High-Performance Library Software for QR Factorization

Erik Elmroth[1] and Fred Gustavson[2]

[1] Department of Computing Science and HPC2N, Umeå University, SE–901 87 Umeå, Sweden.
elmroth@cs.umu.se
[2] IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.
gustav@watson.ibm.com

Abstract. In [5, 6], we presented algorithm RGEQR3, a purely recursive formulation of the QR factorization. Using recursion leads us to a natural way to choose the k-way aggregating Householder transform of Schreiber and Van Loan [10]. RGEQR3 is a performance critical subroutine for the main (hybrid recursive) routine RGEQRF for QR factorization of a general $m \times n$ matrix. This contribution presents a new version of RGEQRF and its accompanying SMP parallel counterpart, implemented for a future release of the IBM ESSL library. It represents a robust high-performance piece of library software for QR factorization on uniprocessor and multiprocessor systems. The implementation builds on previous results [5, 6]. In particular, the new version is optimized in a number of ways to improve the performance; e.g., for small matrices and matrices with a very small number of columns. This is partly done by including mini blocking in the otherwise pure recursive RGEQR3. We describe the salient features of this implementation. Our serial implementation outperforms the corresponding LAPACK routine by 10-65% for square matrices and 10-100% on tall and thin matrices on the IBM POWER2 and POWER3 nodes. The tests covered matrix sizes which varied from very small to very large. The SMP parallel implementation shows close to perfect speedup on a 4-processor PPC604e node.
Keywords: Serial and parallel library software, QR factorization, recursion, register blocking, unrolling, SMP systems, dynamic load balancing.

1 Introduction

A program's ability to efficiently utilize the memory hierarchy is the key to high performance on uniprocessors as well as on SMP systems. In dense linear algebra this is traditionally accomplished by formulating algorithms so that they are rich in Level 3 (i.e., matrix-matrix) operations. This is typically done by expressing matrices as a collection of submatrices (block matrices) and performing the computations as block operations on these submatrices. The block size is typically a fixed parameter that is adjusted for the specific memory hierarchy of the computer.

It has recently been shown that recursion can be used to perform an automatic variable blocking that have a number of advantages over fixed blocking [5–7, 11]. The variable blocking leads to improved performance. It also leads to novel and/or more general algorithms that are less sensitive to parameter settings.

However, for the pure recursive QR factorization presented in [5, 6] there are significant additional costs for creating and performing the updates. This problem was overcome by developing a hybrid recursive algorithm to be used for $m \times n$ matrices where n is large. In general, the hybrid recursive implementation outperforms the corresponding LAPACK algorithm, sometimes significantly.

However, as pointed out in [5, 6], there are situations where the cost of the recursive overhead cannot be overcome by replacing Level 2 operations by Level 3 operations. Typically, this is the case for matrices too small to gain from matrix-matrix operations. This specific problem can be overcome by pruning the recursion tree; i.e., by stopping the recursion when n is small (e.g, $= 4$) instead of $n = 1$.

In this contribution, we present robust library software for QR factorization that is also designed to give high performance for cases where pure recursion provides too much overhead. In Section 2.1, we present the mini blocking, or register blocking, in some more detail. Sections 2.2–2.3 is devoted to concise descriptions of our modifications of the routines RGEQR3 and RGEQRF presented in [5, 6]. Technical descriptions of the new register blocking routines are presented in Sections 2.4–2.6. Performance results for our new set of uniprocessor routines are given in Section 3. In Section 4, we outline the modifications of our new parallel version of RGEQRF. Finally, in Section 5 we give some concluding remarks. The software described will be part of the IBM ESSL library version 3.2.

2 Recursive QR factorization using mini blocking

2.1 Introductory remarks

In [5, 6], we applied a pure recursive algorithm for QR factorization of an $m \times n$ matrix, using n as the recursion variable. In this algorithm a divide and conquer approach is performed until the matrix becomes a single column; i.e., $n = 1$ is used as the stopping criteria. At this point, a Householder transformation $Q = I - \tau v v^T$ is produced from this single column. Through the recursion, Householder transformations are recursively combined into a k-way aggregated transformation by computing the compact WY representation for Q.

However, there are significant additional costs for creating and performing the updates using the compact WY representation which prohibits the efficient use of the recursion for large n. This problem is overcome by using a hybrid recursive algorithm called RGEQRF [5, 6] that uses the pure recursive algorithm for fixed size block columns.

The performance obtained on the PPC604e processor indicated that RGEQRF and the pure recursive RGEQR3 was all that was required. However, on "scientific computing machines" such as the IBM POWER2 the performance gains over traditional block algorithms were less significant, and for small matrices the LAPACK algorithm DGEQRF sometimes outperforms the hybrid recursive algorithm [5]. This contribution focuses on correcting the performance deficiency just described. Our approach here had been followed successfully for both Cholesky factorization and Gaussian elimination with partial pivoting [7, 9].

The new approach for recursion is old and very important for RISC type architectures, especially those with limited memory bandwidth and deep memory hierarchies. The approach is called mini blocking where the blocking parameter is small or tiny. In fact, we only concern ourselves with the tiny case and apply a special type of this blocking called register blocking or loop unrolling [1, 8].

There is another way to view this approach. We call it pruning the recursion tree. Let mb be the mini blocking parameter and assume without loss of generality that mb divides $n = nb$, where n is the column dimension of the $m \times n$ matrix we wish to QR factor. In Figure 1, we give our modified QR factorization algorithm obtained by introducing the mini blocking parameter mb. Briefly, the new recursion tree has d leaves and $d - 1$ interior nodes, where $d = \lceil n/mb \rceil$. Each of the d leaves now becomes a single (m, mb) QR factorization problem consisting of mb leaves and *mb - 1* interior nodes. In terms of the pure recursion problem $(mb = 1)$ we have pruned the tree at nodes where (m, n) has $n \leq mb$.

The d recursive subproblems of the pure recursive QR factorization problem execute at greatly reduced Mflops/s rate on processors like IBM POWER2 and POWER3. By pruning the pure recursion tree, we replace each recursive subproblem by a direct computation which we now show executes at a much higher Mflops/s rate. Overall then, performance improves.

2.2 Adding a blocking parameter mb to RGEQR3

Figure 1 presents the new RGEQR3 algorithm, including the mini blocking parameter mb, for avoiding overhead from recursion for problems to small to gain from Level 3 operations. For further details about RGEQR3 without the mini blocking parameter or, equivalently set $mb = 1$; see [5, 6]. In Figure 1, the if clause computation is now a single call to a subroutine named DQRA4. We have set $mb = 4$.

2.3 The Hybrid Recursive Algorithm RGEQRF

In [5, 6], we only stated algorithm RGEQRF when $m \geq n$. Now we state it for arbitrary m and n in Figure 2. Note that this general algorithm is a trivial change to RGEQRF in [5, 6]; i.e., the variable $k = \min(m, n)$ is used in the outer do loop instead of the variable n. The algorithm is included here for completeness of the description of this new ESSL QR factorization software.

2.4 The mini blocking routine DQR24

It is fairly easy to modify a Level 2 QR factorization routine, like the LAPACK algorithm DGEQR2, to utilize a mini blocking. In this new ESSL internal rou-

RGEQR3 $A(1{:}m, 1{:}n) = (Y, R, T)$! Note, Y and R replaces A, $m \geq n$
if $(n \leq mb)$ **then**
  compute Householder transform $Q = I - YTY^T$ such that
  $Q^T A = R$; Return (Y, R, T)
else
  $n_1 = (\lfloor \lceil n/mb \rceil / 2 \rfloor) * mb$ and $j_1 = n_1 + 1$
  call RGEQR3 $A(1{:}m, 1{:}n_1) = (Y_1, R_1, T_1)$ where $Q_1 = I - Y_1 T_1 Y_1^T$
  compute $A(1{:}m, j_1{:}n) \leftarrow Q_1^T A(1{:}m, j_1{:}n)$
  call RGEQR3 $A(j_1{:}m, j_1{:}n) = (Y_2, R_2, T_2)$ where $Q_2 = I - Y_2 T_2 Y_2^T$
  compute $T_3 \leftarrow T(1{:}n_1, j_1{:}n) = -T_1 (Y_1^T Y_2) T_2$.
  set $Y \leftarrow (Y_1, Y_2)$! Y is m by n unit lower trapezoidal
  return (Y, R, T), where
    $R \leftarrow \begin{pmatrix} R_1 & A(1{:}n_1, j_1{:}n) \\ 0 & R_2 \end{pmatrix}$ and $T \leftarrow \begin{pmatrix} T_1 & T_3 \\ 0 & T_2 \end{pmatrix}$
endif

Fig. 1. Recursive QR factorization routine RGEQR3 with mini blocking.

RGEQRF$(m, n, A, lda, \tau, work)$
$k = \min(m, n)$
do $j = 1, k, nb$! nb is the block size
  $jb = \min(n - j + 1, nb)$
  call RGEQR3 $A(j : m, j + jb - 1) = (Y, R, T)$! T is stored in *work*
  if $(j + jb$ .LE. $n)$ **then**
    compute $(I - YT^TY^T)A(j{:}m, j + jb{:}n)$ using *work* and T via
    a call to DQTC
  endif
enddo

Fig. 2. Hybrid algorithm RGEQRF.

DQR24$(m, n, A, lda, tau, work, ldt, lwork)$
$k = \min(m, n)$
do $j = 0, k - 4, 4$
  ! Generate elementary reflectors $H(j : j + 3)$ and $T(4, 4)$ to produce
  ! block reflector $H(j : j + 3) = I - Y * T * Y^T$ by calling
  DQRA4$(m - j, A(j, j), lda, tau(j), work(ldt * j), ldt)$
  ! Apply $H(j : j + 3)^T$ to $A(j : m - 1, j + 4 : n - 1)$ from the left by calling
  DQRU4$(m{-}j, n{-}j - 4, A(j, j), work(ldt{*}j), ldt, A(j, j + 4), ldt, work(ldt{*}(j + 4)))$
enddo
if $(k \geq j)$ **then**
  if $(m \geq n)$ **then**
    DQRA4$(m - j, k - j, A(j, j), tau(j), work(ldt * j), ldt)$
  else ! $m < n$ and $m = k$
    Apply fixup code for cases when n not a multiple of 4
  endif
endif

Fig. 3. Mini blocking Level 2 routine DQR24.

tine, we used $mb = 4$. The algorithm was named DQR24 and it had exactly the same functionality as DGEQR2. In Figure 3 we give algorithm DQR24.

Note that DQR24 calls subroutines DQRA4 and DQRU4. In DGEQR2, the equivalent routines are DLARFG and DLARF and the corresponding value of $mb = 1$. In our ESSL implementation we chose to either call DQR24 or RGEQRF to QR factor the matrix A. Note that RGEQRF does $\lceil k/nb \rceil$ block steps in the outer do loop on j. Let $km = \max(m, n)$. For a given j the crossover criterion used was $(km - j) * (k - j)$ versus a tunable constant.

2.5 Details on DQRA4

DQRA4 does the QR factorization of $A(0 : m-1, 0 : 3)$ as a single computation. If we look at the recursion tree of an $m \times 4$ matrix A in Figure 4 we see that it calls DLARFG once at each of the four leafs to compute a Householder transformation, DQTC three times and makes three T_3 computations. In the figure, the (m, k) notation at each node represents a factorization of an $m \times k$ matrix. The DQTC (m, n, k) notation represents the update of an $m \times n$ matrix with respect to k Householder transformations. $T(a : b, c : d)$ means compute a T_3 submatrix of that size. For both DQTC and the T_3 computations, the calls are made with parameters $(m, 1, 1)$, $(m, 2, 2)$, and $(m - 2, 1, 1)$ and matrix sizes of T_3 equal to 1×1, 1×1, and 2×2. Recall that DQTC consists of seven subroutine calls, consisting of three calls to DTRMM, two calls to DGEMM, one call to DCOPY, and one call to an matrix subtract routine. For a single T_3 computation, there are five subroutine calls, consisting of three calls to DTRMM, one call to DGEMM, and one call to a matrix transpose routine. Summing over

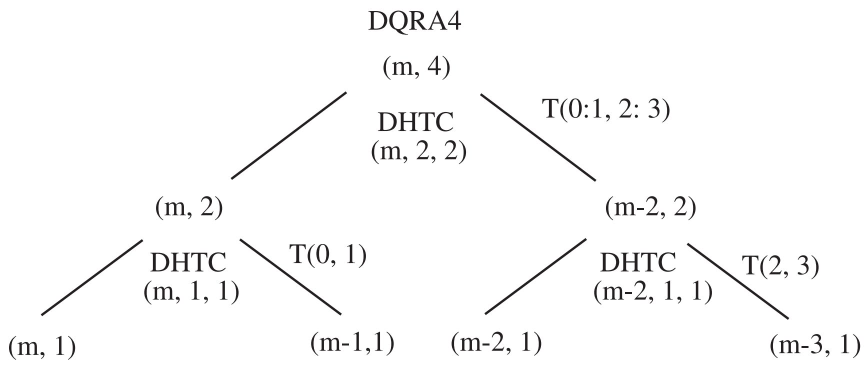

Fig. 4. Recursion tree for DQRA4 on a $m \times 4$ matrix.

the six calls, three to DQTC and three for the T_3 computations, there are 36 subroutine calls, of which 18 are to DTRMM and 9 to DGEMM. Some of the matrix sizes are tiny; e.g., 1×1, and 2×2. Note that ESSL's DTRMM and DGEMM monitors the matrix sizes and if necessary will call Level 2 BLAS such as DGEMV, DGER, and DTRMV. The point we make here is that one has a lot of subroutine calls that have argument checking (DTRMM has 11 arguments and DGEMM has 13 arguments). Additionally, we have not included the costs associated with traversing the recursion tree consisting of three interior nodes and four leaves.

The single routine DQRA4 obviates much of the book keeping described above. Look at the recursion tree again. We see that the two DQTC calls of size $(m, 1, 1)$ and $(m - 2, 1, 1)$ can be replaced by two calls to DDOT and DAXPY and some scalar floating point operations. Similarly, the DQTC call of size $(m, 2, 2)$ consists of a single call to $(2, 2, m)$ DGEMM and a single call to $(2, 2, m)$ DGEMM plus some scalar floating point operations. (Here, (m, n, k) DGEMM refers to a matrix multiplication that performs a rank-k update of an $m \times n$ matrix.) Likewise, two of the three T_3 computations are DDOT calls plus scalar floating point operations while the $T(0:1, 2:3)$ T_3 computation is a call to $(2, 2, m - 2)$ DGEMM plus scalar floating point operations. In all, there are three calls to DGEMM, three to DDOT, and three to DAXPY. Now, additionally we chose not to call DGEMM but to call DGEMM kernel routines. Hence, DQRA4 has minimal subroutine overhead plus it calls DGEMM kernels which has no argument error checking. Performance of DQRA4 is significantly better than DGEQR2 for arbitrary m and $n = 4$. The impact of using this routine for subproblems when factorizing matrices with n large is presented in Section 3.

2.6 DQRU4

This routine takes the output of DQRA4 which is $Q(0:3) = I - YTY^T$ and applies Q^T to an m by n matrix C. Hence, it has the same functionality as DQTC when the blocking parameter nb is 4. DQTC has seven calls (see Section 2.5). Now, DQRU4 has three calls. The first call is to DSCAL to zero out a 4 by n matrix W (work array). The next call computes $W = Y^T C$ by calling a DGEMM kernel of matrix size $(m, n, k) = (4, m, n)$. Now $W = T^T W$ is computed by inline code. T, of course, is LDT by 4 upper triangular. Lastly, a DGEMM kernel of matrix size $(m, n, 4)$ is called to compute $C = C - YW$. Note that Y is lower trapezoidal so the upper triangular part of A $(= Y$ here) must be saved and replaced with ones and zeros. This amounts to saving and restoring ten floating numbers. Finally, we note that RGEQR3 takes advantage of the extra speed of DQRU4; i.e., when $nb = 4$ RGEQR3 calls DQRU4 instead of DQTC.

3 Performance Results

Performance results for the serial RGEQRF are presented for a 160 MHz POWER2 processor and a 200 MHz POWER3 processor. The results for RGEQRF are compared to the DGEQRF which is the corresponding routine in LAPACK. Both routines are providing the same functionality. Values for tuning parameters have been set individually for each routine on each computer system.

Figures 5 and 6 shows the performance in Mflops/s for the new version of the hybrid recursive algorithm RGEQRF and the LAPACK algorithm DGEQRF

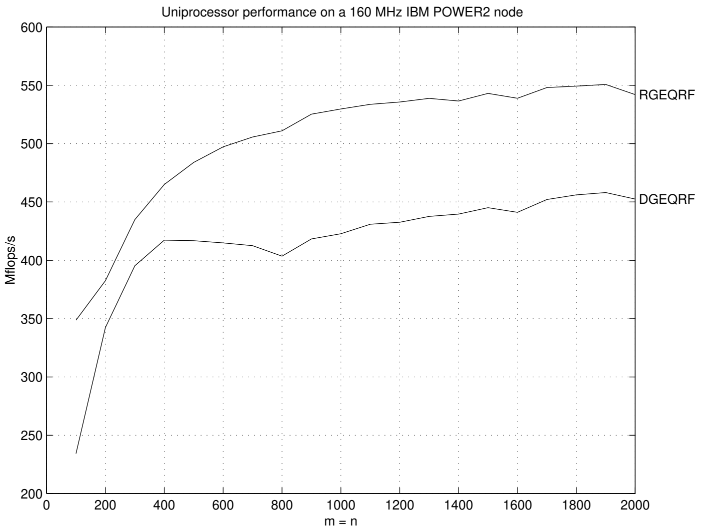

Fig. 5. Uniprocessor performance results in *Mflops* for the hybrid recursive algorithm RGEQRF and DGEQRF of LAPACK on the 160 MHz POWER2 processor.

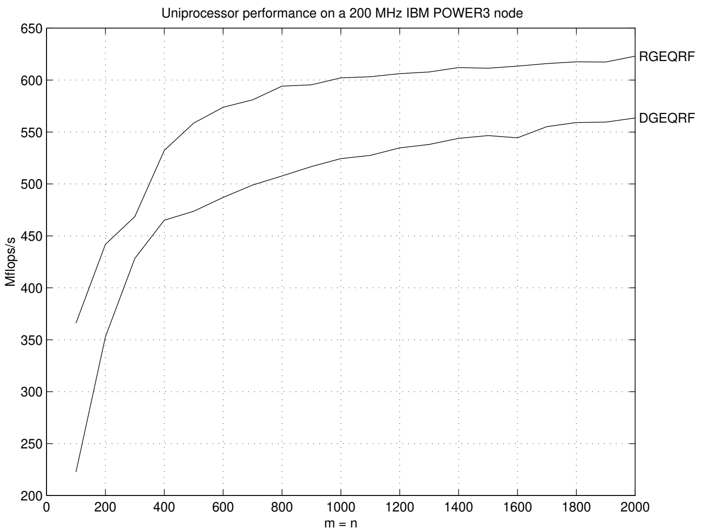

Fig. 6. Uniprocessor performance results in *Mflops* for the hybrid recursive algorithm RGEQRF and DGEQRF of LAPACK on the 200 MHz POWER3 processor.

on square matrices. For $m = n = 100, 200, \ldots, 2000$, RGEQRF outperforms DGEQRF for all tests on both machines, by at least 10%, and most often by around 20%. At best, the difference is 114 Mflops/s (49%) on the POWER2 and and 143 Mflops/s (65%) on the POWER3. In [5], DGEQRF outperformed RGEQRF for $m = n \leq 300$ on the 120 MHz POWER2. Here, we clearly see the positive change of RGEQRF with the new mini blocking incorporated, especially in the critical case for small matrices. For $m = n = 100$, the old RGEQRF was outperformed by DGEQRF by more than 10% on the 120 MHz POWER2 [5], and here the new RGEQRF with mini blocking included is 49% better on the 160MHz version of the same processor.

Performance results for tall and thin matrices are presented in Figures 7 and 8. The figures shows the performance ratio of RGEQRF to DGEQRF. Notably, RGEQRF perform better for all cases. RGEQRF without mini blocking was up to 30% slower than DGEQRF for some cases with $m = 200$. With the mini blocking, this performance deficiency is cured. For $m \geq 800$, RGEQRF outperforms DGEQRF by 30–70% for all values of n tested on POWER2. For the POWER3, the performance difference is up to almost a factor of 2.

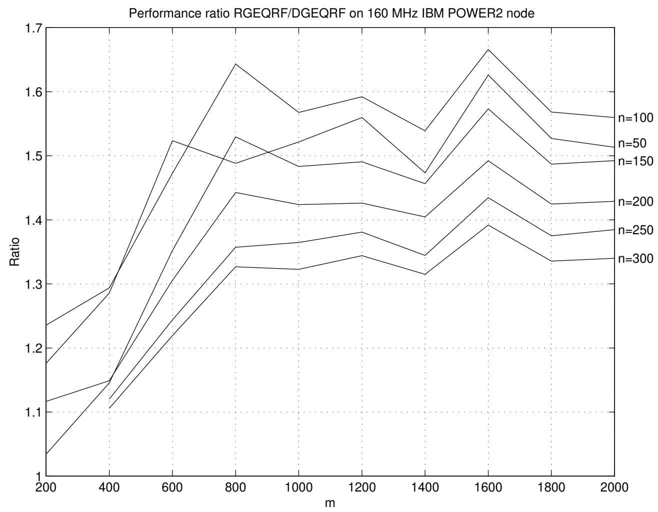

Fig. 7. Performance ratio of RGEQRF to DGEQRF for tall and thin matrices on the 160 MHz POWER2 processor.

In Figures 9 and 10 we present results for tall and extremely thin matrices, i.e., n goes from 4 to 20. Also here, RGEQRF outperforms DGEQRF for all choices of m and n.

4 The Parallel RGEQRF

The parallel version of RGEQRF for the next release of the IBM ESSL library is a slight modification of the software version described in [5, 6]. The mini blocking

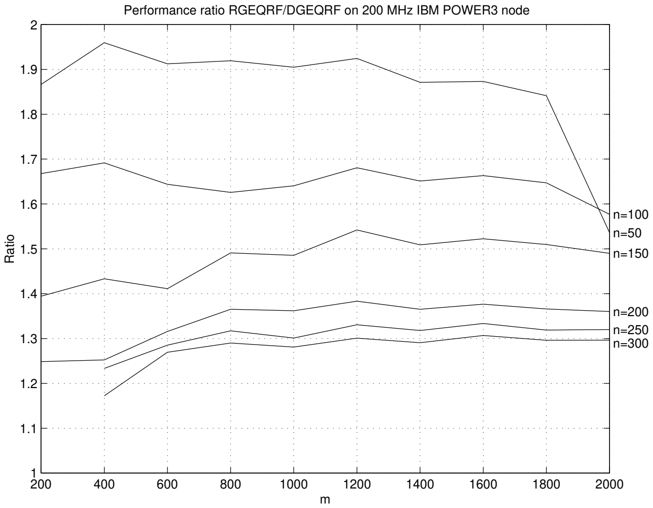

Fig. 8. Performance ratio of RGEQRF to DGEQRF for tall and thin matrices on the 200 MHz POWER3 processor.

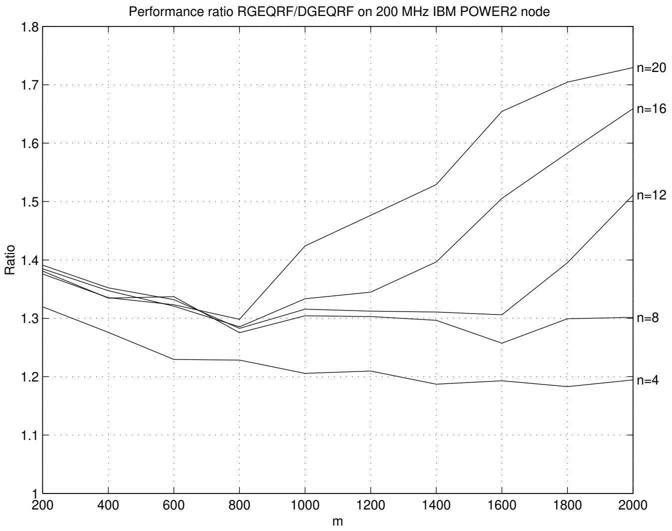

Fig. 9. Performance ratio of RGEQRF to DGEQRF for tall and very thin matrices on the 160 MHz POWER2 processor.

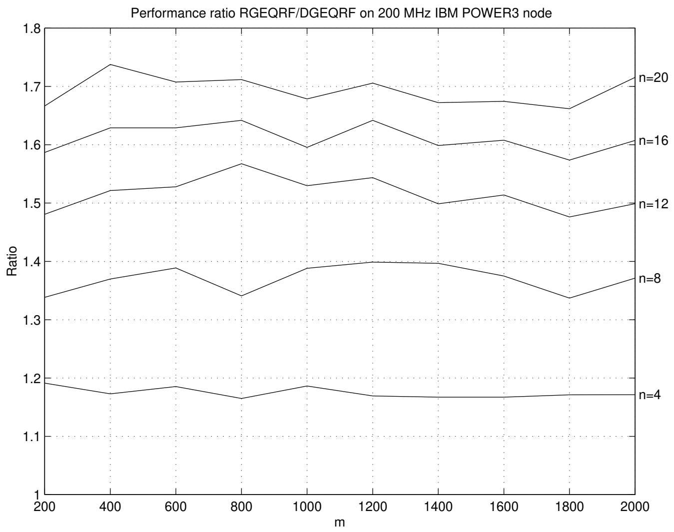

Fig. 10. Performance ratio of RGEQRF to DGEQRF for tall and very thin matrices on the 200 MHz POWER3 processor.

techniques described for the serial RGEQRF is included also in the parallel version. The $n > m$ case is handled in a straight-forward way. The updates in RGEQRF of the right hand side of the matrix are performed by first partitioning the matrix into a number of block columns. The submatrix computations to update these blocks, and to factorize the next block as soon as it is updated, form a pool of tasks. A processor is assigned a new job from the available tasks as soon as its previous job is completed. This is basically performed as described in [5, 6] with some modification in order to allow for updates of columns with column indices $> m$. In [5] parallel speedups of 1.97, 2.99, and 3.97 was obtained for 2, 3, and 4 processors on a 332MHz 4-way SMP PPC604e node. Due to lack of space we do not include any performance figures for the new ESSL version.

5 Concluding remarks

The new IBM ESSL QR factorization routines for uniprocessor and SMP systems have been presented. This contribution builds on the results on recursive QR factorization algorithms in [5, 6] and the main new feature is improved performance especially for small matrices. The improvement comes from pruning the recursion tree, or equivalently, stopping the recursion at $n = mb$, and then performing the QR factorization of the mb wide block column as a direct calculation. We call the concept mini blocking or register blocking. The improved performance provided by the mini blocking leads to a significant performance improvement for small matrices, but as a positive side effect, the overall performance for large matrices is also improved. Notably, the new RGEQRF with mini blocking outperforms the corresponding LAPACK routine for all combinations of m and n tested. The performance gain is at least 10% and at best it is around 100%.

6 Acknowledgements

The authors especially thank Joan McComb of ESSL Development in Poughkeepsie. In particular she was able to track down a subtle timing problem our algorithm encountered using the IBM XLF SMP compiler. We again thank Bo Kågström and Carl Tengwall for setting up our collaboration.

This research was conducted using the resources of the High Performance Computing Center North (HPC2N) and the Danish Computing Centre for Research and Education (UNI•C). Financial support has been provided by the Swedish Research Council for Engineering Sciences under grant TFR 98-604.

References

1. R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Develop*, 38(5):563–576, September 1994.
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* SIAM, Philadelphia, 1994.
3. C. Bischof and C. Van Loan. The *WY* representation for products of householder matrices. *SIAM J. Scientific and Statistical Computing*, 8(1):s2–s13, 1987.
4. A. Chalmers and J. Tidmus. *Practical Parallel Processing.* International Thomson Computer Press, UK, 1996.
5. E. Elmroth and F. Gustavson. Applying Recursion to Serial and Parallel *QR* Factorization Leads to Better Performance. *IBM Journal of Research and Development*, 44, No. 4, 605–624, 2000.
6. E. Elmroth and F. Gustavson. New serial and parallel recursive *QR* factorization algorithms for SMP systems. In B. Kågström et al., editors, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, No. 1541, pages 120–128, 1998.
7. F. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, Vol. 41, No. 6, 1997.
8. F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström and P. Ling. Superscalar GEMM-based Level 3 BLAS – The On-going Evolution of a Portable and High-Performance Library. In Kågström et al. (eds), *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, Vol. 1541, pp 207-215, Springer-Verlag, 1998.
9. F. Gustavson and I. Jonsson. High Performance Cholesky Factorization via Blocking and Recursion that uses Minimal Storage. *This Proceedings.*
10. R. Schreiber and C. Van Loan. A storage efficient *WY* representation for products of householder transformations. *SIAM J. Scientific and Statistical Computing*, 10(1):53–57, 1989.
11. S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM J. Matrix. Anal. Appl.*, 18(4), 1997.