1 \), the algorithm performs a scalar function evaluation \( f(\cdot) \), which we assume takes \( K \) arithmetic steps, i.e., \( T(1) = K \).

The Bartels-Stewart algorithm solves the Sylvester matrix equation by first obtaining \( X_k \) as given by Equation 6.36. The algorithm then proceeds to solve the remaining \( X_i \) for \( i = k - 1, k - 2, \ldots, 1 \) using Equation 6.37. As seen from the two nested loops in Table 6.2, there are \( (k - i) \) scalar-vector products, \( (k - i) \) vector sums, a single scalar addition to the main diagonal of the matrix \( B^T \). Finally, a lower triangular system of size \( k \) is solved. Thus, \( S(k) \) can be given as

$$S(k) = L(k) + k + \sum_{i=1}^{k-1} [2k(k - i) + k + L(k)] = k^3 + kL(k),$$

where \( L(k) \) is the number of arithmetic operations required to solve a lower triangular system of size \( k \), which is easily found as \( L(k) = k^2 \), and thus, \( S(k) = 2k^3 \). Therefore,
the divide-and-conquer algorithm requires

\[ T(n) = 2T(n/2) + \frac{n^3}{2} + \frac{n^2}{4} \]

arithmetic operations with the initial condition \( T(1) = K \). The solution of this recursion is found as

\[ T(n) = Kn + \frac{2n^3}{3} + \frac{n^2}{2} - \frac{7n}{6} \ . \quad (6.38) \]

This analysis applies to both versions (recursive and non-recursive) of the new algorithm. Comparing this figure to that of Parlett's algorithm given by 6.34, we conclude that the new algorithm requires approximately the same number of arithmetic operations. Although the arithmetic complexity of the two methods seems to be the same, we were surprised to observe that the new algorithm has a much better performance than Parlett's algorithm when implemented on a scientific workstation. Table 1 gives the timing results of the Parlett algorithm, the new recursive algorithm, and its non-recursive version for computing the square-root of matrices of size ranging from 8 to 1024. The Matlab (Version 4.1f) routines were run on a HP Apollo Workstation Model 735 with 256KB instruction and 256KB data caches, and 144 MB main memory. The clock speed of the processor is 99 MHz.

The reason behind this 'mysterious' speedup is explained as follows: Scientific workstations (as most computers) come with two levels of memory: the cache and the main memory. The cache is smaller and faster of these two, and if an element is not found in the cache, a whole block of data containing this element is brought from the main memory to the cache. If there is a large amount of data swaps between the cache and the main memory, then the computer spends much of its time performing these operations, and the performance degrades. Thus, it is crucial that we use the data in the cache as much as possible. Parlett's algorithm computes the elements of the matrix \( F \) one superdiagonal element at a time, and due to its data dependency requirements, it causes a large number of data swaps. The recursive and non-recursive algorithms presented in this chapter, on the other hand, are block
Table 6.4. Timing and speedup values for the algorithms.

<table>
<thead>
<tr>
<th>Size</th>
<th>Parlett Time (ms)</th>
<th>Recursive Time (ms)</th>
<th>Speedup</th>
<th>Non-Recursive Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.02</td>
<td>0.03</td>
<td>0.66</td>
<td>0.02</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>0.09</td>
<td>0.09</td>
<td>1.00</td>
<td>0.06</td>
<td>1.50</td>
</tr>
<tr>
<td>32</td>
<td>0.36</td>
<td>0.23</td>
<td>1.56</td>
<td>0.19</td>
<td>1.89</td>
</tr>
<tr>
<td>64</td>
<td>1.58</td>
<td>0.72</td>
<td>2.19</td>
<td>0.65</td>
<td>2.43</td>
</tr>
<tr>
<td>128</td>
<td>7.19</td>
<td>2.40</td>
<td>2.99</td>
<td>2.18</td>
<td>3.29</td>
</tr>
<tr>
<td>256</td>
<td>31.90</td>
<td>10.18</td>
<td>3.13</td>
<td>9.38</td>
<td>3.40</td>
</tr>
<tr>
<td>384</td>
<td>85.06</td>
<td>26.14</td>
<td>3.25</td>
<td>27.12</td>
<td>3.13</td>
</tr>
<tr>
<td>512</td>
<td>173.88</td>
<td>68.10</td>
<td>2.55</td>
<td>68.17</td>
<td>2.55</td>
</tr>
<tr>
<td>640</td>
<td>310.33</td>
<td>114.27</td>
<td>2.71</td>
<td>109.36</td>
<td>2.83</td>
</tr>
<tr>
<td>768</td>
<td>473.13</td>
<td>202.11</td>
<td>2.34</td>
<td>203.21</td>
<td>2.32</td>
</tr>
<tr>
<td>896</td>
<td>752.65</td>
<td>295.19</td>
<td>2.54</td>
<td>290.20</td>
<td>2.59</td>
</tr>
<tr>
<td>1024</td>
<td>1016.70</td>
<td>511.64</td>
<td>1.98</td>
<td>508.64</td>
<td>1.99</td>
</tr>
</tbody>
</table>
algorithms, and tend to use the data much longer before requiring a new data block. It was pointed by Golub and van Loan [28, Page 47] that

... computers having a cache tend to perform better on block algorithms.

In Appendix 1, we give a simplified analysis of data swaps between the cache and the main memory for Parlett's algorithm and the new algorithm. This analysis shows that the divide-and-conquer algorithm requires much fewer data swaps than Parlett’s algorithm, and thus is expected to run faster. Furthermore, the non-recursive algorithm has better performance than the recursive algorithm since the overhead of recursive function calls are avoided.

6.4 Numerical Experiments

We have performed some numerical experiments comparing the results of the divide-and-conquer algorithm to those of Parlett’s algorithm. In the first experiment, we have computed the square-root, cube-root, exponent, and logarithms of randomly generated upper-triangular $64 \times 64$ matrices $T$ with a selected eigenvalue separation $\min |t_{ii} - t_{jj}|$ for $1 \leq i, j \leq 64$. Let $\hat{F}$ and $F$ be the matrices computed by the divide-and-conquer and Parlett’s algorithms, respectively. Table 6.5 shows the relative error values computed by

$$\frac{\|\hat{F} - F\|}{\|F\|},$$

where $\|\cdot\|$ denotes the 2-norm of a matrix.

Also in Table 6.6, we compare the new algorithm to Parlett’s algorithm for computation of square-root and cube-root of upper triangular matrices. Here we calculate the relative error terms using

$$\frac{\|\hat{F}^2 - T\|}{\|T\|} \text{ and } \frac{\|F^2 - T\|}{\|T\|},$$

for the square-root function, and

$$\frac{\|\hat{F}^3 - T\|}{\|T\|} \text{ and } \frac{\|F^3 - T\|}{\|T\|},$$
Table 6.5. Error values for some matrix functions and eigenvalue separations.

|                | min $|t_{ii} - t_{jj}|$ |
|----------------|------------------|
|                | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ |
| square-root    | $4.27 \cdot 10^{-10}$ | $4.16 \cdot 10^{-9}$ | $1.02 \cdot 10^{-8}$ | $2.00 \cdot 10^{-7}$ |
| cube-root      | $4.02 \cdot 10^{-10}$ | $3.70 \cdot 10^{-9}$ | $8.11 \cdot 10^{-9}$ | $2.95 \cdot 10^{-7}$ |
| logarithm      | $8.99 \cdot 10^{-10}$ | $6.42 \cdot 10^{-9}$ | $6.68 \cdot 10^{-8}$ | $1.15 \cdot 10^{-7}$ |
| exponent       | $4.47 \cdot 10^{-15}$ | $2.14 \cdot 10^{-14}$ | $9.43 \cdot 10^{-14}$ | $9.90 \cdot 10^{-14}$ |

for the cube-root functions, where $\hat{F}$ and $F$ are the matrices computed by the divide-and-conquer and Parlett’s algorithms, respectively.

Examining these tables, we conclude that the new algorithm computes these matrix functions almost as accurately as Parlett’s algorithm, perhaps slightly less. Both algorithms produce poor results when the matrix $T$ has close eigenvalues. The numerical problems in the new algorithm are due to the solution of the Sylvester equation. It is shown in [25] that the error in the computed solution of the Sylvester equation can be given as

$$
\frac{\|\hat{F}_2 - F_2\|_f}{\|F_2\|_f} \leq 4u(\|T_1\|_f + \|T_3\|_f) \|\phi^{-1}\|_f,
$$

where $u$ denotes the unit roundoff, $\|\cdot\|_f$ is the Frobenius matrix norm, and

$$
\|\phi^{-1}\| = \left(\min_{X \neq 0} \frac{\|T_1 X - X T_3\|_f}{\|X\|_f}\right)^{-1}.
$$

It can be shown that

$$
\min_{X \neq 0} \frac{\|T_1 X - X T_3\|_f}{\|X\|_f} \leq \min |\lambda - \mu|,
$$

where $\lambda \in \sigma(T_1)$ and $\mu \in \sigma(T_3)$. Thus, the error in the computed solution of the Sylvester equation $\hat{F}_2$ grows as the eigenvalues of $T$ come close.
Table 6.6. Relative error for the square-root and cube-root functions.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>10⁻³</th>
<th>10⁻⁴</th>
<th>10⁻⁵</th>
<th>10⁻⁶</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>square-root</td>
<td>Parlett</td>
<td>1.70 · 10⁻⁸</td>
<td>1.64 · 10⁻⁷</td>
<td>1.54 · 10⁻⁶</td>
<td>1.14 · 10⁻⁵</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>7.08 · 10⁻⁸</td>
<td>7.82 · 10⁻⁷</td>
<td>4.56 · 10⁻⁶</td>
<td>1.14 · 10⁻⁵</td>
</tr>
<tr>
<td>cube-root</td>
<td>Parlett</td>
<td>1.78 · 10⁻¹¹</td>
<td>2.59 · 10⁻¹⁰</td>
<td>3.57 · 10⁻¹⁰</td>
<td>2.19 · 10⁻⁸</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>2.55 · 10⁻¹¹</td>
<td>5.53 · 10⁻¹⁰</td>
<td>2.18 · 10⁻¹⁰</td>
<td>5.11 · 10⁻⁸</td>
</tr>
</tbody>
</table>

6.5 Cache Performance of the Algorithm

Our analysis is similar to that of Golub and van Loan [28]. We partition the matrices $T$ and $F$ into blocks of rows such that each block contains $m$ rows. We assume that the cache can hold approximately $2m + 1$ rows, thus, only 1 block of $T$ and 1 block of $F$ is present in the cache at a given time. If an element of $T$ or $F$ is not found in the cache, then a whole block ($m$ rows) is loaded from the main memory. This operation is called a swap. In the following analysis we count the total number of swaps required by Parlett’s and the divide-and-conquer algorithms.

Parlett’s algorithm first computes the main diagonal entries of $F$, and proceeds by computing the superdiagonals one at a time for $L = 1, 2, \ldots, n - 1$. An element on the $L$th superdiagonal requires its horizontal and vertical neighbors [45]. In order to obtain the vertical neighbors, the algorithm requires approximately $L/m$ swap operations. Since there are $(n - L)$ elements on the $L$th superdiagonal, the total number of swaps is calculated as

$$
\sum_{L=1}^{n-1} (n - L) \frac{L}{m} = \frac{n^3 - n}{6m} \approx \frac{n^3}{6m}.
$$

On the other hand, the divide-and-conquer algorithm goes through $d = \log_2(n)$ steps for $i = 1, 2, \ldots, d$. During the $i$th step the algorithm performs $2 \times (n/2^i) =$
$2^{d-i+1}$ matrix products and $n/2^i = 2^{d-i}$ calls to the subroutine \texttt{svylvester} with matrices of size $2^{i-1} \times 2^{i-1}$. Let $\tau_1(k)$ and $\tau_2(k)$ be the number of swap operations required by the matrix product and Sylvester routines, respectively. Then, the number of swap operations required by the non-recursive matrix function evaluation algorithm is found as

$$\sum_{i=1}^{d} [2^{d-i+1} \tau_1(2^{i-1}) + 2^{d-i} \tau_2(2^{i-1})] .$$

It is shown in [28] that $\tau_1(k) = 2k/m + k^2/m^2$. In order to calculate $\tau_2(k)$, we take a closer look to the Matlab subroutine \texttt{svylvester}. As seen in Table 6.2, first a scalar is added to the diagonal elements of a $k \times k$ matrix. A single swap is required to obtain the scalar element $a(k, k)$, and $k/m$ swaps are required to add it to the diagonal of the matrix $b$. We use a single swap operation to obtain a row of $c$, while $k/m$ swaps are required to solve the lower triangular system. Therefore, the solution of the first system requires $2k/m + 2$ swap operations. Then, the Matlab routine \texttt{svylvester} solves $k-1$ such systems. The $j$ loop needs $k-i$ rows of $x$, which requires $(k-i)/m$ swap operations. There is a single swap operation to obtain $a(i, j)$ for all $j$. Similarly, there is a single swap operation to obtain the $i$th row of $c$. Finally, $2k/m + 2$ swap operations are required to obtain the solution of the lower triangular system. Thus, the total number of swap operations is found as

$$\tau_2(k) = 2 + \frac{2k}{m} + \sum_{i=1}^{k-1} \left( \frac{k-i}{m} + 4 + \frac{2k}{m} \right) = \frac{5k^2 - k}{2m} + 4(k-1) + 2 .$$

The total number of swap operations required by the non-recursive matrix function evaluation is then calculated as

$$\frac{5m+4}{4m^2} n^2 - \frac{8m^2+5m+4}{4m^2} n + \frac{8m^2+7m}{4m^2} n \log(n) + 2 \approx \frac{5n^2}{4m} . \quad (6.41)$$

Comparing 6.40 to 6.41, we conclude that the divide-and-conquer algorithm requires much fewer swap operations than Parlett's algorithm.
Chapter 7
A Parallel Algorithm for Functions of Triangular Matrices

In this chapter we present a parallelization of the divide-and-conquer algorithm for computing arbitrary functions of triangular matrices. The parallel algorithm requires the eigenvalues of the input matrix be distinct, and computes an arbitrary function of a triangular matrix in $O(\log^3 n)$ time using $P = O(n^6)$ processors.

7.1 Introduction

As we have introduced in the previous chapters, computing a function $f(A)$ of an $n$-by-$n$ matrix $A$ is an important problem in linear algebra, engineering and applied mathematics. Especially for transcendental functions of matrices, there are several methods including computing Jordan decomposition $A = MJM^{-1}$ and using the formula $f(A) = Mf(J)M^{-1}$, Schur decomposition, approximation methods such as Taylor expansion, rational Padé approximations, etc. The Jordan decomposition approach has several computational difficulties unless $A$ is diagonalizable and has a well-conditioned matrix of eigenvectors. Whereas Schur decomposition is more stable and can easily be applied for matrix function evaluation. If $A = QTQ^H$ is the Schur decomposition of a full matrix $A$ then

$$f(A) = Qf(T)Q^H.$$  

Therefore we need an effective algorithm for finding the matrix valued function of the triangular matrix $T$.

In this chapter we present a parallelization of the divide-and-conquer algorithm stated in the previous chapter. The functional expansion is the same as the sequential
version, hence the error properties are preserved. The parallel algorithm requires the eigenvalues of the input matrix be distinct, and computes an arbitrary function of a triangular matrix in $O(\log^3 n)$ time using $P = O(n^6)$ processors.

7.2 Derivation of the Algorithm

A divide and conquer algorithm making use of the commutativity relationship of Equation 5.26 has been proposed in [14]. This algorithm is of the same order of complexity as Parlett's algorithm, but the block structure of the algorithm makes it favorable to Parlett's method for computers with two levels of memory. The proposed parallel algorithm is based on this method.

Let $n = 2k$ and the matrices $T$ and $F$ be partitioned as

$$T = \begin{bmatrix} T_1 & T_2 \\ 0 & T_3 \end{bmatrix} \text{ and } F = \begin{bmatrix} F_1 & F_2 \\ 0 & F_3 \end{bmatrix},$$

respectively. Here $T_1, F_1 \in C^{k \times k}$ and $T_3, F_3 \in C^{k \times k}$ are upper triangular, and $T_2, F_2 \in C^{k \times k}$ are full matrices. Here we use the commutativity relationship 5.26, and expand the matrix equation $FT = TF$ in terms of the products of the matrix blocks as

$$T_1 F_1 = F_1 T_1,$$

$$T_3 F_3 = F_3 T_3,$$

$$T_1 F_2 + T_2 F_3 = F_1 T_2 + F_2 T_3.$$

Since $T_1$ and $T_3$ are upper triangular, we have $F_1 = f(T_1)$ and $F_3 = f(T_3)$. Assuming $F_1$ and $F_2$ are already computed, we define $C = F_1 T_2 - T_2 F_3$, and proceed to solve the matrix equation

$$T_1 F_2 - F_2 T_3 = C \quad (7.42)$$

in order to calculate $F_2$. This matrix equation is known as the Sylvester equation [28]. Let $\lambda_i$ and $\mu_i$ for $i = 1, 2, \ldots, k$ be the distinct eigenvalues (diagonal elements) of $T_1$ and $T_3$. The Sylvester equation 7.42 has a unique solution $F_2$ if and only
if \( \lambda_i \neq \mu_j \) for all \( i \) and \( j \). This unique solution can be found using Bartels-Stewart algorithm [5] or Kronecker product method [6]. Both of the two algorithms are \( O(n^3) \) when performed sequentially, and a detailed analysis of the solution for the specific case of upper triangular coefficient matrices have been given in [17].

The new matrix function evaluation algorithm as given in Table 6.1 is a recursive algorithm, however, it can be ‘unrolled’ to obtain a non-recursive algorithm. The progression of the algorithm is similar to the inversion of triangular matrices in [41]. Unwinding the recursion to the lowest level and then building back up again, we produce a simple \( \log n \) phase algorithm for finding \( f(T) \). Let \( n \) be a power of 2, i.e., \( n = 2^d \). The non-recursive algorithm first applies the function \( f \) to the main diagonal. After obtaining the scalar function of the main diagonal, in the first phase the algorithm solves a scalar Sylvester equation which is a linear equation in one unknown \( f_{i,i+1} \),

\[
t_{ii}f_{i,i+1} - f_{i,i+1}t_{i+1,i+1} = f_{i,i+1}t_{i,i+1} - t_{i,i+1}f_{i+1,i+1} \quad \text{for} \quad i = 1, 3, 5, \ldots n - 1 .
\]

It then goes through \( d \) steps for \( k = 2, \ldots, d \) solving Sylvester equations at each step. Prior to the \( k \)th step the evaluation of \( n/2^{k-1} \) matrix blocks (of dimension \( 2^{k-1} \times 2^{k-1} \)) in the main diagonal have been completed. During the \( k \)th step, the algorithm uses these \( n/2^{k-1} \) matrix blocks in pairs, and solves \( n/2^k \) Sylvester equations in order to obtain \( n/2^k \) matrix blocks (of dimension \( 2^k \times 2^k \)) required for the next step. The total number of arithmetic operations for the unrolled divide and conquer algorithm can be given as

\[
T(n) = Kn + \sum_{k=0}^{d-1} \frac{n}{2^k}S(2^k) + \frac{n}{2^{k+1}}U(2^k) = Kn + \frac{2n^3}{3} + \frac{n^2}{2} - \frac{7n}{6} ,
\]

where \( S(n) \) is the number of arithmetic operations required to a Sylvester matrix equation of size \( n \), and \( U(n) \) is the number of arithmetic operations needed to compute the \( n \times n \) matrix \( C \) using \( C = F_1T_2 - T_2F_3 \), which are found as \( S(n) = 2n^3 \) and \( U(n) = 2n^3 + n^2 \) [17].
7.3 The New Parallel Algorithm

We have shown that the divide-and-conquer type algorithm for computing the functions of triangular matrices is a log n phase algorithm, but needs to solve a Sylvester equation of size \(2^k \times 2^k\) at its \(k\)th step. We need to parallelize the solution of the Sylvester equation to assure parallelism at each step of the algorithm.

The proposed algorithm for solving Sylvester's equation is based on the Kronecker product algorithm. Solution of the Sylvester's equation \(AX + XB = C\), where \(A \in \mathcal{R}^{m \times m}, B \in \mathcal{R}^{n \times n}\) are upper triangular matrices and \(C \in \mathcal{R}^{m \times n}\) is a full matrix, is equivalent to solving the \(m^2 \times m^2\) linear equation

\[
H \tilde{X} = \tilde{C},
\]

where \(\tilde{X}\) is a \(m^2 \times 1\) vector formed by stacking the transposed rows of the matrix \(X\). Also \(H\) is an \(m^2 \times m^2\) matrix such that \(H = A \otimes I + I \otimes B^T\), where \(\otimes\) is the Kronecker (or tensor) product. In terms of the matrix blocks the Kronecker product matrix can be represented as \(H = T_1 \otimes I - I \otimes T_3^T\). For example, for \(m = 4\), we have

\[
H = \begin{bmatrix}
    a_{11}I + B^T & a_{12}I & a_{13}I & a_{14}I \\
    0 & a_{22}I + B^T & a_{23}I & a_{24}I \\
    0 & 0 & a_{33}I + B^T & a_{34}I \\
    0 & 0 & 0 & a_{44}I + B^T
\end{bmatrix}
\]

The structure of \(H\) can be exploited to design a parallel algorithm for the solution of the equation \(H \tilde{X} = \tilde{C}\). This algorithm will be similar to the parallel inversion of triangular matrices \([7]\). Let \(D\) be the \(m^2 \times m^2\) diagonal matrix such that \(D_{ii} = H_{ii}\) for \(i = 1, 2, \ldots, m^2\). Let \(J = D^{-1}H\), and \(U = I - J\), where \(U\) is an \(m^2 \times m^2\) matrix with diagonal elements all zero. It can easily be proven that \(U^i = 0\) for \(i \geq 2m - 1\). We will try to analyze this property of the block upper triangular matrix \(U\). The
general form of $U$ be represented as

$$U = \begin{bmatrix}
L_{11} & a_{12}I & a_{13}I & \cdots & a_{1m}I \\
0 & L_{22} & a_{23}I & \cdots & a_{2m}I \\
0 & 0 & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \ddots & L_{mm}
\end{bmatrix}$$

where the block diagonal element $L_{ii}$ is a $n \times n$ lower triangular matrix with zero entries in the main diagonal. The $k$th power of $U$ can be represented as

$$U^k = \begin{bmatrix}
P_{11} & P_{12} & \cdots & P_{1m} \\
0 & P_{22} & \cdots & P_{2m} \\
0 & 0 & \ddots & \vdots \\
0 & 0 & \cdots & P_{mm}
\end{bmatrix},$$

where $P_{ii} = L_{ii}^k$. The structure of $L_{ii}$ can be given as

$$L_{ii} = \begin{bmatrix}
0 & 0 & 0 & \cdots & 0 \\
x & 0 & 0 & \cdots & 0 \\
x & x & \ddots & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & 0 \\
x & \cdots & x & x & 0
\end{bmatrix},$$

where $x$ denotes the nonzero entries of the matrix. The consecutive powers of $L_{ii}$ can be given as

$$L_{ii}^2 = \begin{bmatrix}
0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 \\
x & 0 & \cdots & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & 0 \\
x & \cdots & x & x & 0
\end{bmatrix}, \quad L_{ii}^{m-1} = \begin{bmatrix}
0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & 0 \\
0 & 0 & 0 & \cdots & 0 \\
x & 0 & \cdots & 0 & 0
\end{bmatrix}, \quad L_{ii}^m = 0$$

Therefore for $k \geq m$ the main diagonal matrix blocks can be represented as zero.
matrices, and the block structure of $U^m$ becomes

$$
U^m = \begin{bmatrix}
0 & P'_1 & \cdots & P'_{1m} \\
0 & 0 & \cdots & P'_{2m} \\
0 & 0 & \ddots & \vdots \\
0 & 0 & \cdots & 0
\end{bmatrix}.
$$

Now we can easily prove that

$$J^{-1} = I + U + U^2 + \cdots + U^{2m-2}.$$

By multiplying both sides by $J = I - U$ and expanding the product terms we obtain $JJ^{-1} = I$, which proves the summation shown above. The zero blocks on the diagonal can be utilized to zero out the superdiagonal blocks, and finally $U^{2m-1}$ becomes an $m^2 \times m^2$ zero matrix. In order to compute the summation, we can use the algorithm given in [40] which computes the matrix polynomial $I + Y + Y^2 + \cdots + Y^{n-1}$ using $2[\log_2 n] - 1$ matrix multiplications and $[\log_2 n]$ matrix additions.

As shown in Table 6.1, the divide-and-conquer algorithm solves $n/2^k$ Sylvester equations to obtain the $n/2^k$ unknown blocks of the matrix at the $k$th step of the algorithm where $k = 1, 2, \ldots, \log n - 1$. Let $m = 2^k$ be the size of the matrix blocks at the $k$th step of the algorithm and $p(m) = m^3$ be the number of processors required to multiply two $m \times m$ square matrices in $O(\log m)$ time. Since the size of the Kronecker product matrices are $m^2 \times m^2$, the parallel solution of the linear equation 7.43 needs $p(m^2)$ processors. The coefficient matrices of the Sylvester equation are obtained independently. Therefore the total number of processors needed at the $k$th step of the algorithm can be found as

$$P = \sum_{i=1}^{n/m} p(m^2) = nm^5,$$

where $m = 2^k$. The maximum size of the Sylvester equation is $n/2 \times n/2$, hence the maximum number of processors becomes

$$P = \frac{n^6}{32}.$$
The total number of arithmetic operations at the $k$th step is found as $O(\log^2 m)$.

**Theorem 7.1** Given $P = O(n^6)$ processors, the divide-and-conquer method requires $O(\log^3 n)$ arithmetic steps to compute an arbitrary function of an $n \times n$ triangular matrix.

The algorithm is of divide and conquer type and the arithmetic complexity of each step of the algorithm depends on that of the half sized problem plus parallel solution of the linear system $H\vec{X} = \vec{C}$. The parallel block upper triangular linear system solution is shown to require $p = O(n^6)$ processors and $T_i = O(\log^2 2^i)$ at each step. The following summation gives the total number of arithmetic operations to compute the function of the upper triangular matrix $T$

$$T(n) = \sum_{i=1}^{\log n} \log 2^i = \frac{\log n(\log n - 1)(2\log n - 1)}{6},$$

thus we obtain the total algebraic complexity of the algorithm as $O(\log^3 n)$.

### 7.4 An Example

We can apply this algorithm to a four by four example. Let's consider the square-root of the following four-by-four matrix

$$T = \begin{bmatrix}
16 & -15 & -76 & -14 \\
0 & 1 & -50 & 14 \\
0 & 0 & 81 & -44 \\
0 & 0 & 0 & 4
\end{bmatrix}$$

The main diagonal elements can be obtained by applying the square root function. The first superdiagonal is obtained by solving the scalar Sylvester equation, which is in fact a linear equation in one unknown;

$$f_{12} = \frac{f_{11}t_{12} - t_{12}f_{22}}{t_{11} - t_{22}} = -3$$

$$f_{34} = \frac{f_{33}t_{34} - t_{34}f_{44}}{t_{33} - t_{44}} = -4$$
Since the algorithm is log depth, the second and last step will be computing the sign of the whole matrix. The matrix blocks can be represented as

\[ T_1 = \begin{bmatrix} 16 & -15 \\ 0 & 1 \end{bmatrix}, \quad T_2 = \begin{bmatrix} -76 & -14 \\ -50 & 14 \end{bmatrix}, \quad T_3 = \begin{bmatrix} 81 & -44 \\ 0 & 4 \end{bmatrix}, \]

and the computed matrix blocks of the \( F \) matrix can be shown as

\[ F_1 = \begin{bmatrix} 4 & -3 \\ 0 & 1 \end{bmatrix}, \quad F_3 = \begin{bmatrix} 9 & -4 \\ 0 & 2 \end{bmatrix}. \]

The Kronecker product matrix \( H \) can be represented as

\[
H = T_1 \otimes I - I \otimes T_3^T = \begin{bmatrix}
-65 & 0 & -15 & 0 \\
44 & 12 & 0 & -15 \\
0 & 0 & -80 & 0 \\
0 & 0 & 44 & -3
\end{bmatrix},
\]

and \( J \) becomes

\[
J = D^{-1}H = \begin{bmatrix}
1 & 0 & 0.2308 & 0 \\
3.6667 & 1 & 0 & -1.25 \\
0 & 0 & 1 & 0 \\
0 & 0 & -14.6667 & 1
\end{bmatrix}.
\]

Removing the unity elements along the diagonal we obtain \( U = I - J \). The inverse of \( J \) can be found by the power method

\[
J^{-1} = I + U + U^2 = \begin{bmatrix}
1 & 0 & -0.2308 & 0 \\
-3.6667 & 1 & 19.1795 & 1.25 \\
0 & 0 & 1 & 0 \\
0 & 0 & 14.6667 & 1
\end{bmatrix},
\]

and \( H^{-1} \) becomes

\[
H^{-1} = J^{-1}D^{-1} = \begin{bmatrix}
-0.0154 & 0 & 0.0029 & 0 \\
0.0564 & 0.0833 & -0.2397 & -0.4167 \\
0 & 0 & -0.0125 & 0 \\
0 & 0 & -0.1833 & -0.3333
\end{bmatrix}.
\]
We need to compute \( C \) as follows

\[
C = F_1 T_2 - T_2 F_3 = \begin{bmatrix} 530 & -374 \\ 400 & -214 \end{bmatrix},
\]

and \( \tilde{C} \) is

\[
\tilde{C} = \begin{bmatrix} 530 & -374 & 400 & -214 \end{bmatrix}^T.
\]

Multiplying \( \tilde{C} \) with \( H^{-1} \) we obtain \( \tilde{F}_2 \)

\[
\tilde{F}_2 = H^{-1} \tilde{C} = \begin{bmatrix} -7 & -8 & -5 & -2 \end{bmatrix}^T.
\]

This solves for the unknown block of the \( F \) matrix as

\[
F_2 = \begin{bmatrix} -7 & -8 \\ -5 & -2 \end{bmatrix}.
\]
Chapter 8
Conclusion

A parallel matrix sign function algorithm is introduced and its efficiency with respect to the existing algorithms is compared. A generalization of this method for various sectors in the complex plane is proposed and its convergence properties with respect to the existing algorithms are analyzed. A parallel iterative algorithm for computing the principal \( n \)th roots of symmetric matrices is introduced and implementation results on a Meiko CS-2 is included. Three algorithms for computing arbitrary functions of triangular matrices are introduced. Parallelization of an existing algorithm for computing arbitrary functions of triangular matrices is developed. A new divide-and-conquer algorithm and its parallelization for computing functions of triangular matrices are also developed.

Real-time signal processing, teletraffic, solution of Markov chains and solution of nonlinear matrix equations are some of the few applications of spectrum slicing algorithms that needs to be investigated. Another field of interest is investigating rational Padé approximation methods for matrix sector functions. Finally we believe that implementation of the proposed divide-and-conquer function evaluation algorithm on an MPP platform will be an interesting experimental study.
BIBLIOGRAPHY


