# Efficient Evaluation of Matrix Polynomials*

Niv Hoffman[1], Oded Schwartz[2], and Sivan Toledo[1]

[1] Tel-Aviv University
[2] The Hebrew University of Jerusalem

**Abstract.** We revisit the problem of evaluating matrix polynomials and introduce memory and communication efficient algorithms. Our algorithms, based on that of Patterson and Stockmeyer, are more efficient than previous ones, while being as memory-efficient as Van Loan's variant. We supplement our theoretical analysis of the algorithms, with matching lower bounds and with experimental results showing that our algorithms outperform existing ones.

**Keywords:** Polynomial evaluation, Matrix polynomials, Cache efficiency

## 1 Introduction

In the early 1970s, Patterson and Stockmeyer discovered a surprising, elegant, and very efficient algorithm to evaluate a matrix polynomial [**?**]. Later in the 1970s, Van Loan showed how to reduce the memory consumption of their algorithm [**?**]. There has not been any significant progress in this area since, in spite of dramatic changes in computer architecture and in closely-related algorithmic problems, and in spite of continued interest and applicability [**?**, Section 4.2].

This paper revisits the problem and applies to it both cache-miss reduction methods and new algorithmic tools. Our main contributions are:

- We develop a new block variant of Van-Loan's algorithm, which is usually almost as memory-efficient as Van-Loan's original variant, but much faster.
- We develop two algorithms that reduce the matrix to its Schur form, to speed up the computation relative to both Patterson and Stockmeyer's original algorithm and Van Loan's variants, including the new block variant. One algorithm exploits the fact that multiplying triangular matrices is faster (by up to a factor of 6) than multiplying dense square matrices. The other algorithm partitions the problem into a collection of smaller ones using a relatively recent algorithm due to Davies and Higham.

* This research is supported by grants 1878/14, 1901/14, 965/15 and 863/15 from the Israel Science Foundation, grant 3-10891 from the Israeli Ministry of Science and Technology, by the Einstein and Minerva Foundations, by the PetaCloud consortium, by the Intel Collaborative Research Institute for Computational Intelligence, by a grant from the US-Israel Bi-national Science Foundation, and by the HUJI Cyber Security Research Center.

– We analyze the number of cache misses that the main variants generate, thereby addressing a major cost on modern architecture. The analysis is theoretical and it explains our experimental results, discussed below.
– We evaluate the performance of the direct algorithms (the ones that do not reduce the matrix to Schur form), both existing and new, pinpointing algorithms that are particularly effective.
– We predict the performance of algorithms that reduce the matrix to Schur form using an empirically-based performance model of the performance of their building blocks.

## 2 Building Blocks

This section presents existing algorithmic building blocks for evaluating polynomials of a matrix. We denote the $n$-by-$n$ real or complex input matrix by $A$, the polynomial by $q$, and we assume that it is given by its coefficient $c_0, c_1, \ldots, c_d$. That is, we wish to compute the matrix

$$q(A) = c_0 I + c_1 A + c_2 A^2 + \cdots + c_d A^d \ .$$

We assume that the polynomial is dense, in the sense that either no $c_i$s are zero or too few to be worth exploiting.

*Matrix Multiplication.* Many of the algorithms that we discuss in this paper call matrix multiplication routines. Classical matrix multiplication performs about $2n^3$ arithmetic operations and highly optimized routines are available in Level 3 BLAS libraries [**?**] (DGEMM for double precision numbers).

If the matrix is triangular, classical matrix multiplication performs only about $n^3/3$ operations. We are not interested in polynomials of matrices with special structures, but as we will see below, evaluation of a polynomial of a general matrix can be reduced to evaluation of the same polynomial but of a triangular matrix. Unfortunately, the Level 3 BLAS does not include a routine for multiplying two triangular matrices.

So-called fast matrix multiplication algorithms reduce the asymptotic cost of matrix multiplication to $O(n^{\log_2 7})$. We denote the exponent in fast methods by $\omega_0$; For the Strassen [**?**] and Strassen-Wingograd [**?**] methods $\omega_0 = \log_2 7 \approx 2.81$. The constants of the leading coefficient of these algorithms are larger than those of classical matrix multiplications, but some variants are faster than classical matrix multiplication on matrices of moderate dimensions [**?**,**?**]. Fast algorithms are not as stable as classical ones, and in particular cannot attain elementwise backward stability, but they can attain normwise backward stability [**?**,**?**,**?**]. BLAS libraries do not contain fast matrix multiplication routines, but such routines have been implemented in a number of libraries [**?**,**?**].

One of the algorithms that we discuss multiplies square matrices by vectors or by blocks of vectors. Multiplying a matrix by a vector requires about $2n^2$ arithmetic operations and cannot benefit from Strassen-like fast algorithms.

Multiplying an $n$-by-$n$ matrix by an $n$-by-$k$ matrix requires about $2n^2k$ operations classically, or $O(n^2k^{\omega_0-2})$ if a fast method is used to multiply blocks of dimension $k$.

*Naive Polynomial Evaluation.* There are two naive ways to evaluate a polynomial given its coefficients. One is to construct the explicit powers of $A$ from $A^2$ up to $A^d$ by repeatedly multiplying $A^{k-1}$ by $A$, and to accumulate the polynomial along the way, starting from $Q = c_0 I$ and adding $Q = Q + c_k A^k$ for $k = 1$ to $d$. The other is Horner's rule, which starts with $Q = c_d A + c_{d-1} I$ and repeatedly multiplies $Q$ by $A$ and adds a scaled identity $Q \leftarrow QA + c_k I$ for $k = d-2$ down to 0.

Both methods perform $d-1$ matrix multiplications. The explicit-powers methods also needs to perform matrix scale-and-add operations, whereas Horner only adds a constant to the diagonal of the current $Q$. The explicit-powers method stores two matrices in memory, in addition to $A$, whereas Horner only needs to store one. Clearly, both of these methods can exploit any specialized matrix-matrix multiplication routine, including fast methods.

*Patterson-Stockmeyer Polynomial Evaluation.* Patterson and Stockmeyer discovered a method, which we denote by PS, for evaluating $q(A)$ using only about $2\sqrt{d}$ matrix-matrix multiplications, as opposed to $d-1$ in naive methods [**?**]. The method splits the monomials in $q$ into $s$ consecutive subsets of $p$ monomials each, and represents each subset as a polynomial of degree $p-1$ (or less) in $A$ times a power of $A^p$. Assuming that $d+1 = ps$, we have

$$\begin{aligned} q(A) = c_0 I + c_1 A + \cdots + c_{p-1} A^{p-1} \\ + A^p \left( c_p I + c_{p+1} A + \cdots + c_{2p-1} A^{p-1} \right) + \cdots + \\ + \left(A^p\right)^{s-1} \left( c_{(s-1)p} I + c_{(s-1)p+1} A + \cdots + c_{(s-1)p+p-1} A^{p-1} \right) \ . \end{aligned} \tag{1}$$

In the general case in which $p$ does not divide $d+1$ the last subset has fewer than $p$ terms; if $q$ is sparse, other subsets might have fewer than $p$ terms. In other words, the method represents $q$ as a degree-$(s-1)$ polynomial in $A^p$, in which the coefficients are polynomials of degree $p-1$.

The method computes and stores $A^2, \ldots, A^p$, and it also stores $A$. It then applies the explicit-powers method to compute each degree-$(p-1)$ coefficient polynomial (without computing the powers of $A$ again, of course) and uses Horner's rule to evaluate the polynomial in $A^p$.

Assuming that $p$ divides $d+1$, the total number of matrix multiplications that the method performs is $(p-1)+(s-1) = p+s-2$, the number of matrix scale-and-add operations is $(p-1)s$, and the number of matrices that are stored is $(p-1)+1+1+1 = p+2$. Arithmetic is minimizes by minimizing $p+s$; this happens near $p \approx s \approx \sqrt{d}$.

Note that any matrix multiplication algorithm can be used here, and that if $A$ is triangular, so are all the intermediate matrices that the algorithm computes.

Van Loan proposed a variant of the PS method that requires less memory [**?**]. The algorithm, denoted PS-MV, exploits the fact that polynomials in $A$ and powers of $A$ commute to construct one column of $q(A)$ at a time by applying (**??**) to

**Algorithm 1** Van Loan's memory-efficient version (PS-MV) of the Patterson-Stockmeyer (PS) method.

Compute $A^p$ ($\log_2 p$ matrix-multiplications)
For $j \leftarrow 1, \ldots, n$
  Compute $Ae_j, \ldots, A^{p-1}e_j$ ($p-2$ matrix-vector multiplications)
  Set $Q_j \leftarrow \sum_{\ell=0}^{p-1} c_{d-p+1+\ell} A^\ell e_j$ (vector operations)
  For $k \leftarrow s-1, \ldots, 1, 0$ multiply and add $Q_j \leftarrow A^p Q_j + \sum_{\ell=0}^{p-1} c_{d-kp+\ell+1} A^\ell e_j$
    ($s$ matrix-vector multiplications, $ps$ vector operations)

one unit vector at a time. The algorithm first computes $A^p$ by repeated squaring. Then for every $j$, it computes and stores $Ae_j, \ldots A^{p-1}e_j$ and accumulates $q(A)_{:,j}$ using Horner's rule. The algorithm is presented more formally in Algorithm **??**. The number of artithmetic operations is a little higher than in PS, because of the computation of $A^p$ by repeated squaring. The method stores three matrices, $A$, $A^p$, and the accumulated polynomial, as well as $p-1$ vectors.

*Reduction to Triangular Matrices.* Any square matrix $A$ can be reduced to a Schur form $A = QTQ^*$ where $Q$ is unitary and $T$ is upper triangular [**?**]. When $A$ is real, $T$ may be complex, then one can use the so-called real Schur form in which $T$ is real block upper triangular with 1-by-1 blocks and 2-by-2 blocks. The computation of the Schur form (or the real Schur form) costs $\Theta(n^3)$ arithmetic operations with a fairly large hidden constant; we explore this constant in Section **??**.

*The Parlett-Higham-Davies Method for Triangular Matrices.* Parlett [**?**] discovered that any function $f$ of a triangular matrix $T$ can be computed by substitution as long as its eigenvalues are simple (recall that the diagonal of a triangular matrix contains its eigenvalues). If the eigenvalues are simple but clustered, the method divides by small values and may become numerically unstable. Higham [**?**] generalized Parlett's method to block matrices, in which substitution steps solve a Sylvester equation. These Sylvester equations have a unique solution only if different diagonal blocks do not share eigenvalues, and nearby diagonal values in two blocks cause instability.

Davies and Higham [**?**] developed an algorithm that partitions the eigenvalues of a triangular matrix $T$ into well separated clusters. The algorithm then uses a unitary similarity to transform $T$ into a triangular matrix $T'$ in which the clusters are consecutive, computes the function of diagonal blocks (using a different algorithm), and then uses the block Parlett recurrence to solve for all the off-diagonal blocks of $f(T')$. Because the clusters are well separated, the Sylvester equations that define the off-diagonal blocks are well conditioned. Davies and Higham proposed to use Padé approximations to compute the function of diagonal blocks, but other methods can be used as well (in our context, appropriate methods include Horner's rule, PS, etc.).

## 3 New Algorithms from Existing Building Blocks

Next, we show how building blocks described in Section **??** can be used to construct new algorithms that are more efficient in some settings.

*The Block Patterson-Stockmeyer-Van Loan Algorithm.* In this variant, rather than computing one column of $q(A)$ at a time, we compute $m$ columns at a time. The expressions are a trivial generalization: we replace $e_j = I_{:,j}$ in Algorithm **??** by $I_{:,j:j+m-1}$. This increases the memory requirements to three matrices and $m(p-1)$ vectors. The number of arithmetic operations does not change, but the memory access pattern does; we analyze this aspect below.

*Utilizing Fast Matrix Multiplication.* The naive methods and the PS method are rich in matrix-matrix multiplications; one can replace the classical matrix-multiplication routine with a fast Strassen-like method. Van Loan's PS-MV method cannot benefit from fast matrix multiplication, but the block version can (with savings that are dependent on the block size $m$).

*Simple Schur Methods.* Given the Schur form of $A$, we can express $q(A)$ as $q(A) = q(QTQ^*) = Qq(T)Q^*$. Several methods can be used to evaluate $q(T)$. Because $T$ is triangular, evaluating $q$ on it is generally cheaper than evaluating $q$ on $A$ directly. Whether the savings are worth the cost of computing the Schur form depends on $d$ and the method that is used to evaluate $q$.

Because there are no restrictions on how $q(T)$ is evaluated, this approach is applicable to all representations of $q$, not only to representations by its coefficients. In particular, this approach can be applied to Newton and other interpolating polynomials.

*Parlett-Davies-Higham Hybrids.* If the eigenvalues of $A$ are well separated and the original Schur-Parlett method can be applied, the total arithmetic cost is $O(n^3 + dn)$, where the $dn$ term represents the cost of evaluating $q$ on the eigenvalues of $A$ (on the diagonal of the Schur factor) and the $n^3$ term represents the cost of the reduction to Schur form and the cost of Parlett's recurrence for the offdiagonal elements of $T$. For large values of $d$, this cost may be significantly smaller than in alternative methods in which $d$ or $\sqrt{d}$ multiply a non-linear term (e.g., smaller than $O(\sqrt{d}n^{\omega_0})$ for PS using fast matrix multiplication).

If the eigenvalues of $A$ are not well separated, we can still apply the Parlett-Davies-Higham method to compute off-diagonal blocks of $q(T)$; any non-Parlett method can be used to compute the diagonal blocks, including PS and its variants. In particular, in this case the diagonal blocks are triangular and so will be all the intermediate matrices in PS.

*Remainder Methods.* We note that if we ignore the subtleties of floating-point arithmetic, there is never a need to evaluate $q(A)$ for $d \geq n$. To see why, suppose that $d \geq n$ and let $\chi(A)$ be its characteristic polynomial. We can divide $q$ by $\chi$,

$q(A) = \chi(A)\delta(A) + \rho(A) = \rho(A)$, where $\rho$ is the remainder polynomial of degree at most $n-1$. The second equality holds because $\chi(A) = 0$.

However, it is not clear whether there exist a numerically-stable method to find and evaluate the remainder polynomial $\rho$. We leave this question for future research.

## 4 Theoretical Performance Analyses

We now present a summary of the theoretical performance metrics of the different algorithms.

*Analysis of Post-Reduction Evaluations.* Table **??** summarizes the main performance metrics associated with explicit and PS methods. The table shows the number of matrix multiplications that the methods perform, the amount of memory that they use, and the asymptotic number of cache misses that they generate in a two-level memory model with a cache of size $M$. We elaborate on the model below.

The number of matrix multiplications performed by most of the methods, with the exception of PS-MV variants, require no explanation. Van Loan's original variant performs $\log_2 p$ matrix-matrix multiplications to produce $A^p$, and it also performs matrix-vector multiplications that constitute together $p+s+1$ matrix-matrix multiplications; the latter are denoted in the table as $2\sqrt{d}_{\text{conv}}$, under the assumption that $p \approx \sqrt{d}$. In the block PS-MV variant, the same matrix-matrix multiplications appear as matrix-matrix multiplications involving an $n$-by-$n$ matrix and an $n$-by-$b$ matrix. The memory requirements need no further explanation.

**Table 1.** The main performance metrics associated with explicit and PS methods. The expressions in the cache-misses column assume a relatively small cache size $M$ and classical (non-Strassen) matrix multiplications. The subscripts conv and conv($b$) signify that the methods use repeated matrix-vector multiplications or matrix-matrix multiplication with one small dimension, so the choice of matrix multiplication method is not free.

| Algorithm | Work (# MMs) | Memory (# words) | Cache Misses |
|---|---|---|---|
| Explicit Powers | $d-1$ | $3n^2$ | $O\left(d \cdot \frac{n^3}{\sqrt{M}} + dn^2\right)$ |
| Horner's Rule | $d-1$ | $2n^2$ | $O\left(d \cdot \frac{n^3}{\sqrt{M}} + dn^2\right)$ |
| PS | $p+s-1$ | $(p+1)\,n^2$ | $O\left((p+s-1) \cdot \frac{n^3}{\sqrt{M}} + dn^2\right)$ |
| PS for $p \approx \sqrt{d}$ | $\approx 2\sqrt{d}$ | $\approx \sqrt{d}n^2$ | $O\left(\sqrt{d} \cdot \frac{n^3}{\sqrt{M}} + dn^2\right)$ |
| PS MV | $\log_2 \sqrt{d} + 2\sqrt{d}_{\text{conv}}$ | $3n^2 + \sqrt{d}n$ | $O\left(\sqrt{d} \cdot n^3\right)$ |
| Block PS MV $b \approx \sqrt{M/3}$ | $\log_2 \sqrt{d} + 2\sqrt{d}_{\text{conv}(b)}$ | $3n^2 + \sqrt{d}bn$ | $O\left(\sqrt{d} \cdot \frac{n^3}{\sqrt{M}} + dn^2\right)$ |

We now analyze the number of cache misses generated by each method. Our analysis assumes a two-level memory hierarchy with a cache (fast memory) consisting of $M$ words that are used optimally by a scheduler, and it assumes that every cache miss transfers one word into the cache. It has been widely accepted that this model captures reasonably well actual caches as long as the mapping of addresses to cache lines has high associativity and as long as cache lines are reasonably small (the so-called tall-cache assumption [**?**]).

The cache-miss analysis of the explicit powers, Horner's rule and PS is simple: they perform $d-1$ or $p+s-1$ matrix-matrix multiplications and $d$ matrix scale-and-add operations. Clearly, the scale-and-add operations generate at most $O(dn^2)$ cache misses. If the cache is smaller than the memory requirement of the algorithm by a factor of 2 or more, the scale-and-add will generate this many misses. Similarly, if matrix multiplications are performed using a classical algorithm, the number of cache misses in each multiplication can be reduced to $O(n^3/\sqrt{M})$ [**?**], which implies the correctness of the bounds in the table. The lower bound for classical matrix multiplication is $\Omega(n^3/\sqrt{M})$ [**?**,**?**,**?**]. We can apply the same lower bounds to an entire polynomial evaluation algorithm, by employing the imposed-reads/writes technique in [**?**]. Thus, if the polynomial evaluation algorithm involves $t$ applications of dense $n$-by-$n$ matrix multiplications, then the cost of cache misses is bounded below by $\Omega(tn^3/\sqrt{M})$. If we use a matrix multiplication that performs $\Theta(n^{\omega_0})$ operations, the matrix multiplication cache miss bound reduces to $O(n^{\omega_0}/M^{\omega_0/2-1})$ per matrix multiplication [**?**] and $t$ times that for an entire polynomial evaluation algorithm that invokes fast matrix multiplication $t$ times.

Van-Loan's original PS-MV generates a very large number of cache misses, except for one scenario in which it is very efficient. If $M \geq 3n^2 + \sqrt{d}n$, then all the data structures of the algorithm fit into the cache and it combines a minimal number of operations (among known algorithms) with only $O(n^2)$ compulsory cache misses. If $M$ is small, say $M < n^2$, this variant has essentially no reuse of data in the cache, with $\Theta(\sqrt{d}n^3)$ misses, more than in PS, and with terrible performance in practice.

The memory requirements and cache-miss behavior of our block PS-MV depend on the block size $b$. If we set $b \approx \sqrt{M/3}$, the small dimension in matrix multiplication operations is $b$ which guarantees an asymptotically optimal number of cache misses in matrix multiplications (the multiplications can be performed in blocks with dimension $\Theta(\sqrt{M})$). This leads to the same overall asymptotic number of cache misses as the original Patterson-Stockmeyer algorithm, but requires much less memory. This is an outcome of the fact that when we multiply an $n$-by-$n$ matrix by an $n$-by-$b$ matrix, the data reuse rate improves as $b$ grows from 1 to about $\sqrt{M/3}$ but stays the same afterwards. If classical matrix multiplication is used, this variant is theoretically the most promising: it pays with $\log_2 \sqrt{d}$ matrix multiplications (insignificant relative to $\sqrt{d}$) to reduce the storage requirements dramatically while attaining a minimal number of cache misses. One can apply a fast matrix multiplication algorithm in this case

too, reducing both the arithmetic costs and the cache misses; we omit details due to lack of space.

*Algorithms that Reduce to Schur Form.* In principle, reduction to Schur form is worth its cost for $d$ larger than some constant, but the constant is large because the reduction performs $\Theta(n^3)$ operations with a large multiplier of $n^3$. If $d$ is large enough, it should pay off to reduce the matrix to Schur form and to apply $q$ to the triangular factor using a PS variant; the cost of each matrix multiplication is then reduced by a factor of 6, assuming classical matrix multiplication. If we multiply triangular matrices recursively and use Strassen to multiply full blocks within this recursion, the savings is by a factor of $15/4 = 3.75$. There is a cache-miss efficient algorithm to carry out the reduction .

The decision whether to apply a Parlett-based recurrence to the Schur factor is more complex. It is always worth sorting the eigenvalues and running the Davies-Higham algorithm that clusters the eigenvalues. If they are separated well enough that there are $n$ singleton clusters, Parlett's algorithm can be applied directly at a cost of $O(dn + n^3)$; the algorithm can be applied recursively as in [**?**]. If the eigenvalues are clustered, we need to weigh the cost of the Schur reordering versus the savings from applying the polynomial to diagonal blocks of the Schur form (rather than to the entire Schur factor); solving the Sylvester equations will cost an additional $O(n^3)$ and will achieve good cache efficiency [**?**]. The cost of the Schur reordering is $O(n^3)$ but can be much smaller in special cases; see [**?**] for details.

## 5 Experimental Evaluation

We performed numerical experiments to validate our theoretical results and to provide a quantitative context for them. The experiments were all conducted on a computer running Linux (Kernel version 4.6.3) with an Intel i7-4770 quad-core processor running at 3.4 GHz. The processor has an 8 MB shared L3 cache and four 256 KB L2 caches and four and 32 KB L1 caches (one per core). The computer had 16 GB of RAM and it did not use any swap space or paging area. Our codes are all implemented in C. They were compiled using the Intel C compiler version 17.0.1 and were linked with the Intel Math Kernel Library (MKL) version 2017 update 1. Our code is sequential, but unless otherwise noted, MKL used all four cores.

The input matrices that we used were all random and orthonormal. We used orthonormal matrices to avoid overflow in high-degree polynomials. The code treats the matrices as general and does not exploit their orthonormality.

Figure **??** (top row) presents the performance of Horner and Patterson-Stockmeyer for a range of matrix sizes and for polynomials of a fixed degree $d = 100$. The results show that PS is up to 4 times faster than Horner's rule, but that PS-MV is very slow (by factors of up to about 28). The computational rate (floating-point operations per second) of Horner is the highest, because the vast majority of the operations that it performs are matrix-matrix multiplications;

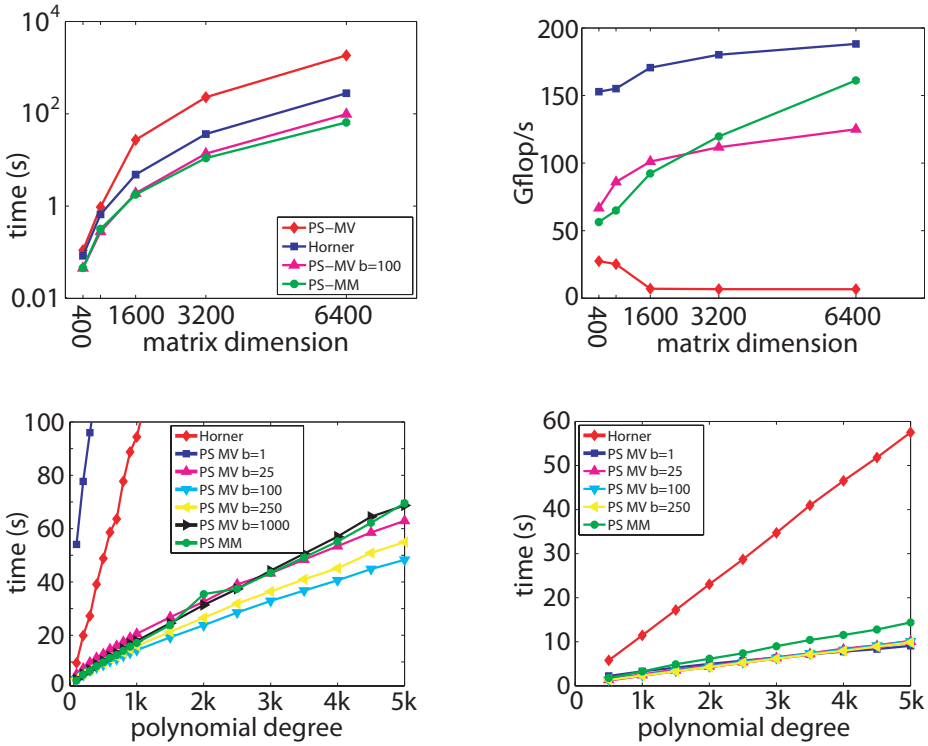


**Fig. 1.** The running times of PS methods. The degree of the polynomial in the top row is $d = 100$. On the bottom row, the dimensions of the matrices are 2000 on the left and 750 on the right.

its performance is close to the performance of DGEMM on a given machine. The computational rate of matrix-matrix PS is lower and it increases significantly as the dimension grows from 400 to 6400. The increase is due to the increasing fraction of operations that are performed within the matrix-matrix multiplication routine (the fraction performed within the matrix addition routine decreases). Even on matrices of dimension 6400 (and 12000 in Figure **??**), the computational rate of PS is significantly lower than that of Horner.

The performance of matrix-vector PS is not only low, it gets worse as matrix dimension grows. On small matrices ($n = 400$ and $n = 800$) the method still enjoys some data reuse in the cache, but on matrices of dimension $n \geq 1600$ that do not fit within the L3 cache performance is uniformly slow; performance is limited by main memory bandwidth.

However, the performance of the block version of the matrix-vector PS is good, close to the performance of matrix-matrix PS. Figure **??**(bottom row) explores this in more detail. We can see that on a large matrices, the matrix-vector variant is very slow; the matrix-matrix variant is much faster. However, the block matrix-vector algorithm is even faster when an effective block size is

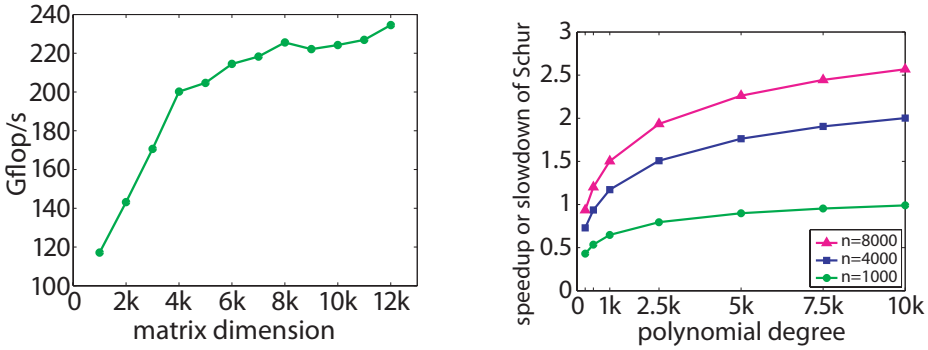


**Fig. 2.** The performance of PS as a function of the dimension for polynomials of degree $d = 10$ (left). The graph on the right shows *predictions* of performance gains (or losses) due to reducing $A$ to Schur form. See text for details.

chosen. In this experiment, the best block size is $b = 100$; other block sizes did not perform as well, but we can also see that performance is not very sensitive to the block size. The memory savings relative to the matrix-matrix variant are large (a factor of about $n/b$).

On a matrix that fits into the level-3 cache ($n = 750$), the matrix-vector algorithms are all significantly faster than the matrix-matrix variant, but the performance is again insensitive to the specific block size.

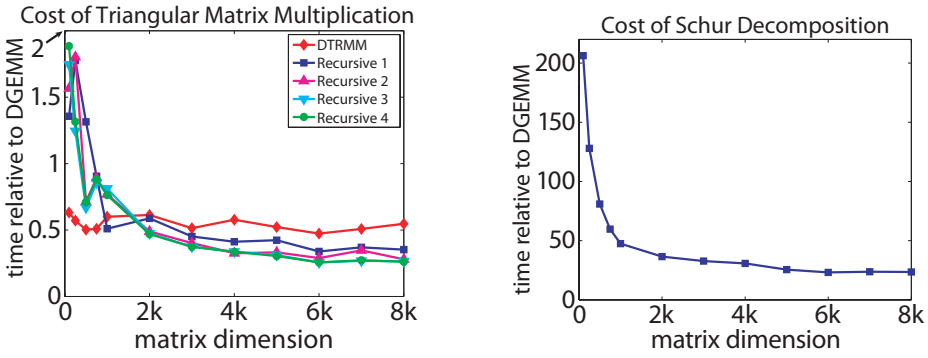


**Fig. 3.** The running times of building blocks of polynomial evaluation methods that reduce the matrix to Schur form. The graphs on the left show the running times of functions that multiply two triangular matrices relative to those of the general matrix multiplication routine DGEMM. The graph on the right shows the running tims of the Schur decomposition routines in LAPACK (DGEHRD, DORGHR, and DHSEQR), again relative to that of DGEMM. The triangular multipliers that we evaluate are DTRMM, a BLAS3 routine that multiplies a triangular matrix by a square one, and four recursive routines that we wrote. Our routines perform between 1 and 4 levels of recursion and then call DTRMM.

The performance of methods that reduce $A$ to a Schur form depend on the cost of the Schur reduction relative to the savings that the Schur form can generate. Figure **??** show that the time to compute Schur decomposition of large matrices is about 23 times higher than the time it takes to multiply matrices (the ratio is higher for small matrices). The figure also shows that multiplying large triangular matrices using a specialized routine takes a factor of 3.8 less time than calling DGEMM; this does not match the operation count (a factor of 6), but it still represents a significant saving.

Figure **??** (right) combines the measurements that we used to plot the graphs in Figure **??** into a performance model that predicts the savings (or losses) generated by reducing $A$ to a Schur form for various values of $n$ and $d$. Specifically, we estimated the performance of square PS as $2\sqrt{d}T_{\text{DGEMM}}$, where $T_{\text{DGEMM}}$ is the empirical time used in Figure **??**, and we estimated the performance of the Schur reduction and triangular PS as $T_{\text{SCHUR}} + 2\sqrt{d}T_{\text{REC(4)}} + 2T_{\text{DGEMM}}$ where $T_{\text{SCHUR}}$ is the empirical time of the Schur reduction (the values that were used in Figure **??**, left graph) and $T_{\text{REC(4)}}$ is the empirical running time of our triangular multiplier with a 4-level recursion. We added two matrix multiplications to the cost of the Schur approach to model the post multiplication of $q(T)$ by the two unitary Schur factors. Clearly, the gains are limited to about a factor of 6, which is the best gain we can hope for in triangular matrix multiplication; the actual ratio is smaller both because the empirical performance of triangular matrix multiplication is not 6 times better than that of DGEMM, and because the Schur reduction itself is expensive.

## 6 Conclusions and Open Problems

Our theoretical and experimental analyses lead to three main conclusions:

1. Our new block variant of the PS method is essentially always faster than both the original PS method and Van Loan's PS-MV. It also uses much less memory than PS and not much more than PS-MV. This variant is also much faster than Horner's rule and similar naive methods.
2. On large matrices and moderate degrees, the performance of fast PS variants is determined mostly by the performance of the matrix-multiplication routine that they use. Therefore, using fast matrix multiplication is likely to be effective on such problems.
3. On large matrices and high degrees, it is worth reducing the matrix to its Schur form. This is true even if the polynomial of the Schur factor is evaluated without first partitioning it using the Davies-Higham method. Although we have not implemented the partitioning method, it is likely to achieve additional savings.

## References

1. G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz. Improving the numerical stability of fast matrix multiplication. *SIAM J. Matrix Anal. Appl.*, 37:1382–1418, 2016.

2. G. Ballard, J. Demmel, O. Holtz, , and O. Schwartz. Minimizing communication in linear algebra. *SIAM J. Matrix Anal. Appl.*, 32:866–901, 2011.
3. G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. of the ACM*, 59:32, 2012.
4. A. R. Benson and G. Ballard. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices*, 50:42–53, 2015.
5. D. Bini and G. Lotti. Stability of fast algorithms for matrix multiplication. *Numerische Mathematik*, 36:63–72, 1980.
6. P. I. Davies and N. J. Higham. A Schur–Parlett algorithm for computing matrix functions. *SIAM J. Matrix Anal. Appl.*, 25:464–485, 2003.
7. E. Deadman, N. J. Higham, and R. Ralha. Blocked schur algorithms for computing the matrix square root. In P. Manninen and P. Öster, editors, *Proc. of the 11th Int. Conf. on Applied Parallel and Sci. Comput. (PARA)*, pages 171–182, 2013.
8. J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg. Fast matrix multiplication is stable. *Numerische Mathematik*, 106:199–224, 2007.
9. J. J. Dongarra, J. D. Cruz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.
10. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.
11. G. Golub and C. V. Loan. *Matrix Computations*. Johns Hopkins, 4th edition, 2013.
12. N. J. Higham. *Functions of Matrices: Theory and Algorithm*. SIAM, 2008.
13. J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn. Implementing Strassen's algorithm with BLIS. *arXiv preprint arXiv:1605.01078*, 2016.
14. J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 690–701. IEEE, 2016.
15. D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. of Par. and Dist. Comput.*, 64:1017–1026, 2004.
16. H. Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 326–333, New York, NY, USA, 1981. ACM.
17. I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular systems: Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. *ACM Transactions on Mathematical Software*, 28:416–435, 2002.
18. D. Kressner. Block algorithms for reordering standard and generalized schur forms. *ACM Transactions on Mathematical Software*, 32:521–532, 2006.
19. C. F. V. Loan. A note on the evaluation of matrix polynomials. *IEEE Trans. Automat. Control*, AC-24:320–321, 1979.
20. B. N. Parlett. Computation of functions of triangular matrices. Memorandum ERL-M481, Electronics Research Laboratory, UC Berkeley, Nov. 1974.
21. M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2:60–66, 1973.
22. V. Strassen. Gaussian elimination is not optimal. *Num. Math.*, 13:354–356, 1969.
23. S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society, 1999.
24. S. Winograd. On multiplication of 2-by-2 matrices. *Linear Algebra and Appl.*, 4:381–388, 1971.