Fast and Parallel Evaluation of Matrix Polynomials

Sivan Toledo, Amit Waisel

August 19, 2018

Abstract

We designed, implemented and evaluated a parallel algorithm for calculating matrix polynomials. The algorithms has some variations, which we will go over in this article. The results, compared to other implementations, are promising.

1 Introduction

This paper introduces some parallel and efficient algorithms for fast matrix polynomial calculations. We describe numerous approaches for calculating the polynomials, and analyze the implications of different properties of the input matrices. Our algorithm uses existing algorithms as building blocks, as discussed below. Our contributions are:

1. Adapt existing algorithms for handling the real-form of the polynomial instead of dealing with complex numbers
2. Develop new algorithms for reducing overall calculation costs. The existing algorithms reduce the matrix to real-schur-form, and cluster the eigenvalues on its diagonal for better concurrency operation. We will analyze the implications of sorting the eigenvalues and show a way to minimize the swaps.
3. Analyze and evaluate different aspects of the algorithms used, such as clustering tolerance, solving Sylvester equations in parallel and the order in which the sub-calculations are executed by.

Many methods for computing matrix functions require the evaluation of a matrix polynomial. We denote the $n \times n$ real or complex input matrix by A, the polynomial by q, and we assume that it is given by its coefficient $c_0, c_1, ..., c_d$. That is, we wish to compute the matrix:

$$q(A) = \sum_{k=0}^{m} c_k \cdot A^k = c_o \cdot I + c_1 \cdot A + c_2 \cdot A^2 + ... + c_d \cdot A^d \qquad (1)$$

where $A \in \mathbb{C}^{n \times n}$ (or $A \in \mathbb{R}^{n \times n}$). 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.

Using Matlab as a baseline, our algorithms shows very promising results, especially for high-rank polynomials on high-rank matrices.

2 Existing Algorithms

This section presents existing algorithmic building blocks for evaluating polynomials of a matrix.

2.1 Matrix Multiplication Methods

Matrix multiplication is a fundamental part in all of the algorithm variations described below. The classic and straight-forward matrix multiplication operation takes about $2n^3$ floating-point operations. An optimized variation of calculating matrix-matrix product is implemented in *?GEMM* (for general matrices) inside BLAS library. More efficient matrix-multiplication algorithms exist, like Strassen or Strassen-Wingograd methods. They execute asymptotically smaller amount of floating-point operations (exponent of about $2^{\lg 7}$) while their constant factors are usually larger than those in classical methods. Their biggest disadvantage is their little numeric stability, especially in small matrix dimensions.

The algorithms introduced below mainly calculate matrix-matrix product of small matrices, while paralleling the calculation of many products. Therefore, Strassen-like methods will not be effective, and even harm the numeric stability of our implementation.

2.2 Polynomial Evaluation by Patterson-Stockmeyer

Whereas Horner's method (nested multiplication, which will be introduced below) is almost always used in the scalar case, for matrix polynomials there are four competing methods.

2.2.1 Explicit powers

The first method for evaluating the polynomial is to calculate its explicit powers. This naive polynomial evaluation simply calculates all powers of A from A^2 to A^d and accumulates the polynomial for each power, starting from $Q = c_0 \cdot I$. The powers are calculated by multiplying A_{k-1} by A for every $k = 1...d$, and adding them to the accumulated matrix $Q = Q + c_k A^k$. Therefore, $d-1$ matrix multiplication calculations are performed, as well as scale-and-add operations.

2.2.2 Horner's Rule

The second approach which was mentioned above, denoted as Horner's rule, is to accumulate the polynomial, meaning we start from $Q = c_d \cdot A + c_{d-1} \cdot I$

Algorithm 1 Evaluate polynomial via explicit powers

This algorithm evaluates the polynomial (1) by explicitly forming matrix powers.

1: $P = A$
2: $Q = c_0 I + c_1 A$
3: **for** $k = 2 : d$ **do**
4: $\quad P = P \cdot A$
5: $\quad Q = Q + c_k P$
6: **end for**

and multiply Q by A in each step, while adding a scaled identity matrix: $Q = Q \cdot A + c_k I$. Horner's rule also calculates $d - 1$ matrix products, while only adding a constant to the diagonal of every Q along the way (as $c_k I$ contains values only on the diagonal), which is cheaper than scale-and-add operations.

Algorithm 2 Horner's method

This algorithm evaluates the polynomial (1) by Horner's method.

1: $Q = c_d X + c_{d-1} I$
2: $Q = c_0 I + c_1 A$
3: **for** $k = d - 2 : -1 : 0$ **do**
4: $\quad Q = A \cdot Q + c_k I$
5: **end for**

Horner's method is not suitable when d is not known at the start of the evaluation, as is often the case when a truncated power series is to be summed. In this case $q(A)$ can be evaluated by explicitly forming each power of A.

2.2.3 Factored form

Another method for calculating the polynomial, factorizes the polynomial $q(A) = c_d(A-\zeta_1)\cdot ...\cdot(A-\zeta_d)$ (where $\zeta_1, ..., \zeta_d$ are the polynomial roots) and then evaluates this factorized form for the input matrix A.

Algorithm 3 Evaluate polynomial in factored form

This algorithm evaluates the polynomial (1) given the roots $\zeta_1, ..., \zeta_d$ of $q(A)$.

1: $Q = A - \zeta_d \cdot I$
2: **for** $k = d - 1 : -1 : 1$ **do**
3: $\quad Q = Q \cdot (A - \zeta_k \cdot I)$
4: **end for**

One drawback to Algorithm 3 is that some of the roots ζ_j may be complex and so complex arithmetic may be required even when the polynomial and A are both real.

2.2.4 Patterson-Stockmeyer

The fourth and least obvious method is that of Patterson and Stockmeyer. They developed an algorithm for evaluating $q(A)$ using only about $2\sqrt{d}$ matrix-matrix multiplications, as opposed to $d-1$ operations in the naive methods described above. Patterson and Stockmeyer algorithm splits the d monomials ($c_i A^i$) in $q(A)$ into s subsets of (approximately) p monomials each. Each subset is a summary of at-most p monomials, so it can be represented as a polynomial of degree $p-1$ at most for A, times some power of A^p.

For clarity, let's assume that $d+1$ divides to integers p, s such that $d+1 = ps$. We can define $q(A) = c_0 I + c_1 A + c_2 A^2 + ... + c_d A^d$ as a summary of s polynomials of degree $p-1$ each:

$$\begin{aligned} q(A) = (c_0 I + c_1 A + \cdots + c_{p-1} A^{p-1})(A^p)^0 + \\ (c_p I + c_{p+1} A + \cdots + c_{2p-1} A^{p-1})(A^p)^1 + \\ (c_{2p} I + c_{2p+1} A + \cdots + c_{3p-1} A^{p-1})(A^p)^2 + \\ \cdots + \\ + (c_{(s-1)p} I + c_{(s-1)p+1} A + \cdots + c_{sp-1} A^{p-1})(A^p)^{s-1} \end{aligned} \quad (2)$$

In fact, $q(A)$ is now represented as a degree-$(s-1)$ polynomial of A^p, with coefficients that are polynomials of degree $p-1$ each. This representation assumes that q is dense, and that $d+1$ can be divided to two integers without a remainder. In the case where $q(A)$ is sparse, some monomials will be missing from the sub-polynomials (of degree $p-1$) - so their coefficients c_i-s can be treated as 0. In addition, when $d-1$ cannot be factorized to integers, the last sub-polynomial will be of smaller degree and will have less than p monomials.

Implementing Patterson-Stockmeyer algorithms requires calculating $A^2, ..., A^p$ in advance (to be used in all sub-polynomials) and calculates each sub-polynomial using any of the naive approaches described above. Then, calculating the polynomial $q(A)$ of A^p with s coefficients (all the sub-polynomials) can be applied using Horner's rule.

In the full-and-dense case where $d+1 = ps$, the total number of matrix multiplications that the method performs is $(p-1)(s-1) = p+s-2$. 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.

In addition, the number of matrix scale-and-add operations for each sub-polynomials is $p-1$, which totals to $s(p-1)$ across all sub-polynomials together.

To avoid re-calculation of matrix powers or coefficients, the number of matrices that have to be stored is $(p-1)+1+1+1 = p+2$. $p-1$ stands for $A^2, ..., A^p$ that are calculated and cached to be used for all sub-polynomials, and the 3 additional matrices are the accumulated $(A^p)^k$, the matrix used for calculating each sub-polynomial, and the one used to accumulate all sub-polynomials (multiplied by $(A^p)^k$) calculated so far.

The calculation cost is minimized by minimizing $p+s$, which happens near $p \approx s \approx \sqrt{d}$.

2.3 Reduction to Triangular Matrices and Schur Decomposition

Using well-conditioned similarity transformations is a key requirement in order to maintain numerical stability of the calculation. In particular, restricting to unitary transformations tightly keeps the numerical stability. In general, the best method to transform a matrix to diagonal form via unitary similarities is the Schur triangular form. The Schur decomposition factors $A \in \mathbb{C}^{n\times n}$ as $A = QTQ^*$, where $Q \in \mathbb{C}^{n\times n}$ is unitary and $T \in \mathbb{C}^{n\times n}$ is upper triangular. The eigenvalues of A appear on the diagonal of T.

When A is real, T may be complex. Calculating a real-schur-form yields a real upper quasi-triangular matrix T, with 1-by-1 and 2-by-2 blocks on its diagonal. 2-by-2 blocks will be standardized in the form $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ where $b \cdot c < 0$. The eigenvalues of such a block are $a \pm i\sqrt{bc}$.

The Schur decomposition can be computed with perfect backward stability by the QR algorithm, and hence it is a standard tool in numerical linear algebra. Since $f(A) = Qf(T)Q^*$, this decomposition reduces the $f(A)$ problem to that of computing $f(T)$ for a triangular matrix. Therefore, building a solution for calculating the polynomial of triangular matrices will also cover general matrices.

2.4 Parlett Recurrence

The Parlett Recurrence method (introduced by Parlett-David-Higham) can be applied once the matrix polynomial calculation is reduced to (quasi) triangular matrix, .

This method requires the matrix function (polynomial calculation in particular) to be calculated directly for smaller blocks on the diagonal, and then applies the following algorithm to calculate the elements outside those blocks.

Algorithm 4 Parlett Recurrence

Given an upper triangular $T \in \mathbb{C}^{n\times n}$ with distinct diagonal elements and a function f defined on the spectrum of T, this algorithm computes $F = f(T)$

1: $f_{ii} = f(t_{ii})$ ▷ Calculate function f for all diagonal elements
2: **for** $j = 2 : n$ **do**
3: **for** $i = j - 1 : -1 : 1$ **do**
4:

$$f_{ij} = t_{ij} \cdot \frac{f_{ii} - f_{jj}}{t_{ii} - t_{jj}} + \sum_{k=i+1}^{j-1} (f_{ik}t_{kj} - t_{ik}f_{kj}) / t_{ii} - t_{jj} \tag{3}$$

5: **end for**
6: **end for**

Parlett's recurrence fails to perform the calculation when $t_{ii} = t_{jj}$ for some $i \neq j$. As the diagonal elements are T's eigenvalues, this condition is valid when

T has repeated eigenvalues. In this situation (3) provides no information about f_{ij}. To handle this situation, a blocked version of Parlett recurrence can be applied.

2.4.1 Blocked Parlett Recurrence

Let $T = (Ti_j)$ be block upper triangular with square diagonal blocks, possibly of different sizes. Then $F = (Fij)$ has the same block structure. Equating (i, j) blocks in $TF = FT$ leads to

$$T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1} (F_{ik}T_{kj} - T_{ik}F_{kj}),\ i < j \qquad (4)$$

Algorithm 5 Block Parlett Recurrence

Given a triangular matrix $T = (Tij) \in \mathbb{C}^{n\times n}$ partitioned in block form, with no two diagonal blocks having an eigenvalue in common, and a function f defined on the spectrum of T, this algorithm computes $F = f(T)$ using the block form of Parlett's recurrence.

1: $F_{ii} = f(T_{ii})$ - Calculate function f for all diagonal blocks
2: **for** $j = 2 : n$ **do**
3: **for** $i = j - 1 : -1 : 1$ **do**
4: Solve the Sylvester equation $T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1}(F_{ik}T_{kj} - T_{ik}F_{kj})$ for F_{ij}
5: **end for**
6: **end for**

This recurrence can be used to compute F either a block superdiagonal[1] at a time or a block column at a time, provided we can evaluate the diagonal blocks $F_{ii} = f(T_{ii})$ and solve the Sylvester equations (4) for the F_{ij}. The Sylvester equation (4) is non-singular if and only if T_{ii} and T_{jj} have no eigenvalue in common.

Therefore, in order to use this block recurrence, we need first to reorder the matrix T so that no two diagonal blocks have an eigenvalue in common; Here, reordering means applying a unitary similarity transformation to permute the diagonal elements whilst preserving "triangularity". A reordering optimization is introduced below, as applying the unitary similarity transformation is computationally heavy. The two variations for calculating the recurrence are explored and analyzed below, as well as a parallel variant for solving Sylvester equations.

2.5 Reorder matrix diagonal

To apply the blocked Parlett recurrence and maintain numeric stability, no two blocks of eigenvalues can contain identical (or close) values. Denote $\hat{T}$ as the

[1] sub-diagonal of blocks

reordered and partitioned form of T. Let $\delta > 0$ be a blocking parameter - aka a tolerance for checking whether two values or blocks are close. The blocks must be well separated and hold the following conditions:

1. The blocks are separated at least by tolerance :

$$\min\left\{|\lambda - \mu| : \lambda \in \Lambda(\tilde{T}_{ii}),\ \mu \in \Lambda(\tilde{T}_{jj}),\ i \neq j\right\} > \delta$$

2. All the values within a block are close to one another at most by tolerance δ: for every block $\tilde{T}_{ii}$ with dimension bigger than 1, for every $\lambda \in \Lambda(\tilde{T}_{ii})$ there is a $\mu \in \Lambda(\tilde{T}_{ii})$ such that $\lambda \neq \mu$, $|\lambda - \mu| \leq \delta$. This implies that $\max\{|\lambda - \mu| : \lambda, \mu \in \Lambda(\tilde{T}_{ii}),\ \lambda \neq \mu\} \leq (m-1) \cdot \delta$ for $\tilde{T}_{ii} \in \mathbb{R}^{m \times m}$ $(m > 1)$ and this bound is attained when, for example, $\Lambda(\tilde{T}_{ii}) = \{\delta, 2\delta, \ldots, m\delta\}$.

To reorder the upper triangular Schur factor T into blocks following the conditions described above, T has to be transformed into a partitioned upper triangular matrix $\tilde{T} = U^* \cdot T \cdot U = (\tilde{T}_{ij})$, where U is unitary. The transformation has to be applied using the following algorithm:

Algorithm 6 Reorder Matrix Diagonal

Given an upper tiangular Schur factor T and its diagonal blocks, this algorithm reorders the diagonal blocks following the conditions described aboce, by transforming T into a partitioned-upper-triangular matrix $\tilde{T} = U^* \cdot T \cdot U = (\tilde{T}_{ij})$

1: Cluster T's eigenvalues to their designated blocks
2: Calculate the best permutation of the blocks, given the original location of each eigenvalue on the diagonal, to minimize the number of unitary similarity transformations applied to reorder the eigenvalues
3: Plan and apply the reordering according to the calculated permutation.

2.5.1 Cluster Eigenvalues to Designated Blocks

The first step of clustering T's eigenvalues into their designated clusters, as described in algorithm 6 in step 1, is solved by algorithm 7 [ref to ch9] that provides a mapping from each eigenvalue λ_i of T to an integer q_i such that the set S_{q_i} contains λ_i

2.5.2 Permutation calculation

The next phase in algorithm 6, described in step 2, is to calculate the best permutation of the blocks defined by the previous step. This problem is equivalent to finding a method for swapping adjacent elements in q to obtain a confluent permutation $\hat{q}$. A confluent permutation of n integers, $q_1, \ldots, q_n$, is a permutation such that any repeated integers q_i are next to each other. The permutation is agnostic to the order of the elements within the same block. The best confluent permutation is the one that requires a minimal number of swaps to transform q

Algorithm 7 Block pattern

Given a triangular matrix $T_{ii} \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_i \equiv t_{ii}$ and a blocking parameter $\delta > 0$, this algorithm produces a block pattern, defined by an integer vector q, for the block Parlett recurrence; The eigenvalue λ_i is assigned to the set S_{q_i}, and it satisfies the conditions that $\min\{|\lambda_i - \lambda_j| : \lambda_i \in S_p,\ \lambda_j \in S_q,\ p \neq q\} > \delta$ and, for each set S_i with more than one element, every element of S_i is within distance at most δ from some other element in the set. For each set S_q, all the eigenvalues in S_q are intended to appear together in an upper triangular block $\tilde{T}_{ii}$ of $\tilde{T} = U^* \cdot T \cdot U$.

$p = 1$
Initialize $S_q = \emptyset$ for every q
for $i = 1 : n$ **do**
    if $\lambda_i \neq S_q$ for all $1 \leq q < p$ **then**
        Assign λ_i to S_p
        $p = p + 1$
    end if
    for $j = i + 1 : n$ **do**
        Denote by S_{q_i} the set that contains λ_i
        if $\lambda_i \notin S_{q_i}$ **then**
            if $|\lambda_i - \lambda_j| \leq \delta$ **then**
                if $|lambda_j \notin S_k$ for all $1 \leq k < p$ **then**
                    Assign λ_j to S_{q_i}
                else
                    Move the elements of $S_{\max(q_i,q_j)}$ to $S_{\min(q_i,q_k)}$ ▷ Can be done concurrently over all elements
                    Reduce by 1 the indices of sets S_q for $q > \max(q_i, q_j)$ ▷ Can be done concurrently over all $q > \max(q_i, q_j)$
                    $p = p - 1$
                end if
            end if
        end if
    end for
end for

to $\hat{q}$. Finding such a permutation is an NP-complete problem [ref to ch9], where the minimum number of swaps required to obtain a given confluent permutation is bounded above by $\frac{n^2}{2}(1-\frac{1}{k})$, where k is the number of distinct q_i (the number of different clusters).

The following method works well in practice: find the average index of the integers in q and then order the integers in q´ by ascending average index.

3 Algorithm Design and Analysis

In this section we will cover the advancements and improvements made in designing and implementing the matrix polynomial calculation algorithm. The section covers algorithms for rclustering and reordering eigenvalues for matrices, applying the calculation functions using smaller blocks, and parallelism of some aspects that reduce the execution time of the algorithm.

3.1 Eigenvalues reordering - Smart Swapping vs. Bubble Sort

The final step in the reordering algorithm 6 on page 7 (described in step 3) applies the chosen permutation on the input triangular matrix T.

Swapping two adjacent diagonal elements of T is implemented in LAPACK's function *?TRSEN* with computational cost of $20n$ flops for the complex flavor and $6n$ flops for the real flavor, plus another identical number of flops to update the Schur vectors matrix. A more robust LAPACK function *?TREXC*, introduced by Bai/Demmel (1993) [XXX ref], allows transferring an element from one index to another, by repetitively swapping adjacent blocks. Thus, all the element in between the source and target indices are shifted. This function also handles matrices in real-schur-form, meaning moving 1-by-1 or 2-by-2 blocks to other indices of existing 1-by-1 or 2-by-2 blocks. The computational cost of this function is the product of the cost for swapping, and the number of swaps.

As it appears from the evaluation shown below, this step is computationally heavy, so the number of swaps must be minimized to achieve better and competitive results.

A "naive" reordering can be implemented by applying the bubble-sort approach to swap adjacent diagonal elements, until they are located in their designated blocks. The swapping operation can be applied by either *?TRSEN* or *?TREXC*. This approach does not yield the best results, as seen in the evaluation.

To reduce the number of swaps needed, a smart reordering has so be applied, which calculates the series of required swaps in advance. As the swapping operation of diagonal blocks is computationally expensive, the sorting operation can be applied on the elements indices, creating the desired order of blocks inside each cluster. Algorithm 8 generates a list of ordered diagonal blocks, given an ordered list $c_1, \ldots, c_k$ of k clusters (the chosen permutation mentioned above) and a list of the m input blocks $b_1, ..., b_m$. Each block b_i belong to one of the

clusters $c_1, \ldots, c_k$. We denote that block b_i belongs to cluster c_j as $b_i \in c_j$, and denote c_{b_i} as c_j for j where $b_i \in c_j$.

The algorithm generates the ordered list by sorting the input blocks according to the input clusters order $c_1, \ldots, c_k$. The required permutation, calculated by algorithm 7, does not indicate where the 2-by-2 blocks are located inside each cluster. So, within a cluster, the location of every block type (1-by-1 or 2-by-2) is set, while the algorithm is agnostic to the specific block that will be placed in each location.

For example, assume the input diagonal blocks are clustered as 3,1,4,2,3,2,2,1 with sizes 1,2,2,1,2,1,2,2 respectively, and the clusters are sorted 4,3,1,2. The algorithm will output the ordered blocks to be clustered 4,3,3,1,1,2,2,2 with sizes respectively 2,1,2,2,2,1,1,2.

The algorithm operates by bubble-sorting the input blocks according to the given permutation. In the "naive" reordering, the swapping operation would swap adjacent diagonal blocks in the matrix, while in the smart reordering will just swap their references without performing the actual replacement.

Algorithm 8 Permute cluster blocks

The algorithm generates the ordered list by sorting the input blocks according to the input clusters order $c_1, \ldots, c_k$.

```
1:  s = m
2:  while s > 0 do
3:      t = 0
4:      for j = i + 1 : n do
5:          Denote c_{b_{i-1}} the cluster that contains b_{i-1}
6:          Denote c_{b_i} the cluster that contains b_i
7:          if q_{i-1} > q_i then
8:              Swap b_{i-1} ⟷ b_i                ▷ Blocks are misplaced
9:              t = i
10:         end if
11:     end for
12:     s = t
13: end while
```

Given the ordered blocks indices$ob_1, \ldots, ob_m$ generated by algorithm 8, the series of required moves can be generated. We denote the size of block b_i to be s_{b_i}. Algorithm 10 first classifies each input block ob_j to be either misplaced or well-placed, according to the ordered blocks $ob_1, \ldots, ob_m$ clusters and sizes.

For example, if b_i is a 2-by-2 block of cluster j and ob_i is also a 2-by-2 block of cluster j - then b_i is well placed. Otherwise, if the block is of the wrong size ($s_{b_i} \neq s_{ob_i}$) or of the wrong cluster ($c_{b_i} \neq c_{ob_i}$), it is considered as misplaced.

Then, the algorithm iterates over the misplaced blocks list, and finds a candidate b_i for each location j containing a misplaced block b_j. The iteration is done in ascending order over the locations. Therefore, all the blocks before the current location in each iteration are well-placed by previous iterations. By

Algorithm 9 Classify eigenvalue blocks

The algorithm classifies each block to be well-placed or misplaced, according to the given permutation

1: Initialize sets for storing good locations and bad locations:$G = \emptyset, B = \emptyset$
2: Initialize a list of output moves to be empty: $moves = []$
3: **for** $i = 1 : m$ **do**
4: Denote c_{ob_i} the cluster that contains ob_i
5: Denote c_{b_i} the cluster that contains b_i
6: **if** $c_{b_i} \neq c_{ob_i} or s_{b_i} \neq s_{ob_i}$ **then**
7: Append b_i to B
8: **else**
9: Append b_i to G
10: **end if**
11: **end for**

moving b_j to location i, all blocks in between i and j locations are shifted. The algorithm re-classifies them as misplaced or well-placed as needed.

Note that no blocks before location j are shifted, as they are already well-placed. So, the maximal number of moves is m at most.

Algorithm 10 places the required block in the minimal misplaced location in each iteration. The required block originates from a location after the minimal misplaced location (otherwise, the minimal misplaced location will not be minimal). Each replacement shifts the block between the source and target locations, so no blocks located before the minimal misplaced location are moved. By assuring that the algorithm does not spoil any well-placed blocks in each iteration, the number of swaps performed by the smart reordering is at most m, while each swap moves a block m steps at most.

3.1.1 Reordering overview

The reordering algorithm can be summarized as follows:

The asymptotic cost of both approaches, the native reordering and the smart reordering, is almost the same. The smart permutation works faster under the evaluation shown below.

3.2 Calculate Parlett Recurrence

After the diagonal blocks are clustered and reordered, the required polynomial calculation can be performed directly on each diagonal cluster. There is no point in splitting the clusters into smaller chunks, because the eigenvalues are too close (thus they are located in the same clusters), and the calculation may be numerically unstable.

After the polynomial function is calculated for all the diagonal blocks clusters, the function has to be applied for all the other elements in the triangular matrix, that are located outside the diagonal blocks. This is done by using

Algorithm 10 Reorder misplaced blocks

The algorithm finds and reorders the misplaced blocks

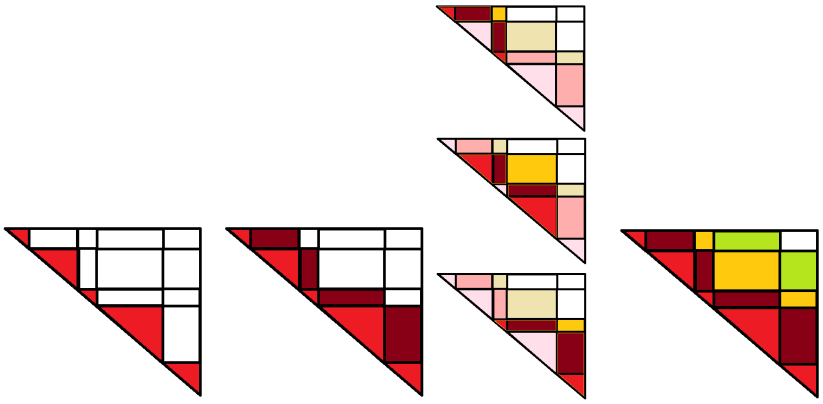
Require: B, G sets generated by algorithm 9

while $B \neq \emptyset$ **do**
  Denote by b_j a block from B with the minimal index
  Get a location candidate i such that $b_i \in B$, $c_{b_i} = c_{ob_j}$ and $s_{b_i} = s_{ob_j}$ ▷ The block b_i is misplaced, it belongs to the cluster that should be placed in j location and has the required block size to be placed in j location. Note that $j < i$ as j is the minimal index in B by its definition
  $moves = moves + (b_i, b_j)$
  $B = B \setminus \{b_j\}$
  $nB = \emptyset$ ▷ New misplaced blocks
  for $k = j : i$ **do**
    Take the block b_k either from B or G. Remove it from B if exists in it.
    if $c_{ob_{k+1}} \neq c_{b_k} or s_{ob_{k+1}} \neq s_{b_k}$ **then** ▷ Check if moving b_k to index $k+1$, results a new misplaced or well-placed block
      Append b_k to nB as $k+1$
    else
      Append b_k to G as $k+1$
    end if
  end for
  $B = B \cup nB$
end while

Algorithm 11 Reorder matrix by clusters

This algorithm expands algorithm 6 on page 7 using the algorithms described above.

1: Based on the locations of the blocks belong to each cluster, build a permutation p that optimizes the number of swaps required to transform the input list into that permutation, using algorithm 8
2: Cluster the input m eigenvalue blocks by according to the blocking parameter $\delta > 0$ using algorithm 7
3: Use algorithm 9 to classify all blocks as well-placed or misplaced.
4: Use algorithm 10 to build reordering moves by iterating over the misplaced locations in ascending order, and fixing each location by placing a candidate block from a higher index in each iteration.
5: Apply the moves generated by algorithm 10 to reorder the diagonal blocks of the input triangular matrix.



(a) Iteration #0 (b) Iteration #1 (c) Iteration #2 (d) Iteration #3

Figure 1: Superdiagonal iterations

Parlett Recurrence (algorithm 7). This recurrence can be used to compute F either a block superdiagonal at a time or a block column at a time, provided we can evaluate the diagonal blocks $F_{ii} = f(T_{ii})$ and solve the Sylvester equations (4) for the F_{ij}. We analyze those two approaches for calculating the recurrence.

3.2.1 Superdiagonal iterations

Assume the polynomial was calculated for the diagonal clusters, as shown in figure 1a. The diagonal cluster blocks are the first layer of blocks calculated. In each iteration, the superdiagonal algorithm takes the next layer of blocks on the superdiagonal adjacent to the one from the previous iteration (figure (1b)), and applies the Parlett Recurrence (4) to calculate the blocks in that layer. Each block F_{ij} is calculated as described in algorithm (5), using the values in the blocks below it and on its left side (figure (1c))). Block F_{ij} solves the Sylvester equation (4). All T_{ab} blocks are known, as T is real-schur-form of the input triangular matrix, and only already-calculated F_{cd} blocks are used.

Therefore, calculating each block on the superdiagonal is independent from the other blocks on the same superdiagonal, meaning that all blocks on the same superdiagonal can be calculated in parallel to one another (figure (1c))

The algorithm for superdiagonal iterations requires to change algorithm (5) (Block Parlett Recurrence) and is described in algorithm

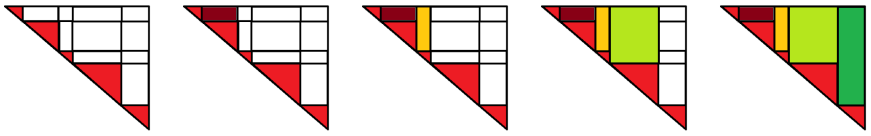
3.2.2 Sequential iterations

After the polynomial is calculated for the diagonal cluster blocks (figure (2a)) marked as A-blocks, Parlett Recurrence can be applied to calculate the top-left non-diagonal block (figure (2b)) marked as B block. After B-block is calculated, the block adjacent to it (figure (2c)) can be calculated (marked as C block), as

Algorithm 12 Superdiagonal Block Parlett Recurrence

Given a triangular matrix $T = (Tij) \in \mathbb{C}^{n \times n}$ partitioned in block form, with no two diagonal blocks having an eigenvalue in common, and a function f defined on the spectrum of T, this algorithm computes $F = f(T)$ using the block form of Parlett's recurrence using the superdiagonal approach demonstrated in figure 1.

1: $F_{ii} = f(T_{ii})$ - Calculate function f for all diagonal blocks
2: **for** $d = 2 : n$ **do** ▷ Those are the superdiagonal layers
3: **for** $i = 1 : n - d$ **do** ▷ Run in parallel over all range $1 \dots n - d$
4: $j \leftarrow i + d$ ▷ This is the column for superdiagonal d in row i
5: Solve the Sylvester equation $T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1}(F_{ik}T_{kj} - T_{ik}F_{kj})$ for F_{ij}
6: **end for**
7: **end for**



(a) Iteration #0 (b) Iteration #1 (c) Iteration #2 (d) Iteration #3 (e) Iteration #4

Figure 2: Sequential iterations

the Sylvester equation that it solves, contains only A and B blocks (figures (2d),(2e)). This approach solves less Sylvester equations ($O(m)$, opposed to $O(m^2)$ in the superdiagonal algorithm) where each Sylvester equation contains larger-scale matrices.

The calculation of the block in each iteration relies on the blocks calculated before it, so this approach is sequential and cannot be paralleled.

The Sylvester equation solver algorithm can be paralleled internally, as shown in algorithm 14.

Algorithm descibes the operation of the sequential approach.

3.3 Blocked Recursive Sylvester Solver

The recursive algorithm Blocked recursive Sylvester solver (14) on page 16 for solving Sylvester equations, operates by recursively splitting the dimensions of the matrices, calculating their solutions, and unifying the calculated results together [ref to article]. Some of the recursive steps can be paralleled, as their are independent to one another. This implementation had empirically better results than the straight-forward LAPACK version *?TRSYL*.

Algorithm 13 Sequential iterations

Given a triangular matrix $T = (Tij) \in \mathbb{C}^{n \times n}$ partitioned in block form, with no two diagonal blocks having an eigenvalue in common, and a function f defined on the spectrum of T, this algorithm computes $F = f(T)$ using the block form of Parlett's recurrence using the sequential approach demonstrated in figure 2.

1: $F_{ii} = f(T_{ii})$ - Calculate function f for all diagonal blocks
2: **for** $i = 2 : n$ **do** ▷ Those are sequential blocks.
3: Denote F_i the i-th top-left non-diagonal block. Denote F_{ii},T_{ii} the diagonal blocks of F, T in index i.
4: Let F_{jj} be the sub-matrix of F with blocks $1, \ldots, i-1$ ▷ This is the triangular block on the left side of the current calculated block
5: Let T_{jj} be the sub-matrix of T with blocks $1, \ldots, i-1$
6: Solve the Sylvester equation $T_{ii}F_i - F_iT_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj}$ for F_i
7: **end for**

3.4 Schur-Parlett algorithm

The complete algorithm for calculating the polynomial$q(A)$ is described in algorithm Schur-Parlett algorithm for $q(A)$ (15) on page 17

4 Implementation

We implemented the algorithms described above using Intel® Parallel Studio XE 2018 for Windows, under C++ 14 standards using Visual Studio 2017 IDE and Intel® C++ Compiler 18.0 for Windows. The LAPACK functions used are also part of the Intel Parallel Studio package, specifically from Intel MKL library. [ref to GitHub]

4.1 Parallelism

The parallelism was done using Intel® Cilk™ Plus library integrated with Intel Parallel Studio package, on the following algorithms:

1. Calculate the sum of two matrices by calculating the sum of every element in parallel (using array notations)

   (a) Adding matrix M to current matrix *this* in offset *from*

```
cilk_for(int col = 0; col < M.ncols(); col++)
{
  this->data(from.first, from.second + col)[0:M
      .nrows()] += M.data(0, col)[0:M.nrows()];
}
```

2. Multiply a matrix by a scalar, by calculating the product of every element in parallel

Algorithm 14 Blocked recursive Sylvester solver

The algorithm recursively and concurrently solves Sylvester equation based on the input matrices

Ensure: $A \in \mathbb{R}^{n \times n}$and $B \in \mathbb{R}^{m \times m}$ in upper quasi-triangular (Schur) form. $C \in \mathbb{R}^{n \times m}$ dense matrix. $blks$ is the maximal block size to use an algorithm for solving small-sized triangular Sylvester equation.

Ensure: Function $GEMM$ implements the $GEMM$-operation $C = C + A \cdot B$. Function $TRSYLV$ implements an algorithm for solving triangular Sylvester kernel problems.

1: **function** RTRSYLV($A, B, C, blks$)
2: **if** $1 \leq M, N \leq blks$ **then**
3: $X =$TRSYLV(A, B, C)
4: **else if** $1 \leq M \leq \frac{N}{2}$ **then** ▷ Case 1: Split A by rows and columns and C by rows only
5: $X_2 =$RTRSYLV($A_{22}, B, C_2, blks$)
6: $C_1 =$GEMM($-A_{12}, X_2, C_1$)
7: $X_1 =$RTRSYLV($A_{11}, B, C_1, blks$)
8: $X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$
9: **else if** $1 \leq N \leq \frac{M}{2}$ **then** ▷ Case 2: Split B by rows and columns and C by columns only
10: $X_1 =$RTRSYLV($A, B_{11}, C_1, blks$)
11: $C_2 =$GEMM(X_1, B_{12}, C_2)
12: $X_2 =$RTRSYLV($A, B_{22}, C_2, blks$)
13: $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$
14: **else** ▷ Case 3: Split A, B and C by rows and columns
15: $X_{21} =$RTRSYLV($A_{22}, B_{11}, C_{21}, blks$)
16: $C_{22} =$GEMM(X_{21}, B_{12}, C_{22}),$C_{11} =$GEMM($-A_{12}, X_{21}, C_{11}$) ▷ Run in parallel
17: $X_{22} =$RTRSYLV($A_{22}, B_{22}, C_{22}, blks$),$X_{11} =$RTRSYLV($A_{11}, B_{11}, C_{11}, blks$) ▷ Run in parallel
18: $C_{12} =$GEMM($-A_{12}, X_{22}, C_{12}$)
19: $C_{12} =$GEMM(X_{11}, B_{12}, C_{12})
20: $X_{12} =$RTRSYLV($A_{11}, B_{22}, C_{12}, blks$)
21: $X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$
22: **end if**
23: **end function**

Algorithm 15 Schur-Parlett algorithm for $q(A)$

Given $A \in \mathbb{R}^{n \times n}$ and a polynomial $q(A) = \sum_{k=0}^{m} c_k \cdot A^k$ (1), this algorithm computes $F = q(A)$.

1: Compute the real-form Schur decomposition $A = QTQ^*$ (Q unitary, T upper quasi-triangular).
2: **if** T is diagonal **then**
3: $F = q(T)$
4: go-to line (10)
5: **end if**
6: Using Algorithm (7) with blocking parameter $\delta = 0.1$, assign each eigenvalue λ_i to a set S_{q_i}.
7: Choose a confluent permutation $\hat{q}$ of q ordered by average index, and use algorithm (8) to apply the permutation.
8: Reorder T according to $\hat{q}$ using the smart permutation algorithm (11), and update Q. ▷ Now $A = QTQ^*$ is a reordered Schur decomposition, with m diagonal clustered blocks in T.
9: Apply Parlett Recurrence using either of Superdiagonal iterations or Sequential iterations techniques.
10: $F = Q \cdot F \cdot Q^*$

(a) Scale current matrix *this* in offset *from* by scalar *alpha*

```
cilk_for(int col = 0; col < n; col++)
{
  this->data(from.first, from.second + col)[0:m
      ] *= alpha;
}
```

3. Apply the polynomial calculation using Patterson-Stockmeyer algorithm as described in equation (2) on page 4. Every $p-1$-degree polynomial can be calculated in parallel, using the cached $A^2, ..., A^{p-1}$ matrices. The final polynomial is calculated using Horner's method as described in algorithm 2 on page 3, sequentially.

(a) Given the polynomial *coefficients* (degree d), calculate the polynomial using Patterson-Stockmeyer algorithm. Assume *Aps* contains all powers of $A^2, ..., A^{p-1}$ and $Ap = A^p$. Each iteration calculatation is stored in *pPolynomial* and added to the final qA using Horner's rule (algorithm 2 on page 3)

```
// Polynomial degree, including I (degree 0)
int d = coefficients.size();
size_t p = (size_t)sqrt(d);
size_t s = (d + 1) / p; // d+1 = ps
// Reverse order - Horner Rule
for (int si = s - 1; si >= 0; si--)
```

```
{
  Matrix pPolynomial(n, n);
  cilk_for (size_t i = 0; i < Aps.size(); i++)
  {
    int pi = si * p + i;
    Matrix Ai(Aps[i]);
    Ai.multiply(coefficients[pi]);
    pPolynomial.add(Ai); // pPolynomial += Ai
  }
  // qA = qA * Ap + pPolynomial
  qA.multiply(Ap);
  qA.add(pPolynomial, origin);
}
return qA;
```

4. Calculate the Sylvester equations using the recursive algorithm 14 on page 16, performing some of the calculations concurrently as described in the algorithm definition.

   (a) The following snippet shows the part of the parallel code using Cilk's syntax in algorithm 14 on page 16 (case 3)

```
auto X21 = rtrsylv(A22, B11, C21, block_size);
cilk_spawn Matrix::multiply_ex(X21, B12, C22,
    X21.nrows(), B12.ncols(), X21.ncols());
A12.multiply(-1);
Matrix::multiply_ex(A12, X21, C11, A12.nrows(),
     X21.ncols(), A12.ncols());
cilk_sync;
auto X22 = cilk_spawn rtrsylv(A22, B22, C22,
    block_size);
auto X11 = rtrsylv(A11, B11, C11, block_size);
cilk_sync;
Matrix::multiply_ex(A12, X22, C12, A12.nrows(),
     X22.ncols(), A12.ncols());
Matrix::multiply_ex(X11, B12, C12, X11.nrows(),
     B12.ncols(), X11.ncols());
auto X12 = rtrsylv(A11, B22, C12, block_size);
auto X1 = Matrix::concat_cols(X11, X12);
auto X2 = Matrix::concat_cols(X21, X22);
return Matrix::concat_rows(X1, X2);
```

5. Perform some eigenvalue clustering steps of algorithm (7) as described in lines (15),(16)

   (a) `cilk_for (int m = 0; m < n; m++)`

```
{
  // Move the elements of Smax(qi,qj) to Smin(
      qi,qj)
  if (clusters[m] == max_qiqj)
    clusters[m] = min_qiqj;
  // Reduce by 1 the indices of sets Sq for q >
       max(qi, qj).
  if (clusters[m] > max_qiqj)
    clusters[m]--;
}
```

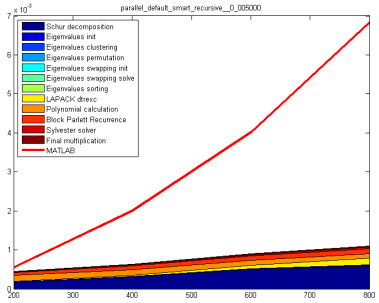
6. Apply Parlett Recurrence over superdiagonal blocks, as described in algorithm 12 on page 14 - calculate all the blocks that belong to the same superdiagonal concurrently.

7. Benchmarking the performance of the implementation requires precise timing of each step. The timings are accumulated using Cilk's reducer to support parallelism.

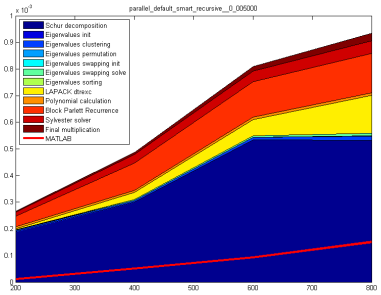
It is important to note that Patterson-Stockmeyer's algorithm was used, rather than Val-Loan's variation as described in [ref to Sivan's article]. The Van-Loan algorithm exploits that parallelism of computing multiple columns concurrently, but still requires calculating and caching the explicit powers $A^2, \ldots, A^p \ldots, (A^p)^s$. Therefore, it will neither improve the parallelism of the algorithm, nor the storage requirements or floating-point calculations.

Intel MKL 2018 provides highly optimized, threaded, and vectorized math functions that maximize performance on Intel processor architectures. It is compatible across many different compilers, languages, operating systems, linking, and threading models. In particular, the Intel MKL *?GEMM* function for matrix-matrix multiplication is highly tuned for small matrices. Intel MKL primitives take advantage of Intel Advanced Vector Extensions 512 (Intel AVX-512) and the capabilities of the latest generations of the Intel Xeon Phi™ processors. *?GEMM* chooses the code path at runtime based on characteristics of the matrices and the underlying processor's capabilities. As a result, applications that rely on *?GEMM* automatically benefit from these optimizations without needing any modification of the code merely be relinking with Intel MKL. Thus, it is not necessary to implement a parallel-form of the matrix multiplication operation, as *?GEMM* is already highly optimized.

~~In addition, parallel matrix multiplication implementations, such as Strassen algorithm (which uses $O(n^{2.81})$ flops), will not improve performance, as the polynomial calculation algorithm splits the input matrix into small matrix chunks. Strassen multiplies the matrix using 7 multiply operations and 18 add operations, while the naive multiplication uses 8 multiply operations and 7 add operations. For small matrix sizes/chunks, the overhead of Strassen-like multiplications overweights the advantage in parallelism.~~



(a) High-degree polynomial evaulation



(b) Low-degree polynomial evaulation

Figure 3: Parallel Matrix Ploynomial Benchmark
Calculation time (seconds) by the size of the input matrix (elements). All displayed timings are normalized by the input matrix degree.

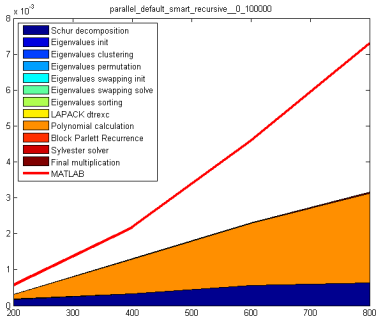
5 Evaluation

The implementation is parameterized and can use many variations. We will start with showing and analyzing of the best variant, and later analyze the impact of different parameters on the performance. For the optimal variation of the algorithm, we chose the following parameters:

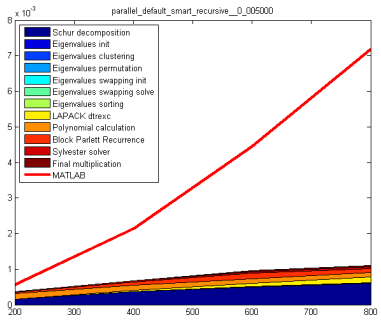
1. Use real-schur-form decomposition as implemented in LAPACK library
2. Use the parallel superdiagonal variation of the Parlett Recurrence described in algorithm 12 on page 14
3. Use the blocked recursive/parallel variation of Sylvester solver described in algorithm 14 on page 16
4. Use the smart reordering algorithm 11 to reorder the triangular matrix's eigenvalues according to the desired permutation of clusters.
5. Use default clustering tolerance of $\lambda = 0.005$ (as large number of clusters is paralleled well)

Figure (3) shows the following findings:

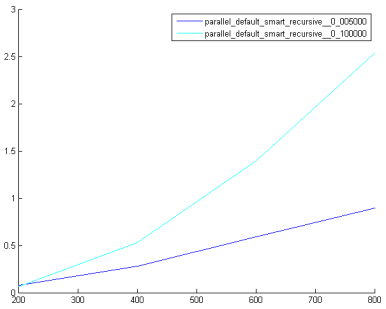
1. As shown in figure (3a), high-degree polynomial evaluation exploits the benefits of parallel computation - thus takes less time to evaluate than the naive Matlab implementation.
2. For low-degree polynomial calculations, Matlab's naive implementation takes less computation time even than the Schur decomposition itself
3. As shown in figures (3a) and (3b), more than half of the parallel algorithm execution time is spent on the Schur decomposition calculation.



(a) Large blocks



(b) Small blocks



(c) Execution time comparison

Figure 4: Block-size effect on execution time

4. Other significant calculation-time steps are the eigenvalues reordering (yellow-colored) and Parlett Recurrence calculation (red-orange-colored).

5. For high-degree polynomials (figure (3a)), the polynomial calculation takes approximately the same computation time as the Parlett Recurrence.

5.1 Large vs. Small Diagonal Clusters

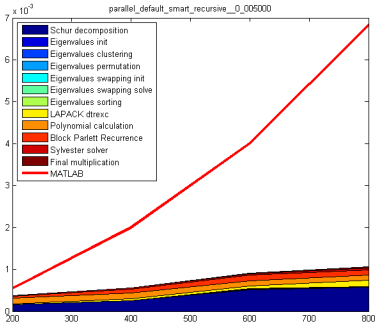
Figure (4) shows the differences between splitting the input matrix into small blocks and large blocks.

The following findings can be deducted:

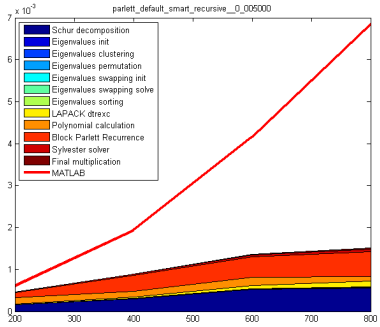
1. As seen in figure (4c), small blocks exploit better parallelism, resulting lower execution times.

2. As seen in figures (4a) and (4b):

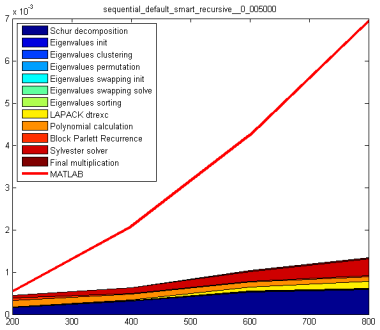
   (a) Polynomial calculation time is significantly higher in the large blocks case (figure (4a)) than the small blocks (figure (4b)), supporting the



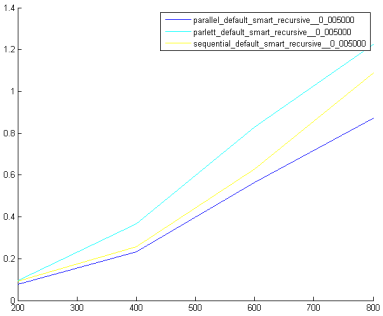
(a) Parallel version for superdiagonal calculation (algorithm 12 on page 14)



(b) Non-parallel version for superdiagonal calculation



(c) Sequential calculation (algorithm 13 on page 15)



(d) Execution time comparison

Figure 5: Parlett Recurrence calculation

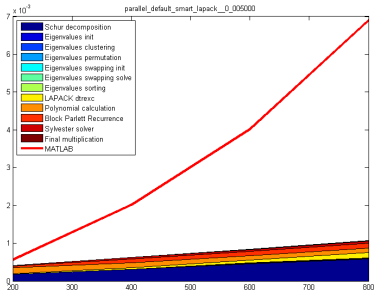
claim that the overall polynomial calculation can be evaluated by applying Parlett Recurrence calculation on smaller blocks.

(b) Schur decomposition is not affected by the blocks clustering

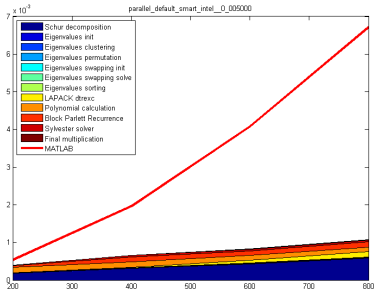
(c) Smaller blocks (figure (4b)) result longer blocks reordering than the large blocks (figure (4a))

5.2 Parlett vs. Sequential vs. Superdiagonal

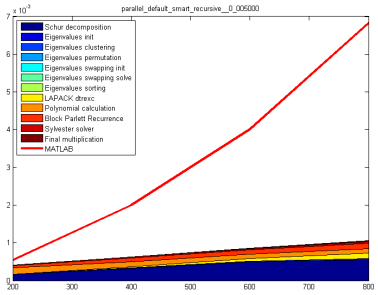
Figure (5) shows the differences between Parlett Recurrence calculation techniques.



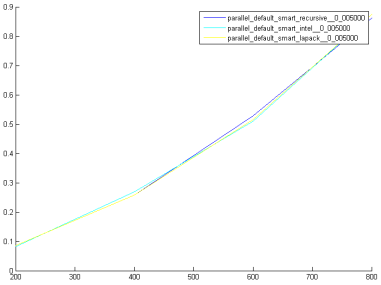
(a) LAPACK implementation



(b) Native Intel MKL implementation



(c) Recursive implementation (algorithm 14 on page 16)



(d) Execution time comparison

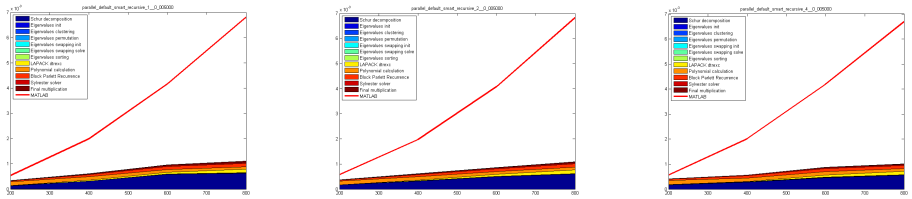
Figure 6: Sylvester solver types

5.3 Sylvester Solver Types

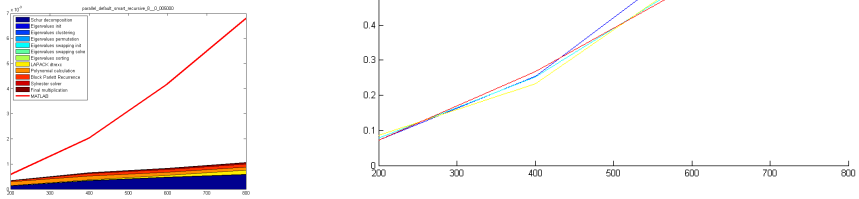
Figure (6) shows the differences between sylvester equation solver implementations.

5.4 Number of cores

Figure (7) shows the differences between executing the parallel polynomial calculation algorithm on various processing cores.



(a) 1 core (b) 2 cores (c) 4 cores



(d) Execution time comparison (e) Execution time comparison

Figure 7: Computation cores comparison