A recursive formulation of Cholesky factorization of a matrix in packed storage*

Bjarne S. Andersen† Fred G. Gustavson‡
Jerzy Waśniewski†

Abstract

A new compact way to store a symmetric or triangular matrix called RPF for Recursive Packed Format is fully described. Novel ways to transform RPF to and from standard packed format is included. A new algorithm, called RPC for Recursive Packed Cholesky that operates on the RPF format is presented. Algorithm RPC is level 3 BLAS based and require algorithms **TRSM** and **SYRK** that work on RPF. We thus introduce and fully describe novel recursive algorithms **RP_TRSM** and **RP_SYRK** that the RPC algorithm requires. It turns out, that both **RP_TRSM** and **RP_SYRK** only call **GEMM**. Hence RPC mostly calls **GEMM** during execution.

The advantage of this storage scheme compared to traditional packed storage is demonstrated. First, both storage schemes use the minimal amount of storage for the symmetric or triangular matrix. Second, RPC gives a level 3 implementation of Cholesky factorization that only requires standard full format **GEMM** whereas standard packed implementations are only level 2. Hence, performance wise our RPC implementation is decidedly superior.

We present performance measurements on several current architectures that demonstrate order of magnitude improvements over the traditional packed routines. Also SMP parallel computations on the IBM SMP computer are made. The Graphs, which are attached in the appendix of the paper, show that the RPC algorithms are superior by a factor of 2 to 9 over the traditional packed algorithms.

*This work has been submitted for publication. Copyright may be transferred without further notice and the accepted version may then be posted by the publisher.

†Danish Computing Center for Research and Education (UNI•C), DTU, Building 304, DK-2800 Lyngby, Denmark, `Bjarne.Stig.Andersen@uni-c.dk` and `Jerzy.Wasniewski@uni-c.dk` respectively.

‡IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA, `gustav@watson.ibm.com`

1 Introduction

A very important class of linear algebra problems are those in which the coefficient matrix A is symmetric and positive definite [5, 11, 23]. Because of the symmetry it is only necessary to store either the upper or lower triangular part of the matrix.

Lower triangular case

$$\begin{pmatrix} 1 & & & & & & \\ 2 & 9 & & & & & \\ 3 & 10 & 17 & & & & \\ 4 & 11 & 18 & 25 & & & \\ 5 & 12 & 19 & 26 & 33 & & \\ 6 & 13 & 20 & 27 & 34 & 41 & \\ 7 & 14 & 21 & 28 & 35 & 42 & 49 \end{pmatrix}$$

Upper triangular case

$$\begin{pmatrix} 1 & 8 & 15 & 22 & 29 & 36 & 43 \\ & 9 & 16 & 23 & 30 & 37 & 44 \\ & & 17 & 24 & 31 & 38 & 45 \\ & & & 25 & 32 & 39 & 46 \\ & & & & 33 & 40 & 47 \\ & & & & & 41 & 48 \\ & & & & & & 49 \end{pmatrix}$$

Figure 1: The mapping of $7\times$ 7 matrix for the LAPACK Cholesky Algorithm using the **full** storage (**LDA**= 7 if in Fortran77)

Lower triangular case

$$\begin{pmatrix} 1 & & & & & & \\ 2 & 8 & & & & & \\ 3 & 9 & 14 & & & & \\ 4 & 10 & 15 & 19 & & & \\ 5 & 11 & 16 & 20 & 23 & & \\ 6 & 12 & 17 & 21 & 24 & 26 & \\ 7 & 13 & 18 & 22 & 25 & 27 & 28 \end{pmatrix}$$

Upper triangular case

$$\begin{pmatrix} 1 & 2 & 4 & 7 & 11 & 16 & 22 \\ & 3 & 5 & 8 & 12 & 17 & 23 \\ & & 6 & 9 & 13 & 18 & 24 \\ & & & 10 & 14 & 19 & 25 \\ & & & & 15 & 20 & 26 \\ & & & & & 21 & 27 \\ & & & & & & 28 \end{pmatrix}$$

Figure 2: The mapping of 7×7 matrix for the LAPACK Cholesky Algorithm using the **packed** storage

1.1 LAPACK POTRF and PPTRF subroutines

The LAPACK library[3] offers two different kind of subroutines to solve the same problems, for instance POTRF[1] and PPTRF both factorize symmetric, positive

[1]Four names SPOTRF, DPOTRF, CPOTRF and ZPOTRF are used in LAPACK for real symmetric and complex Hermitian matrices[3], where the first character indicates the precision and arithmetic versions: S – single precision, D – double precision, C – complex and Z – double complex. LAPACK95 uses one name LA_POTRF for all versions[7]. POTRF and/or PPTRF express, in this paper, any precision, any arithmetic and any language version of the PO and/or PP matrix factorization algorithms.

definite matrices by means of the Cholesky algorithm. The only difference is the way the triangular matrix is stored (see figures 1 and 2).

In the POTRF case the matrix is stored in one of the lower left or upper right triangles of a full square matrix[16, page 64] [2], the other triangle is wasted (see figure 1). Because of the uniform storage scheme, blocking and level 3 BLAS[8] subroutines can be employed, resulting in a high speed solution.

In the PPTRF case the matrix is kept in *packed* storage ([1], [16, page 74, 75]), which means that the columns of the lower or upper triangle are stored consecutively in a one dimensional array (see figure 2). Now the triangular matrix only occupies the strictly necessary storage space but the nonuniform storage scheme means that use of full storage BLAS is impossible and only the level 2 BLAS[20, 9] packed subroutines can be employed, resulting in a low speed solution.

To summarize, there is a choice between high speed with waste of memory versus low speed with no waste of memory.

1.2 A new Way of Storing Real Symmetric and Complex Hermitian and, in either case, Positive Definite Matrices

Together with some new recursively formulated linear algebra subroutines, we propose a new way of storing a lower or upper triangular matrix that solves this dilemma[14, 24]. In other words *we obtain the speed of* POTRF *with the amount of memory used by* PPTRF. The new storage scheme is named RPF, *recursive packed format*(see figure 4), and it is explained below.

The benefit of recursive formulations of the Cholesky factorization and the LU decomposition is described in the works of Gustavson [14] and Toledo [22]. The symmetric, positive definite matrix in the Cholesky case is kept in full matrix storage, and the emphasis in these works are the better data locality and thus better utilization of the computers memory hierarchy, that recursive formulations offer. However, the recursive packed formulation also has this property.

We will provide a very short introduction on a computer memory hierarchy and the Basic Linear Algebra Subprograms (BLAS) before describing the Recursive Packed Cholesky (RPC) and the Recursive Packed Format (RPF).

1.3 The Rationale behind introducing our New Recursive Algorithm, RPC and the New Recursive Data Format, RPF

Computers have several levels of memory. The flow of data from the memory to the computational units is the most important factor governing performance of engineering and scientific computations. The object is to keep the functional units running at their peak capacity. Through the use of a memory hierarchy

[2] In Fortran column major, in C row major.

system (see figure 3), high performance can be achieved by using locality of reference within a program. In the present context this is called blocking.

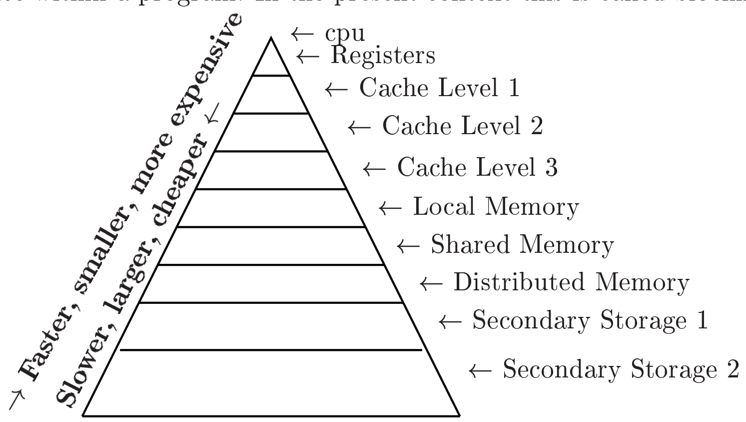


Figure 3: A computer memory hierarchy

At the top of the hierarchy is a Central Processing Unit (CPU). It communicates directly with the registers. The number of the registers is usually very small. A Level 1 cache is directly connected to the registers. The computer will run with almost peak performance if we are able to deliver the data to the L1 (level 1) cache in such way that the CPU is permanently busy. There are several books describing problems associated with the computer memory hierarchy. The literature in [10, 5, 11] is adequate for Numerical Linear Algebra specialists.

The memories near the CPU (registers and caches) have a faster access to CPU than the memories further away. The fast memories are very expensive and this is one of the reason that they are small. The register set is tiny. Cache memories are much larger than the set of registers. However, L1 cache is still not large enough for solving scientific problems. Even a subproblem like matrix factorization does not fit into cache if the order of the matrix is large.

A special set of Basic Linear Algebra Subprograms (BLAS) have been developed to address the computer memory hierarchy problem in the area of Numerical Linear Algebra. The BLAS are documented in [20, 9, 8, 6]. BLAS are very well summarized and explained for Numerical Linear Algebra specialists in [10, 5].

There are three levels of BLAS: Level 1 BLAS shows vector vector operations, Level 2 BLAS shows vector matrix (and/or matrix vector) operations, and Level 3 BLAS shows matrix matrix operations.

For Cholesky factorization one can make the following three observations with respect to the BLAS.

1. Level 3 implementations using full storage format run fast.
2. Level 3 implementations using packed storage format rarely exist. A level 3 implementation was previously used in [16], however, at great programming cost. Conventional Level 2 implementations using packed storage

format run, for large problem sizes, considerably slower than the full storage implementations.

3. Transforming conventional packed storage to RPF and using our RPC algorithm produces a Level 3 implementation using the same amount of storage as packed storage.

1.4 Overview of the Paper

Section 2 describes the new packed storage data format and the data transformations to and from conventional packed storage. Section 2.1 describes conventional lower and upper triangular packed storage. Section 2.2 discusses how to transform in place either a lower or upper trapezoid packed data format to recursive packed data format and vice versa. Section 2.3 describes the possibility to transpose the matrix while it is reordered from packed to recursive packed format and vice versa. Finally, in Section 2.4 the recursive aspects of the data transformation is described. These four subsections describe the in place transformation pictorially via several figures.

In Sections 3.1 and 3.2, recursive TRSM and SYRK, both which work on RPF, are described. Both routines do almost all their required floating point operations by calling level 3 BLAS GEMM. Finally, in Section 3.3, the RPC algorithm is described in terms of using the recursive algorithms of Sections 3.1 and 3.2. As in Section 2, all three algorithms are described pictorially via several figures. Note that the RPC algorithm only uses one Level 3 BLAS subroutine, namely GEMM. Usually the GEMM routine is very specialized, highly tuned and done by the computer manufacturer. If not, the ATLAS[25] GEMM can be used.

Section 4 explains that the RPC algorithm is numerically stable.

Section 5 describes performance graphs of the packed storage LAPACK[3] algorithms and of our recursive packed algorithms on several computers; the most typical computers like COMPACQ, HP, IBM SP, IBM SMP, INTEL Pentium, SGI and SUN were considered (figures 11, ..., 17). All these results show that the recursive packed Cholesky factorization (RP_PPTRF) and the solution (RP_PPTRS) are 4 – 9 times faster than the traditional packed subroutines. There are three more graphs. One demonstrates successful use of OpenMP[17, 18] parallelizing directives (figure 18). The second graph shows that the recursive data format is also effective for the complex arithmetic (figure 19). The third one shows the performance of all three algorithms for the Cholesky factorization (POTRF, PPTRF and RP_PPTRF) and the solution (POTRS, PPTRS and RP_PPTRS) (figure 20).

Section 6 discusses the most important developments in this paper.

2 The recursive packed storage

A new way to store triangular matrices in packed storage called *recursive packed* is presented. This is a storage scheme by its own right, and a way to explain it, is to describe the conversion from packed to recursive packed storage and vice versa (see figures 2 and 4).

Lower triangular case

$$
\left(\begin{array}{ccc|cccc}
1 & & & & & & \\
2 & 4 & & & & & \\
3 & 5 & 6 & & & & \\
\hline
7 & 11 & 15 & 19 & & & \\
8 & 12 & 16 & 20 & 21 & & \\
9 & 13 & 17 & 22 & 24 & 26 & \\
10 & 14 & 18 & 23 & 25 & 27 & 28
\end{array}\right)
$$

Upper triangular case

$$
\left(\begin{array}{ccc|cccc}
1 & 2 & 3 & 7 & 10 & 13 & 16 \\
 & 4 & 5 & 8 & 11 & 14 & 17 \\
 & & 6 & 9 & 12 & 15 & 18 \\
\hline
 & & & 19 & 20 & 22 & 24 \\
 & & & & 21 & 23 & 25 \\
 & & & & & 26 & 27 \\
 & & & & & & 28
\end{array}\right)
$$

Figure 4: The mapping of 7×7 matrix for the Cholesky Algorithm using the **recursive packed** storage. The recursive block division is illustrated.

2.1 Lower and upper triangular packed storage

Symmetric, complex hermitian or triangular matrices may be stored in *packed* storage form (see LAPACK Users' Guide [3], IBM ESSL Library manual[16, pages 66–67] and figure 2). The columns of the triangle are stored sequentially in a one dimensional array starting with the first column. The mapping between positions in full storage and in packed storage for a triangular matrix of size m is,

$A_{i,j}$	i,j	**UPLO**
$\mathbf{AP}_{i+(j-1)j/2}$	$1 \leq j \leq m$, $1 \leq i \leq j$	'U'
$\mathbf{AP}_{i+(j-1)(2m-j)/2}$	$1 \leq j \leq m$, $j \leq i \leq m$	'L'

[3]

The advantage of this storage is the saving of almost half [4] the memory compared to full storage.

[3] For UPLO = 'U' upper triangular and for UPLO = 'L' lower triangular of A is stored.

[4] At least $m \times (m-1)/2$. This formulae is a function of **LDA** (leading dimension of **A**) and m in Fortran77. The saving in Fortran77 is $m \times (2 \times \mathbf{LDA} - m - 1)/2$.

2.2 Reordering of a lower and upper trapezoid

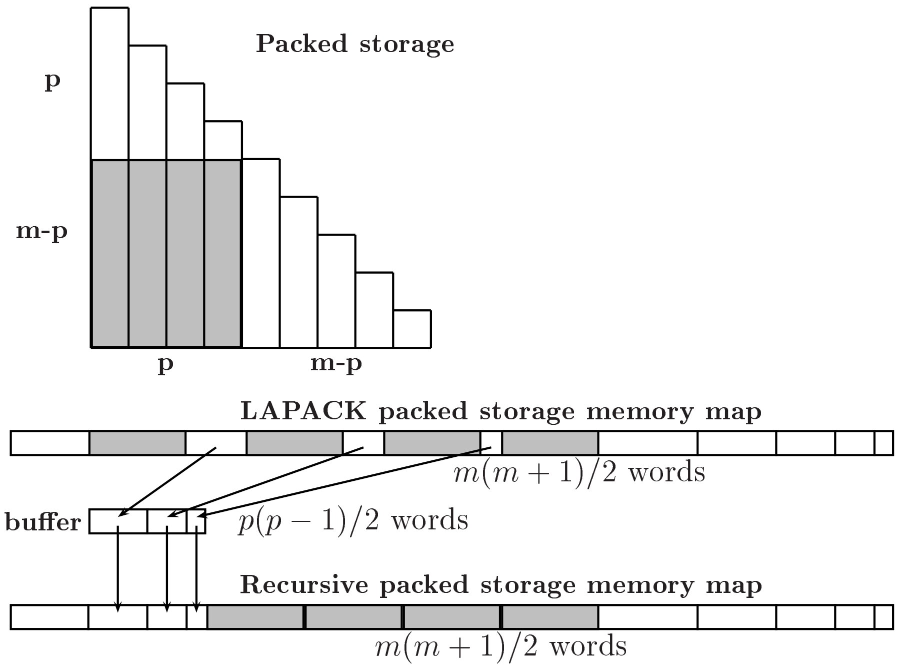


Figure 5: Reordering of the lower packed matrix. First, the last $p-1$ columns of the leading triangle are copied to the buffer. Then, in place, the columns of the accentuated rectangle are assembled in the bottom space of the trapezoid. Last, the buffer is copied back to the top of the trapezoid.

It is assumed that the matrices are stored in column major order, but the concepts in the paper are fully applicable also if the matrices are stored in row major order. As an intermediate step to transform a packed triangular matrix to a recursive packed matrix, the matrix is divided into two parts along a column thus dividing the matrix in a trapezoidal and a triangular part as shown in fig. 5 and 6. The triangular part remains in packed form, the trapezoidal part is reordered so it consists of a triangle in packed form, and a rectangle in *full* storage form. The reordering demands a buffer of the size of the triangle minus the longest column. The reordering in the lower case, fig. 5, takes the following steps. First the columns of the triangular part of the trapezoid are moved to the buffer (note that the first column is in correct place), then the columns of the rectangular part of the trapezoid are moved into consecutive locations and finally the buffer is copied back to the correct location in the reordered array. If p in figure 5 is chosen to $\lfloor m/2 \rfloor$ the rectangular submatrix will be square or deviate from a square only by a single column. The buffer size is $p(p-1)/2$ and the addresses of the lead-

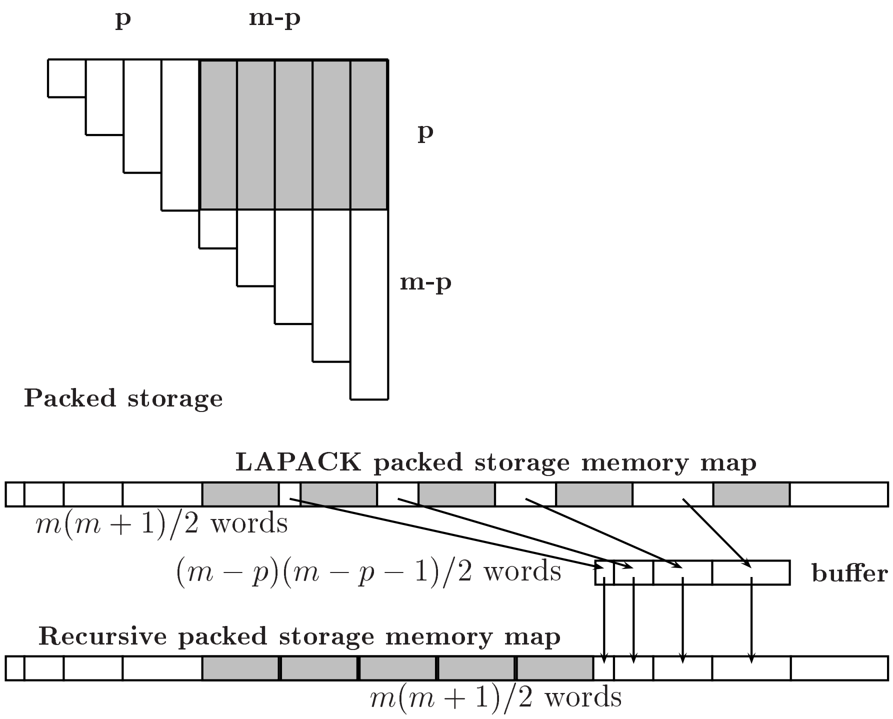


Figure 6: Reordering of the upper packed matrix. First, the first $m-p-1$ columns of the trailing triangle are copied to the buffer. Then, in place, the columns of the accentuated rectangle are assembled in the top space of the trapezoid. Last, the buffer is copied back to the bottom of the trapezoid.

ing triangle, the rectangular submatrix and the trailing triangle are given by,

- 1
- $1+p(p+1)/2$
- $1+mp-p(p-1)/2$

After the reordering the leading and trailing triangles are both in the same lower or upper packed storage scheme as the original triangular matrix. The reordering can be implemented as subroutines,

subroutine $TPZ\_TO\_TR(m, p, AP)$

and

subroutine $TR\_TO\_TPZ(m, p, AP)$

where TPZ_TO_TR means the reordering of the trapezoidal part from packed format to the triangular-rectangular format just described. TR_TO_TPZ is the opposite reordering.

2.3 Transposition of the rectangular part

The rectangular part of the reordered matrix are now kept in full matrix storage. If desired, this offers an excellent opportunity to transpose the matrix while it is transformed to recursive packed format. If the rectangular submatrix is square the transposition can be done completely in-place. If it deviates from a square by a column, a buffer of the size of the columns is necessary to do the transposition, for this purpose we can reuse the buffer used for the reordering.

2.4 Recursive application of the reordering

The method of reordering is applied recursively to the leading and trailing triangles which are still in packed storage, until finally the originally triangular packed matrix is divided in rectangular submatrices of decreasing size, all in full storage. The implementation of the complete transformation from packed to recursive packed format, P_TO_RP is (compare the figures 2 and 4),

recursive subroutine $P\_TO\_RP(m, AP)$
  if $(m > 1)$ **then**
    $p = \lfloor m/2 \rfloor$
    call $TPZ\_TO\_TR(m, p, AP)$
    call $P\_TO\_RP(p, AP)$
    call $P\_TO\_RP(m - p, AP(1 + mp - p(p-1)/2))$
  end if
end

and the inverse transformation from recursive packed to packed, RP_TO_P is,

recursive subroutine $RP\_TO\_P(m, AP)$
  if $(m > 1)$ **then**
    $p = \lfloor m/2 \rfloor$
    call $RP\_TO\_P(p, AP)$
    call $TR\_TO\_TPZ(m, p, AP)$
    call $RP\_TO\_P(m - p, AP(1 + mp - p(p-1)/2))$
  end if
end

The examples shown here concerns the lower triangular matrix, but the upper triangular transformation, and the transformation with transposition follows the same pattern. The figure 7 illustrates the recursive division of small lower and upper triangular matrices.

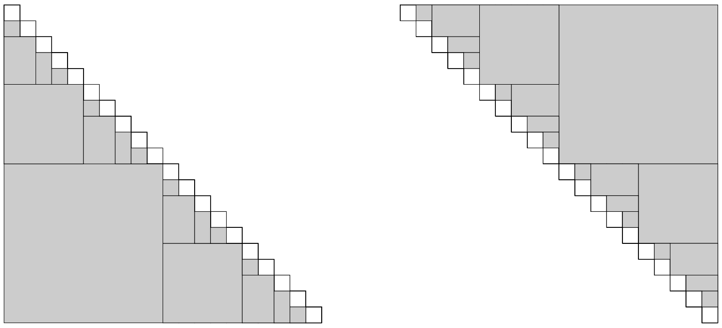


Figure 7: The lower and upper triangular matrices, in recursive packed storage data format, for $m = 20$. The rectangular submatrices, shown in the figures, are kept in full storage in column major order, in the array containing the whole matrices.

3 Recursive formulation of the Cholesky algorithm and its necessary BLAS

Two BLAS[6] operations, the triangular solver with multiple right hand sides, TRSM[5] and the rank k update of a symmetric matrix, SYRK are needed for the recursive Cholesky factorization and solution, RP_PPTRF[6] and RP_PPTRS[2]. In this section RP_TRSM, RP_SYRK, RP_PPTRF and RP_PPTRS are formulated recursively and their use of *recursive packed* operands are explained. TRSM, SYRK, PPTRF and PPTRS operate in various cases depending of the operands and the order of the operands. In the following we only consider single specific cases, but the deduction of the other cases follows the same guidelines.

All the computational work in the recursive BLAS routines RP_TRSM and RP_SYRK (and also RP_TRMM) is done by the non recursive matrix-matrix multiply routine GEMM[19, 25]. This is a very attractive property, since GEMM usually is or can be highly optimized on most current computer architectures.

The GEMM operation is very well documented and explained in [12, 13, 6]. The speed of our computation depends very much from the speed of a good GEMM. Good GEMM implementations are usually developed by computer manufacturers. The model implementation of GEMM can be obtained from

[5] On naming of TRSM, SYRK, HERK and GEMM see footnote of POTRF on page 1.

[6] The prefix RP_ indicates that the subroutine belongs to the Recursive Packed library, for example RP_PPTRF is the Recursive Packed Cholesky factorization routine.

netlib [6]; it works correctly but slowly. The Innovative Computing Laboratory at the University of Tennessee in Knoxville developed an automatic system called ATLAS[25] which usually can produce a very fast GEMM subroutine. Another automatic code generator scheme for GEMM was developed at Berkeley[4].

In ESSL, see [1], GEMM and all other BLAS are produced via blocking and high performance kernel routines. For example, ESSL produces a single kernel routine, DATB, which has the same function as the ATLAS on chip GEMM kernel. The principles underlying the production of both kernels are similar. The major difference is that ESSL's GEMM code is written by hand whereas ATLAS' GEMM code is parametrized and run over all parameter settings until a best parameter setting is found for the particular machine.

3.1 Recursive TRSM based on non-recursive GEMM

Fig. 8 shows the splitting of the TRSM operands. The operation now consists of the three suboperations,

$$\begin{array}{rcll} X_{11}A_{11}^T & = & \alpha B_{11} & \textbf{RP\_TRSM} \\ \hat{B}_{12} & = & B_{12} - \alpha^{-1} X_{11} A_{21}^T & \textbf{GEMM} \\ X_{12}A_{22}^T & = & \alpha \hat{B}_{12} & \textbf{RP\_TRSM} \end{array}$$

Based on this splitting, the algorithm can be programmed as follows,

recursive subroutine $RP\_TRSM(m, n, \alpha, AP, B)$
    if $(n == 1)$ **then**
        do $i = 1, m$
            $B(i,1) = \alpha B(i,1)/AP(1)$
        end do
    else
        $p = \lfloor n/2 \rfloor$
        call $RP\_TRSM(m, p, \alpha, AP, B)$
        call $GEMM('N','T', m, n-p, p, -\alpha^{-1}, B, AP(1+np-p(p-1)/2),$
                $n-p, 1, B(1, p+1))$
        call $RP\_TRSM(m, n-p, \alpha, AP(1+np-p(p-1)/2), B(1, p+1))$
    end if
end

3.2 Recursive SYRK based on non-recursive GEMM

Fig. 9 shows the splitting of the **SYRK** operands. The operation now consists of the three suboperations,

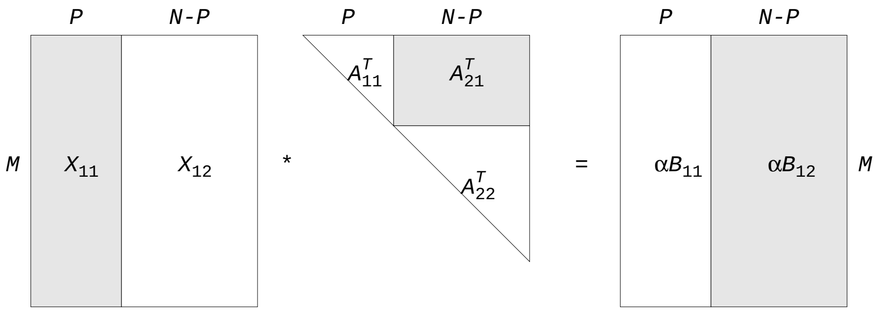


Figure 8: The recursive splitting of the matrices in the RP_TRSM operation for the case where SIDE=Right, UPLO=Lower and TRANSA=Transpose.

$$
\begin{array}{rcll}
C_{11} & = & \beta C_{11} + \alpha A_{11} A_{11}^T & \quad \textbf{RP\_SYRK} \\
C_{21} & = & \beta C_{21} + \alpha A_{21} A_{11}^T & \quad \textbf{GEMM} \\
C_{22} & = & \beta C_{22} + \alpha A_{21} A_{21}^T & \quad \textbf{RP\_SYRK}
\end{array}
$$

Based on this splitting, the algorithm can be programmed as follows,

```
recursive subroutine RP_SYRK(m, n, α, A, β, CP)
    if (m == 1) then
        CP(1) = β CP(1)
        do j = 1, m
            CP(1) = CP(1) + α A(1, j)^2
        end do
    else
        p = ⌊m/2⌋
        call RP_SYRK(p, n, α, A, β, CP)
        call GEMM('N', 'T', m − p, p, n, α, A(p + 1, 1), A,
                  β, CP(1 + p(p + 2)/2))
        call RP_SYRK(m − p, n, α, A(p + 1, 1), β, CP(1 + mp − p(p − 1)/2)
    end if
end
```

3.3 Recursive PPTRF and PPTRS based on recursive TRSM and recursive SYRK

Fig. 10 shows the splitting of the PPTRF operand. The operation now consists of four suboperations,

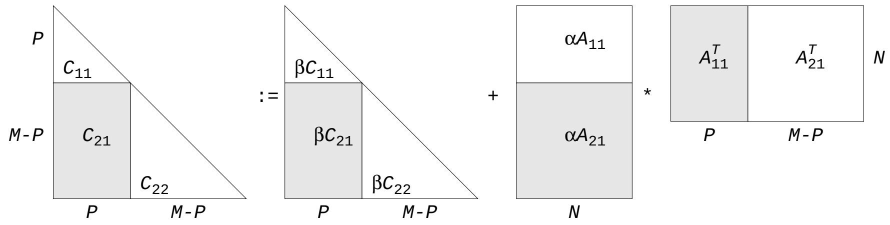


Figure 9: The recursive splitting of the matrices in the **RP_SYRK** operation for the case where UPLO=Lower and TRANS=No transpose.

$$
\begin{array}{rcll}
A_{11} & = & L_{11}L_{11}^T & \textbf{RP\_PPTRF} \\
A_{21} & = & L_{21}L_{11}^T & \textbf{RP\_TRSM} \\
\hat{A}_{22} & = & A_{22} - L_{21}L_{21}^T & \textbf{RP\_SYRK} \\
\hat{A}_{22} & = & L_{22}L_{22}^T & \textbf{RP\_PPTRF}
\end{array}
$$

Based on this splitting the algorithm can be programmed as follows,

```
recursive subroutine RP_PPTRF(m, AP)
    if (m == 1) then
        AP(1) = sqrt(AP(1))
    else
        p = ⌊m/2⌋
        call RP_PPTRF(p, AP)
        call RP_TRSM(m − p, p, 1.0, AP, AP(1 + p(p + 1)/2), m − p)
        call RP_SYRK(m − p, p, −1.0, AP(1 + p(p + 1)/2),
                     m − p, 1.0, AP(1 + mp − (p − 1)/2))
        call RP_PPTRF(m − p, AP(1 + mp − p(p − 1)/2)
    end if
end
```

The solution subroutine RP_PPTRS performs consecutive triangular solutions to the transposed and the non-transposed Cholesky factor. This routine is not explicitly recursive, as it just calls the recursive RP_TRSM twice.

4 Stability of the Recursive Algorithm

The paper [24] shows that the recursive Cholesky factorization algorithm is equivalent to the traditional algorithms in the books[5, 11, 23]. The whole the-

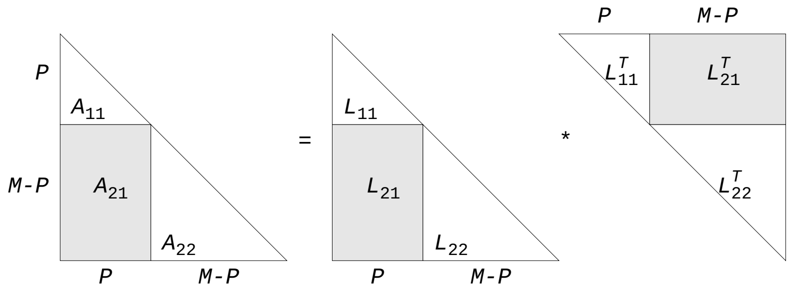


Figure 10: The recursive splitting of the matrix in the RP_PPTRF operation for the case where UPLO=Lower.

ory of the traditional Cholesky factorization and BLAS (TRSM and SYRK) algorithms carries over to the recursive Cholesky factorization and BLAS (TRSM and SYRK) algorithms described in Section 3. The error analysis and stability of these algorithms is very well described in the book of Nicholas J. Higham[15]. The difference between LAPACK algorithms PO, PP and RP[7] is how inner products are accumulated. In each case a different order is used. They are all mathematically equivalent, and, stability analysis shows that any summation order is stable.

5 Performance results

IBM	4 x PowerPC 604e	@ 332 MHz
IBM	Power2	@ 160 MHz
SUN	UltraSparc II	@ 400 MHz
SGI	R10000	@ 195 MHz
COMPAQ	Alpha EV6	@ 500 MHz
HP	PA-8500	@ 440 MHz
INTEL	Pentium III	@ 500 MHz

Table 1: Computer names

The new recursive packed BLAS (RP_TRSM and RP_SYRK), and the new recursive packed Cholesky factorization and solution (RP_PPTRF and RP_PPTRS) routines were compared to the traditional LAPACK subroutines, both concerning the results and the performance. The comparisons were made on seven

[7] full, packed and recursive packed.

different architectures, listed in the Table 1. The result graphs are attached in the appendix of this paper. The double precision arithmetic in Fortran90[21] was used in all cases.

IBM-PPC	ESSL 3.1.0.0	-lesslsmp
IBM-PW2	ESSL 2.2.2.0	-lesslp2
SUN	Sun Performance Library 2.0	-lsunperf
SGI	Standard Execution Environment 7.3	-lblas
COMPAQ	DXML V3.5	-ldxmp_ev6
HP	HP-UX PA2.0 BLAS Library 10.30	-lblas
INTEL	ATLAS 3.0 BETA	-latlas

Table 2: Computer library versions

The following procedure was used in carrying out our performance tests.

- On each machine the recursive and the traditional routines were compiled with the same compiler and compiler flags and they call the same vendor optimized, or otherwise optimized, BLAS library. The BLAS library versions can be seen in Table 2.
- The compared recursive and traditional routines received the same input and produced the same output for each time measurement. The time spent in reordering the matrix *to and from*[8] recursive packed format is included in the run time for both RP_PPTRF and RP_PPTRS. For the traditional routines there was no data transformation cost.
- The CPU time is measured by the timing function ETIME except on the PowerPC machine, which is a 4 way SMP. On this machine the run time was measured by the wall clock time by means of a special IBM utility function called RTC. Except for the operating system no other programs were running during these test runs.
- For each machine the timings were made for a sequence of matrix sizes ranging from $n = 300$ to $n = 3000$ in steps of $n = 100$. In case of the HP and Intel machines the matrix size starts at $n = 500$. We start at $n = 500$ because the resolution of the ETIME utility was too coarse. The number of right hand sides were taken to be $nrhs = n/10$. Due to memory limitations on the actual HP machine, this test series could only range to $n = 2500$.
- The operation counts for Cholesky factorization and solution are

$$NFP_{fac} = n^3/3 \quad \text{and} \quad NFP_{sol} = 2\,(nrhs)\,n^2,$$

[8] However it is only necessary to perform the *to* transformation in RP_PPTRF and no transformation in RP_PPTRS, to get the correct results.

where n is the number of equations and *nrhs* the number of right hand sides. These counts were used to convert run times to Flop rates.

Ten figures (figure 11, ..., figure 20) show performance graph comparisons, between the new RPC algorithms and the traditional LAPACK algorithms. The RPC algorithms use the RPF data format in all comparisons. As mentioned the cost of transforming from packed format to RPF and from RPF to packed format is included in the both the recursive packed factor and solve routines. The LAPACK subroutines DPPTRF, ZPPTRF, DPPTRS and ZPPTRS use packed data format, and DPOTRF and DPOTRS use full data format. Figure 20 compares all three algorithms RPC, LAPACK full storage and LAPACK packed storage.

Every figure has two subfigures and one caption. The upper subfigure shows comparison curves for Cholesky factorization. The lower subfigures show comparison curves of forward and backward substitutions. The captions describe details of the performance figures. The first seven figures (Figure 11, ..., 17) describe the same comparison of performance on several different computers.

5.1 The IBM SMP PowerPC

Figure 11 shows the performance on the the IBM 4-way PowerPC 604e 332 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) performs at about 100 MFlops. Performance of the 'U' graph is a little better than the 'L' graph. Performance remains constant as the order of the matrix increases.

The performance of the RPC factorization routine increases as n increases. The 'U' graph increases from 50 MFlops to almost 600 MFlops and the 'L' graph from 200 MFlops to 650 MFlops. The 'U' graph performance is better than the 'L' graph performance. The relative ('U', 'L') RPC algorithm performance is (4.9, 7.2) times better than the DPPTRF algorithm for large matrix sizes.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The DPPTRS routine performs about 100 MFlops for all matrix sizes. The RPC algorithm curve increases from 250 MFlops to almost 800 MFlops. The relative ('U', 'L') performance of the RPC algorithm is (5.7, 5.5) times faster than the DPPTRS algorithm for large matrix sizes.

The matrix size varies from 300 to 3000 on these subfigures.

5.2 The IBM Power2

Figure 12 shows the performance on the IBM Power2 160 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) 'U' graph performs at about 200 MFlops, the 'L' graph performs at about 150 MFlops. There is no increase in both graphs as the size of the matrix grows.

The performance graphs of the RPC factorization routine both increase, the 'U' graph from 300 to a little more than 400 MFlops, and the 'L' graph from

200 MFlops to 450 MFlops. The 'L' graph is better than the 'U' graph when the matrix sizes are between 750 and 3000. The 'U' graph is better than the 'L' graph when the matrix sizes are between 300 and 750. Both graphs grow very rapidly for matrix sizes between 300 and 500. The relative ('U', 'L') RPC algorithm performance is (1.9, 3.1) times faster than the DPPTRF algorithm for large matrix sizes.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The performance of the DPPTRS algorithm stays constant at about 250 MFlops decreasing slightly as n ranges from 300 to 3000. The performance of the RPC algorithm increases from 350 to more than 500 MFlops. The relative ('U', 'L') RPC algorithm performance is (2.3, 2.3) times faster than the DPPTRS algorithm for large matrix sizes.

The matrix size varies from 300 to 3000 on these subfigures.

5.3 The Compaq Alpha EV6

Figure 13 shows the performance on the the COMPAQ Alpha EV6 500 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) 'U' graph performs better than the 'L' graph. The difference is about 50 MFlops. The performance starts at about 300 MFlops, increases to 400 MFlops and than drops down to about 200 MFlops.

The performance of the RPC factorization routine increases as n increases. Both graphs (the 'U' and 'L' graphs) are almost equal. The 'U' graph is a little higher for matrix sizes between 300 and 450. The relative ('U', 'L') RPC algorithm performance is (3.4, 5.0) times faster than the DPPTRF algorithm for large matrix sizes.

For the routine DPPTRS the shape of the solution performance curves (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The performance of the DPPTRS routine decreases from 450 to 250 MFlops as n increases from 300 to 3000. The RPC performance curves increases from about 450 MFlops to more than 750 MFlops. The performance ('U', 'L') of the RPC algorithm is (3.3, 3.3) times faster than DPPTRS algorithm for large matrix sizes.

The matrix size varies from 300 to 3000 on these subfigures.

5.4 The SGI R10000

Figure 14 shows the performance on the the SGI R10000 195 MHz computer, on one processor only.

The LAPACK routine DPPTRF (the upper subfigure) 'U' graph performs better than the 'L' graph for matrix sizes from 300 to about 2000, after which both the 'U' and the 'L' graphs are the same. The DPPTRF performance slowly decreases.

The performance of the RPC factorization routine ('U' and 'L' graphs) increases from about 220 to 300 MFlops as n increases from 300 to about 1000, and stays constant as n increase to 3000. The relative ('U', 'L') RPC algorithm performance is (4.9, 4.9) times faster than the DPPTRF algorithm for large matrix sizes.

For the routine DPPTRS the shape of the solution performance curves (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The performance of the DPPTRS routine decreases from 130 MFlops to 60 MFlops as n increases from 300 to 3000. The Performance of the RPC solution routine increases in the beginning, and then runs constantly at about 300 MFlops. The performance ('U', 'L') of the RPC algorithm is (5.1, 5.2) times faster than the DPPTRS algorithm for large matrix sizes.

The matrix size varies from 300 to 3000 on these subfigures.

5.5 The SUN UltraSparc II

Figure 15 shows the performance on the the SUN UltraSparc II 400 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) 'U' and 'L' graphs show almost equal performance when $n > 1500$. These functions start between 200 and 225 MFlops and then decrease down to about 50 MFlops.

For the RPC factorization routine, the performance of the 'U' and 'L' graphs, are also almost equal over the whole interval. Their function values start from 250 MFlops, quickly rise to 350 MFlops and then slowly increase to about 450 MFlops. The RPC factorization ('U', 'L') algorithm is (9.7, 10.2) times faster than the DPPTRF algorithm for large matrix sizes.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The DPPTRS performance graphs decreases from 225 to 50 MFlops. The performance for the RPC solution graphs increases from 330 to almost 450 MFlops. The RPC solution ('U', 'L') algorithm is (10.0, 9.4) times faster than the DPPTRS algorithm for large matrix sizes.

The matrix size varies from 300 to 3000 on these subfigures.

5.6 The HP PA-8500

Figure 16 shows the performance on the the HP PA-8500 440 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) 'U' and 'L' graphs are decreasing functions. The 'U' graph function values go from about 370 to 100 MFlops. The 'L' graph function goes from 280 to about 180 MFlops.

The performance of the RPC factorization graphs are increasing functions as the matrix size increases from 1000 to 3000. The performance varies for matrix sizes between 500 and 1500. The 'U' graph function values range from about 700 MFlops to almost 800 MFlops, the 'L' graph function values range from 600 MFlops to a little more than 700 MFlops. The RPC algorithm ('U', 'L') is (4.7, 6.7) times faster than the DPPTRF algorithm for large matrix sizes.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The DPPTRS routine performance decreases from 300 MFlops to 200 MFlops. The RPC algorithm curve increases from 550

MFlops to almost 810 MFlops. The RPC algorithm ('U', 'L') is (5.2, 5.0) times faster than the DPPTRS algorithm for large matrices in the solution case.

The matrix size varies from 500 to 2500 on these subfigures.

5.7 The INTEL Pentium III

Figure 17 shows the performance on the INTEL Pentium III 500 MHz computer.

The LAPACK routine DPPTRF (the upper subfigure) 'U' and 'L' graphs are decreasing functions. The 'U' graph function ranges from about 100 to 80 MFlops. The 'L' graph function ranges from less than 50 to about 25 MFlops.

For the RPC factorization routine the 'U' and the 'L' graphs are almost equal. The graphs are increasing functions from about 200 to 310 MFlops. The RPC factorization algorithm ('U', 'L') is (4.2, 9.2) times faster than the DPPTRF algorithm for large matrices.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The DPPTRS performance graphs decreases from about 80 to about 50 MFlops. The RPC algorithm curves increases from 240 to about 330 MFlops. The RPC algorithm ('U', 'L') is (5.9, 6.0) times faster than the DPPTRS algorithm for large matrices.

The matrix size varies from 500 to 3000 on these subfigures.

5.8 The IBM SMP PowerPC with OpenMP directives

Figure 18 shows the performance on the the IBM 4-way PowerPC 604e 332 MHz computer.

These graphs demonstrate successful use of OpenMP[17, 18] parallelizing directives. The curves LAPACK(L), LAPACK(U), Recursive(L) and Recursive(U) are identical to the corresponding curves of figure 11. We compare curves Recursive(L), Recursive(U), Rec.Par(L) and Rec.Par(U). The Rec.Par(L) and Rec.Par(U) curves result from double parallelization. The RPC algorithms call a parallelized DGEMM and they are parallelized themselves by the OpenMP directives.

The Rec.Par(L) curve is not much faster than Recursive(L), sometimes it is slower. The Rec.Par(U) is the fastest, specially for large size matrices. The doubly parallelized RPC algorithm (Rec.Par(U)) is about 100 MFlops faster than the ordinary RPC algorithm (Recursive(U)). The relative ('U', 'L') RPC factorization algorithm performance is (5.6, 7.6) times faster than the DPPTRF algorithm for large matrices.

The RPC double parallelization algorithm for the solution (lower subfigure) exceeds 800 MFlops. The relative ('U', 'L') RPC solution algorithm performance is (6.5, 6.6) times faster than the DPPTRF algorithm for large matrices.

The matrix size varies from 300 to 3000 on these subfigures.

5.9 The INTEL Pentium III running Complex Arithmetic

Figure 19 shows the performance on the INTEL Pentium III 500 MHz computer.

This figure demonstrate the successful use of RPC algorithm for Hermitian positive definite matrices. The performance is measured in Complex MFlops. To compare with the usual real arithmetic MFlops the Complex MFlops should be multiplied by 4.

The LAPACK routine ZPPTRF (the upper subfigure) 'U' graph performs a little better than the 'L' graph. These routine performs at about 80 MFlops.

The RPC Hermitian factorization routine 'U' graph performs better than the 'L' graph. The RPC performance graphs are increasing functions. They go from 240 up to 320 MFlops. The RPC Hermitian factorization algorithm ('U', 'L') is (3.8, 4.3) times faster than the ZPPTRF algorithm for the large size matrices.

The performance of the RPC solution routine (the lower subfigure) for the 'L' and 'U' graphs are almost equal. The ZPPTRS performance decreases from about 108 to 80 MFlops. The RPC solution algorithm increases from about 240 up to more than 320 MFlops. The RPC algorithm ('U', 'L') is (3.9, 3.7) times faster than the ZPPTRS algorithm for large Hermitian matrices.

The matrix size varies from 500 to 3000 on these subfigures.

5.10 The INTEL Pentium III with all three Cholesky Algorithms

Figure 20 shows the performance on the INTEL Pentium III 500 MHz computer.

The graphs on this figure depict all three Cholesky algorithms, the LAPACK full storage (DPOTRF and DPOTRS) algorithms, the LAPACK packed storage (DPPTRF and DPPTRS) algorithms and the RPC (factorization and solution) algorithms.

The LAPACK packed storage algorithms (DPPTRF and DPPTRS) are previously explained on figure 17.

The DPOTRF routine (the upper subfigure), for both the 'U' and 'L' cases, performs better than the RPC factorization routine for smaller matrices. For larger matrices the RPC factorization algorithm performs equally well or slightly better than the DPOTRF algorithm.

The performance of the DPOTRS algorithms ('U' and 'L' graphs) are better than the RPC performance for this computer.

However, the POTRF and POTRS storage requirement is almost twice the storage requirement of the RPC algorithms.

The matrix size varies from 500 to 3000 on these subfigures.

6 Conclusion

We summarize and emphasize the most important developments described in our paper.

- A recursive packed Cholesky factorization algorithm based on BLAS Level 3 operations has been developed.

- The RPC factorization algorithm works with almost the same speed as the traditional full storage algorithm but occupies the same data storage as the traditional packed storage algorithm. Also see bullet 4.
- The user interface of the new packed recursive subroutines (RP_PPTRF and RP_PPTRS) is exactly the same as the traditional LAPACK subroutines (PPTRF and PPTRS). The user will see identical data formats. However, the new routines run much faster.
- Two separate routines are described here: RP_PPTRF and RP_PPTRS. The data format is always converted from LAPACK packed data format to the recursive packed data format before the routine starts its operation and converted back to LAPACK data format afterwards. The RP_PPSV subroutine exists in our package which is equivalent to the LAPACK PPSV routine. In the RP_PPSV subroutine the data is not converted between the factorization and the solution.
- New recursive packed Level 3 BLAS, RP_TRSM and RP_SYRK, written in Fortran90[21] were developed. They only call the GEMM routine.
- This GEMM subroutine can be developed either by the computer manufacturer or generated by ATLAS system[25]. The ATLAS generated GEMM subroutine is usually compatible with the manufacturer developed routine.

Acknowledgements

This research was partially supported by the LAWRA project, the UNI•C collaboration with the IBM T.J. Watson Research Center at Yorktown Heights. The last two authors were also supported by the Danish Natural Science Research Council through a grant for the EPOS project (Efficient Parallel Algorithms for Optimization and Simulation).

References

[1] R.C. Agawal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism on power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5):563–576, September 1994.

[2] B.S. Andersen, F. Gustavson, A. Karaivanov, J. Waśniewski, and P.Y. Yalamov. LAWRA – Linear Algebra with Recursive Algorithms. In R. Wyrzykowski, B. Mochnacki, H. Piech, and J. Szopa, editors, *Proceedings of the* 3^{th} *International Conference on Parallel Processing and Applied Mathematics, PPAM'99*, pages 63–76, Kazimierz Dolny, Poland, 1999. Technical University of Częstochowa.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHIPAC: a portable, high-performance, ansi c coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, Jul 1997. ACM Sigarc.

[5] J.W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[6] J. Dongarra et al. BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/. Ongoing Projects at the Innovative Computing Laboratory, Computer Science Department, University of Tennessee at Knoxville, USA.

[7] J. Dongarra and J. Waśniewski. High Performance Linear Algebra Package – LAPACK90. In P.M. Pardalos and S. Rajasekaran, editors, *Advances in Randomized Parallel Computing*, volume 5 of *Combinatorial Optimization*, pages 241–275. Kluwer Academic Publishers, 1999. Available also from http://www.netlib.org/lapack/lawns/lawn134.ps.

[8] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–28, March 1990.

[9] J. J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–32, March 1988.

[10] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High–Performance Computers*. SIAM, 1998.

[11] G. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[12] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive Blocked Data Formats and BLAS' for Dense Linear Algebra Algorithms. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Proceedings of the* 4^{th} *International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'98*, number 1541 in Lecture Notes in Computer Science Number, pages 195–206, Umeå, Sweden, June 1998. Springer.

[13] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar GEMM-based Level 3 BLAS – The On-going Evolution of Portable and High-Performance Library. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Proceedings of the* 4^{th} *International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'98*, number 1541 in Lecture Notes in Computer Science Number, pages 207–215, Umeå, Sweden, June 1998. Springer.

[14] F.G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6), November 1997.

[15] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[16] IBM. *IBM Engineering and Scientific Subroutine Library for AIX*, Version 3, Volume 1 edition, December 1997. Pub. number SA22–7272–0.

[17] IBM. *XL Fortran AIX, Language Reference*, first edition, Dec 1997. Version 5, Release 1.

[18] IBM. *XL Fortran AIX, User's Guide*, first edition, Nov 1997. Version 5, Release 1.

[19] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[20] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[21] M. Metcalf and J. Reid. *FORTRAN 90/95 Explained*. Oxford University Press, Oxford, UK, second edition, 1996.

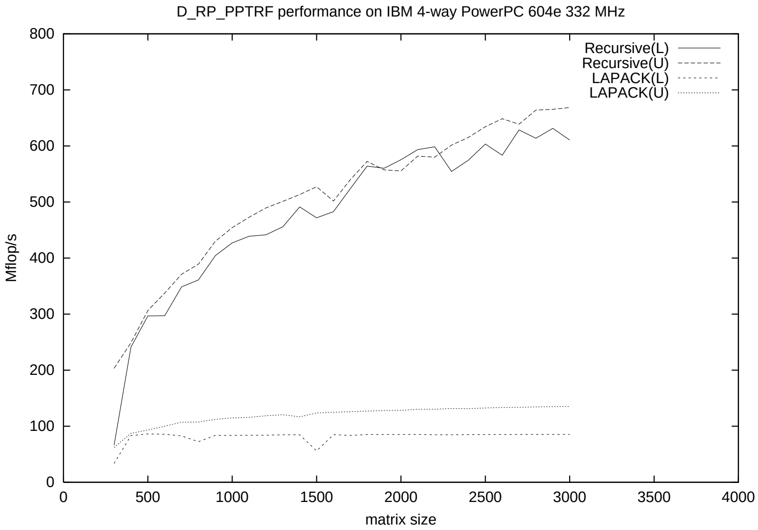
[22] S. Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4), 1997.

[23] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[24] J. Waśniewski, B.S. Andersen, and F. Gustavson. Recursive Formulation of Cholesky Algorithm in Fortran 90. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Proceedings of the* 4^{th} *International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'98*, number 1541 in Lecture Notes in Computer Science Number, pages 574–578, Umeå, Sweden, June 1998. Springer.

[25] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). http://www.netlib.org/atlas/, 1999. University of Tennessee at Knoxville, Tennessee, USA.

Appendix: Performance Graphs



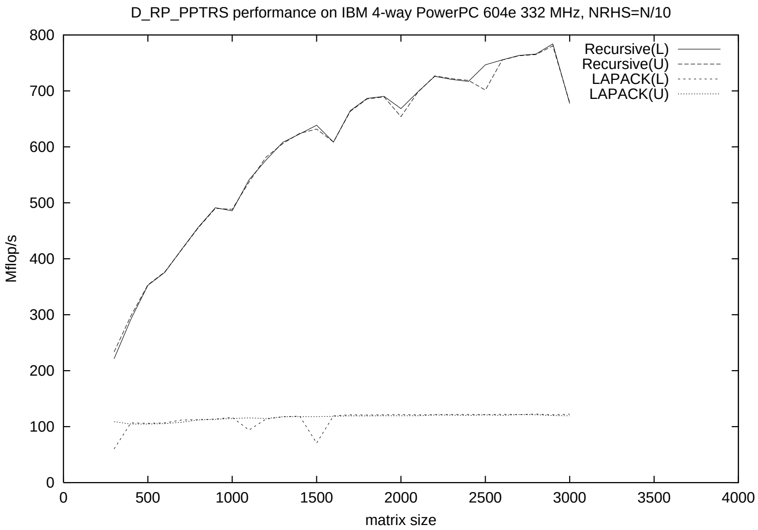
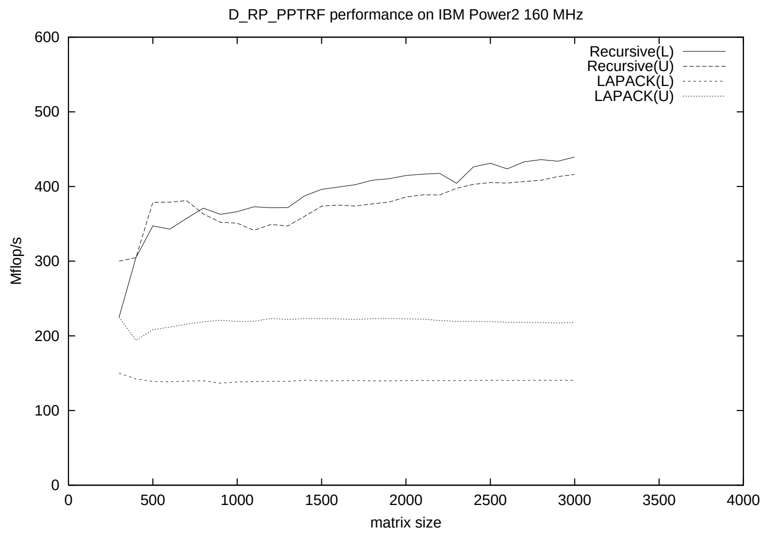


Figure 11: Performance of the recursive Cholesky factorization and solution on IBM 4 x PowerPC 604e, @ 332 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for the PowerPC architecture.



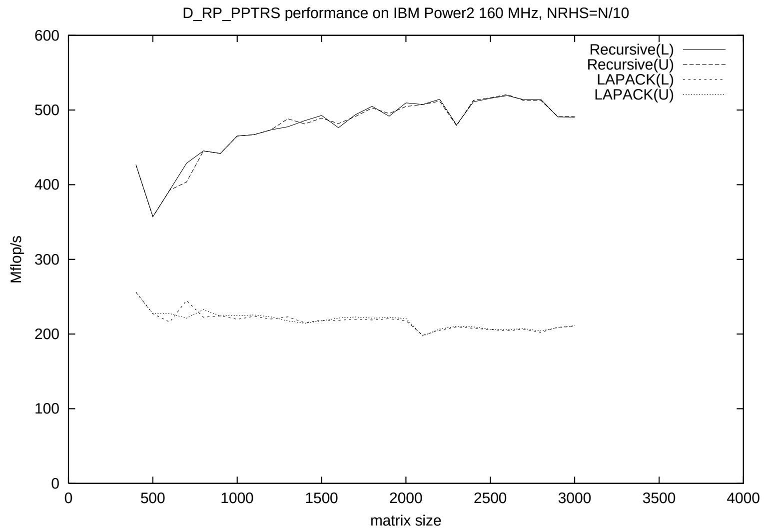
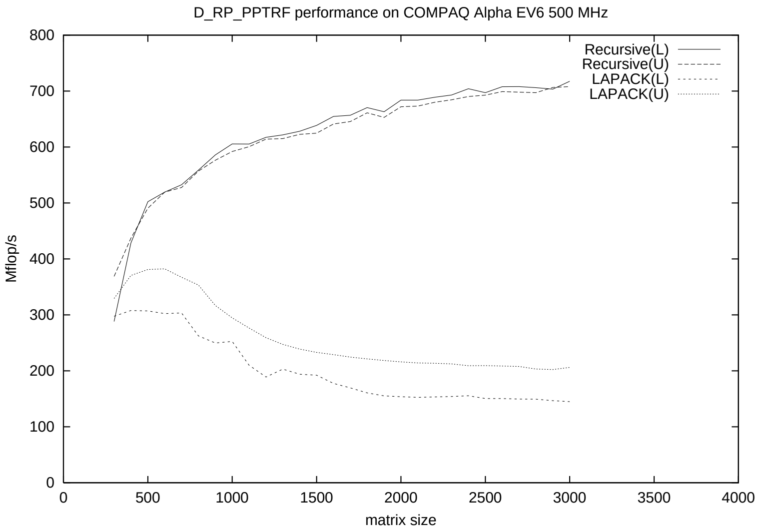


Figure 12: Performance of the recursive Cholesky factorization and solution on IBM Power2, @ 160 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for the Power2 architecture.



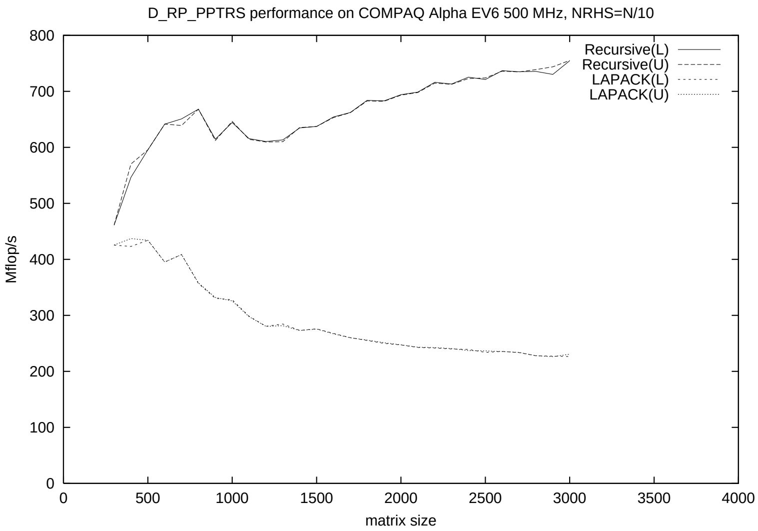


Figure 13: Performance of the recursive Cholesky factorization and solution on COMPAQ Alpha EV6, @ 500 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for the Alpha architecture.

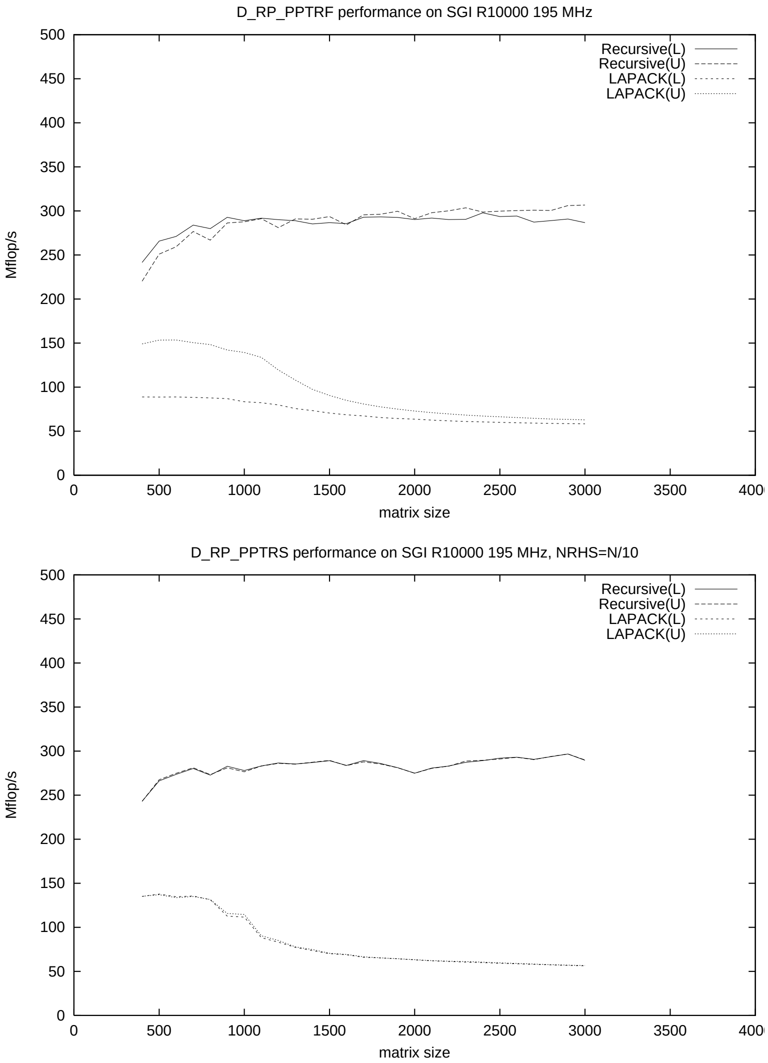


Figure 14: Performance of the recursive Cholesky factorization and solution on SGI R10000 @ 195 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for this SGI architecture.

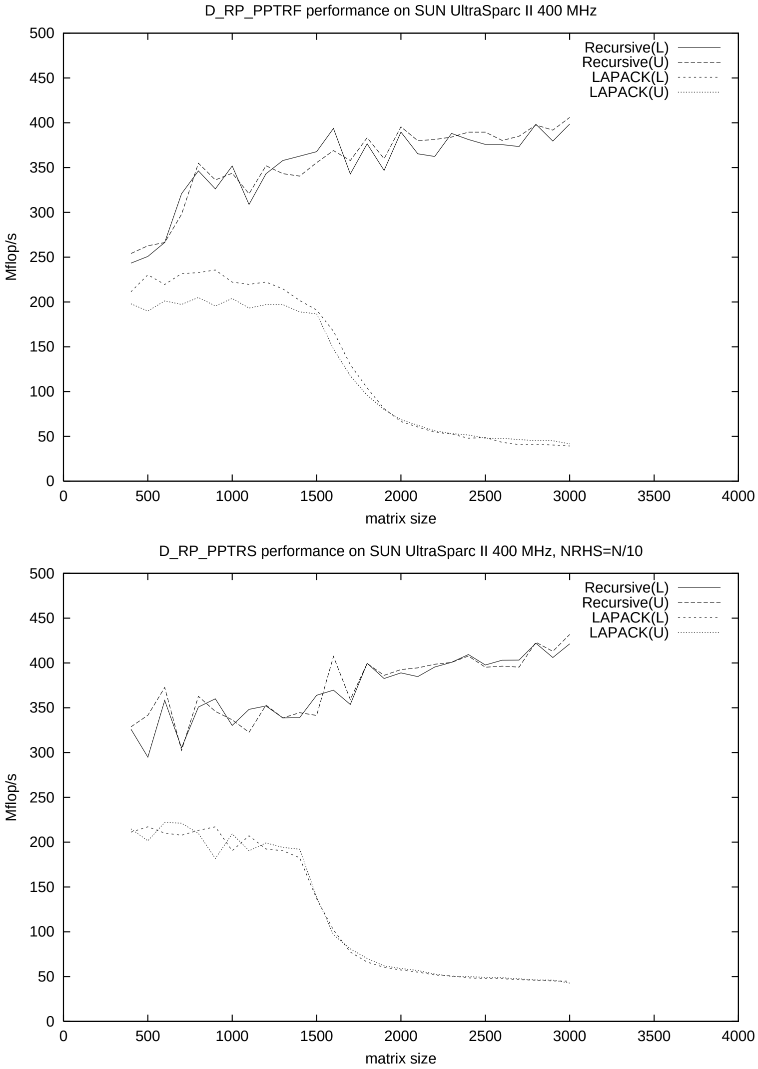


Figure 15: Performance of the recursive Cholesky factorization and solution on SUN UltraSparc II, @ 400 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for this SUN architecture.

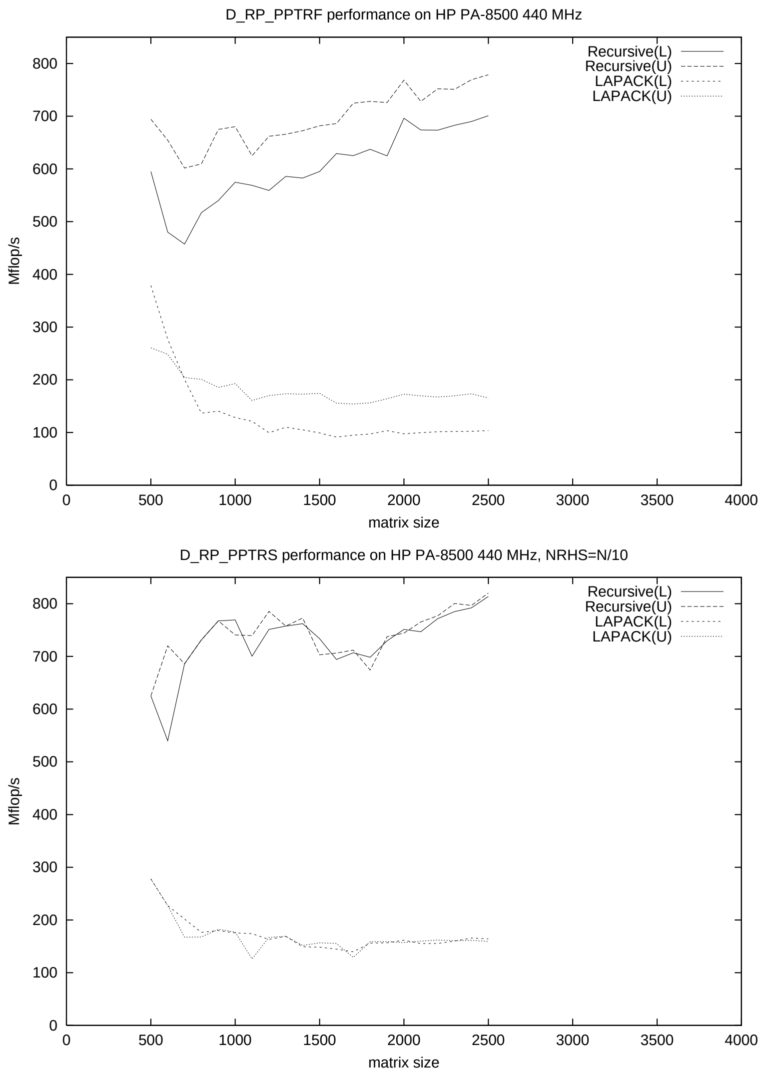
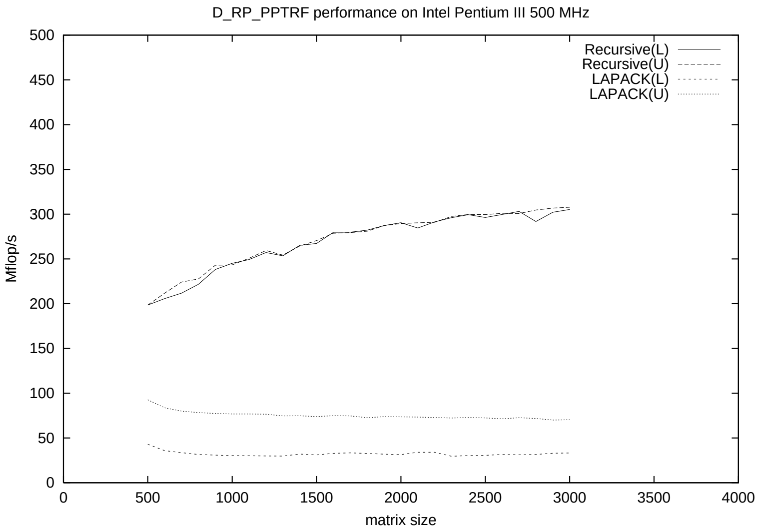


Figure 16: Performance of the recursive Cholesky factorization and solution on HP PA-8500, @ 440 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for this HP architecture.



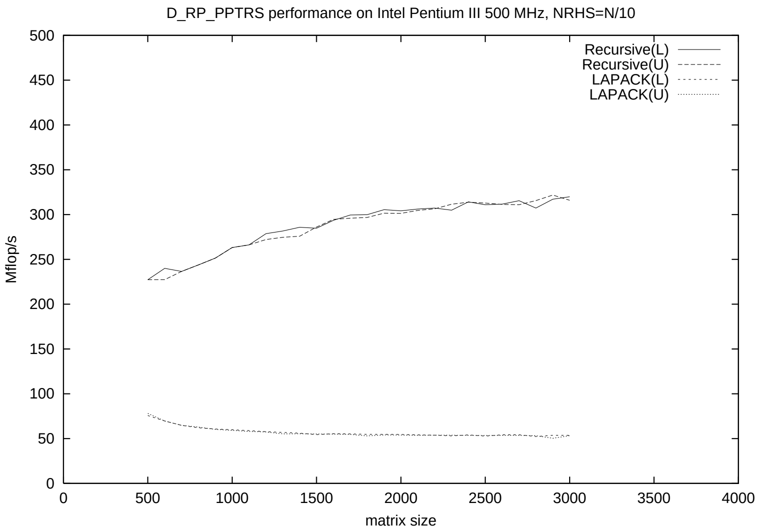
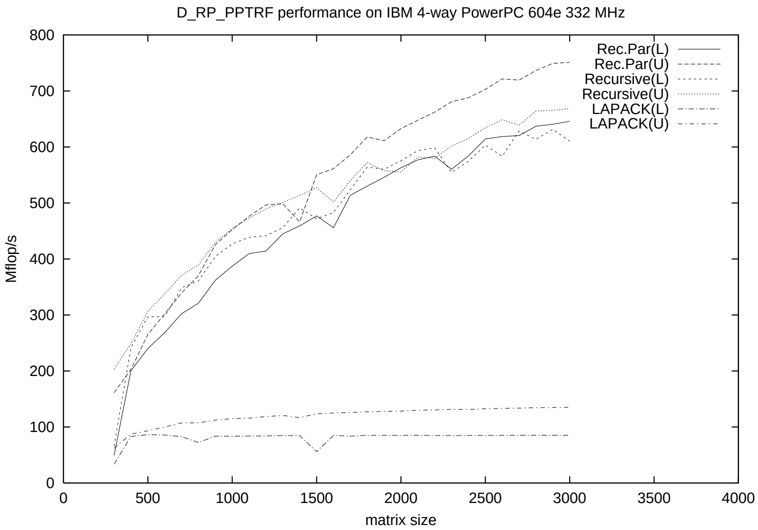


Figure 17: Performance of the recursive Cholesky factorization and solution on INTEL Pentium III, @ 500 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized ATLAS BLAS.



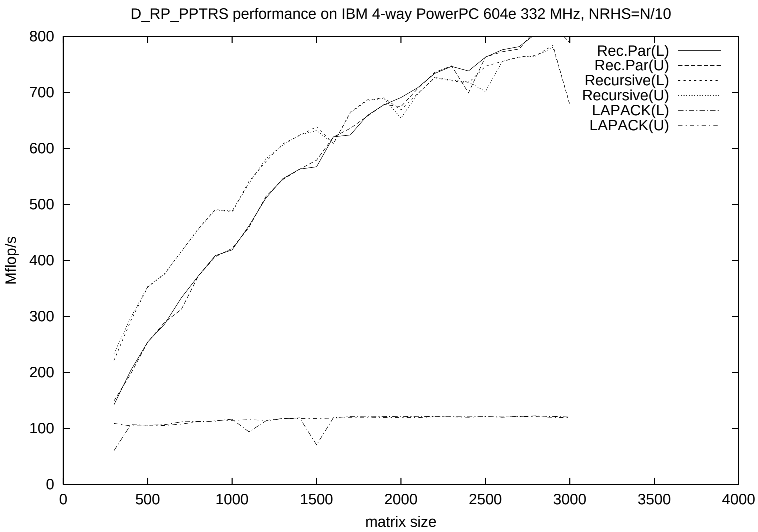
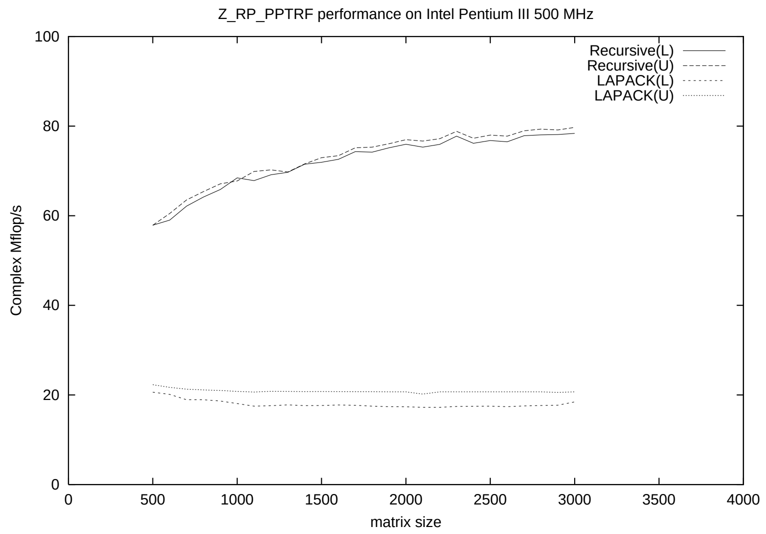


Figure 18: Performance of the recursive Cholesky factorization and solution on IBM 4 x PowerPC 604e, @ 332 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized BLAS for the PowerPC architecture. These graphs demonstrate successful use of OpenMP parallelizing directives. The Rec.Par(L) and Rec.Par(U) curves are results of the doubly parallelized RPC algorithms. They call the parallelized ESSL DGEMM and are parallelized themselves by the OpenMP directives.



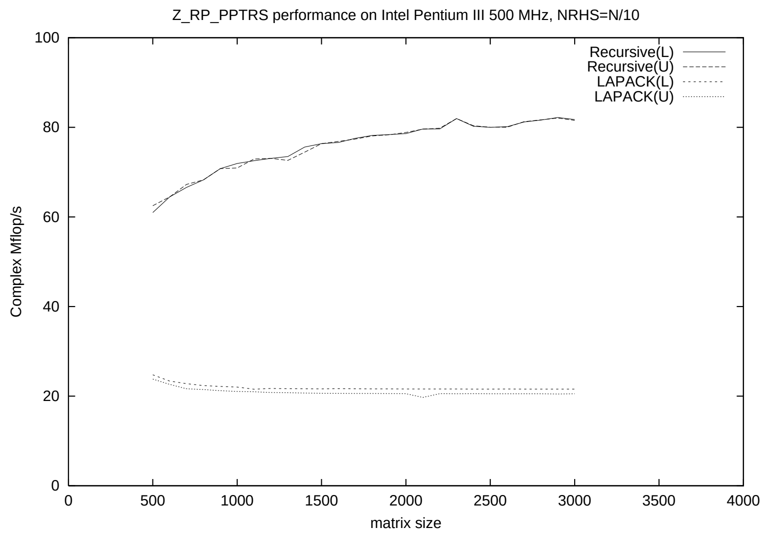


Figure 19: Performance of the recursive Hermitian Cholesky factorization and solution on INTEL Pentium III, @ 500 MHz. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized ATLAS BLAS (ZGEMM).

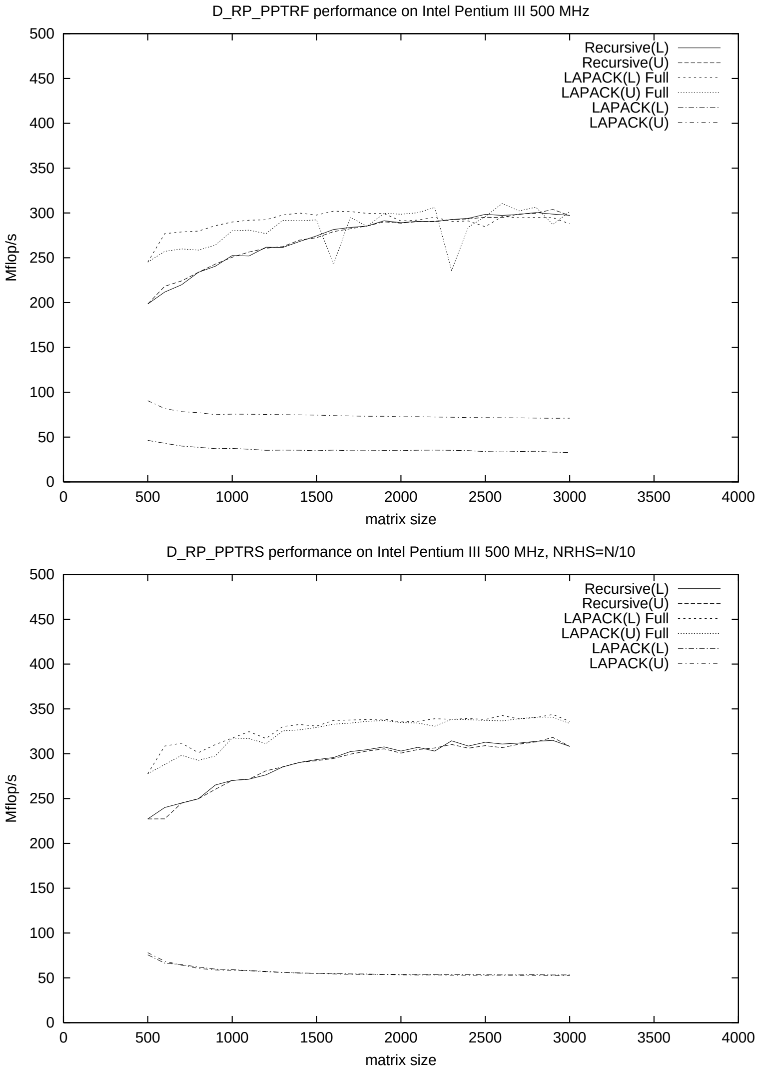


Figure 20: Performance of the recursive Cholesky factorization and solution on INTEL Pentium III, @ 500 MHz. The curves on this figure compare all three Cholesky factorization and solution algorithms. The LAPACK full storage (DPOTRF and DPOTRS), the LAPACK packed storage (DPPTRF and DPPTRS) and RPC (factorization and solution) algorithms. The recursive results include the time consumed by converting from packed to recursive packed storage and vice versa. All routines call the optimized ATLAS BLAS routines.