A Parallelization of Parlett's Algorithm for Functions of Triangular Matrices *

B. Bakkaloğlu, K. Erciyeş[†] and Ç. K. Koç
Department of Electrical and Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Abstract

We present a parallelization of Parlett's algorithm for computing arbitrary functions of upper triangular matrices. The parallel algorithm preserves the numerical stability properties of the serial algorithm, and is suitable for implementation on coarse-grain parallel computers. Our experiments on a 16-processor Meiko CS-2 multiprocessor indicate that the algorithm obtains nearly constant efficiency (linear speedup) for small number of processors and for matrices of size greater than 500.

1 Computing Matrix Functions

Computing functions of square matrices is an important topic in linear algebra, engineering, and applied mathematics. There are several methods for this task: Jordan decomposition, Schur decomposition, and approximation methods, e.g., Taylor expansion and rational Padé approximations. The approximation methods may not be suitable for arbitrary functions, since specific properties of the function are exploited. The Jordan and Schur decomposition methods are more general in the sense that an arbitrary function of a given square matrix can be computed using these algorithms. Let A be an $n \times n$ matrix with entries from the real or complex field, and $f(\cdot)$ be the function. The Jordan decomposition algorithm is used to obtain $A = MJM^{-1}$, and then $f(A)$ is computed using the formula $f(A) = Mf(J)M^{-1}$. However, there are some computational difficulties with the Jordan decomposition approach, unless A can be diagonalized and has well-conditioned eigenvectors [3]. The Schur decomposition, on the other hand, is more stable, and can be used for computing arbitrary functions of matrices. Let $A = QTQ^H$ be the Schur decomposition of the full matrix A, then $f(A) = Qf(T)Q^H$, where T is an upper triangular matrix. This way the computation of $f(A)$ for an arbitrary matrix A is reduced to the computation of $f(T)$ for an upper triangular matrix T.

Let $F = f(T)$, and f_{ij} and t_{ij} be the elements in the ith row and jth columns of the upper triangular matrices F and T, respectively. One approach in computing the entries f_{ij} is to obtain explicit expressions for each f_{ij} in terms of t_{ij} for all possible values of i and j. However, these expressions become very complicated as we move away from the main diagonal, and do not allow cost-effective computation of the entries of F. Let $\lambda_i = t_{ii}$. It is shown in [2, 3] that $f_{ii} = f(\lambda_i)$ for $1 \leq i \leq n$ and $f_{ij} = 0$ for $1 \leq j < i \leq n$. Furthermore, for all $1 \leq i < j \leq n$, we have

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

*This work is supported in part by the National Science Foundation under grant ECS–9312240 and CDA–9216172.
[†]On leave from Ege University, Izmir, Turkey.

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

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

2 Parlett's Algorithm

A fast algorithm for computing $F = f(T)$ was proposed by Parlett [5]. This method is derived from the commutativity property $FT = TF$. Parlett shows that by expanding the matrix multiplication and solving for f_{ij} in the above, we obtain the summation formula

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

which requires that $t_{ii} \neq t_{jj}$ for all $i \neq j$. Parlett's algorithm starts with computing the main diagonal entries of F. Since the main diagonal entries t_{ii} are the eigenvalues of T, f_{ii} is calculated by applying f to each t_{ii}, i.e., $f_{ii} = f(t_{ii})$. After computing the main diagonal entries, the algorithm computes the superdiagonals one at a time, using the summation expression (1). Parlett's algorithm is given in Figure 1.

Figure 1: Parlett's algorithm.

```
for i = 1 to n
  f_ii = f(t_ii)
end
for L = 1 to n - 1
  for i = 1 to n - L
    j = i + L
    s = t_ij(f_jj - f_ii)
    for k = i + 1 to j - 1
      s = s + t_ik f_kj - f_ik t_kj
    end
    f_ij = s/(t_jj - t_ii)
  end
end
```

Note that the number of terms in the summation becomes larger as the algorithm proceeds over the superdiagonals. As an example, the expressions for f_{3j} for $j = 3, 4, 5, 6$ are given below:

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

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

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

Figure 2: Data dependency to compute f_{36}.

f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}
	f_{22}	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}
		f_{33}	f_{34}	f_{35}	f_{36}	f_{37}	f_{38}
			f_{44}	f_{45}	f_{46}	f_{47}	f_{48}
				f_{55}	f_{56}	f_{57}	f_{58}
					f_{66}	f_{67}	f_{68}
						f_{77}	f_{78}
							f_{88}

Parlett's algorithm performs n function evaluations to obtain the main diagonal entries of F. After the elements in the main diagonal are obtained, the summation formula (1) is used to compute the $(n-L)$ elements in the Lth superdiagonal, each of which requires $4L$ arithmetic operations. Thus, assuming a single scalar function evaluation requires K arithmetic operations, Parlett's algorithm requires a total of

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

arithmetic operations to compute all elements of the upper triangular matrix F. However, we must remark that if T has close eigenvalues, this algorithm will give inaccurate results. Alternative methods for dealing with the repeated eigenvalue case can be found in [5, 3].

3 Parallelization of Parlett's Algorithm

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

The parallel algorithm has n phases; a superdiagonal vector is computed at each phase using all the available processors. The number of processors p is assumed to be less than or equal to the matrix size n. Note that as we proceed away from the main diagonal, the length of the superdiagonal vector decreases and the number of arithmetic operations required per element increases. Thus, we perform an approximately equal amount of work at each phase, i.e., the parallel algorithm is well-balanced. In order to achieve a low communication penalty we distribute the upper triangular matrices F and T to all processors. This provides access to all elements of F and T by all processors at the beginning. In order to maintain this property, we broadcast the computed superdiagonal at the end of each phase. The processors then update their copy of the F matrix, and thus, have the fresh elements at the beginning of every phase.

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

Figure 3: The parallel algorithm.

```
broadcast the matrices T and F
for all i that processor P owns
  f_ii = f(t_ii)
end
broadcast the main diagonal of F
for L = 1 to n − 1
  for all i that processor P owns
    j = i + L
    s = t_ij(f_jj − f_ii)
    for k = i + 1 to j − 1
      s = s + t_ik f_kj − f_ik t_kj
    end
    f_ij = s/(t_jj − t_ii)
  end
  broadcast the Lth superdiagonal of F
end
```

As illustrated in Figure 2, in order to compute the element f_{ij}, we need to have access to f_{ik} for $k = i, i+1, \ldots, j-1$ and f_{kj} for $k = i+1, \ldots, j$. Since the data dependency pattern becomes more complex and the length of the summation terms increases from phase to phase, the distribution of all elements of the matrices F and T to all processors seems justified. With this distribution, we achieve low communication penalty. Furthermore, the work distribution of the processors is easily handled. At each phase, each processor simply picks a starting and ending index in the superdiagonal to be computed during this phase. This processor is then responsible for computing the elements in this range. We must note that, with this partitioning technique, we may not be able to fit very large matrices to the memory available in each processor. However, the resulting parallel algorithm is efficient, and matrices of dimension up to two thousand (with double-precision floating-point entries) can be dealt with using 16 MB of memory per processor.

Figure 4: The distribution of the elements for $n = 16$ and $p = 4$.

P	Phases of the algorithm															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16
	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13				
	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10								
	4,4	4,5	4,6	4,7												
2	5,5	5,6	5,7	5,8	4,8	4,9	4,10	4,11	3,11	3,12	3,13	3,14	2,14	2,15	2,16	
	6,6	6,7	6,8	6,9	5,9	5,10	5,11	5,12	4,12	4,13	4,14					
	7,7	7,8	7,9	7,10	6,10	6,11	6,12									
	8,8	8,9	8,10													
3	9,9	9,10	9,11	8,11	7,11	7,12	7,13	6,13	5,13	5,14	5,15	4,15	3,15	3,16		
	10,10	10,11	10,12	9,12	8,12	8,13	8,14	7,14	6,14	6,15						
	11,11	11,12	11,13	10,13	9,13	9,14										
	12,12	12,13														
4	13,13	13,14	12,14	11,14	10,14	10,15	9,15	8,15	7,15	7,16	6,16	5,16	4,16			
	14,14	14,15	13,15	12,15	11,15	e1,16	10,16	9,16	8,16							
	15,15	15,16	14,16	13,16	12,16											
	16,16															

In case we have fewer than n processors, there is very little change in the structure of the parallel algorithm. At each phase, the processors compute the starting and the ending indices and perform

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

the summations in this range according to the formula (1). At the end of the phase, the entire superdiagonal is broadcast to all p processors, and the matrix F is updated to get ready for the next phase. Figure 4 shows the distribution of the elements over the processors at each step of the algorithm for $n = 16$ and $p = 4$.

4 Analysis of Efficiency and Implementation Results

As seen from Figure 3, the parallel version of Parlett's algorithm first performs $\lceil n/p \rceil$ function evaluations. Assuming a single function evaluation requires K arithmetic steps, the parallel computation of the main diagonal requires

$$K \left\lceil \frac{n}{p} \right\rceil \leq K \left(\frac{n}{p} + 1 \right)$$

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

$$\sum_{L=1}^{n-1} \left\lceil \frac{n-L}{p} \right\rceil (4L) \leq \sum_{L=1}^{n-1} \left(\frac{n-L}{p} + 1 \right) (4L) \;=\; \frac{2}{3p} \left(n^3 - n\right) + 2n^2 - 2n \;.$$

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

$$T_p = K \left(\frac{n}{p} + 1 \right) + \frac{2}{3p} \left(n^3 - n\right) + 2n^2 - 2n \;.$$

In order to calculate the communication penalty, we take a closer look at the communication steps of the algorithm. After a processor computes $\lceil (n - L)/p \rceil$ elements of the Lth superdiagonal, these elements are broadcast to all the other processors. This operation is called a multi-node broadcast operation. A naive method of accomplishing this task is to perform n sequentially-arranged single-node broadcast operations. A better strategy is to perform simultaneous broadcast operations in order to achieve maximum concurrency [1]. Let T_b be the time required to perform a multi-node broadcast operation on p single-precision floating-point numbers residing on p processors. For example, on the hypercube architecture, we have $T_b = 2p - 2$. Since during the Lth phase $\lceil (n - L)/p \rceil$ elements are to be broadcast, we calculate the total communication penalty of the parallel algorithm as

$$T_c = \sum_{L=0}^{n-1} \left\lceil \frac{n-L}{p} \right\rceil T_b \leq \sum_{L=0}^{n-1} \left(\frac{n-L}{p} + 1 \right) T_b \;=\; \left(\frac{n^2+n}{2p} + n \right) T_b \;.$$

The efficiency of the parallel algorithm can be estimated using the arithmetic and communication complexity values T, T_p, and T_c. Let τ be the ratio of the time required to transfer a floating-point number to an adjacent node to the time required to perform a floating-point operation. Using this value of τ, we can calculate the estimated efficiency of the parallel algorithm as

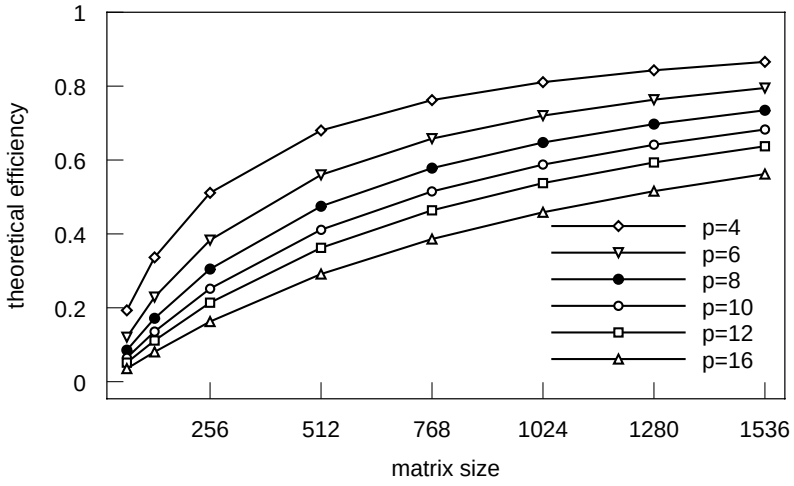
$$E_{est} = \frac{T}{p(T_p + \tau T_c)} \;.$$

In order to compute the estimated efficiency we need to estimate the size of K, which is the number of arithmetic operations required to compute the scalar function $f(\cdot)$. Our experiments indicated

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

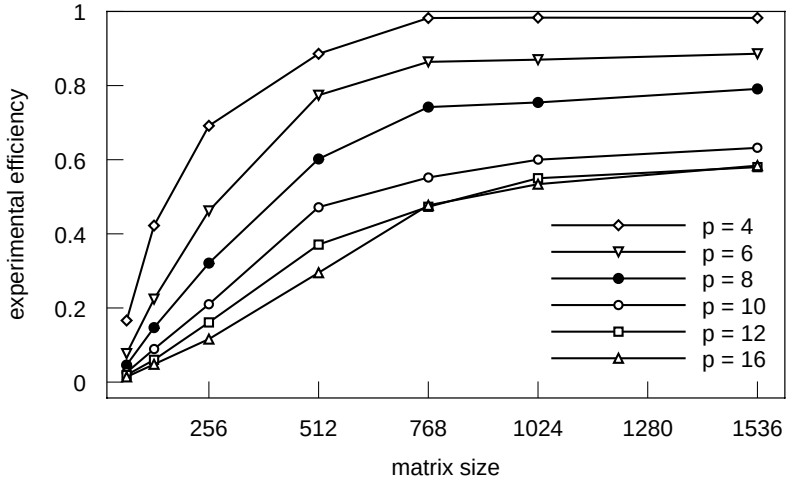
that for most common functions, e.g., logarithm, square-root, trigonometric, and exponential functions, the value of K is between 5 and 15. Thus, we can take an average value $K = 10$. Furthermore, we found that the value of τ on the Meiko CS-2 approximately equals to 35. Using these values, we plot the estimated efficiency of the parallel algorithm in Figure 5 for matrix sizes between 64 and 1536.

Figure 5: Theoretical efficiency as a function of the matrix size.



We implemented the parallel algorithm on a Meiko CS-2 multiprocessor with 16 processors, in which each node is a Sparc processor equipped with 256 MBytes of memory. In our experiments, we computed the functions of matrices of dimensions ranging from 64 to 1536. In Figure 6, we give the actual (experimental) efficiency of the parallel algorithm as a function of the matrix size. The timing values used to compute these efficiency values are the average timings for computing matrix square-root, logarithm, and exponential functions. As can be seen from Figures 5 and 6, the estimated efficiency values are nearly equal to the actual efficiency values obtained. Furthermore, the efficiency is almost constant for matrix size greater than 500.

Figure 6: Actual efficiency as a function of the matrix size.



Parallel Algorithms and Applications, 11(1-2):61-69, 1997.

5 Conclusions

We have presented a parallelization of Parlett's algorithm for computing functions of upper triangular matrices. The parallel algorithm preserves the numerical properties of the serial algorithm, and is suitable for implementing on coarse-grain parallel computers. Our experiments on a Meiko CS-2 has indicate that the parallel algorithm obtains nearly constant efficiency (linear speedup) for small number of processors and for matrices of size larger than 500.

The presented parallel algorithm computes an arbitrary function of an $n \times n$ upper triangular matrix in $O(n^2)$ time using n processors. However, it is possible to compute the matrix function in $O(n \log n)$ time by parallel computation of the summation terms given by (1), which would require $O(n^2)$ processors. It is an open question whether Parlett's algorithm can be further parallelized, more specifically whether an $O(\log n)$-time parallel algorithm can be obtained, which uses Parlett's summation (1). However, it is shown in [4] that the commutativity property and Bartels-Stewart algorithm for solving Sylvester's equation yield a divide-and-conquer algorithm for computing functions of upper triangular matrices. The resulting algorithm requires approximately the same number of arithmetic operations as Parlett's algorithm, and allows further parallelization. The parallel divide-and-conquer algorithm given in [4] requires $O(\log^3 n)$ time using $O(n^6)$ processors to compute an arbitrary function of an $n \times n$ upper triangular matrix.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] C. Davis. Explicit functional calculus. *Linear Algebra and its Applications*, 6:193–199, 1973.

[3] G. H. Golub and C. F. van Loan. *Matrix Computations.* Baltimore, MD: The Johns Hopkins University Press, 2nd edition, 1989.

[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.

Parallel Algorithms and Applications, 11(1-2):61-69, 1997.