PERFORMANCE OPTIMIZATION OF SYMMETRIC FACTORIZATION ALGORITHMS

SILVIO TARCA

Abstract. Nonlinear optimization algorithms that use Newton's method to determine the search direction exhibit quadratic convergence locally. In the predominant case where the Hessian is positive definite, Cholesky factorization is a computationally efficient algorithm for evaluating the Newton search direction $-\nabla^2 f(x^{(k)})^{-1}\nabla f(x^{(k)})$. If the Hessian is indefinite, then modified Cholesky algorithms make use of symmetric indefinite factorization to perturb the Hessian such that it is sufficiently positive definite and reasonably well-conditioned, while preserving as much as possible the information contained in the Hessian. This paper measures and compares the performance of algorithms implementing Cholesky factorization, symmetric indefinite factorization and modified Cholesky factorization. From these performance data we estimate the work (runtime) involved in symmetric pivoting and modifying the symmetric indefinite factorization. Furthermore, we evaluate the effect of the degree of indefiniteness of the symmetric matrix on performance. For each of these matrix factorizations we developed routines that implement a variety of performance optimization techniques including loop reordering, blocking, and the use of tuned Basic Linear Algebra Subroutines.

1. Introduction. Nonlinear optimization algorithms generate a minimizing sequence of iterates $x^{(k)}$, where $x^{(k+1)} = x^{(k)} + t^{(k)}\Delta x^{(k)}$. The step length $t^{(k)}$ is positive, and the search direction $\Delta x^{(k)}$ in a descent method must satisfy $\nabla f(x^{(k)})^T\Delta x^{(k)} < 0$. The sequence of iterates $x^{(k)}$ terminates when an optimal point with sufficient accuracy has been found. In moving from one iterate to the next, nonlinear optimization algorithms determine a search direction $\Delta x^{(k)}$ and choose a step length $t^{(k)}$. A natural choice for the search direction is the negative gradient, $\Delta x^{(k)} = -\nabla f(x^{(k)})$. The gradient descent method moves in the direction $-\nabla f(x^{(k)})$ at every step, which provides a computational advantage since it only requires calculation of the gradient. It typically exhibits linear convergence, but can be very slow, even for problems where the Hessian is moderately well-conditioned. By comparison, the Newton method, with search direction given by $\Delta x^{(k)} = -\nabla^2 f(x^{(k)})^{-1}\nabla f(x^{(k)})$, exhibits quadratic convergence locally [5]. When the Hessian $\nabla^2 f(x^{(k)})$ is positive definite, the Newton direction is guaranteed to be a descent direction; when the Hessian is indefinite, the Newton direction may not exist or satisfy the downhill condition. In the latter case, nonlinear optimization algorithms modify the Hessian to be sufficiently positive definite and reasonably well-conditioned, while preserving as much as possible the information contained in the Hessian.

Rearranging the equation for the Newton direction and introducing conventional linear algebra notation, we have $Ax = \nabla^2 f(x^{(k)})\Delta x^{(k)} = -\nabla f(x^{(k)}) = b$. In the predominant case in which the Hessian, $A = \nabla^2 f(x^{(k)})$, is positive definite, nonlinear optimization algorithms make use of Cholesky factorization to efficiently solve for the Newton direction. Cholesky algorithms factors A by computing the unique lower triangular matrix L with positive diagonal entries, where $A = LL^T$. Given $Ax = LL^Tx = b$, the Newton direction is then determined by solving the triangular systems of equations $Ly = b$ and $L^Tx = y$. Cholesky factorization involves $\frac{1}{3}n^3 + O(n^2)$ floating point operations (flops), while solving for the Newton direction adds $O(n^2)$ flops to the computation.

If the Hessian is indefinite, nonlinear optimization algorithms make use of modified Cholesky factorization to efficiently modify the Hessian and solve for the Newton direction. Modified Cholesky algorithms make use of symmetric indefinite factorization to find a matrix $\widehat{A} = A + E$, where $\widehat{A}$ is sufficiently positive definite, so that the Newton direction satisfies the downhill condition. Symmetric indefinite factorization takes the form $PAP^T = LDL^T$, where D is diagonal or block diagonal, L is unit lower triangular, and P is a permutation matrix for symmetric pivoting. We

consider two approaches to modified Cholesky factorization: the method proposed by Gill, Murray and Wright [14], where D is diagonal, and A is modified as the factorization proceeds; and the one proposed by Cheng and Higham [8], where D is block diagonal with block order 1 or 2, and $\widehat{A}$ is found by modifying a computed factorization of A. For both approaches the cost of modifying the factorization is a small multiple of n^2 flops. For symmetric indefinite factorization, numerical stability comes at the expense of pivoting. Partial pivoting makes only $O(n^2)$ comparisons of matrix elements but has the worst accuracy; complete pivoting involves $O(n^3)$ comparisons; and rook pivoting involves between $O(n^2)$ and $O(n^3)$ comparisons. The Gill-Murray-Wright algorithm uses a form of partial pivoting, while our implementation of the Cheng-Higham algorithm employs either Bunch-Kaufman [6] (partial) or bounded Bunch-Kaufman [2] (rook) pivoting. Given these estimates of the work (flops) involved in computing a modified Cholesky factorization, we anticipate that much of the variation in performance between modified Cholesky algorithms will be explained by the pivoting strategy employed.

This paper measures and compares the performance of algorithms implementing Cholesky factorization, symmetric indefinite factorization, and modified Cholesky factorization (Gill-Murray-Wright and Cheng-Higham algorithms). From these performance data we estimate the work (runtime) involved in symmetric pivoting and modifying the symmetric indefinite factorization. Additionally, for symmetric indefinite and modified Cholesky factorizations, we evaluate the effect of the degree of indefiniteness of the symmetric matrix on performance. For each of these matrix factorizations we implement a variety of performance optimization (tuning) techniques including loop reordering, blocking, and the use of tuned BLAS (Basic Linear Algebra Subroutines). By comparing the performance of these algorithms with a benchmark, the corresponding LAPACK (Linear Algebra PACKage) routine, this paper assesses the efficacy of various performance optimization techniques.

Section 2 lists the software developed for this research, and provides technical specifications for hardware, compilers and libraries. Section 3 introduces performance optimization techniques such as loop reordering and blocking to maximize data locality, and describes the use of efficient libraries including tuned BLAS and LAPACK. Matrix multiplication is used to demonstrate these performance optimization concepts. Unblocked and blocked algorithms for Cholesky factorization are explained in Section 4. We measure the performance of our implementation of basic and "optimized" algorithms for Cholesky factorization and compare it with that of the corresponding LAPACK routine to assess their efficiency. Section 5 outlines Bunch-Kaufman (partial), bounded Bunch-Kaufman (rook) and Bunch-Parlett [7] (complete) pivoting strategies, and discusses unblocked and blocked algorithms for symmetric indefinite factorization. In addition to measuring the performance of our implementation of these algorithms, we also evaluate the effect of pivoting strategy employed and the degree of indefiniteness of the symmetric matrix on performance. Section 6 builds on the discussion of standard Cholesky and symmetric indefinite factorizations to explain the Gill-Murray-Wright and Cheng-Higham algorithms for modified Cholesky factorization. Again, we measure the performance of our implementation of basic and optimized versions of these algorithms, and compare the work (runtime) involved in modifying the symmetric indefinite factorization with the work to perform symmetric pivoting. Finally, Section 7 introduces parallel programming using the MPI (Message-Passing Interface) library, and demonstrates concepts of speedup and efficiency using Fox's algorithm for parallel matrix multiplication. This paper focuses on performance optimization of serial algorithms implementing matrix factorizations, so an obvious extension to this research would develop parallel algorithms for these matrix factorizations and measure their performance.

2. Hardware and Software. Timing experiments to measure the performance of algorithms analyzed in this research were conducted primarily on an Intel Xeon 5345 processor, 2.33 GHz, with 4 dual-cores, each dual-core sharing 4096 KB of cache. The operating system is Red Hat Linux release 5.1 running kernel `2.6.18.-128.7.1.el5_lustre.1.8.1.1` on CPU architecture `x86_64`. Software is coded in the C programming language [18], and compiled using Intel C compiler version 11.1.046 with `-O3` optimization level. Installed on this machine is Intel MKL version 10.2.2, which is compliant with LAPACK release 3.1. Parallel programming is implemented using MPI library functions, MVAPICH version 1.1.0. Unless explicitly stated otherwise, performance data presented in this paper are for routines executed on this machine, and references to BLAS and LAPACK in the context of this machine should be read as Intel MKL implementations of BLAS and LAPACK libraries. Timing experiments (jobs) were submitted using a portable batch system script.

In order to provide a comparison of performance across hardware, compilers and libraries, some timing experiments were also conducted on an alternative machine — AMD Opteron 180, 2.4 GHz, dual-core with 1024 KB of L2 cache per core. The operating system is Red Hat Linux release 5.4 running kernel `2.6.18-194.3.1.el5` on CPU architecture `x86_64`. Programs are compiled using GNU C compiler version 4.1.2 with `-O3` optimization level. Installed on this machine are ATLAS (Automatically Tuned Linear Algebra Software) versions of LAPACK and BLAS routines (`atlas.x86_64 3.6.0-15.el5`). References to BLAS and LAPACK in the context of this machine should be read as ATLAS implementations of BLAS and LAPACK libraries.

Source code developed for matrix factorization and matrix multiplication algorithms includes:

`lufact.c` Gaussian elimination (LU factorization)[1]
`cholfact.c` Cholesky factorization
`ldltfact.c` symmetric indefinite factorization with Bunch-Kaufman, bounded Bunch-Kaufman and Bunch-Parlett pivoting
`modchol.c` modified Cholesky algorithms (Gill-Murray-Wright and Cheng-Higham)
`matmult.c` matrix multiplication
`matmultp.c` parallel matrix multiplication

All matrix computations are performed using double-precision arithmetic.

To ensure that algorithms coded for this research perform matrix computations accurately, testing harnesses were developed for matrix factorization (`mfactest.c`) and serial and parallel matrix multiplication (`mmultest.c` and `mmultstp.c`, respectively). Then to measure the performance of, and profile these algorithms, timing harnesses were developed for matrix factorization (`mfactime.c`) and serial and parallel matrix multiplication (`mmultime.c` and `mmultmp.c`, respectively). Timing harnesses write performance data to an output file. Common matrix operations are coded in `matcom.c`, and timing functions used for measuring performance and profiling are coded in `timing.c`.

The source code, header files and `Makefiles` [21, 23] developed for this research are listed in the appendix, a separate document (`perf_optm_sym_factor_appx.pdf`) that accompanies this article. These documents, along with a tar archive file containing the source code, header files and `Makefiles`, are posted on the web pages `http://www.cs.cornell.edu/~bindel/students.html` and `http://www.cs.nyu.edu/overton/msadvising.html`.

[1] Although not discussed in this paper, LU factorization routines were developed as part of the research effort. LU factorization is normally used to introduce the concepts of matrix factorization and pivoting before one progresses to standard Cholesky factorization, symmetric indefinite factorization and modified Cholesky factorization.

3. Performance Optimization Guidelines. In optimizing the performance of matrix factorization code we consider: data locality; the use of efficient libraries; and compiler optimization levels. The goal in high performance computing is to keep the functional units of the central processing unit (CPU), which perform computations on input data, running at their peak capacity. Since moving data between levels of the memory hierarchy to the CPU (memory access) is the major performance bottleneck, high performance is achieved through data locality. At the top of a typical memory hierarchy are the registers of the CPU, followed by two levels of cache, then main memory, and finally disk storage. As one proceeds down the memory hierarchy, memory size increases but so does access time (latency) of the CPU to data in memory. To provide some orders of magnitude, approximate access times range from immediate for registers, one clock cycle for L1 cache, ten cycles for L2 cache and as many as one-hundred cycles for main memory [15].

There are two basic types of data locality: temporal and spatial. Temporal data locality means that if data stored in some memory location is referenced, then it likely will be referenced again in the near future. Spatial data locality means that if data stored in some memory location is referenced, then it is likely that data stored in nearby memory locations will be referenced in the near future. Efficient algorithms for matrix factorizations, the primary concern of this paper, employ memory access optimization techniques including loop reordering and blocking to maximize data locality. In general, blocked factorization algorithms iteratively factor a diagonal block, then solve a triangular system of equations to yield a column block of a lower triangular matrix or a row block of an upper triangular matrix, and finally update the trailing sub-matrix. Blocked algorithms for matrix factorization spend the bulk of their time updating the trailing sub-matrix, which is essentially a matrix multiplication operation. Therefore, we begin by demonstrating the maximization of data locality on matrix multiplication, and measuring the performance gains attributable to loop reordering and blocking. The observations we make will inform the performance optimization of matrix factorizations discussed in the subsequent sections of this paper.

```
for i = 1 : n
        for j = 1 : n
                for k = 1 : n
                        C(i, j) = C(i, j) + A(i, k)B(k, j)
                end
        end
end
```

FIG. 3.1. *Inner product method for matrix multiplication, $C = C + AB$.*

```
for j = 1 : n
        for k = 1 : n
                for i = 1 : n
                        C(i, j) = C(i, j) + A(i, k)B(k, j)
                end
        end
end
```

FIG. 3.2. *Implementation of SAXPY operation for matrix multiplication, $C = C + AB$.*

```
for j = 1 : n : r
    for k = 1 : n : r
        for i = 1 : n : r
            C(i:i+r−1, j:j+r−1) = C(i:i+r−1, j:j+r−1) +
                A(i:i+r−1, k:k+r−1)B(k:k+r−1, j:j+r−1)
        end
    end
end
```

FIG. 3.3. *Simple blocking algorithm for matrix multiplication,* $C = C + AB$*.*

Consider the matrix multiplication operation $C = C + AB$, where A, B and C are n-by-n matrices stored in column-major order. The simplest implementation of matrix multiplication, the inner product method outlined in Figure 3.1, employs ijk indexing. From a data locality perspective it is far from optimal, since the inner-most loop makes row and column accesses. The combined scalar multiplication and vector addition method, or SAXPY[2] operation, adds a scalar multiple of a column to another column. The SAXPY operation, which employs jki indexing, achieves better spatial data locality than the inner product method through loop reordering (Figure 3.2). Data locality is further improved by partitioning the matrices into blocks, where block size is chosen such that the three matrix blocks referenced in the inner-most loop can be stored in fast access cache [10]. Figure 3.3 outlines a simple blocking algorithm for matrix multiplication in pseudocode. The step size r of the nested loops is the dimension of the matrix blocks. Throughout this research, unblocked and simple blocking algorithms operate on matrices stored in column-major order[3].

Given n-by-n matrices A, B and C, the matrix multiplication operation $C = C + AB$ involves $2n^3$ floating point operations (flops). Figure 3.4 plots the performance gains achieved by improving data locality through loop reordering and blocking. Our implementation of the SAXPY operation produces a three-fold increase in performance over the inner product method, while our implementation of simple blocking roughly doubles performance relative to the SAXPY operation.

To minimize the cost of communication latency across memory hierarchy levels for matrix multiplication, sub-matrices optimally sized for different levels of the memory hierarchy should be stored contiguously so that data reuse is maximized within each hierarchy level [3]. In order to examine this idea, we implemented blocked algorithms that operate on data structures designed around the memory hierarchy. One data structure, which we refer to as contiguous block storage, stores matrix blocks sized for L2 cache contiguously. Another, which we refer to as recursive contiguous block storage, stores matrix blocks sized for L2 cache contiguously, and within each block, sub-blocks sized for L1 cache are stored contiguously. Throughout this research, contiguous blocking algorithms operate on matrix blocks stored in column-major order, while recursive contiguous blocking algorithms operate on sub-blocks stored in column-major order. Some experimentation led us to choose a block size equal to 96-by-96 and sub-block size equal to 8-by-8 for the matrix multiplication on the Intel machine. While our implementation of contiguous blocking marginally

[2]The term SAXPY comes from the BLAS single-precision routine which computes a scalar multiple of a vector added to another vector. In this paper we use SAXPY in the generic sense to refer to the combined scalar multiplication and vector addition operation irrespective of arithmetic precision. Routines developed for this research perform matrix computations using double-precision arithmetic.

[3]The C programming language stores two-dimensional arrays in row-major order. We store an n-by-n matrix A in a one-dimensional array $A[n \times n]$ in column-major order.

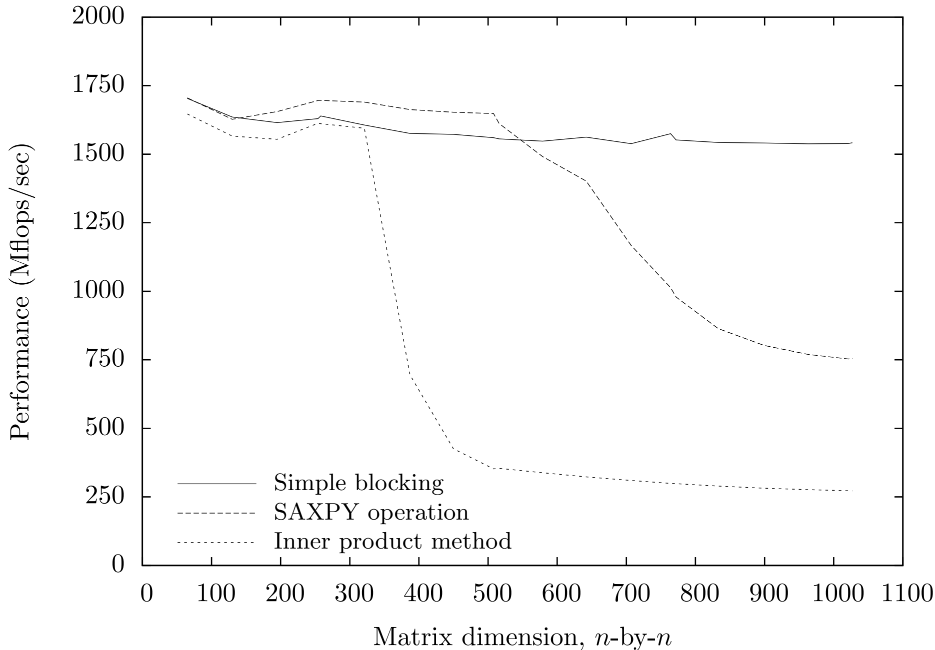

Fig. 3.4. *Data locality in matrix multiplication.*

outperforms simple blocking, recursive contiguous blocking[4] boosts performance by more than 40% relative to simple blocking (Figure 3.5).

Common matrix computations, such as matrix multiplication, have been standardized as Basic Linear Algebra Subroutines (BLAS) [4] and optimized by computer manufacturers for their machines. There is a hierarchy of BLAS routines: level 1 BLAS routines, including inner product and combined scalar multiplication and vector addition, perform $O(n)$ flops on vectors; level 2 BLAS routines, such as matrix-vector multiplication and solving a triangular system of matrix-vector equations, perform $O(n^2)$ flops on matrices and vectors; and level 3 BLAS routines, including matrix multiplication and solving a triangular system of matrix equations, perform $O(n^3)$ flops on matrices [9]. The software library LAPACK (Linear Algebra PACKage) provides routines for solving linear systems of equations, and least squares, eigenvalue and singular value problems. LAPACK routines implement blocked algorithms and use highly efficient level 3 BLAS to the fullest extent possible [20]. Ultimately, our goal is to reorganize the matrix factorization algorithms that we develop to exploit BLAS, in much the same manner that LAPACK does. Figure 3.5 compares the performance of our implementation of blocked algorithms with that of BLAS. The level 3 BLAS matrix multiplication routine `DGEMM` [4] runs at triple the speed of any of the blocked routines that we developed.

[4]To control looping in the matrix multiplication kernel, which operates on sub-blocks, we use a symbolic constant. This requires setting elements of the fringe sub-blocks that are not in the matrices to zero. Symbolic constants are evaluated during compilation, while variables are evaluated at run time. Performance is almost halved when using variables to control looping in the matrix multiplication kernel.

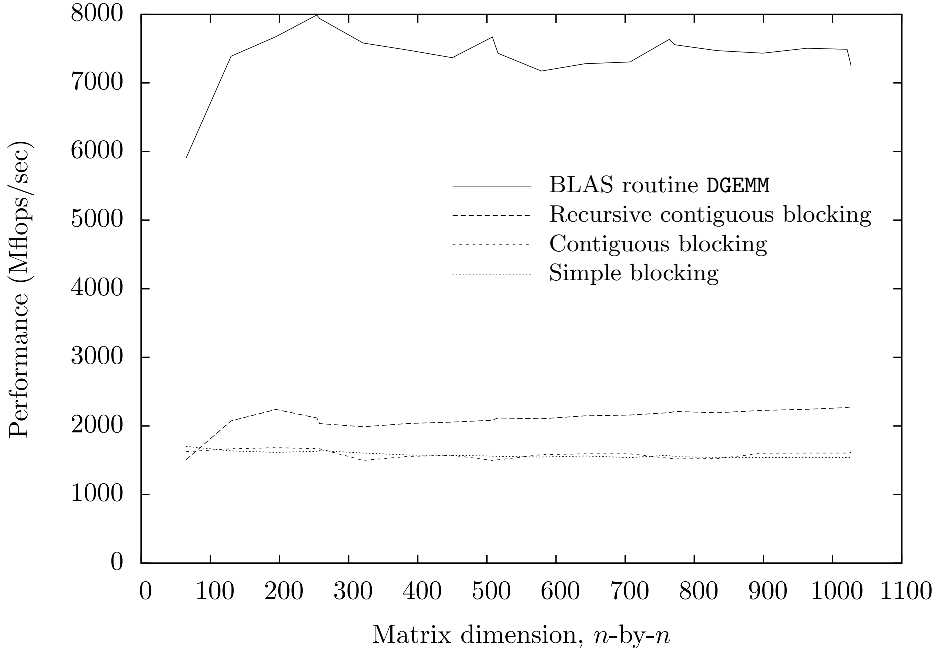

FIG. 3.5. *Blocked algorithms for matrix multiplication.*

In this research we evaluate the efficiency of our implementation of basic and optimized algorithms for matrix factorization by comparing their performance with that of the corresponding LAPACK routines. In this manner we can estimate the contribution to performance of the various performance optimization techniques. The LAPACK routines choose an optimal block size for the local environment based on the routine and matrix leading dimension. Our implementation of blocked algorithms uses the same block size as the corresponding LAPACK routine for a given matrix leading dimension, unless a different block size is found to produce superior performance.

Programs measuring the performance of algorithms are compiled at `-O3` optimization level, which turns on expensive optimizations with speed-space tradeoffs [17]. Our timing experiments find that even at an optimization level of `-O1`, optimizing floating point operations by hand using techniques such as loop unrolling and software pipelining is unnecessary.

4. Cholesky Factorization.

In order to solve a symmetric positive definite system of equations $Ax = b$ in a computationally efficient manner, we make use of a numerically stable factorization known as Cholesky factorization. If $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, then the Cholesky factor is the unique lower triangular matrix $L \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that $A = LL^T$. Given $Ax = LL^Tx = b$, x is recovered by solving the triangular systems of equations $Ly = b$ and $L^Tx = y$. Cholesky factorization takes $\frac{1}{3}n^3 + O(n^2)$ flops, while solving the triangular systems of equations involves $O(n^2)$ work. The numerical stability of Cholesky factorization follows from the inequalty $l_{ij}^2 \leq \sum_{k=1}^{i} l_{ik}^2 = a_{ii}$, which shows that the entries of lower triangular factor L are nicely bounded [16].

For this research, we implemented a number of Cholesky factorization algorithms employing a variety of performance optimization techniques. In the interests of clarity and brevity we identify each algorithm by its function name from the source code listings in the appendix when it is introduced in the discussion. Thereafter, we reference the algorithm by its function name.

```
for k = 1 : n
        A(k,k) = sqrt(A(k,k))
        for i = k+1 : n
                A(i,k) = A(i,k)/A(k,k)
        end
        for j = k+1 : n
                for i = j : n
                        A(i,j) = A(i,j) - A(i,k) * A(j,k)
                end
        end
end
```

Fig. 4.1. *Outer product method for Cholesky factorization.*

```
for j = 1 : n
        for k = 1 : j-1
                for i = j : n
                        A(i,j) = A(i,j) - A(i,k) * A(j,k)
                end
        end
        A(j,j) = sqrt(A(j,j))
        for i = j+1 : n
                A(i,j) = A(i,j)/A(j,j)
        end
end
```

Fig. 4.2. *Implementation of SAXPY operation for Cholesky factorization.*

We implemented two unblocked algorithms for computing the standard Cholesky factorization of a symmetric positive definite matrix. The outer product method (`chol_outer_product`) outlined in Figure 4.1 employs kji indexing. Each pass through the k loop subtracts the outer product of column $A(k+1:n,k)$ with its transpose from the lower triangular part of the trailing sub-matrix $A(k+1:n,k+1:n)$. With matrix elements stored in column-major order, each pass through the k loop accesses each row of the trailing sub-matrix, which makes for less than optimal data locality. An implementation of the SAXPY operation (`chol_saxpy`) in Figure 4.2, which uses jki indexing, improves data locality through loop reordering. In this case, each pass through the j loop subtracts a multiple (element $A(j,k)$) of column $A(j:n,k)$ from column $A(j:n,j)$ for $k=1,\ldots,j-1$. That is, the inner-most loop performs a combined scalar multiplication and vector addition operation. Because of symmetry, the Cholesky factorization algorithms need only update elements on and below the diagonal. Both unblocked algorithms overwrite $A(i,j)$ with $L(i,j)$ for $i \geq j$.

In order to factor large matrices efficiently, we turn to blocked algorithms. Suppose that we have factored the symmetric positive definite matrix A; then the factorization may be written in block form:

$$\begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix}.$$

We examine the block matrix operations required to factor matrix A by multiplying triangular block matrices L and L^T together, and equating terms with blocks of A. Because of symmetry we need only consider the lower triangular blocks of A.

$$\begin{pmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11}L_{11}^T & & \\ L_{21}L_{11}^T & A_{22} & \\ L_{31}L_{11}^T & A_{32} & A_{33} \end{pmatrix},$$

where

$$\begin{pmatrix} A_{22} & \\ A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix} + \begin{pmatrix} L_{22} & \\ L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{22}^T & L_{32}^T \\ & L_{33}^T \end{pmatrix}.$$

A_{11} can be factored into $L_{11}L_{11}^T$ using an unblocked Cholesky algorithm, say, `chol_saxpy`. With L_{11} known, we can recover L_{21} and L_{31} by solving the triangular systems of equations $L_{21}L_{11}^T = A_{21}$ and $L_{31}L_{11}^T = A_{31}$, respectively. Note that a rectangular version of an unblocked Cholesky algorithm could factor A_{11} and recover L_{21} and L_{31} without directly solving triangular systems of equations. If matrix A is stored in column-major order, the data locality of these two approaches is equivalent and neither has a performance advantage. Then, rearranging the equation pertaining to the trailing sub-matrix yields

$$\begin{pmatrix} \widetilde{A}_{22} & \\ \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{22} & \\ L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{22}^T & L_{32}^T \\ & L_{33}^T \end{pmatrix} = \begin{pmatrix} A_{22} & \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix}.$$

Proceeding in the same manner with the trailing sub-matrix, we have

$$\begin{pmatrix} \widetilde{A}_{22} & \\ \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{22}L_{22}^T & \\ L_{32}L_{22}^T & L_{32}L_{32}^T + L_{33}L_{33}^T \end{pmatrix},$$

where

$$\widehat{A}_{33} = L_{33}L_{33}^T = \widetilde{A}_{33} - L_{32}L_{32}^T.$$

Again, $\widetilde{A}_{22}$ can be factored into $L_{22}L_{22}^T$, and L_{32} can be recovered by solving the triangular system of equations $L_{32}L_{22}^T = \widetilde{A}_{32}$. Finally, we factor $\widehat{A}_{33}$ into $L_{33}L_{33}^T$, and we have the lower triangular factor L of A. The blocked algorithm we have just described is a right-looking version, which computes a column block at each step and uses it to update the trailing sub-matrix [10].

Our simple blocking algorithm (`chol_block`), which factors a symmetric positive definite matrix stored in column-major order, is a right-looking one. Figure 4.3 outlines the algorithm in pseudocode. A crucial parameter for optimizing performance of a blocked algorithm is block dimension, variable r in the pseudocode listing, which specifies the step size for the outer loop. Note that lower triangular factor L overwrites the lower triangular part of A. For ease of exposition, the pseudocode for blocked algorithms in this paper assumes that the matrix dimension n is evenly divisible by the block dimension r.

```
factor A(1:r, 1:r) into lower triangular block L(1:r, 1:r)
for k = r+1 : n : r
        for i = k : n : r
                solve triangular system of equations:
                    L(i:i+r−1, k−r:k−1)L(k−r:k−1, k−r:k−1)^T = A(i:i+r−1, k−r:k−1)
        end
        for j = k : n : r
                for i = j : n : r
                        update trailing sub-matrix block:
                            A(i:i+r−1, j:j+r−1) = A(i:i+r−1, j:j+r−1) −
                                L(i:i+r−1, k−r:k−1)L(j:j+r−1, k−r:k−1)^T
                end
        end
        factor A(k:k+r−1, k:k+r−1) into lower triangular block L(k:k+r−1, k:k+r−1)
end
```

FIG. 4.3. *Cholesky factorization, simple blocking, right-looking algorithm.*

In the previous section we demonstrated some principles of performance optimization using matrix multiplication and found that recursive contiguous block storage produced a significant performance improvement over simple blocking. For standard Cholesky factorization we implemented contiguous blocking (`chol_contig_block`) and recursive contiguous blocking (`chol_recur_block`) algorithms. In the former, elements of matrix A are copied into an array where matrix blocks sized for L2 cache are stored contiguously, then a blocked algorithm performs Cholesky factorization on the contiguous blocks, and finally the contiguous blocks are copied back into matrix A in column-major order. In the latter, elements of matrix A are copied into an array where matrix blocks sized for L2 cache are stored contiguously, and within each block, sub-blocks sized for L1 cache are stored contiguously. Then a blocked algorithm performs Cholesky factorization on the recursive contiguous blocks, and finally the recursive contiguous blocks are copied back into matrix A in column-major order. Both `chol_contig_block` and `chol_recur_block` are left-looking algorithms, which compute one column block at a time using previously computed columns [10].

Left-looking `chol_contig_block` performs the same block matrix operations as described above for right-looking `chol_block`, only the block matrix operations are performed in a different sequence and matrix blocks are stored contiguously. Table 4.1 and Table 4.2 contrast the sequences of matrix block operations for right-looking and left-looking Cholesky factorization algorithms, respectively. Assume that the symmetric positive definite matrix A can be partitioned into 4-by-4 blocks, and its lower triangular factor L overwrites the lower triangular part of A as the factorization proceeds. $L_{jj} = \texttt{chol}(A_{jj})$ represents the Cholesky factorization of diagonal block A_{jj}; solving the triangular system of equations for lower triangular block L_{ij} is represented as $L_{ij}L_{jj}^T = A_{ij}$; and updating trailing sub-matrix block A_{ij} is represented as $A_{ij} = A_{ij} - L_{ik}L_{jk}^T$. The superscript preceding each equation enumerates the sequence of the block matrix operations.

Algorithm `chol_recur_block` adds an additional data layer and level of blocking. Sub-blocks are stored contiguously within contiguous blocks, and the left-looking sequence of block matrix operations enumerated in Table 4.2 is repeated for sub-blocks within each diagonal block when performing Cholesky factorization. A left-looking sequence of sub-matrix operations also exists for each of the other block matrix operations — triangular solve and trailing sub-matrix reduction.

Table 4.1

Sequence of block matrix operations for right-looking Cholesky factorization algorithm.

[1] $L_{11} = \mathtt{chol}(A_{11})$			
[2] $L_{21}L_{11}^T = A_{21}$	[5] $A_{22} = A_{22} - L_{21}L_{21}^T$ [11] $L_{22} = \mathtt{chol}(A_{22})$		
[3] $L_{31}L_{11}^T = A_{31}$	[6] $A_{32} = A_{32} - L_{31}L_{21}^T$ [12] $L_{32}L_{22}^T = A_{32}$	[8] $A_{33} = A_{33} - L_{31}L_{31}^T$ [14] $A_{33} = A_{33} - L_{32}L_{32}^T$ [17] $L_{33} = \mathtt{chol}(A_{33})$	
[4] $L_{41}L_{11}^T = A_{41}$	[7] $A_{42} = A_{42} - L_{41}L_{21}^T$ [13] $L_{42}L_{22}^T = A_{42}$	[9] $A_{43} = A_{43} - L_{41}L_{31}^T$ [15] $A_{43} = A_{43} - L_{42}L_{32}^T$ [18] $L_{43}L_{33}^T = A_{43}$	[10] $A_{44} = A_{44} - L_{41}L_{41}^T$ [16] $A_{44} = A_{44} - L_{42}L_{42}^T$ [19] $A_{44} = A_{44} - L_{43}L_{43}^T$ [20] $L_{44} = \mathtt{chol}(A_{44})$

Table 4.2

Sequence of block matrix operations for left-looking Cholesky factorization algorithm.

[1] $L_{11} = \mathtt{chol}(A_{11})$			
[2] $L_{21}L_{11}^T = A_{21}$	[5] $A_{22} = A_{22} - L_{21}L_{21}^T$ [8] $L_{22} = \mathtt{chol}(A_{22})$		
[3] $L_{31}L_{11}^T = A_{31}$	[6] $A_{32} = A_{32} - L_{31}L_{21}^T$ [9] $L_{32}L_{22}^T = A_{32}$	[11] $A_{33} = A_{33} - L_{31}L_{31}^T$ [13] $A_{33} = A_{33} - L_{32}L_{32}^T$ [15] $L_{33} = \mathtt{chol}(A_{33})$	
[4] $L_{41}L_{11}^T = A_{41}$	[7] $A_{42} = A_{42} - L_{41}L_{21}^T$ [10] $L_{42}L_{22}^T = A_{42}$	[12] $A_{43} = A_{43} - L_{41}L_{31}^T$ [14] $A_{43} = A_{43} - L_{42}L_{32}^T$ [16] $L_{43}L_{33}^T = A_{43}$	[17] $A_{44} = A_{44} - L_{41}L_{41}^T$ [18] $A_{44} = A_{44} - L_{42}L_{42}^T$ [19] $A_{44} = A_{44} - L_{43}L_{43}^T$ [20] $L_{44} = \mathtt{chol}(A_{44})$

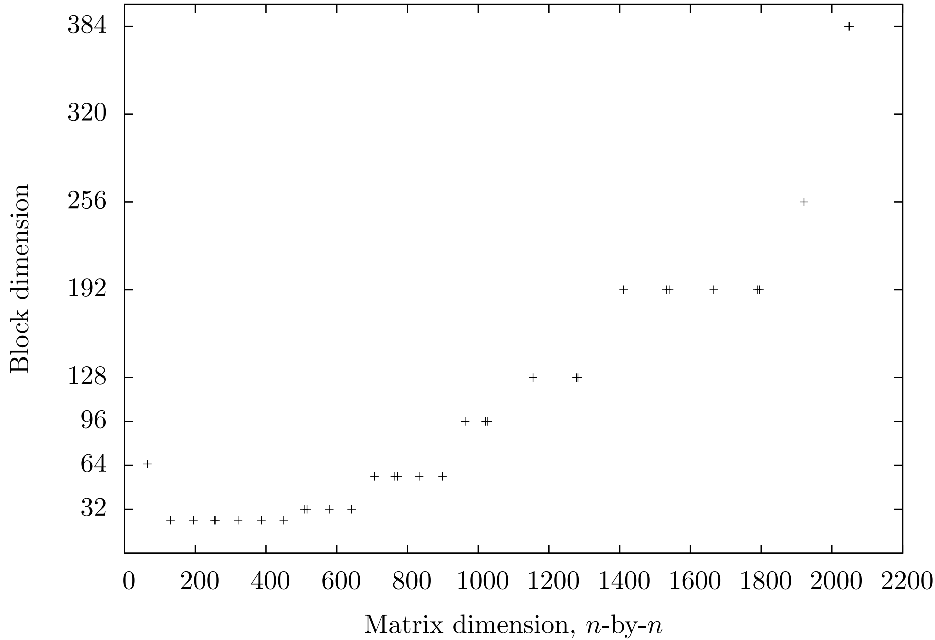

Fig. 4.4. *Blocking parameter for Cholesky factorization.*

Figure 4.4 plots the block size chosen by LAPACK for its Cholesky factorization routine `DPOTRF` [20] on the Intel machine over a range of matrix sizes. Our implementation of blocked algorithms for Cholesky factorization uses the same block size for a given matrix leading dimension as that chosen for `DPOTRF`. Figure 4.5 illustrates the performance gains attributable to the improved data locality of `chol_block` relative to `chol_saxpy` achieved through blocking, and `chol_saxpy` relative to `chol_outer_product` achieved through loop reordering. The performance gains are not as dramatic as we saw for matrix multiplication, but nonetheless significant — for large matrices `chol_saxpy` outperforms `chol_outer_product` by roughly 40%, and `chol_block` outperforms `chol_saxpy` by a further 40%. The promise of performance gains through recursive contiguous block storage, as evidenced for matrix multiplication, did not materialize with our implementation of Cholesky factorization. With a 32-by-32 sub-block size `chol_recur_block` is 20% slower than `chol_block`. Perhaps at optimization level `-O3` the speed-space tradeoff chosen by the compiler to optimize an additional data layer and level of blocking is detrimental to performance. By contrast, Elmroth et al. [11] report that on a typical RISC (Reduced Instruction Set Computer) processor, their packed storage recursive Cholesky algorithm using BLAS is about 5% faster than full storage LAPACK routine `DPOTRF` for large matrices. With respect to our implementation of contiguous block storage, algorithm `chol_contig_block` is marginally faster than `chol_block` for large matrices (Figure 4.6).

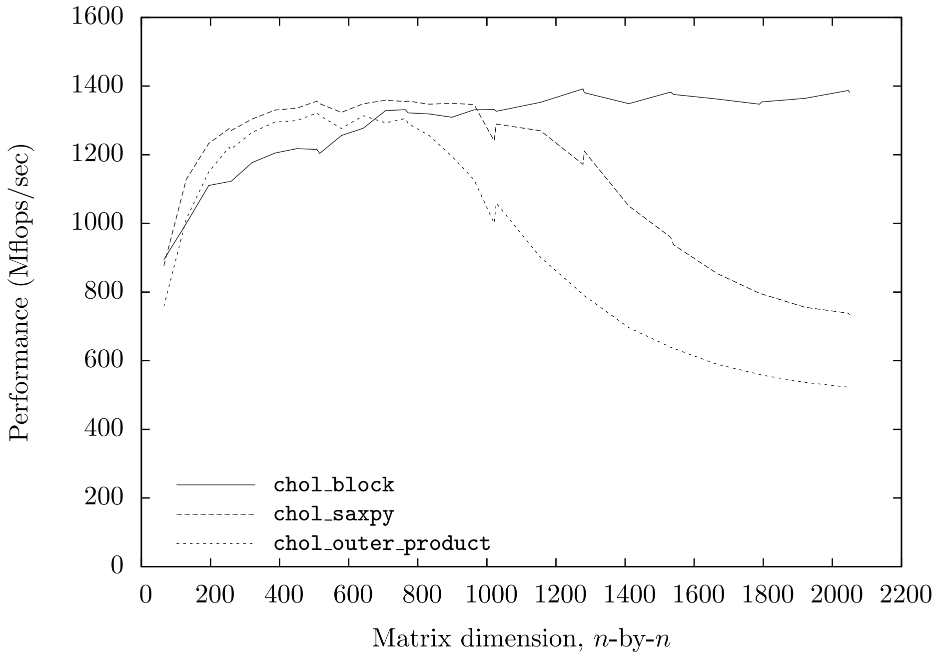

Fig. 4.5. *Data locality in Cholesky factorization.*

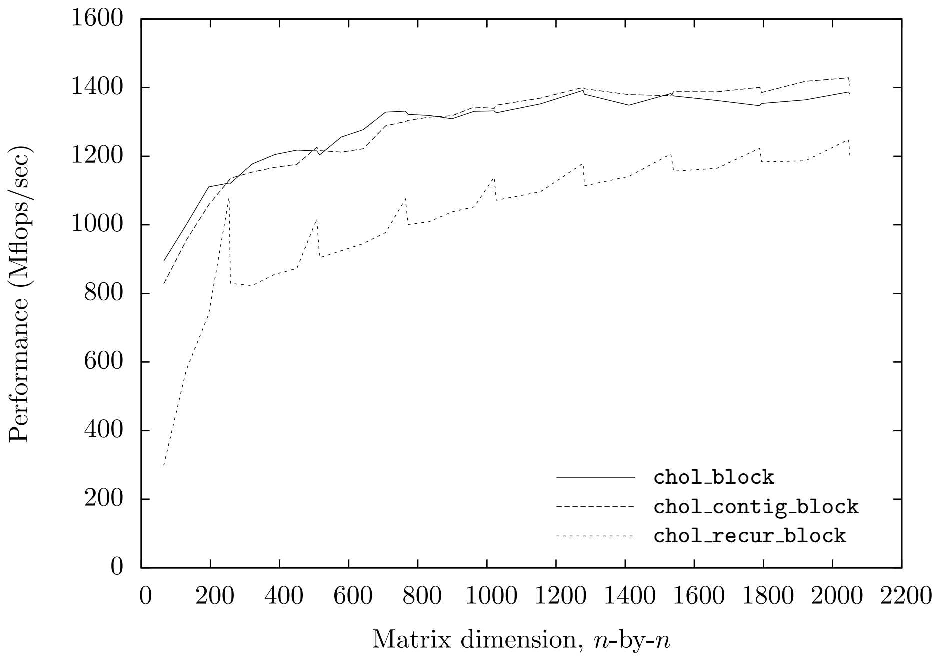

Fig. 4.6. *Blocked algorithms for Cholesky factorization.*

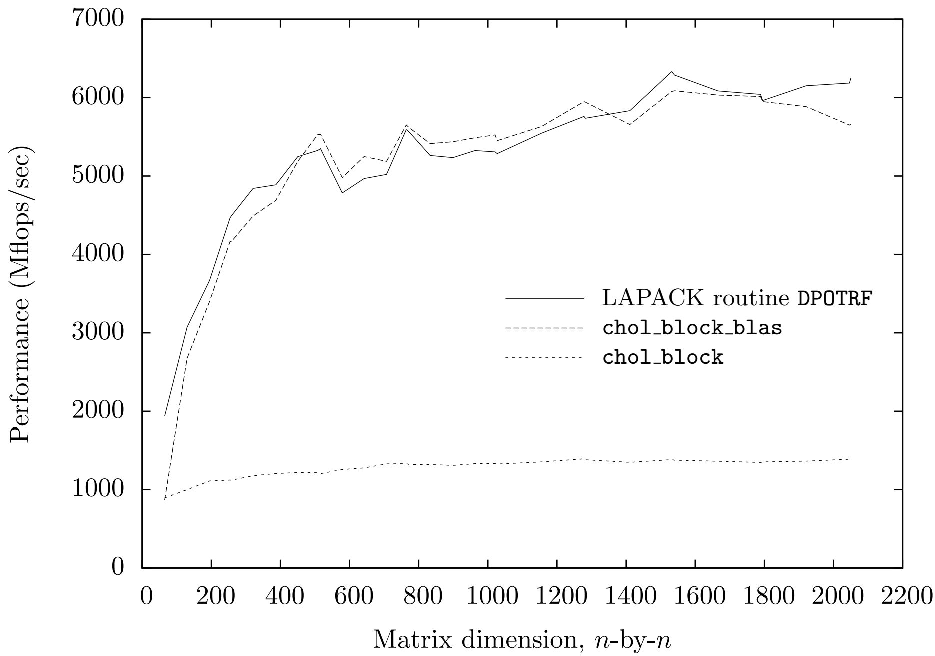

Fig. 4.7. *Cholesky factorization using BLAS and LAPACK libraries.*

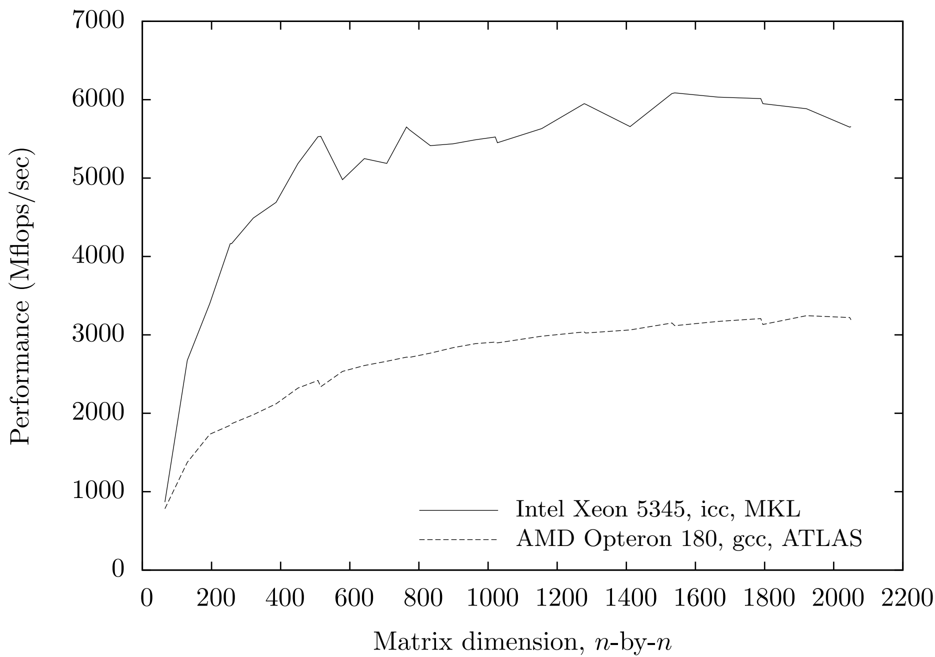

Fig. 4.8. *Performance variability across hardware, compilers and libraries.*

Our implementation of blocked algorithms achieves less than one-quarter of the flop rate attained by LAPACK routine `DPOTRF`, which is a right-looking blocked algorithm. Profile data for `chol_block` factoring 2000-by-2000 symmetric positive definite matrices reveal that the algorithm spends approximately 98% of its time performing block triangular solves (18%) and trailing sub-matrix reductions (80%). If we were to use `chol_saxpy` to perform Cholesky factorization on diagonal blocks, but call BLAS routines `DTRSM` and `DSYRK` [4], respectively, to perform block triangular solves and trailing sub-matrix reductions, we would expect a substantial performance boost. We identify this blocked algorithm as `chol_block_blas`. Indeed, Figure 4.7 shows that `chol_block_blas` performs in line with LAPACK routine `DPOTRF`. The time to factor diagonal blocks is the same for `chol_block` and `chol_block_blas` — both algorithms invoke `chol_saxpy` — but level 3 BLAS routines `DTRSM` and `DSYRK` perform block triangular solves and trailing sub-matrix reductions much more efficiently. As a consequence the proportion of time spent by `chol_block_blas` on factoring diagonal blocks rises to 8% and the overall time to perform Cholesky factorization is cut by more than 75%.

Finally, in Figure 4.8 we use `chol_block_blas` to demonstrate the variability in performance across hardware, compilers and libraries.

5. Symmetric Indefinite Factorization. To solve an n-by-n symmetric, possibly indefinite, system of equations $Ax = b$ in a computationally efficient and numerically stable manner, we make use of the factorization $PAP^T = LDL^T$, where $A \in \mathbb{R}^{n\times n}$ is symmetric, $L \in \mathbb{R}^{n\times n}$ is unit lower triangular, D is block diagonal with block order 1 or 2, and $P \in \mathbb{R}^{n\times n}$ is a permutation matrix for pivoting. Given $Ax = P^T(LDL^T)Px = b$, x is recovered by solving the triangular systems of equations $Lw = Pb$, $Dz = w$, $L^Ty = z$ and $Px = y$. Although the cost of symmetric indefinite factorization depends on the pivoting strategy chosen, a lower bound is provided by Cholesky factorization, which takes $\frac{1}{3}n^3 + O(n^2)$ flops. Solving the triangular systems of equations involves $O(n^2)$ work [16].

$\alpha = (1+\sqrt{17})/8$
$\lambda = |a_{r1}| = \max\{|a_{21}|, \ldots, |a_{m1}|\}$
`if` $\lambda > 0$
  `if` $|a_{11}| \geq \alpha\lambda$
    use a_{11} as 1-by-1 pivot
  `else`
    $\sigma = |a_{pr}| = \max\{|a_{1r}|, \ldots, |a_{r-1,r}|, |a_{r+1,r}|, \ldots, |a_{mr}|\}$
    `if` $|a_{11}|\sigma \geq \alpha\lambda^2$
      use a_{11} as 1-by-1 pivot
    `else if` $|a_{rr}| \geq \alpha\sigma$
      use a_{rr} as 1-by-1 pivot
    `else`
      use $\begin{bmatrix} a_{11} & a_{r1} \\ a_{r1} & a_{rr} \end{bmatrix}$ as 2-by-2 pivot
    `end`
  `end`
`end`

FIG. 5.1. *Bunch-Kaufman (partial) pivoting algorithm.*

Numerical stability of a matrix factorization is assured when entries in the factors are nicely bounded. For symmetric indefinite factorization, numerical stability comes at the expense of symmetric pivoting. We outline three pivoting strategies for symmetric indefinite factorization: Bunch-Kaufman [6] (partial pivoting), bounded Bunch-Kaufman [2] (rook pivoting), and Bunch-Parlett [7] (complete pivoting). Partial pivoting involves $O(n^2)$ comparisons, complete pivoting $O(n^3)$ comparisons, and rook pivoting between $O(n^2)$ and $O(n^3)$ comparisons. Clearly, Bunch-Kaufman pivoting has a performance advantage over bounded Bunch-Kaufman and Bunch-Parlett, but unfortunately it has the worst accuracy of the three.

Suppose that A is an n-by-n symmetric matrix, and at the kth step of the factorization we have the reduced symmetric trailing sub-matrix, or Schur complement,

$$A_k = \begin{pmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{pmatrix} \text{ where } A_k \in \mathbb{R}^{(n-k+1)\times(n-k+1)}.$$

Each of the pivoting strategies compares entries of the Schur complement A_k in making its pivot selection at the kth step.

The algorithm for Bunch-Kaufman pivoting is outlined in Figure 5.1. The parameter α bounds the element growth of the sequence of trailing sub-matrices. With the value of α set to $(1+\sqrt{17})/8$, the bound on trailing sub-matrix growth for a 2-by-2 pivot equals that for two consecutive 1-by-1 pivots, and thereby minimizes the worst case growth for any arbitrary sequence of pivot selections. Ashcraft, Grimes and Lewis [2] show that in cases where a_{11} is a 1-by-1 pivot

```
α = (1 + √17)/8
λ = |a_r1| = max{|a_21|, ..., |a_m1|}
if λ > 0
    if |a_11| ≥ αλ
        use a_11 as 1-by-1 pivot
    else
        i = 1
        do
            σ = |a_pr| = max{|a_1r|, ..., |a_r-1,r|, |a_r+1,r|, ..., |a_mr|}
            if |a_rr| ≥ ασ
                use a_rr as 1-by-1 pivot
            else if λ = σ
                use [ a_ii  a_ri ; a_ri  a_rr ] as 2-by-2 pivot
            else
                i = r
                λ = σ
                r = p
            end
        until pivot selected
    end
end
```

Fig. 5.2. *Bounded Bunch-Kaufman (rook) pivoting algorithm.*

```
α = (1 + √17)/8
ξ = max_{i≠j}{|a_ij|}
η = |a_ss| = max_k{|a_kk|}
if ξ > 0 or η > 0
        if |a_11| ≥ αλ
                if η ≥ αξ
                        use a_ss as 1-by-1 pivot
                else
                        use [ a_ii  a_ji ; a_ji  a_jj ] as 2-by-2 pivot
                end
        end
end
```

FIG. 5.3. *Bunch-Parlett (complete) pivoting algorithm.*

with $|a_{11}|\sigma \geq \alpha\lambda^2$ or $\begin{pmatrix} a_{11} & a_{r1} \\ a_{r1} & a_{rr} \end{pmatrix}$ is a 2-by-2 pivot, there is no upper bound on the magnitude of entries in the lower triangular factor L. In both these aberrant cases it is necessary to control the ratio σ/λ, as defined in Figure 5.1, in order to bound the entries of L. Ashcraft, Grimes and Lewis denote their variant of the Bunch-Kaufman algorithm as bounded Bunch-Kaufman (Figure 5.2). The idea is to only permit the two aberrant cases when the ratio $\sigma/\lambda = 1$, otherwise replace a_{11} with a_{rr} and proceed with comparisons until a pivot is selected. The condition that $\sigma/\lambda = 1$ eliminates the first aberrant case, since $|a_{11}|\sigma \geq \alpha\lambda^2$ reduces to the case where $|a_{11}| \geq \alpha\lambda$, and in the second aberrant case the entries of L are nicely bounded. The bounded Bunch-Kaufman algorithm provides the stability of complete pivoting at a cost that is potentially little more than that of partial pivoting, and no higher than the cost of complete pivoting. Figure 5.3 outlines the Bunch-Parlett algorithm for complete pivoting in pseudocode.

As in the previous section we identify LDL^T factorization algorithms that we implemented by their function names from the source code listings in the appendix. To identify the pivoting strategy employed by these algorithms we use the acronyms BK, BBK and BP for Bunch-Kaufman, bounded Bunch-Kaufman and Bunch-Parlett, respectively. So, for example, `ldlt_saxpy(BK)` signifies our implementation of the SAXPY operation with Bunch-Kaufman pivoting.

We implemented two unblocked algorithms for symmetric indefinite factorization: an outer product version (`ldlt_outer_product`) which employs kji indexing; and a version of the SAXPY operation (`ldlt_saxpy`) which uses jki indexing. With each pass through the k loop of algorithm `ldlt_outer_product`, pivot selection is performed on an updated trailing sub-matrix. Suppose that a pivot has been selected and symmetric pivoting performed at the kth step, and denote the reduced symmetric trailing sub-matrix by

$$P_k A_k P_k^T = \widetilde{A}_k = \begin{pmatrix} D_k & C_k^T \\ C_k & \bar{A}_k \end{pmatrix} = \begin{pmatrix} I & \\ C_k D_k^{-1} & I \end{pmatrix} \begin{pmatrix} D_k & \\ & \bar{A}_k - C_k D_k^{-1} C_k^T \end{pmatrix} \begin{pmatrix} I & \\ C_k D_k^{-1} & I \end{pmatrix}^T,$$

where either D_k is a diagonal entry, $C_k D_k^{-1} \in \mathbb{R}^{(n-k)\times 1}$ is a column of the unit lower triangular matrix, and $A_{k+1} = \bar{A}_k - C_k D_k^{-1} C_k^T \in \mathbb{R}^{(n-k)\times(n-k)}$ is the updated trailing sub-matrix (before pivot selection) at step k+1; or D_k is a 2-by-2 symmetric diagonal block, $C_k D_k^{-1} \in \mathbb{R}^{(n-k-1)\times 2}$ is a column block of the unit lower triangular matrix, and $A_{k+2} = \bar{A}_k - C_k D_k^{-1} C^T \in \mathbb{R}^{(n-k-1)\times(n-k-1)}$

is the updated trailing sub-matrix at step $k+2$. Our implementation of the outer product algorithm is presented in Figure 5.4. It calls one of the pivoting algorithms outlined above based on the pivot strategy passed in the argument list. The permutation matrix is encoded in the pivot vector $piv[\,]$, and symmetric pivoting only interchanges row and column entries on and below the diagonal.

```
k = 1
while k < n
    perform pivot selection on A(k:n, k:n)
    if k ≠ piv[k]
        interchange row and column k with piv[k]
    end
    if 1-by-1 pivot
        for i = k + 1 : n
            A(i, k) = A(i, k)/A(k, k)
        end
        for j = k + 1 : n
            for i = j : n
                A(i, j) = A(i, j) − A(i, k) ∗ A(j, k) ∗ A(k, k)
            end
        end
        k = k + 1
    else if 2-by-2 pivot
        if k + 1 ≠ piv[k + 1]
            interchange row and column (k+1) with piv[k+1]
        end
        compute column block of unit lower triangular matrix,
          A(k+2:n, k:k+1) = A(k+2:n, k:k+1)A(k:k+1, k:k+1)^{-1}
        update trailing sub-matrix,
          A(k+2:n, k+2:n) = A(k+2:n, k+2:n) −
            A(k+2:n, k:k+1)A(k:k+1, k:k+1)^{-1} A(k+2:n, k:k+1)^T
        k = k + 2
    end
end
```

Fig. 5.4. *Outer product method for symmetric indefinite factorization.*

Now, with each pass through the j loop of algorithm `ldlt_saxpy`, trailing sub-matrix updates $k = 1, \ldots, j{-}1$ are applied after pivot selection to column $\widetilde{A}(j:n, j)$. That is, when pivot selection is performed at the beginning of the jth pass through the outer loop, no trailing sub-matrix updates have been applied to $\widetilde{A}(j:n, j:n)$. Hence, trailing sub-matrix updates must be applied during pivot selection to candidate columns before comparisons are made. Both unblocked algorithms overwrite lower triangular entries of symmetric matrix A with unit lower triangular factor L and block diagonal factor D.

We repeat the examination of Cholesky block matrix operations for symmetric indefinite factorization to illustrate the matrix operations performed by blocked algorithms. Suppose we have factored symmetric matrix A; then the factorization may be written in block form:

$$P \begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix} P^T = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix}.$$

Making use of symmetry we equate the lower triangular blocks of A with the product of block matrices L, D and L^T:

$$P \begin{pmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{pmatrix} P^T = \begin{pmatrix} \widetilde{A}_{11} & & \\ \widetilde{A}_{21} & \widetilde{A}_{22} & \\ \widetilde{A}_{31} & \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{11}D_{11}L_{11}^T & & \\ L_{21}D_{11}L_{11}^T & \widetilde{A}_{22} & \\ L_{31}D_{11}L_{11}^T & \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix},$$

where

$$\begin{pmatrix} \widetilde{A}_{22} & \\ \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} D_{11} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix} + \begin{pmatrix} L_{22} & \\ L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} D_{22} & \\ & D_{33} \end{pmatrix} \begin{pmatrix} L_{22}^T & L_{32}^T \\ & L_{33}^T \end{pmatrix}.$$

A rectangular (unblocked) symmetric indefinite factorization algorithm factors $\widetilde{A}_{11}$ into $L_{11}D_{11}L_{11}^T$, and solves for L_{21} and L_{31}. Then rearranging the equation pertaining to the trailing sub-matrix yields

$$\begin{pmatrix} \widehat{A}_{22} & \\ \widehat{A}_{32} & \widehat{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{22} & \\ L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} D_{22} & \\ & D_{33} \end{pmatrix} \begin{pmatrix} L_{22}^T & L_{32}^T \\ & L_{33}^T \end{pmatrix}$$

$$= \begin{pmatrix} \widetilde{A}_{22} & \\ \widetilde{A}_{32} & \widetilde{A}_{33} \end{pmatrix} - \begin{pmatrix} L_{21}D_{11} \\ L_{31}D_{11} \end{pmatrix} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix}.$$

Proceeding in the same manner with the trailing sub-matrix, we have

$$\begin{pmatrix} \widehat{A}_{22} & \\ \widehat{A}_{32} & \widehat{A}_{33} \end{pmatrix} = \begin{pmatrix} L_{22}D_{22}L_{22}^T & \\ L_{32}D_{22}L_{22}^T & L_{32}D_{22}L_{32}^T + L_{33}D_{33}L_{33}^T \end{pmatrix},$$

where

$$\bar{A}_{33} = L_{33}D_{33}L_{33}^T = \widehat{A}_{33} - L_{32}D_{32}L_{32}^T.$$

Again, a rectangular (unblocked) symmetric indefinite factorization algorithm factors $\widehat{A}_{22}$ into $L_{22}D_{22}L_{22}^T$, and recovers L_{32}. Finally, after computing $\bar{A}_{33}$, it is factored into $L_{33}D_{33}L_{33}^T$, and the symmetric indefinite factorization of matrix A is complete.

We implemented the simple blocking algorithm (`ldlt_block`) outlined in Figure 5.5, a right-looking algorithm that operates on matrices stored in column-major order. Variable r in the pseudocode listing represents the blocking parameter. The following procedure is repeated for each step r of the outer loop of `ldlt_block`. Suppose that the factorization has progressed to the kth column ($k = ar+1, a \in \mathbb{N}$), and denote the Schur complement by $P_k A_k P_k^T = \widetilde{A}_k = \begin{pmatrix} \widetilde{A}_{ii} & \widetilde{A}_{ji}^T \\ \widetilde{A}_{ji} & \widetilde{A}_{jj} \end{pmatrix}$. Algorithm `ldlt_block` invokes unblocked algorithm `ldlt_saxpy` to factor r-by-r matrix block $\widetilde{A}_{ii}$ into $L_{ii}D_{ii}$ and solve for $(n-k-r+1)$-by-r column block L_{ji}. Then, `ldlt_block` proceeds to update the trailing sub-matrix by computing $\widehat{A}_{jj} = \widetilde{A}_{jj} - L_{ji}D_{ii}L_{ji}^T$. Before incrementing the outer loop by r, pivot selections made during the factorization of $\widetilde{A}_{ii}$ are applied to columns 1 through $k-1$ of $\widetilde{A} = P_k A P_k^T$. Entries of unit lower triangular matrix L and block diagonal matrix D

overwrite the lower triangular entries of symmetric matrix A. Given the disappointing performance of recursive contiguous blocking for Cholesky factorization, we did not pursue recursive contiguous block storage, nor contiguous block storage, for symmetric indefinite factorization. In any case, the pivoting requirement apparently precludes the development of effective left-looking blocked algorithms [10].

```
factor A(1:r,1:r) into unit lower triangular block L(1:r,1:r) and block diagonal D(1:r,1:r)
solve for unit lower triangular column block L(r+1:n,1:r)
for k = r+1 : n : r
        for j = k : n : r
                for i = j : n : r
                        update trailing sub-matrix block,
                            A(i:i+r-1, j:j+r-1) = A(i:i+r-1, j:j+r-1) -
                                L(i:i+r-1, k-r:k-1)D(k-r:k-1, k-r:k-1)L(j:j+r-1, k-r:k-1)^T
                end
        end
        factor A(k:k+r-1, k:k+r-1) into unit lower triangular block L(k:k+r-1, k:k+r-1)
            and block diagonal D(k:k+r-1, k:k+r-1)
        solve for unit lower triangular column block L(k+r:n, k:k+r-1)
        for i = k : k+r-1
                if i ≠ piv[i]
                        interchange row vector L(i,1:k-1) with L(piv[i],1:k-1)
                end
        end
end
```

FIG. 5.5. *Symmetric indefinite factorization, simple blocking, right-looking algorithm.*

While the optimal block size chosen by LAPACK for Cholesky factorization on the Intel machine varies with matrix leading dimension, LAPACK chooses a constant block size of 64-by-64 for symmetric indefinite factorization routine `DSYTRF` [20]. However, we find that algorithm `ldlt_block` is nearly 20% faster when using a block size of 128-by-128 rather than the optimal block size chosen for `DSYTRF`. Therefore, performance data for `ldlt_block` presented in this section were generated with block size set to 128-by-128.

The performance improvement achieved through optimization of memory access on our implementation of algorithms for symmetric indefinite factorization with Bunch-Kaufman pivoting is illustrated in Figure 5.6. (Note that we use the lower bound on floating point operations given by Cholesky factorization in our calculation of Mflops/sec.) Algorithm `ldlt_block` achieves only 30% of the flop rate attained by LAPACK routine `DSYTRF`, which is not inconsistent with the performance observed for blocked algorithms implementing Cholesky factorization. One difference between the designs of `DSYTRF` and `ldlt_block` is that during the factorization of a column block of matrix A, `DSYTRF` stores the product of the lower triangular column block of L and the diagonal block of D in working storage. In terms of matrix blocks representing the Schur complement outlined above, `DSYTRF` stores $W = \begin{pmatrix} W_{ii} \\ W_{ji} \end{pmatrix} = \begin{pmatrix} L_{ii} \\ L_{ji} \end{pmatrix} \begin{pmatrix} D_{ii} \end{pmatrix}$ and updates the trailing sub-matrix by computing $\widehat{A}_{jj} = \widetilde{A}_{jj} - W_{ji} L_{ji}^T$. Working array W is also used to compute trailing sub-matrix updates on columns of the Schur complement during pivot selection. This design is more efficient in terms of floating point operations, since columns of W are available after pivot selection but before solving

for columns of the unit lower triangular factor, and hence the product of the lower triangular column block of L and the diagonal block of D need not be recomputed during the trailing sub-matrix reduction. The use of working array W most likely explains LAPACK's choice of constant block size 64-by-64 for `DSYTRF`. When we incorporated this design feature in `ldlt_block` and used the block size chosen by LAPACK for `DSYTRF`, we found that it ran 20% slower than our implementation of `ldlt_block` described above. So, we rolled back this potential performance enhancement to `ldlt_block`. For our implementation of simple blocking the cost of additional memory allocation for the working array and its effect on data locality seems to overwhelm the saving in floating point operations.

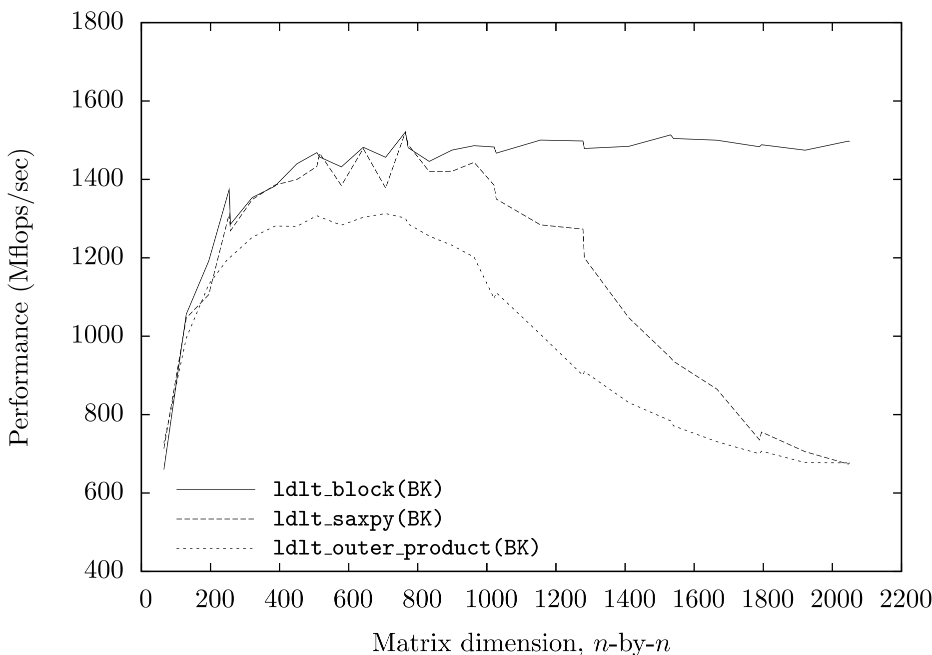

Fig. 5.6. *Data locality in LDL^T factorization, Bunch-Kaufman pivoting.*

Another difference between `DSYTRF` and `ldlt_block` is the use of BLAS by `DSYTRF`. `DSYTRF` calls level 3 BLAS routine `DGEMM` (matrix-matrix multiplication) and level 2 BLAS routine `DGEMV` (matrix-vector multiplication) to perform trailing sub-matrix updates. In addition, it calls a number of level 1 BLAS routines (`DCOPY`, `DSWAP`, `DSCAL` and `IDAMAX`) during pivot selection, symmetric pivoting and solving for columns of the unit lower triangular factor [4]. Clearly `DSYTRF` is highly efficient, due largely to its use of BLAS, but it is not as modular as `DPOTRF`, LAPACK's Cholesky factorization routine. Unlike `DPOTRF`, `DSYTRF` does not invoke an unblocked version of symmetric indefinite factorization to factor diagonal blocks, and level 3 BLAS routines to solve for lower triangular column blocks and perform trailing sub-matrix updates. LAPACK does provide an unblocked version of symmetric indefinite factorization, routine `DSYTF2`, which uses the outer product method. However, with its use of working array W, `DSYTRF` only invokes `DSYTF2` to factor the last diagonal block. Hence, our task of calling BLAS routines to perform most of the work in our blocked algorithm for symmetric indefinite factorization is not as straightforward as it was for Cholesky

factorization. When we selectively replaced code loops of `ldlt_block` with BLAS routines `DGEMM` and `DGEMV` and set block size to 128-by-128, this version of simple blocking reached 75% of the flop rate attained by `DSYTRF`. Alternatively, if we incorporate working array W to store the product of the lower triangular column block of L and the diagonal block of D, and replace code loops with `DGEMM` and `DGEMV`, this blocked algorithm (`ldlt_block_blas`) achieves 80% of the flop rate attained by `DSYTRF` (Figure 5.7). We set block size to 64-by-64 for `ldlt_block_blas`, consistent with the block size chosen for `DSYTRF`, but found no deterioration in performance using a block size of 128-by-128. Finally, we remark that invoking the aforementioned level 1 BLAS routines in addition to `DGEMM` and `DGEMV` makes no real discernible difference to the performance of `ldlt_block_blas`.

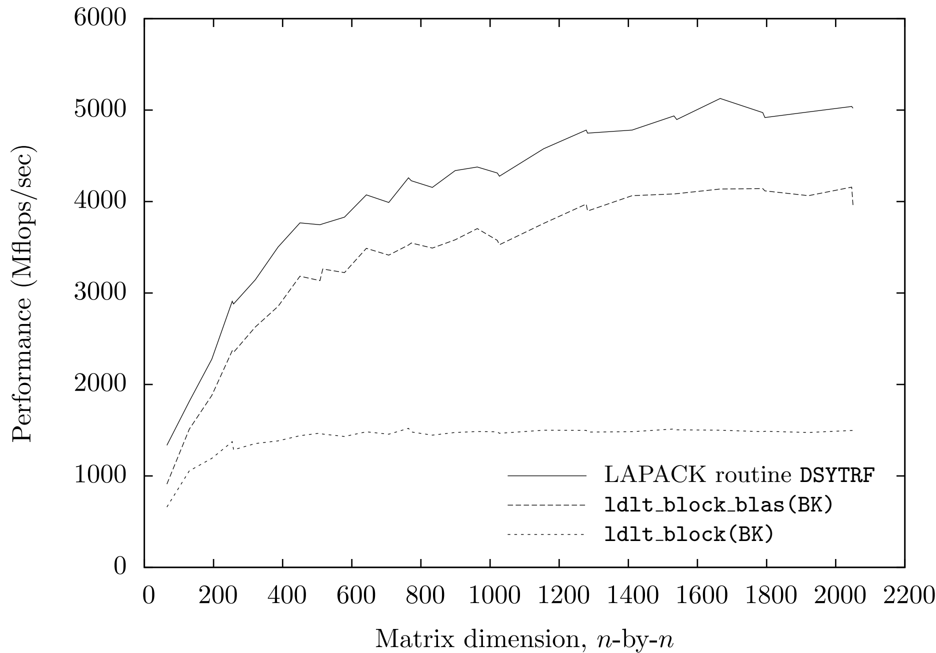

Fig. 5.7. *Blocked algorithms for LDL^T factorization, Bunch-Kaufman pivoting.*

Profile data for `ldlt_block(BK)` and `ldlt_block_blas(BK)` affirms that the superior performance of the latter is attributable to the use of BLAS. Profiling indicates that while factoring a 2000-by-2000 randomly generated symmetric matrix A, `ldlt_block(BK)` spends 89% of its time performing trailing sub-matrix updates, 6% pivoting, and 5% factoring column blocks of A into diagonal blocks of D and unit lower triangular blocks of L. By comparison, `ldlt_block_blas(BK)` spends 79% of its time performing trailing sub-matrix updates, 13% pivoting, and 8% factoring column blocks of A. Algorithm `ldlt_block_blas(BK)` cuts the overall time for LDL^T factorization by 70% relative to `ldlt_block(BK)`. Profile data looks much the same with bounded Bunch-Kaufman pivoting.

Figure 5.8 plots the number of symmetric pivots (row and column interchanges) for a sample of randomly generated 2000-by-2000 symmetric matrices, starting with a positive definite one and progressively increasing the degree of indefiniteness of each matrix, as proxied by the number of Bunch-Kaufman pivots. Note that the sample of randomly generated symmetric matrices

was selected such that the number of Bunch-Kaufman pivots occurs at regular intervals. For the same matrix, the number of row and column interchanges introduced by bounded Bunch-Kaufman pivoting averages one and one-half times that made by Bunch-Kaufman pivoting — the cost of ensuring numerical stability. Complete pivoting performs comparisons across all entries in the trailing sub-matrix, so the number of row and column interchanges for Bunch-Parlett pivoting hovers near the matrix dimension irrespective of the degree of indefiniteness. Also, we remark that the charts above, which plot performance for a range of matrix dimensions, do so for randomly generated symmetric matrices whose degree of indefiniteness, or ratio of number of symmetric pivots to matrix dimension, is near the top end of Figure 5.8.

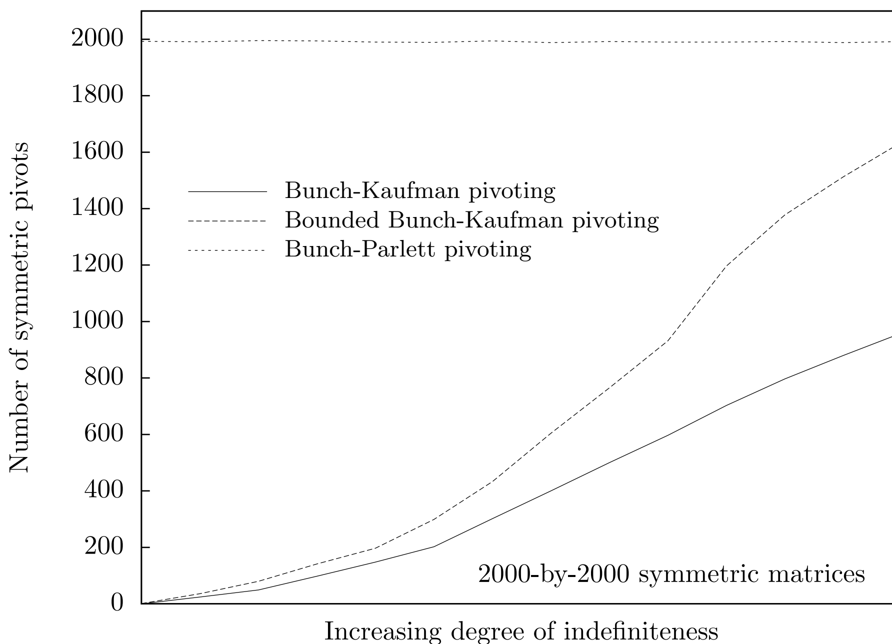

FIG. 5.8. *Number of symmetric pivots (row and column interchanges).*

Intuitively, we expect that LDL^T factorization of a symmetric positive definite matrix would be faster than LDL^T factorization of a symmetric indefinite matrix of the same dimension, with the difference in speed a function of the amount of symmetric pivoting. Curiously enough, on our simple blocking algorithm `ldlt_block` we observe the inverse relationship — the time to factor symmetric positive definite matrices is longer. It turns out that the inner-most loop executed by the function that updates the trailing sub-matrix depends on whether the diagonal block of D referenced in the computation is 1-by-1 or 2-by-2. For each 1-by-1 diagonal block, a multiple of a column of L is subtracted from a column of the trailing sub-matrix, where the multiple is a product of two local variables corresponding to the diagonal element and an element of L^T. For each 2-by-2 diagonal block, multiples of two adjacent columns of L are subtracted from a column of the trailing sub-matrix, where the multiples are some linear combination of five local variables corresponding to the three elements of the (symmetric) diagonal block and two adjacent elements of a column of L^T. With each pass through the 2-by-2 diagonal block more floating point operations are executed

than for every two passes through 1-by-1 diagonal blocks, and in the context of LDL^T factorization of a 2000-by-2000 randomly generated symmetric indefinite matrix (D is block diagonal with block order 1 or 2), the total number of floating point operations is a fraction of one percent higher than that for LDL^T factorization of a symmetric positive definite matrix (D is diagonal) of the same dimension. However, with more local variables in the inner-most loop that processes 2-by-2 diagonal blocks, the compiler is able to better exploit on-chip parallelism through optimization of floating point operations, and as a consequence LDL^T factorization of a symmetric indefinite matrix is faster than that of a symmetric positive definite one of the same dimension.

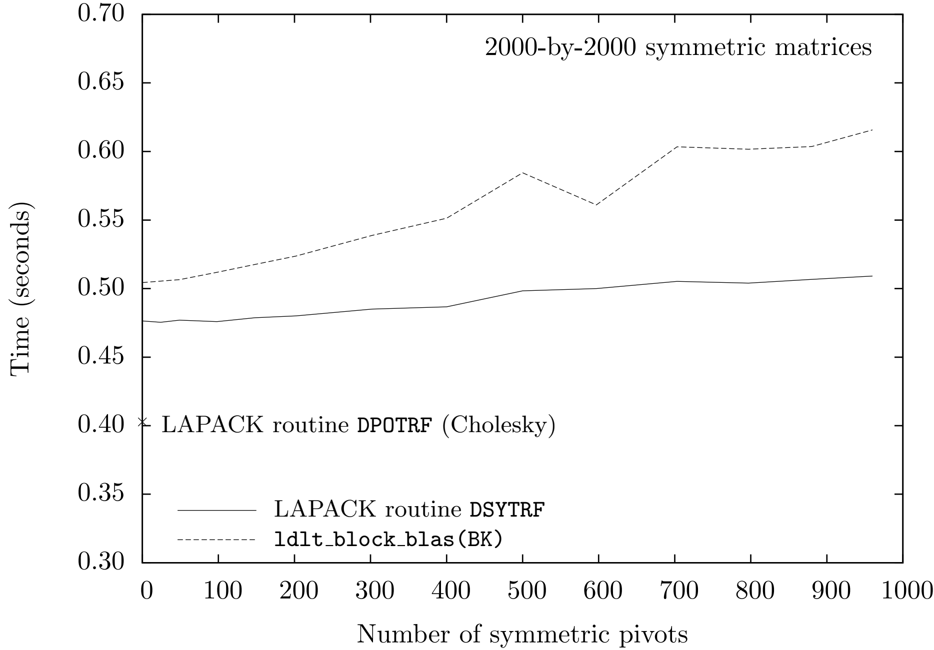

Fig. 5.9. *Cost of Bunch-Kaufman pivoting with increasing indefiniteness, LDL^T factorization.*

In Figure 5.9 the relationship between the time to perform LDL^T factorization of 2000-by-2000 symmetric matrices and the number of symmetric pivots, as a proxy for degree of indefiniteness, exhibits the anticipated positive gradient for algorithm `ldlt_block_blas` and LAPACK routine `DSYTRF`. It also plots a point on the vertical axis representing the time to perform Cholesky factorization on the symmetric positive definite matrix using LAPACK routine `DPOTRF`. For a 2000-by-2000 symmetric positive definite matrix there is a 18% premium on the cost of computing the LDL^T factorization using `DSYTRF` over the cost of computing its Cholesky factorization. The performance advantage of `DPOTRF` can probably be attributed to its use of level 3 BLAS routines `DTRSM` and `DSYRK` to solve for a column block of the lower triangular factor and update the trailing sub-matrix, respectively. As the degree of indefiniteness of the sample symmetric matrices increases, the time to perform LDL^T factorization using `DSYTRF` rises modestly, adding a further 7% to the cost of factoring the symmetric positive definite matrix.

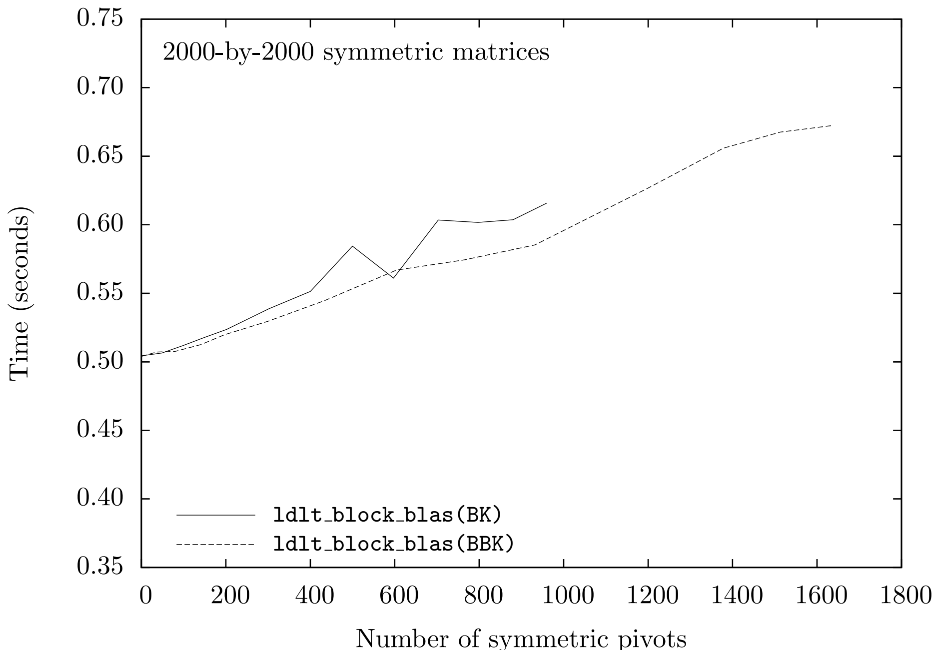

FIG. 5.10. *Cost of symmetric pivoting, blocked algorithm using BLAS.*

Figure 5.10 plots the time to perform LDL^T factorization using `ldlt_block_blas(BK)` and `ldlt_block_blas(BBK)` against the number of row and column interchanges for Bunch-Kaufman and bounded Bunch-Kaufman pivoting, respectively. For a given symmetric matrix, bounded Bunch-Kaufman pivoting introduces at least as many row and column interchanges as Bunch-Kaufman pivoting, and for the sample 2000-by-2000 symmetric matrices the ratio averages 3 : 2 (Figure 5.8). Since complete pivoting performs comparisons across all entries in the trailing sub-matrix, Bunch-Parlett pivoting is only an option for the LDL^T outer product method. Furthermore, when updating the trailing sub-matrix even blocked algorithms are limited to an outer product operation on column vectors (1-by-1 pivots) or multiplying column blocks of order 2 (2-by-2 pivots). As such, LDL^T factorization with Bunch-Parlett pivoting is an order of magnitude slower — taking approximately 9 seconds on 2000-by-2000 symmetric matrices — than LDL^T factorization with either Bunch-Kaufman or bounded Bunch-Kaufman pivoting. In Figure 5.11 we compare the performance of algorithm `ldlt_block_blas` employing partial (Bunch-Kaufman), rook (bounded Bunch-Kaufman) and complete (Bunch-Parlett) pivoting. For large randomly generated symmetric matrices, employing bounded Bunch-Kaufman pivoting in lieu of Bunch-Kaufman pivoting adds approximately 11% to the cost of symmetric indefinite factorization.

Both Figure 5.10 and Figure 5.11 provide measures of performance of LDL^T factorization comparing Bunch-Kaufman pivoting with bounded Bunch-Kaufman pivoting, while Figure 5.8 compares the number of row and column interchanges introduced by the respective pivoting strategies for sample symmetric matrices of varying degrees of indefiniteness. It would be interesting to have some measure of efficiency for these pivoting algorithms, which would require an estimate of extra work done or duplicate computations performed by them. Pivot comparisons are performed on updated

columns of the trailing sub-matrix, and the SAXPY operation underlying the blocked algorithms only applies trailing sub-matrix updates (associated with columns 1 through $j-1$) to column j when the outer-loop processes column j. That is, in a manner analogous to the code outlined in Figure 4.2, it computes $A(j:n,j) = A(j:n,j) - L(j:n,1:j-1)D(1:j-1,1:j-1)L(j,1:j-1)^T$. If, for example, the current column is j, then `ldlt_saxpy(BK)` applies trailing sub-matrix updates associated with columns 1 through $j-1$ to column j. Furthermore, suppose that column j satisfies the condition requiring pivot comparisons with elements of column r. Then trailing sub-matrix updates associated with columns 1 through $j-1$ are also applied to column r. Now, if a 1-by-1 pivot is selected, only trailing sub-matrix updates applied to the column corresponding to the selected pivot are retained; trailing sub-matrix updates applied to the column corresponding to the discarded pivot are recomputed when pivot comparisons are next made on that column. In this case one column of trailing sub-matrix updates is considered a duplicate computation, or extra work performed by the algorithm. We find that for the sample 2000-by-2000 symmetric matrices the duplicate computation of trailing sub-matrix updates on columns as a proportion of the number of row and column interchanges averages 0.9 for Bunch-Kaufman pivoting and 1.1 for bounded Bunch-Kaufman pivoting. Coupled with the observation that the ratio of bounded Bunch-Kaufman pivots to Bunch-Kaufman pivots averages 3:2, this estimate of extra work performed suggests that the cost of bounded Bunch-Kaufman pivoting is much nearer $O(n^2)$ than $O(n^3)$, or little more than the cost of partial pivoting.

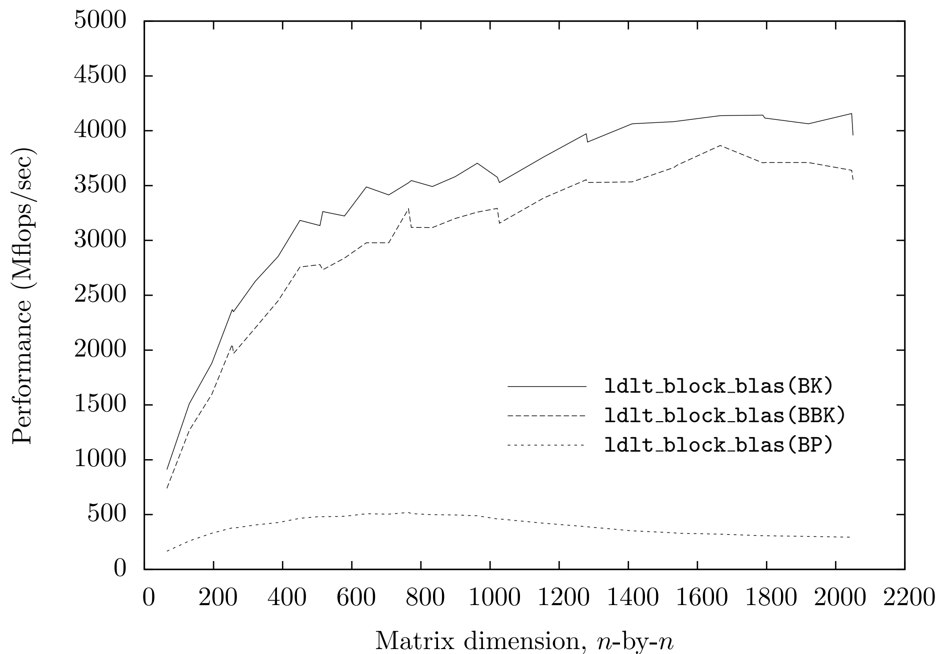

FIG. 5.11. *LDL^T factorization, blocked algorithm using BLAS.*

6. Modified Cholesky Algorithms. Given a symmetric, possibly indefinite, n-by-n matrix A, modified Cholesky algorithms find a matrix $\widehat{A} = A + E$, where $\widehat{A}$ is sufficiently positive definite and reasonably well-conditioned, while preserving as much as possible the information of A. Fang and O'Leary catalog modified Cholesky algorithms and analyze the asymptotic cost of the different approaches [13]. Their research evaluates how close these different approaches come to achieving the goal of keeping the cost of the algorithm to a small multiple of n^2 higher than that of standard Cholesky factorization, which takes $\frac{1}{3}n^3 + O(n^2)$ flops.

Fang and O'Leary analyze three factorizations of a symmetric matrix A:

1. $PAP^T = LDL^T$, where D is diagonal, L is unit lower triangular and P is a permutation matrix for symmetric pivoting.
2. $PAP^T = LBL^T$, where B is block diagonal with block order 1 or 2.
3. $PAP^T = LTL^T = L(\widetilde{P}^T\widetilde{L}\widetilde{B}\widetilde{L}^T\widetilde{P})L^T$, where T is tridiagonal with off-diagonal elements in the first column all zero, and $\widetilde{P}T\widetilde{P}^T = \widetilde{L}\widetilde{B}\widetilde{L}^T$ is the LBL^T factorization of T.

Existing modified Cholesky algorithms typically use either the LDL^T or LBL^T factorization. Fang and O'Leary propose a new modified Cholesky algorithm, which uses a sandwiched LTL^T-LBL^T factorization and modifies a computed factorization. Aasen introduced an algorithm for reducing an n-by-n symmetric matrix to tridiagonal form, which involves $\frac{1}{3}n^3 + O(n^2)$ flops and exhibits numerical stability comparable to that of Gaussian elimination with partial pivoting [1]. Fang and O'Leary show that the LBL^T factorization of a symmetric tridiagonal matrix remains sparse, and the cost of symmetric pivoting is no more than $O(n^2)$, since pivot selection requires at most $3k$ comparisons for a k-by-k Schur complement [12]. Finally, the computed $L(\widetilde{P}^T\widetilde{L}\widetilde{B}\widetilde{L}^T\widetilde{P})L^T$ factorization is modified using the method proposed by Cheng and Higham [8]. This new approach to modified Cholesky factorization achieves the objective of keeping the cost of the algorithm to a small multiple of n^2 higher than that of standard Cholesky factorization.

This paper analyzes the performance of modified Cholesky algorithms based on the more typical LDL^T and LBL^T factorizations. In particular, we consider the modified LDL^T algorithm proposed by Gill, Murray and Wright [14], and the modified LBL^T algorithm proposed by Cheng and Higham [8]. Note that the discussion of the previous section on symmetric indefinite factorization is referred to here as LBL^T factorization, consistent with the notation used by Fang and O'Leary [13].

The Gill-Murray-Wright algorithm modifies the matrix A as the factorization proceeds. Suppose pivot selection and symmetric pivoting have been performed at the kth step of the LDL^T factorization of an n-by-n symmetric positive definite matrix $\widehat{A} = A+E$. Let $P_k\widehat{A}_kP_k^T = \widetilde{A}_k = \begin{pmatrix} a_k & c_k^T \\ c_k & \bar{A}_k \end{pmatrix}$ be the Schur complement, where $a_k \in \mathbb{R}$, $c_k \in \mathbb{R}^{(n-k)\times 1}$ and $\bar{A}_k \in \mathbb{R}^{(n-k)\times(n-k)}$. Note that if $\widehat{A}$ is positive definite, so are $P_k\widehat{A}_kP_k^T$ and its principal sub-matrices, and all diagonal entries are positive. So, $\hat{a}_k = a_k + \delta_k > 0$. The Gill-Murray-Wright algorithm sets

$$\hat{a}_k = \max\left\{\delta, |a_k|, \frac{\|c_k\|_\infty^2}{\beta_2}\right\}$$

where $\delta = \epsilon_M$ (machine epsilon), $\|c_k\|_\infty = \max_{k<j\le n}|a_{kj}|$, $\beta^2 = \max\left\{\eta, \frac{\xi}{\sqrt{n^2-1}}, \epsilon_M\right\}$, and η and ξ are the maximum magnitude of the diagonal and off-diagonal elements of A, respectively. Then,

$$D(k,k) = \hat{a}_k,\ L(k{+}1\!:\!n,k) = \frac{c_k}{\hat{a}_k},\ \text{and}\ \ \widehat{A}_{k+1} = \bar{A}_k - \frac{c_kc_k^T}{\hat{a}_k}.$$

To ensure numerical stability, the Gill-Murray-Wright algorithm pivots on the maximum magnitude diagonal element. That is, at the kth step, rows and columns are symmetrically interchanged such that $|a_k| \geq |\widehat{A}_k(j,j)|$ for $j = 1, \ldots, n-k+1$. The cost of this form of partial pivoting is $O(n^2)$.

Any symmetric positive definite matrix has an LDL^T factorization, where the diagonal elements of D are positive. The Gill-Murray-Wright algorithm, which modifies the matrix A as the factorization proceeds, factors a symmetric positive definite matrix $\widehat{A} = A + E$, such that $P\widehat{A}P^T = LDL^T$. For symmetric matrices, in general, an LDL^T factorization may fail to exist, but any symmetric matrix has an LBL^T factorization. Therefore, modified Cholesky algorithms that modify a computed factorization, including the Cheng-Higham algorithm, use the LBL^T factorization.

The Cheng-Higham algorithm first computes $PAP^T = LBL^T$, and then perturbs B such that $P\widehat{A}P^T = P(A+E)P^T = L(B+\Delta B)L^T = L\widehat{B}L^T$ and $\widehat{A} = A + E$ is symmetric positive definite. Given an LBL^T factorization of A, the Cheng-Higham algorithm modifies each 1-by-1 diagonal block d in B to be $\hat{d} = \max\{\delta, d\}$. For each 2-by-2 diagonal block D in B, the algorithm calculates its spectral decomposition $D = U \begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix} U^T$, sets $\hat{\lambda}_k = \max\{\delta, \lambda_k\}$ for $k = 1, 2$, and computes $\widehat{D} = U \begin{pmatrix} \hat{\lambda}_1 & \\ & \hat{\lambda}_2 \end{pmatrix} U^T$. The parameter $\delta = \sqrt{\epsilon_M/2}\|A\|_\infty$ is the preset modification tolerance, where $\|A\|_\infty = \max_{1\leq i\leq n} \left\{ \sum_{j=1}^n |a_{ij}| \right\} = \max_{1\leq j\leq n} \left\{ \sum_{i=1}^n |a_{ij}| \right\}$ for a symmetric matrix.

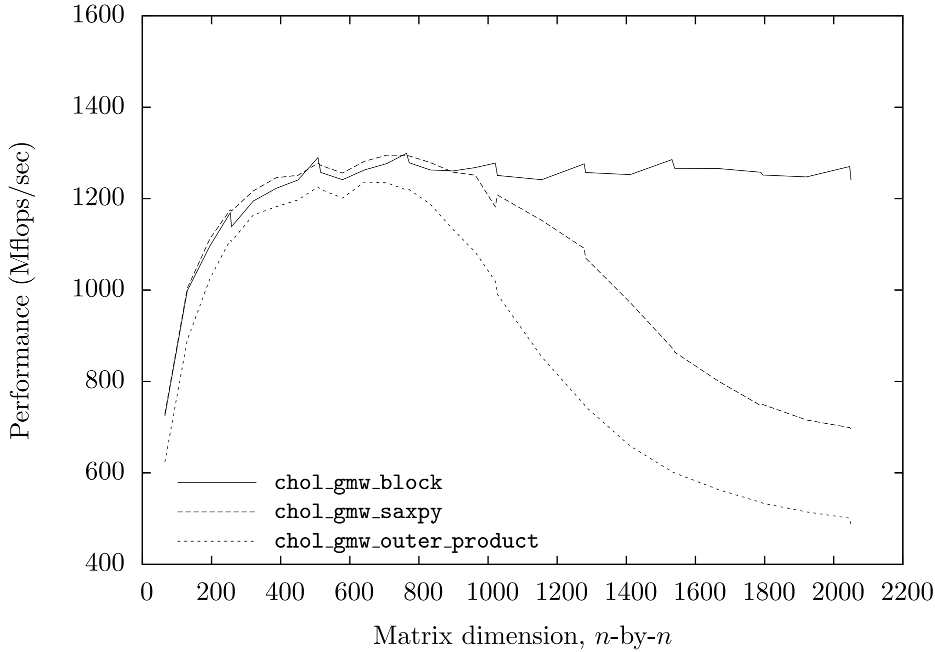

Fig. 6.1. *Data locality in modified Cholesky, Gill-Murray-Wright algorithm.*

We have seen that Cholesky factorization, which provides a lower bound on modified Cholesky factorization, involves $\frac{1}{3}n^3+O(n^2)$ flops, and symmetric pivoting involves between $O(n^2)$ and $O(n^3)$ flops. More specifically, the cost of partial pivoting is $O(n^2)$ flops, complete pivoting $O(n^3)$ flops, and rook pivoting between $O(n^2)$ and $O(n^3)$ flops. The Gill-Murray-Wright algorithm employs partial pivoting, while our implementation of the Cheng-Higham algorithm employs either Bunch-Kaufman (partial) or bounded Bunch-Kaufman (rook) pivoting. In the previous section we estimated that the cost of bounded Bunch-Kaufman pivoting is little more than the cost of partial pivoting. Given that both the Gill-Murray-Wright and Cheng-Higham modifications to the symmetric indefinite factorization involve a small multiple of n^2 flops, we expect variations in performance between modified Cholesky algorithms to be largely explained by the pivoting strategy employed.

Here, we adopt the same naming convention as used in earlier sections of this paper to reference modified Cholesky algorithms. Since the Cheng-Higham algorithm modifies a computed symmetric indefinite factorization, our implementation of the outer product method (`chol_ch_outer_product`), SAXPY operation (`chol_ch_saxpy`), simple blocking (`chol_ch_block`) and blocked routine using BLAS (`chol_ch_block_blas`) invoke the respective symmetric indefinite factorization algorithms discussed in the previous section. On the other hand, the Gill-Murray-Wright algorithm modifies the symmetric matrix as the factorization proceeds, so we developed separate and distinct routines: outer product method (`chol_gmw_outer_product`), SAXPY operation (`chol_gmw_saxpy`), simple blocking (`chol_gmw_block`) and blocked routine using BLAS (`chol_gmw_block_blas`).

Firstly, to complete the data locality picture, Figure 6.1 plots the effect of loop reordering (`chol_gmw_saxpy`) and blocking (`chol_gmw_block`) on performance for the Gill-Murray-Wright algorithm.

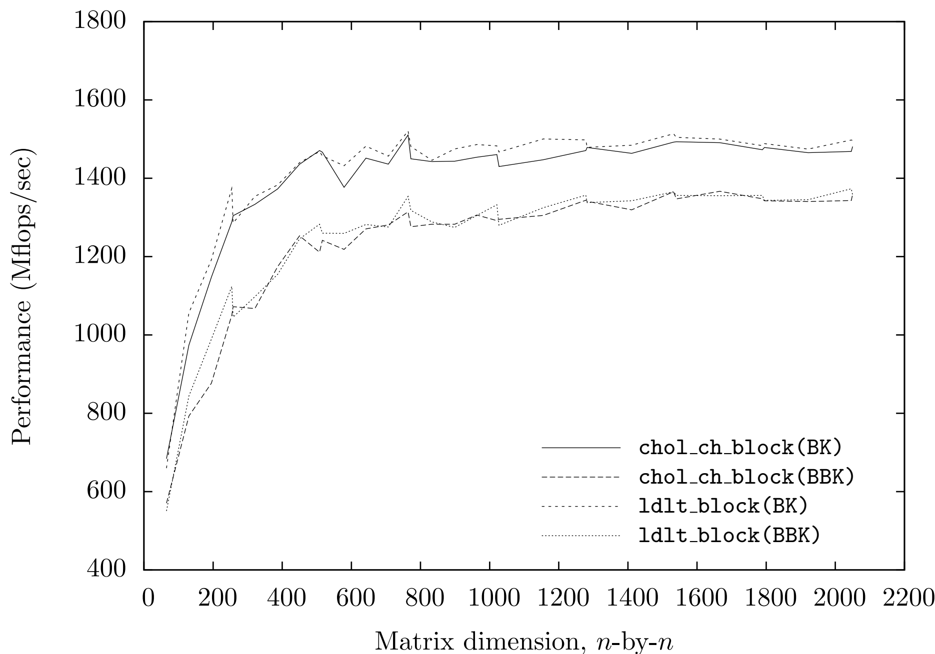

FIG. 6.2. *Cost of modifying LDL^T factorization, simple blocking.*

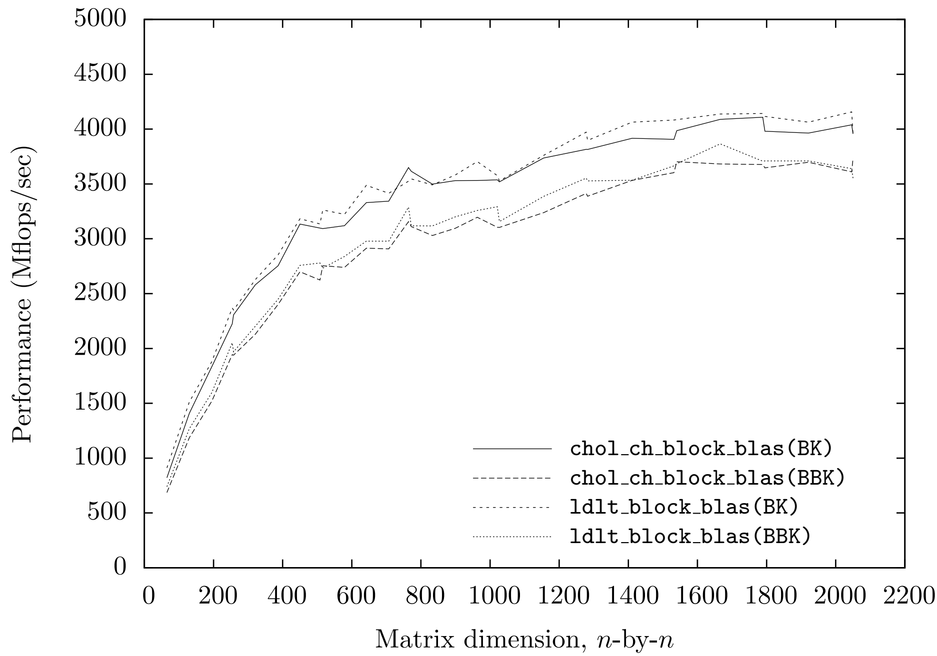

FIG. 6.3. *Cost of modifying LDL^T factorization, blocked routines using BLAS.*

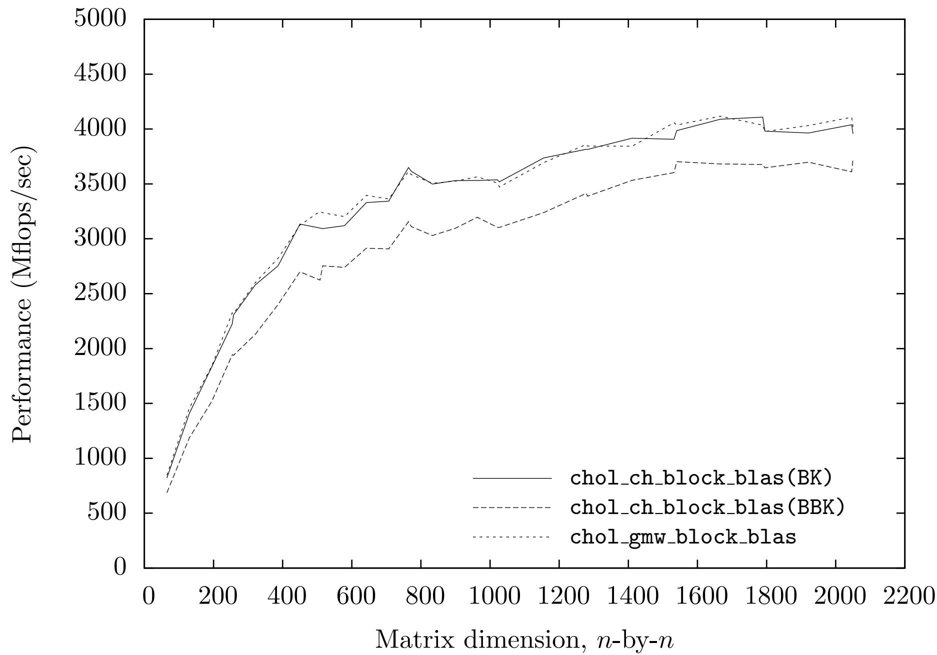

FIG. 6.4. *Modified Cholesky, blocked routines using BLAS.*

One way to measure the cost of modifying a symmetric indefinite factorization is to compare the performance of modified Cholesky algorithms with the corresponding algorithms implementing symmetric indefinite factorization. Figure 6.2 plots the performance of our implementation of simple blocking for the Cheng-Higham algorithm and symmetric indefinite factorization, each employing Bunch-Kaufman and bounded Bunch-Kaufman pivoting. Figure 6.3 makes the same performance comparisons for our blocked routines using BLAS. Both charts reveal that the incremental cost of modifying the symmetric indefinite factorization is small relative to the difference in cost between Bunch-Kaufman and bounded Bunch-Kaufman pivoting. Consistent with this observation, Figure 6.4 shows that modified Cholesky algorithms employing partial pivoting perform in line with one another (`chol_gmw_block_blas` and `chol_ch_block_blas(BK)`), while modified Cholesky factorization with rook pivoting (`chol_ch_block_blas(BBK)`) is more expensive.

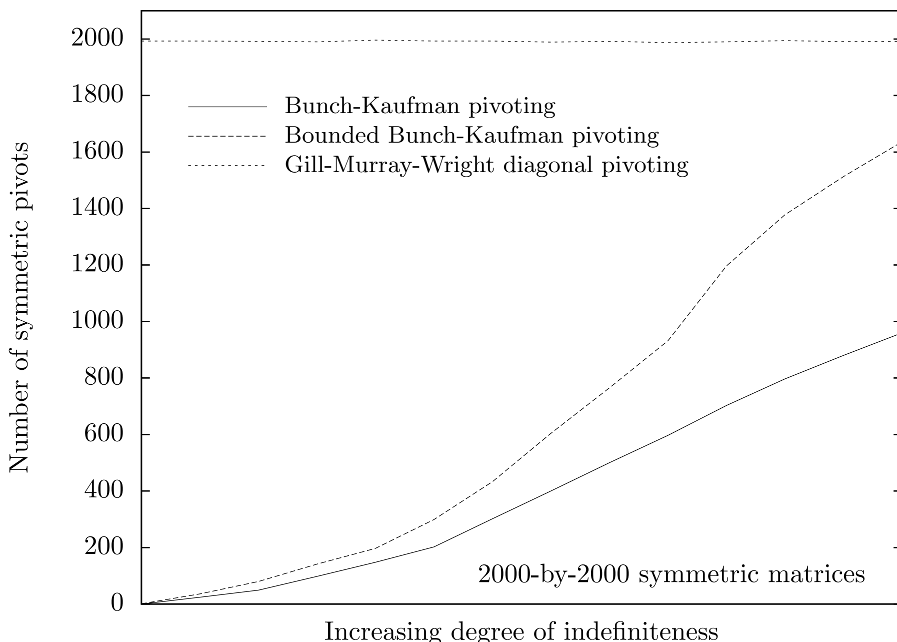

FIG. 6.5. *Number of symmetric pivots (row and column interchanges).*

Figure 6.5 contrasts the number of symmetric pivots (row and column interchanges) introduced by the Gill-Murray-Wright algorithm with that made by Bunch-Kaufman and bounded Bunch-Kaufman pivoting for a sample of randomly generated 2000-by-2000 symmetric matrices of varying degrees of indefiniteness. (This is the same sample of matrices used in the analysis of symmetric indefinite factorization, where the degree of indefiniteness is proxied by the number of Bunch-Kaufman pivots.) Since the Gill-Murray-Wright algorithm pivots on the maximum magnitude diagonal element, the number of row and column interchanges hovers near the matrix dimension irrespective of the degree of indefiniteness, not unlike Bunch-Parlett pivoting. Note that the charts above, which plot performance for a range of matrix dimensions, do so for randomly generated symmetric matrices whose degree of indefiniteness, or ratio of number of symmetric pivots to matrix dimension, is near the top end of Figure 6.5.

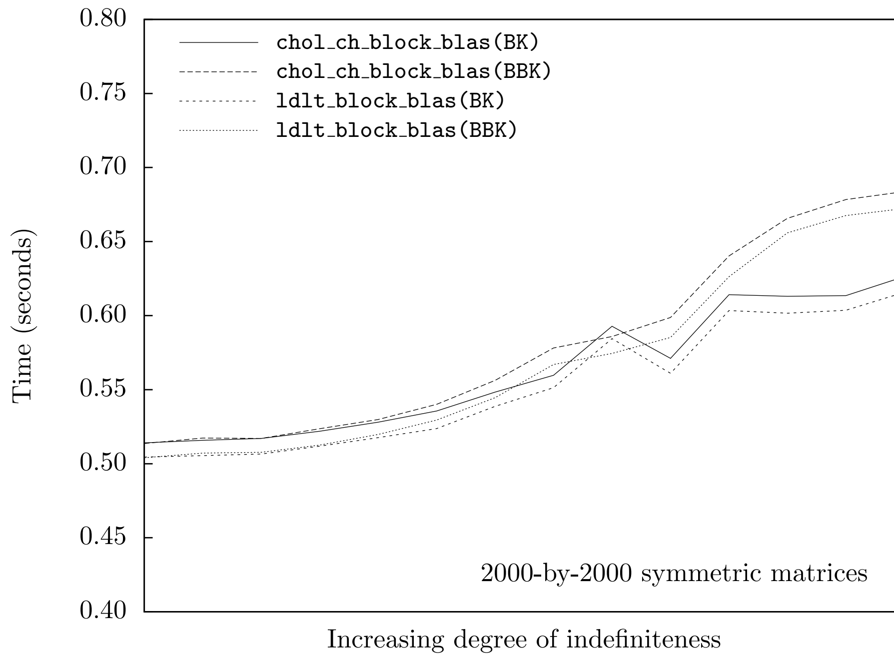

Fig. 6.6. *Cost of modifying LDL^T factorization with increasing degree of indefiniteness.*

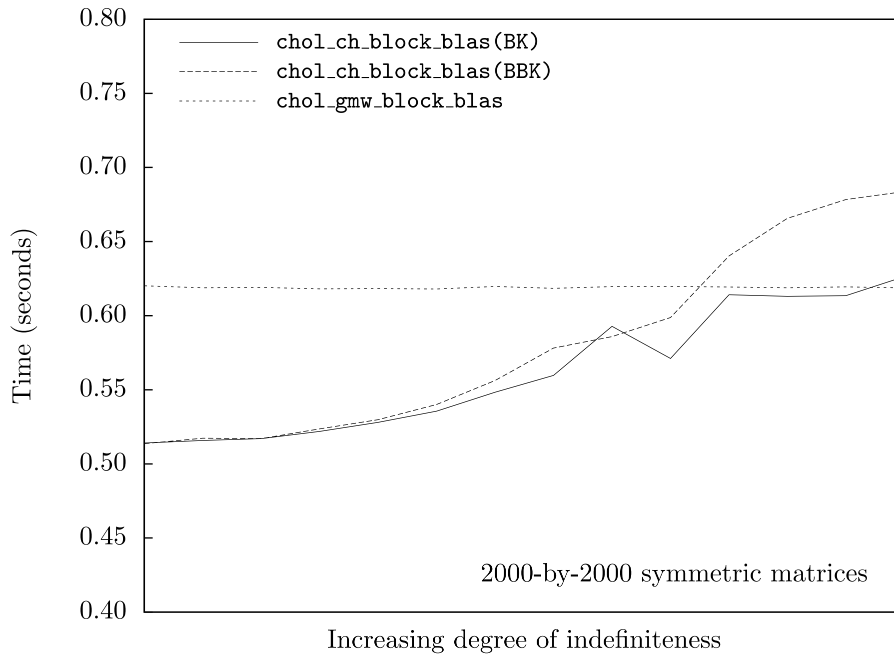

Fig. 6.7. *Modified Cholesky algorithms with increasing degree of indefiniteness.*

In Figure 6.6 we provide an alternate measure (runtime) of the cost associated with modifying the symmetric indefinite factorization. For the sample of symmetric matrices, the incremental cost of the Cheng-Higham algorithm employing Bunch-Kaufman and bounded Bunch-Kaufman pivoting over the corresponding symmetric indefinite factorization algorithms is fairly constant across the spectrum of indefiniteness. We can explain the observed performance by remarking that the computation of the preset modification tolerance by the Cheng-Higham algorithm involves $O(n^2)$ flops, while its modification of 1-by-1 and 2-by-2 diagonal blocks takes only $O(n)$ flops.

The time for algorithms `chol_ch_block_blas(BK)` and `chol_ch_block_blas(BBK)` to perform modified Cholesky factorization on the sample of symmetric matrices of varying degrees of indefiniteness tracks that of `ldlt_block_blas(BK)` and `ldlt_block_blas(BBK)`, respectively — time increases as the number of symmetric pivots rises, and for Bunch-Kaufman and bounded Bunch-Kaufman pivoting the number of symmetric pivots rises with increasing degree of indefiniteness. As illustrated in Figure 6.5, the number of symmetric pivots for the Gill-Murray-Wright algorithm approaches the dimension of the matrix irrespective of the degree of indefiniteness, so the time for `chol_gmw_block_blas` to perform modified Cholesky factorization does not depend on the degree of indefiniteness of the symmetric matrix (Figure 6.7).

7. Parallel Programming. This paper evaluates techniques for optimizing the performance of serial algorithms implementing Cholesky factorization, symmetric indefinite factorization and modified Cholesky factorization. A natural extension of this research would develop parallel algorithms for these matrix factorizations, and measure their speedup and efficiency.

Speedup is the ratio of the runtime of the fastest serial algorithm to the runtime of a parallel version of the algorithm run on p parallel processes, $S_p = T_\sigma / T_\pi$. Amdahl's Law states that the speedup of a parallel program is limited by the serial fraction of the program. Let s be the serial fraction of the program, where $0 \leq s \leq 1$, then

$$S_p = \frac{p}{(p-1)s+1}, \text{ and } S_p \to \frac{1}{s} \text{ as } p \to \infty,$$

providing an upper bound on speedup. However, for many problems the serial fraction of the program decreases as the problem size increases, leading to scalability. Efficiency is the ratio of the runtime, or work done, by the serial program to the sum of the runtimes of p processes of the parallel program solving the same problem. Interprocess communication, idle time and extra computation are the main sources of overhead — extra work done by the parallel program over the serial program — which determines the efficiency of a parallel program. Linear speedup, $S_p = p$, implies a perfectly parallelized program with 100% efficiency. Returning to the concept of scalability, a parallel program is scalable if it is possible to maintain a level of efficiency by increasing the number processes with problem size [22].

As a demonstration of these ideas we implemented Fox's algorithm for parallel matrix multiplication using the Message-Passing Interface (MPI) library [22, 19]. On 10000-by-10000 matrices using 25 processors, Fox's algorithm achieves a speedup of 21.7 with 83% efficiency relative to the (serial) simple blocking algorithm — on each processor matrix blocks are multiplied using the simple blocking algorithm (Figure 7.1).

Finally, we note that ScaLAPACK is a library of functions for solving linear algebra problems on distributed-memory systems, and it has an implementation that uses MPI for its communication.

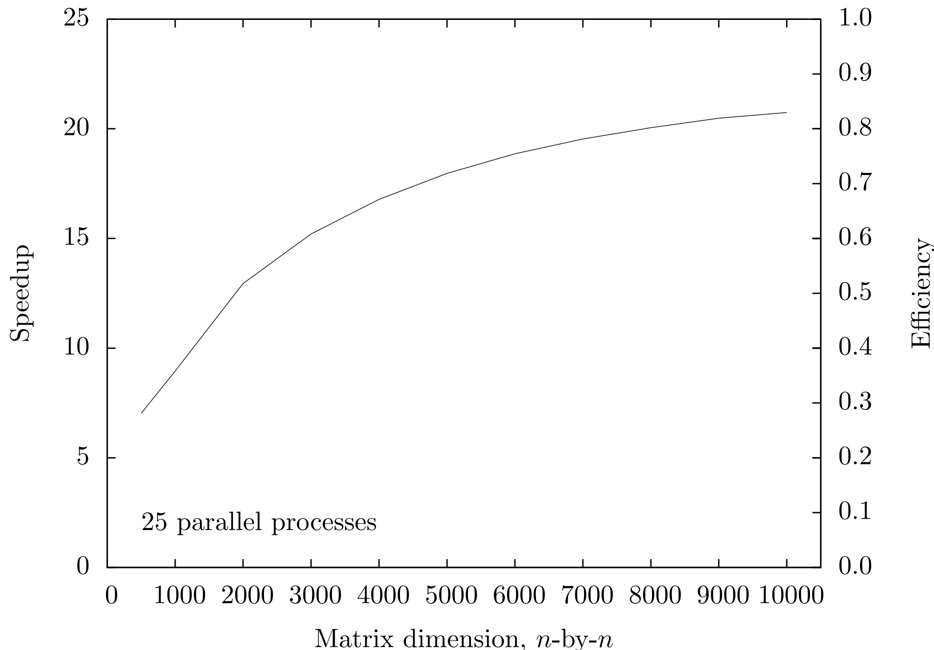

FIG. 7.1. *Fox's algorithm for parallel matrix multiplication using MPI.*

8. Conclusion. In this paper we measure and compare the performance of algorithms implementing Cholesky factorization, symmetric indefinite factorization, and modified Cholesky factorization (Gill-Murray-Wright and Cheng-Higham algorithms). For each of these matrix factorizations we developed routines that implement a variety of performance optimizations techniques including loop reordering, blocking and the use of tuned BLAS. We summarize observations made from the performance data generated by our timing experiments.

- High performance is achieved by maximizing data locality. For unblocked algorithms the SAXPY operation improves data locality relative to that of the outer product method through loop reordering. Data locality is further improved by partitioning matrices into blocks where matrix blocks are stored in fast access cache. For the matrix factorizations analyzed in this paper, the SAXPY operation outperforms the outer product method, and simple blocking outperforms the SAXPY operation. The promise of performance gains through recursive contiguous block storage did not materialize for our implementation of Cholesky factorization, so we did not pursue these techniques for symmetric indefinite and modified Cholesky factorizations.
- Reorganizing blocked algorithms to make use of tuned BLAS, and in particular level 3 BLAS, to the fullest extent possible produces the highest performance.
- Cholesky factorization is more efficient than LDL^T factorization of a symmetric positive definite matrix, primarily due to the greater use of level 3 BLAS by the Cholesky algorithm. We estimate the cost premium for LDL^T factorization of a symmetric positive definite matrix at 18% over the cost of its Cholesky factorization. As the degree of indefiniteness increases from symmetric positive definite to random symmetric we estimate that the cost

of LDL^T factorization with Bunch-Kaufman pivoting increases by a further 7% due to the increasing number of row and column interchanges.
- For a given degree of indefiniteness the ratio of bounded Bunch-Kaufman pivots to Bunch-Kaufman pivots averages 3 : 2. Coupled with estimates of duplicate computations associated with trailing sub-matrix updates applied to columns during pivot selection, we contend that bounded Bunch-Kaufman provides the numerical stability of complete pivoting at little more than the cost of partial pivoting. Employing bounded Bunch-Kaufman (rook) pivoting in lieu of Bunch-Kaufman (partial) pivoting adds approximately 11% to the cost of symmetric indefinite factorization.
- The cost of modifying a symmetric indefinite factorization is a small multiple of n^2 flops, so variations in performance between modified Cholesky algorithms is largely explained by the pivoting strategy employed. For randomly generated symmetric matrices, the Gill-Murray-Wright algorithm with partial pivoting and the Cheng-Higham algorithm with Bunch-Kaufman pivoting perform in line with one another, while the Cheng-Higham algorithm with bounded Bunch-Kaufman pivoting is more expensive.
- For the Cheng-Higham algorithm with either Bunch-Kaufman or bounded Bunch-Kaufman pivoting, the time to perform modified Cholesky factorization increases with increasing degree of indefiniteness of the symmetric matrix, since the number of symmetric pivots rises with increasing degree of indefiniteness. By contrast, the Gill-Murray-Wright algorithm pivots on the maximum magnitude diagonal element, so the number of symmetric pivots hovers near the matrix dimension irrespective of the degree of indefiniteness. Hence the time it takes to perform modified Cholesky factorization does not depend on the degree of indefiniteness of the symmetric matrix.

In conclusion, we remark that there are possibilities for continuing or extending this research effort. Section 6 introduced the modified Cholesky algorithm proposed by Fang and O'Leary, which uses a sandwiched LTL^T-LBL^T factorization that performs pivot selection on a (sparse) symmetric tridiagonal matrix. So, one possibility for future research is to develop basic and optimized versions of the Fang-O'Leary algorithm and compare their performance with that of the Gill-Murray-Wright and Cheng-Higham algorithms, which perform pivot selection on dense symmetric matrices. Another possibility, as discussed in the previous section, is to develop parallel algorithms for standard Cholesky, symmetric indefinite and modified Cholesky factorizations, and measure their speedup and efficiency. Presently, LAPACK provides unblocked and blocked routines for symmetric indefinite factorization, which only employ Bunch-Kaufman pivoting. As a final suggestion for future work, we propose enhancing LAPACK to provide the capability for symmetric indefinite factorization with bounded Bunch-Kaufman pivoting.

Acknowledgements. We are especially grateful to Professor David Bindel for advising on this thesis. His guidance on the organization of the research, and his expertise in high performance computing and linear algebra were invaluable. Additional thanks to Professor Michael Overton for his constructive comments during the review process. The experiments in this research were conducted on the high performance computing resources at New York University. We wish to acknowledge the technical support provided by the NYU Information Technology Services staff.

REFERENCES

[1] Jan O. Aasen, *On the reduction of a symmetric matrix to tridiagonal form*, BIT, 11 (1971), pp. 233–242.

[2] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis, *Accurate symmetric indefinite linear equation solvers*, SIAM Journal on Matrix Analysis and Applications, 20 (1998), pp. 513–561.

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz, *Communication-optimal parallel and sequential cholesky decomposition*, Tech. Report UCB/EECS-2009-29, University of California at Berkeley, 2009.

[4] *BLAS – Basic Linear Algebra Subroutines*. `http://www.netlib.org/blas/`, 2005.

[5] Stephen Boyd and Lieven Vandenberghe, *Convex Optimization*, Cambridge University Press, Cambridge, United Kingdom, 2004.

[6] James R. Bunch and Linda Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Mathematics of Computation, 31 (1977), pp. 163–179.

[7] James R. Bunch and Beresford N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM Journal on Numerical Analysis, 8 (1971), pp. 639–655.

[8] Sheung Hun Cheng and Nicholas J. Higham, *A modified Cholesky algorithm based on a symmetric indefinite factorization*, SIAM Journal on Matrix Analysis and Applications, 19 (1998), pp. 1097–1110.

[9] James W. Demmel, *Applied Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[10] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[11] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström, *Recursive blocked algorithms and hybrid data structures for dense matrix library software*, SIAM Review, 46 (2004), pp. 3–45.

[12] Haw-ren Fang and Dianne P. O'Leary, *Stable factorizations of symmetric tridiagonal and triadic matrices*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 576–595.

[13] ———, *Modified Cholesky algorithms: a catalog with new approaches*, Mathematical Programming (Series A), 115 (2008), pp. 319–349.

[14] Philip E. Gill, Walter Murray, and Margaret H. Wright, *Practical Optimization*, Academic Press, New York, NY, 1981.

[15] Stephan Goedecker and Adolfy Hoisie, *Performance Optimization of Numerically Intensive Codes*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2001.

[16] Gene H. Golub and Charles F. van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, MD, third edition ed., 1996.

[17] Brian Gough, *An Introduction to GCC*, Network Theory Limited, Bristol, United Kingdom, 2004.

[18] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, Upper Saddle River, NJ, second edition ed., 1988.

[19] Salman Khalid, *Fox's algorithm for matrix multiplication in a parallel environment*. Seminar on parallel algorithms, Center for Solid State Physics, Lahore, Pakistan, 2001.

[20] *LAPACK – Linear Algebra PACKage*. `http://www.netlib.org/lapack/`, June 2010. Version 3.2.2.

[21] Robert Mecklenburg, *Managing Projects with GNU Make*, O'Reilly, Cambridge, MA, third edition ed., 2005.

[22] Peter S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[23] Richard M. Stallman, Roland McGrath, and Paul Smith, *GNU Make: A Program for Directing Recompilation*, Free Software Foundation, Boston, MA, 2004.