A Divide-and-Conquer Algorithm for Functions of Triangular Matrices [*]

Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Technical Report, June 1996

Abstract

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. Like Parlett's algorithm, the new algorithm also requires that the eigenvalues (main diagonal elements) of the input matrix be distinct, and computes the matrix function nearly as accurately.

1 Introduction

Let A be an $n \times n$ matrix with entries from the real or complex field, and $f(\cdot)$ be an analytic function. We are interested in computing the matrix function $f(A)$. Specific methods have been developed for specific functions, e.g., matrix square-root, matrix sign function, matrix exponential, and so on. The preferred approach for computing an arbitrary function of a square matrix is via Schur decomposition: We first decompose A as $A = QTQ^H$, and then compute $f(A) = Qf(T)Q^H$, where T is an upper triangular matrix [3]. This way the computation of $f(A)$ for an arbitrary matrix A is reduced to the computation of $F = f(T)$ for an upper triangular matrix T. Parlett has given an $O(n^3)$ algorithm for computing arbitrary functions of upper triangular matrices [5]. 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 property that the matrices T and F commute, i.e.,

$$FT = TF \ . \tag{1}$$

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

$$f_{ij} = t_{ij} \, \frac{f_{jj} - f_{ii}}{t_{jj} - t_{ii}} + \frac{1}{t_{jj} - t_{ii}} \sum_{k=i+1}^{j-1} (t_{ik} f_{kj} - f_{ik} t_{kj}) \ . \tag{2}$$

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., $f_{ii} = f(t_{ii})$.

[*]This work is supported in part by the National Science Foundation under grant ECS–9312240.

This step requires Kn arithmetic operations, assuming a single scalar function evaluation requires K arithmetic operations. After computing the main diagonal entries, the algorithm computes the superdiagonals one at a time, using the summation expression (2). The Lth 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

$$Kn + \sum_{L=1}^{n-1}(n-L)(4L) \ = \ Kn + \frac{2}{3}\,(n^3 - n)\ . \tag{3}$$

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 [5, 3].

In this paper we provide a divide-and-conquer algorithm as an alternative to Parlett's algorithm, which is also derived from the commutativity relationship (1). The new algorithm is of the same order of complexity as Parlett's algorithm, however, it seems to have some advantages.

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 $T_1, F_1 \in \mathcal{C}^{k\times k}$ and $T_3, F_3 \in \mathcal{C}^{k\times k}$ are upper triangular, and $T_2, F_2 \in \mathcal{C}^{k\times k}$ are full matrices. Here we use the commutativity relationship (1), and expand the matrix equation $FT = TF$ in terms of the products of the matrix blocks as

$$\begin{aligned} T_1F_1 &= F_1T_1\ , \\ T_3F_3 &= F_3T_3\ , \\ T_1F_2 + T_2F_3 &= F_1T_2 + F_2T_3\ . \end{aligned}$$

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 have already been computed, we calculate $C = F_1T_2 - T_2F_3$, and proceed to solve the matrix equation

$$T_1F_2 - F_2T_3 = C \tag{4}$$

in order to compute F_2. This matrix equation is known as the Sylvester equation [3]. Let λ_i and μ_i for $i = 1, 2, \ldots, k$ be the eigenvalues (diagonal elements) of T_1 and T_3, respectively. The Sylvester equation (4) 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 the Bartels-Stewart algorithm [1].

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 the Bartels-Stewart algorithm in order to compute F_2. In Figure 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 $F = f(T)$.

Figure 1: Recursive matrix function evaluation.

```
function f = tfun(t,fun);
[n,mm] = size(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
```

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

$$(a_{kk}I_m - B^T)X_k^T = C_k^T \ , \tag{5}$$

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

$$(a_{ii}I_m - B^T)X_i^T = \left(C_i^T - \sum_{j=i+1}^{k} a_{ij}X_j^T \right) \tag{6}$$

for $i = k-1, k-2, \ldots, 1$. We give the Matlab routine in Figure 2.

Figure 2: The Bartels-Stewart algorithm for the Sylvester equation.

```
function x = sylvester(a,b,c);
[k,kk] = size(a);
[m,mm] = size(b);
b = -b';
x = zeros(k,m);
x(k,:) = ((b+a(k,k)*eye(m))\(c(k,:))')';
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
```

The new matrix function evaluation algorithm as given in Figure 1 is a recursive algorithm, however, it can be 'unrolled' to obtain a non-recursive algorithm. 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. It then goes through d steps for $i = 1, 2, \ldots, d$. Prior to the ith step the evaluation of $n/2^{i-1}$ matrix blocks (of dimension $2^{i-1} \times 2^{i-1}$) in the main diagonal has been completed. During the ith 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 3 for $n = 8$ and the square-root function.

Figure 3: Computation of the square-root of an 8×8 matrix.

$$\begin{tikzcd}
\begin{matrix}
16 & -15 & -76 & -6 & 4 & 28 & -30 & -20 \\
0 & 1 & -50 & 16 & -52 & -39 & 17 & -84 \\
0 & 0 & 81 & -40 & 48 & 104 & 34 & 126 \\
0 & 0 & 0 & 1 & 32 & 18 & -32 & 29 \\
0 & 0 & 0 & 0 & 49 & 112 & -6 & 89 \\
0 & 0 & 0 & 0 & 0 & 81 & 60 & -51 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 36
\end{matrix} \arrow[r] &
\begin{matrix}
4 & -15 & -76 & -6 & 4 & 28 & -30 & -20 \\
0 & 1 & -50 & 16 & -52 & -39 & 17 & -84 \\
0 & 0 & 9 & -40 & 48 & 104 & 34 & 126 \\
0 & 0 & 0 & 1 & 32 & 18 & -32 & 29 \\
0 & 0 & 0 & 0 & 7 & 112 & -6 & 89 \\
0 & 0 & 0 & 0 & 0 & 9 & 60 & -51 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6
\end{matrix} \arrow[d] \\
\begin{matrix}
4 & -3 & -7 & -8 & 4 & 28 & -30 & -20 \\
0 & 1 & -5 & -2 & -52 & -39 & 17 & -84 \\
0 & 0 & 9 & -4 & 48 & 104 & 34 & 126 \\
0 & 0 & 0 & 1 & 32 & 18 & -32 & 29 \\
0 & 0 & 0 & 0 & 7 & 7 & -6 & 8 \\
0 & 0 & 0 & 0 & 0 & 9 & 6 & -3 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6
\end{matrix} \arrow[d] &
\begin{matrix}
4 & -3 & -76 & -6 & 4 & 28 & -30 & -20 \\
0 & 1 & -50 & 16 & -52 & -39 & 17 & -84 \\
0 & 0 & 9 & -4 & 48 & 104 & 34 & 126 \\
0 & 0 & 0 & 1 & 32 & 18 & -32 & 29 \\
0 & 0 & 0 & 0 & 7 & 7 & -6 & 89 \\
0 & 0 & 0 & 0 & 0 & 9 & 60 & -51 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6
\end{matrix} \arrow[l] \\
\begin{matrix}
4 & -3 & -7 & -8 & 5 & 1 & 5 & -2 \\
0 & 1 & -5 & -2 & -3 & 0 & 6 & -3 \\
0 & 0 & 9 & -4 & 4 & 4 & 3 & 7 \\
0 & 0 & 0 & 1 & 4 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 7 & 7 & -6 & 8 \\
0 & 0 & 0 & 0 & 0 & 9 & 6 & -3 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6
\end{matrix} &
\end{tikzcd}$$

We give the non-recursive algorithm in Figure 4 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 $F = f(T)$.

Figure 4: The non-recursive matrix function evaluation.

```
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) & i ~= d
       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
```

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

```
function m = mod(a,b)
m = a - floor(a/b)*b;
```

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 analysis although the algorithm is suitable for any n. As seen in the Matlab routine given in Figure 1, we apply the matrix function evaluation algorithm to each of the half-sized blocks $F_1 = f(T_1)$ and $F_3 = f(T_3)$, and then solve a Sylvester matrix equation in order to compute F_2. Thus, the number of arithmetic operations required to compute $F = f(T)$ for an $n \times n$ matrix is given as

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

where $S(k)$ is the number of arithmetic operations required for solving 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 $C = F_1T_2 - T_2F_3$, 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 (5). The algorithm then proceeds to solve the remaining X_i for $i = k-1, k-2, \ldots, 1$ using Equation (6). As seen from the two nested loops in Figure 2, there are $(k-i)$ scalar-vector products, $(k-i)$ vector sums, and 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

$$T(n) = Kn + \frac{2n^3}{3} + \frac{n^2}{2} - \frac{7n}{6} \ . \tag{7}$$

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 (3), we conclude that the new algorithm requires approximately the same number of arithmetic operations. Although we found the arithmetic complexity of the two methods 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 Parlett's algorithm, the new recursive algorithm, and its non-recursive version for computing the square-root of matrices of size ranging from 8 to 1024. We used the Matlab function `funm.m`, which implements Parlett's algorithm, and the routines given in Figures 1, 2, and 3. The Matlab (Version 4.1) routines were run on an 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.

Table 1: Timing and speedup values for the algorithms.

	Parlett	Recursive		Non-Recursive	
Size	Time (ms)	Time (ms)	Speedup	Time (ms)	Speedup
8	0.02	0.03	0.66	0.02	1.00
16	0.09	0.09	1.00	0.06	1.50
32	0.36	0.23	1.56	0.19	1.89
64	1.58	0.72	2.19	0.65	2.43
128	7.19	2.40	2.99	2.18	3.29
256	31.90	10.18	3.13	9.38	3.40
384	85.06	26.14	3.25	27.12	3.13
512	173.88	68.10	2.55	68.17	2.55
640	310.33	114.27	2.71	109.36	2.83
768	473.13	202.11	2.34	203.21	2.32
896	752.65	295.19	2.54	290.20	2.59
1024	1016.70	511.64	1.98	508.64	1.99

As can be seen from Table 1, the recursive and non-recursive versions of the new algorithm is up to 3 times faster than Parlett's algorithm. The reason behind this 'mysterious' speedup is due to the block structure of the new algorithm. Scientific workstations (and most computers) come with two levels of memory: the cache and the main memory. The cache is the 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 swapping between the cache and the main memory, then the computer spends much of its time performing these operations, and the performance is degraded. 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 requires a large number of data swaps due to its data dependency requirements.

The recursive and non-recursive algorithms presented in this paper, on the other hand, are block algorithms, and tend to use the data much longer before requiring a new data block. It was pointed out by Golub and van Loan [3, Page 47] that

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

In Appendix I, 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 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.

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 64×64 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 2 shows the relative error values computed by $\hat{F} - F\|/\|F\|$, where $\| \cdot \|$ denotes the 2-norm of a matrix.

Table 2: 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	$4.47 \cdot 10^{-15}$	$2.14 \cdot 10^{-14}$	$9.43 \cdot 10^{-14}$	$9.90 \cdot 10^{-14}$

Also in Table 3, 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 $\|\hat{F}^2 - T\|/\|T\|$ and $\|F^2 - T\|/\|T\|$ for the square-root function, and $\|\hat{F}^3 - T\|/\|T\|$ and $\|F^3 - T\|/\|T\|$ for the cube-root function. Here $\hat{F}$ and F are the matrices computed by the divide-and-conquer and Parlett's algorithms, respectively.

Table 3: Relative error for the square-root and cube-root functions.

		$\min \|t_{ii} - t_{jj}\|$			
		10^{-3}	10^{-4}	10^{-5}	10^{-6}
square-root	Parlett	$1.70 \cdot 10^{-8}$	$1.64 \cdot 10^{-7}$	$1.54 \cdot 10^{-6}$	$1.14 \cdot 10^{-5}$
	New	$7.08 \cdot 10^{-8}$	$7.82 \cdot 10^{-7}$	$4.56 \cdot 10^{-6}$	$1.14 \cdot 10^{-5}$
cube-root	Parlett	$1.78 \cdot 10^{-11}$	$2.59 \cdot 10^{-10}$	$3.57 \cdot 10^{-10}$	$2.19 \cdot 10^{-8}$
	New	$2.55 \cdot 10^{-11}$	$5.53 \cdot 10^{-10}$	$2.18 \cdot 10^{-10}$	$5.11 \cdot 10^{-8}$

Examining the tables, we conclude that the new algorithm computes these matrix functions almost as accurately as Parlett's algorithm, perhaps slightly less. Parlett's algorithm and the new algorithm both 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 [2] that the error in the computed solution of the Sylvester equation satisfies

$$\frac{\|\hat{F}_2 - F_2\|_f}{\|F_2\|_f} \leq 4u(\|T_1\|_f + \|T_3\|_f) \ \|\phi^{-1}\| \ , \tag{8}$$

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.

5 Conclusions

We have given a divide-and-conquer algorithm for computing functions of upper triangular matrices. The algorithm requires approximately the same number of arithmetic operations as Parlett's algorithm, however, runs up to 3 times faster on computers having a cache due to its block structure. The numerical properties of the algorithm seem to be similar to those of Parlett's algorithm.

Finally we note that the divide-and-conquer algorithm can be parallelized to compute an arbitrary function of an $n \times n$ triangular matrix in $O(\log^3 n)$ time using $O(n^6)$ processors [4].

References

[1] R. H. Bartels and G. W. Stewart. Solution of the matrix equation $AX + XB = C$. *Communications of the ACM*, 15(9):820–826, 1972.

[2] G. H. Golub, S. Nash, and C. F. van Loan. A Hessenger-Schur method for the problem $AX + XB = C$. *IEEE Transactions on Automatic Control*, 24(6):909–913, December 1979.

[3] G. H. Golub and C. F. van Loan. *Matrix Computations.* Baltimore, MD: The Johns Hopkins University Press, 3rd edition, 1996.

[4] Ç. K. Koç and B. Bakkaloğlu. A parallel algorithm for functions of triangular matrices. *Computing*, 57(1):85–92, 1996.

[5] B. N. Parlett. A recurrence among the elements of functions of triangular matrices. *Linear Algebra and its Applications*, 14:117–121, 1976.

Appendix I

Our analysis is similar to that of Golub and van Loan [3]. 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 Lth superdiagonal requires its horizontal and vertical neighbors [5]. In order to obtain the vertical neighbors, the algorithm requires approximately L/m swap operations. Since there are $(n-L)$ elements on the Lth superdiagonal, the total number of swaps is calculated as

$$\sum_{L=1}^{n-1} (n-L)\frac{L}{m} = \frac{n^3-n}{6m} \approx \frac{n^3}{6m} \ . \tag{9}$$

On the other hand, the divide-and-conquer algorithm goes through $d = \log_2(n)$ steps for $i = 1, 2, \ldots, d$. During the ith 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 `sylvester` 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 [3] that $\tau_1(k) = 2k/m + k^2/m^2$. In order to calculate $\tau_2(k)$, we take a closer look at the Matlab subroutine `sylvester` given in Figure 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, $k-1$ such systems is solved. The j loop needs $k-i$ rows of x $(k-i)/m$ 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 ith 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

$$\frac{5m+4}{4m^2}\, n^2 - \frac{8m^2+5m+4}{4m^2}\, n + \frac{8m^2+7m}{4m^2}\, n\log(n) + 2 \approx \frac{5n^2}{4m} \ . \tag{10}$$

Comparing (9) to (10), we conclude that the divide-and-conquer algorithm requires fewer swap operations than Parlett's algorithm.