Efficiency Enhancement by Blocking

Wilfried N. Gansterer[1]
Dieter F. Kvasnicka[2]
Christoph W. Ueberhuber[1]

[1] Institute for Applied and Numerical Mathematics,
University of Technology, Vienna
`ganst@aurora.tuwien.ac.at`
`christof@uranus.tuwien.ac.at`
[2] Institute for Physical and Theoretical Chemistry,
University of Technology, Vienna
`dieter@titania.tuwien.ac.at`

August 1998

AURORA TR 1998-10

Abstract

When developing high performance algorithms blocking is a standard procedure to increase the locality of reference. Conflicting factors which influence the choice of blocking parameters are described in this paper. These factors include cache size, load balancing, memory overhead, algorithmic issues, and others. Optimal block sizes can be determined with respect to each of these factors. The resulting block sizes are independent of each other and can be implemented in several levels of blocking within a program. A tridiagonalization algorithm serves as an example to illustrate several blocking techniques.

The work described in this paper was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

1 Introduction

In modern computer systems the gap between raw computing power and memory throughput increases tremendously. This leads to the effect that for many algorithms the processors are not running at full speed since they have to wait for data from memory.

One solution to this problem is to insert one or several levels of fast cache memories between main memory and the processor. These caches hold a subset of the data stored in main memory. If the processor accesses a data item from memory and this data item is resident in cache the memory access time is much shorter. The *locality of reference* is high if a high fraction of the memory accesses can be served from the high levels of the memory hierarchy, which consist (in top down order) of registers, one to three levels of caches, local memory, distributed memory, and secondary storage.

Many algorithms can be restructured to reuse data by rearranging the order of the operations performed. This method is called *blocking* (see Section 2).

Section 3 summarizes conflicting aspects which sometimes favor large blocks and under other circumstances favor small blocks. Section 5 describes a tridiagonalization algorithm, which serves as an example for the various blocking techniques discussed in this paper. Subsequent sections go into detail with these blocking methods.

2 Blocked Memory Access

The key to high-performance computing is to reuse data which have been transferred to a faster level of the memory hierarchy as often as possible before they are replaced by new data and restored at a slower level (see e. g., Lam et al. [18]).

In order to transform a matrix algorithm in such a way that the highest levels of the memory hierarchy are exploited to a high degree, the algorithm is reformulated as a block algorithm which operates on submatrices (blocks) of the original matrix. (The concept of blocking as well as hierarchical blocking of matrix operations is explained in detail in Ueberhuber [20].) The submatrices are chosen small enough such that for each operation of the block algorithm, all blocks needed as arguments can be stored in the respective level of the memory hierarchy. On the other hand the blocks should be large enough so that the number of operations executed on elements of those blocks is as large as possible.

Other factors that influence the optimal block sizes are cache line sizes and page sizes. If a cache line is already transferred to a high level of the memory hierarchy, it should be used completely before fetching the next cache line. As a result of the complexity of all these factors, the optimal blocking strategy can often only be determined by way of experiment.

3 Block Size

At first sight the size of submatrices to be worked on should be chosen so that they fit into a certain level of the memory hierarchy. To exploit more than one level several blocking levels, i. e., hierarchical blocking, should be applied. However, as will be shown in this paper, blocking techniques are also influenced by aspects which were not mentioned in Section 2.

Large Block Sizes. The main reasons for choosing large block sizes are (i) the exploitation of all cache levels (Section 4), (ii) the combination of messages in data parallel implementations (Section 10), and (iii) the reduction of algorithmic complexity (obtained, for instance, using the Strassen algorithm or particular complex matrix multiplication algorithms; Sections 8 and 9).

Small Block Sizes. The main reasons for choosing small block sizes are (i) the reduction of the number of matrix-vector operations in relation to matrix-matrix operations (Section 6), (ii) load balancing purposes in data parallel implementations (Section 10), (iii) algorithmic requirements, which can sometimes produce significant overhead when using large block sizes (Section 7), (iv) storage requirements (Section 11), and (v) memory throughput for complex floating-point numbers (Section 9).

All these aspects are investigated in more detail in the sections indicated above.

4 Basic Linear Algebra Subprograms

To account for the great variety of architectures available and in order to combine efficiency with portability, a set of matrix routines has been defined:

the Basic Linear Algebra Subprograms (BLAS; see Dongarra et al. [5]), which are optimized for each new computer system by hardware or compiler vendors. These routines perform hardware-optimized blocking and usually run at the highest floating-point performance achieveable on a certain computer. Table 1 illustrates the advantage of Level 3 (matrix-matrix) BLAS through a comparison of the ratios of floating-point operations to data movement for three closely related operations from the Level 1, 2, and 3 BLAS (see Dongarra et al. [5]).

Table 1: Ratio of floating-point operations to data movement.

BLAS	Routine	Memory Accesses	Flops	Flops per Memory Access
Level 1	`daxpy`	$3n$	$2n$	$2/3$
Level 2	`dgemv`	n^2	$2n^2$	2
Level 3	`dgemm`	$4n^2$	$2n^3$	$n/2$

Numerical algorithms should exploit Level 3 BLAS as much as possible in order to achieve highest floating-point performance on a wide variety of computers. In block algorithms BLAS routines are called on submatrices. The size of these submatrices should be large enough to give enough possibility to exploit hardware features.

5 Tridiagonalization

An algorithm whose performance is strongly determined by an appropriate choice of blocking parameters is the tridiagonalization algorithm. This algorithm is used in the solution of eigenvalue problems to transform a symmetric input matrix to a similar tridiagonal matrix (Golub, Van Loan [11]).

Unblocked tridiagonalization of the symmetric matrix A, $T = Q^T AQ$, requires the product $Q = H_n H_{n-1} \dots H_1$ of Householder reflectors $H_i = I - \tau u_i u_i^T$. The vector u_i is chosen to eliminate the ith column of A except for the diagonal and subdiagonal elements.

The matrix Q is not explicitly accumulated, but the Householder reflectors are applied to the matrix A as soon as they are available: $A^{(i)} = H_i^T A^{(i-1)} H_i$, with $A^{(0)} \equiv A$ and $A^{(n-1)} \equiv T$. So the computation of T is actually done by a series of rank-two updates to the matrix A.

6 Blocked Tridiagonalization in One Sweep

LAPACK provides a blocked *one-sweep* tridiagonalization algorithm (see Anderson et al. [1] and Dongarra et. al [6, 7]). Like in most linear algebra routines, this blocked tridiagonalization routine computes a matrix panel from a block of columns and then applies this panel to the rest of the matrix in an update operation. The computation of this panel cannot be implemented using efficient Level 3 matrix operations. So the optimum blocking parameter is heavily influenced by the amount of work which has to be done using Level 2 (matrix-vector) operations. The influence of the blocking parameter is illustrated in Fig. 1. Note, the optimum block size decreases with increasing matrix size n, so that the current panel fits into the largest cache. This is, however, a minor effect: for a 300×300 matrix block sizes from 6 to 300 lead to a performance which is within a 10 % range near the optimum floating-point performance. The execution times of this algorithm (and of the two-sweep algorithm presented in the next chapter) are shown in Fig. 2.

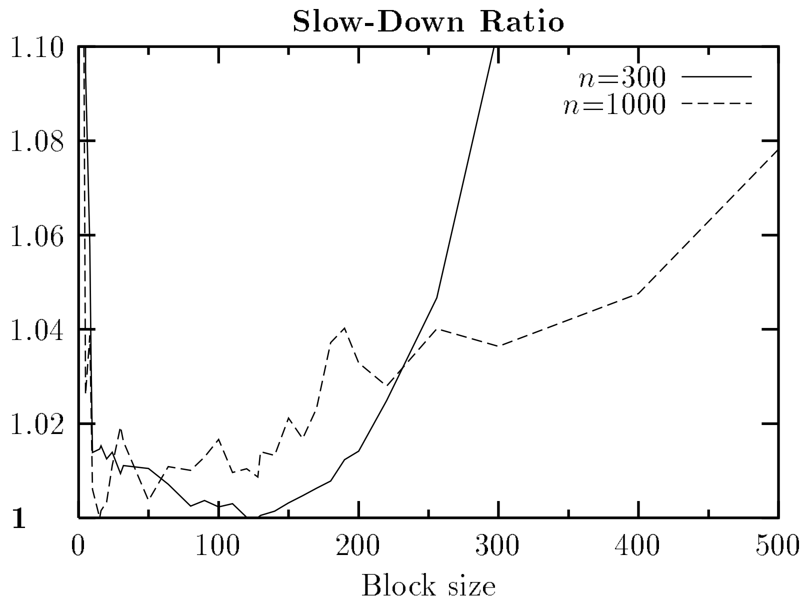

Figure 1: Slow-down effect of suboptimal choice of blocking parameters in the blocked one-sweep tridiagonalization. The minimum (optimum) execution time is scaled to 1 for each matrix size ($n = 300$ and $n = 1000$).

The one-sweep tridiagonalization routine may use Level 3 operations for about one half of the floating-point operations. However, the default blocking parameter in LAPACK is set to 1 for the tridiagonalization, which was best for most architectures at the time LAPACK was released. This blocking parameter can be adapted to a given architecture during installation (but most installations do not take advantage of that feature).

The Level 3 operations of the one-sweep algorithms occur as rank-$2k$ updates. The remaining half of the floating-point operations are performed by a matrix-vector product, i. e., a Level 2 operation. This matrix-vector product cannot be avoided in a one-sweep tridiagonalization (see Haunschmid, Kvasnicka [12] and Gansterer, Kvasnicka [9]). The tridiagonalization of large matrices on cache-based computers may spend more than 80 % of the overall execution time in the 50 % of floating-point operations required by the matrix-vector product.

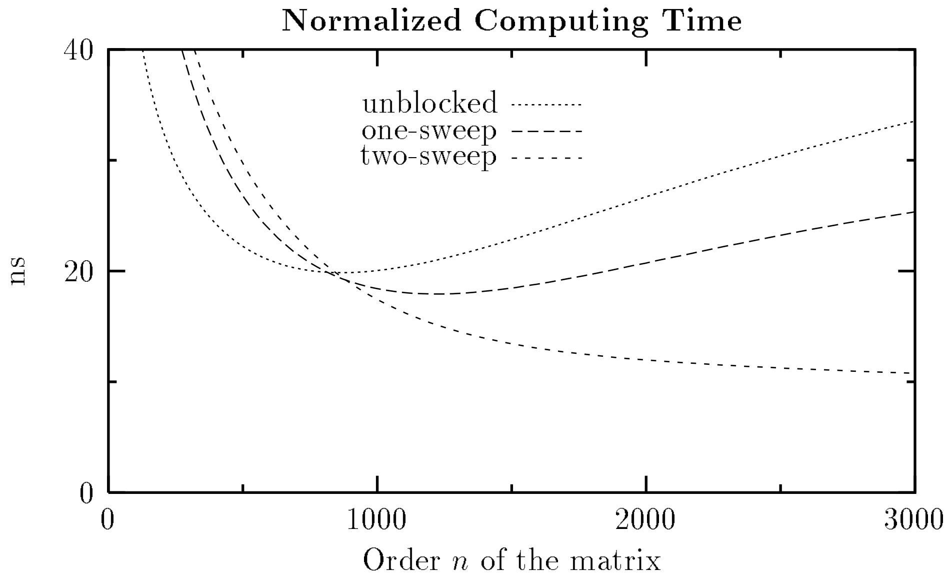

Figure 2: Comparison of normalized execution times (T/n^3) of the unblocked, the one-sweep and the two-sweep tridiagonalization.

To enable the utilization of Level 3 BLAS for all accesses to the full matrix, the algorithm has to be modified. A *two-sweep* algorithm has to be used (see Fig. 3, Gansterer, Kvasnicka [9], and Kvasnicka et al. [17]): A first sweep reduces the full matrix A to a band matrix B using Level 3 BLAS and a second sweep reduces B to a tridiagonal matrix T. If the band width of B is chosen reasonably small, most computational effort is spent in the first sweep, where fast Level 3 BLAS routines can be applied.

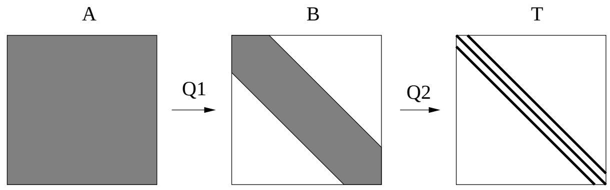

Figure 3: Tridiagonalization in two sweeps.

7 Blocked Tridiagonalization in Two Sweeps

In the two-sweep tridiagonalization of Section 6, increasing the band width of B increases not only the number of operations which cannot be performed in Level 3 routines, but does effectively increase the total number of operations.

Thus the optimum blocking parameter enables the efficient use of Level 3 operations while the total number of operations is not increased excessively. The influence of the choice of the blocking parameter is shown in Fig. 4. Note, the range of useful blocking parameters is narrower than for the one-sweep algorithm. This range increases with increasing n, because the overhead is of lower complexity than n^3. Computing times required by the various tridiagonalization algorithms of this paper can be found in Fig. 2.

The SBR toolbox (see Bischof et al. [2]) uses a QR factorization algorithm to reduce a dense symmetric matrix to a band matrix. This algorithm (see Algorithm 1) reveals that, in contrast to the LAPACK blocking, no access to the rest of the matrix A has to be made to accumulate the update matrices. There are two accesses to the matrix A in the matrix update. Both of them are blocked with block size $\bar{p}$.

Details of this algorithm can be found in Gansterer, Kvasnicka [9], an implementation is included in the SBR toolbox (see Bischof et al. [2]). A complete solution of the symmetric eigenproblem is described in Gansterer, Kvasnicka [10] and in Kvasnicka et al. [17]).

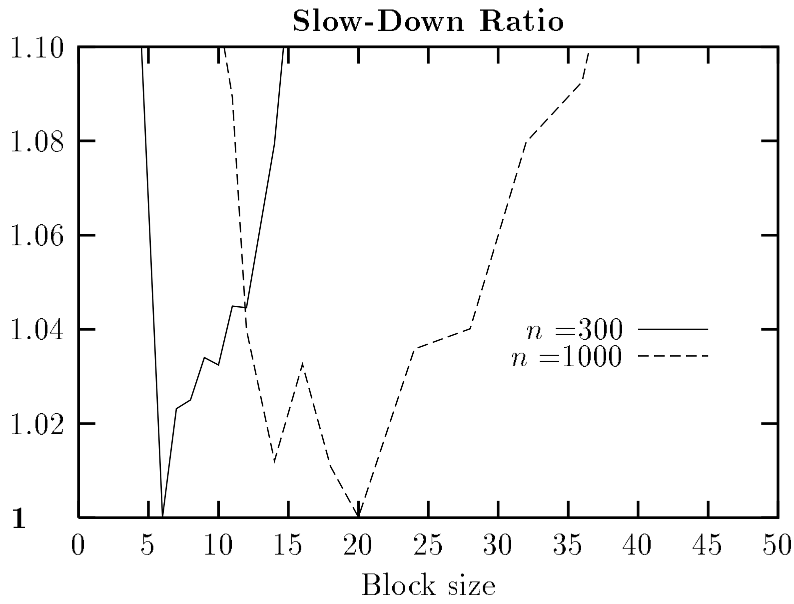

Figure 4: Slow-down effect of suboptimal choice of blocking parameters in the blocked tridiagonalization. The minimum (optimum) execution time is scaled to 1 for each matrix size.

8 The Strassen Algorithm

Using Strassen's algorithm for the multiplication of matrices of arbitrary order the complexity of this Level 3 operation is less than $28n^{\log_2 7}$ arithmetic operations (see Strassen [19]).

It is not easy to implement the Strassen algorithm efficiently. Thus, for a long time the Strassen algorithm was regarded to be of theoretical interest only. However, Douglas et al. [8] published efficient implementations for use on both single-processor and parallel computers. Some software libraries (e. g., IBM's ESSL) already comprise implementations of the Strassen algorithm, as well.

On most computers the break-even point $n_{\min}$, i. e., the minimum matrix order for which a Strassen implementation takes less time than the corresponding Blas-3 subroutine, is between 32 and 256. For larger matrices a significant complexity and run-time reduction is achieved (see Fig. 5 and Ueberhuber [20]).

Thus, to exploit the capabilities of the Strassen algorithm in blocked

Algorithm 1

n is the order of the matrix A
$\bar{p}$ is the block size
$\bar{N} = (n-2)/\bar{p}$
$A^{(0)} = A$
do $k = 1, \bar{N}$
  $\bar{s} = (k-1)\bar{p} + 1$
  $\bar{r} = k\bar{p}$
  – QR factorization with $Q^{(k)} = I - W^{(k)}(Y^{(k)})^T$
  $(W^{(k)}, Y^{(k)}) = \mathrm{QR}(A^{(k-1)}(\bar{s}:\bar{r}))$
  – matrix update
  $A^{(k)} = (I - W^{(k)}(Y^{(k)})^T)^T A^{(k-1)}(I - W^{(k)}(Y^{(k)})^T)$
end do
$B = A^{(\bar{N})}$

tridiagonalization the block size used in matrix-matrix multiplications should be as large as possible.

It has to be mentioned that the Strassen algorithm is less stable than the conventional method for multiplying two matrices. In a way similar to the modified version for multiplying two complex matrices (cf. Section 9) the best bound known for the error is a *normwise* and not an elementwise bound (Higham [14]). But in most practical cases the computed solution is of acceptable quality.

9 Multiplication of Complex Matrices

Blocked algorithms are characterized by a high fraction of Level 3 operations. This can be utilized to reduce the overall complexity (see Higham[15]). The multiply-add operation, $a = c \times d + e$, in its complex form

$$(a_r + a_i\mathrm{i}) = (c_r + c_i\mathrm{i}) \times (d_r + d_i\mathrm{i}) + (e_r + e_i\mathrm{i}),$$

can be written using four real multiplications and four additions

$$a_r = c_r \times d_r - c_i \times d_i + e_r, \qquad a_i = c_r \times d_i - c_i \times d_r + e_i$$

and in a form using only three real multiplications and seven additions

$$X_1 = c_r \times d_r, \qquad X_2 = c_i \times d_i, \qquad X_3 = (c_r + c_i) \times (d_r + d_i)$$

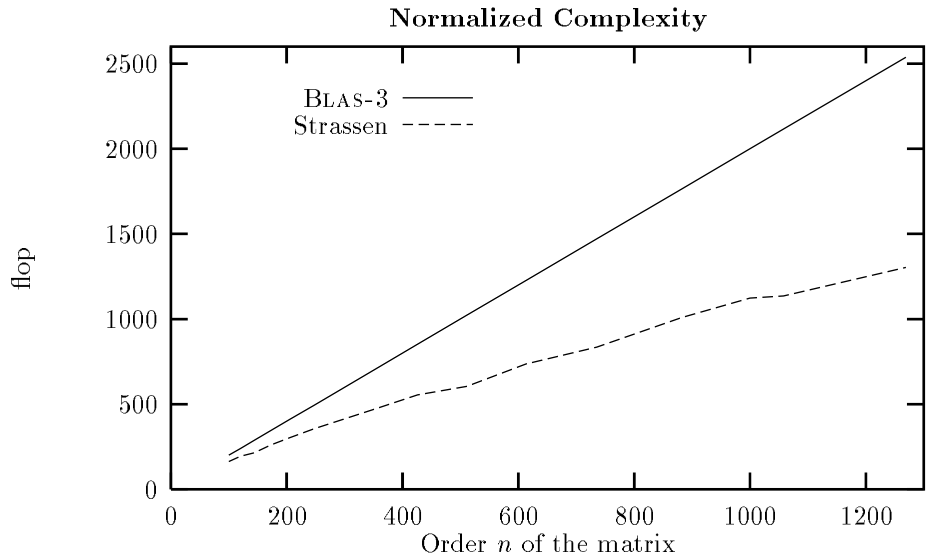

Figure 5: Number *flop* of floating-point operations *per element* of the resulting matrix required by the Strassen algorithm and the BLAS 3 implementation based on standard matrix-matrix multiplication.

$$a_r = X_1 - X_2 + e_r, \qquad a_i = X_3 - X_1 - X_2 + e_i.$$

On modern computer architectures, the latter method has the major drawback of a larger number of add instructions. However, for blocked matrix algorithms, a new side of this method is revealed: A multiply-add operation on subblocks requiring only three block-multiplies instead of four enables a great complexity reduction. A matrix multiply requires (roughly) n^3 multiply-add instructions. A matrix add requires n^2 add instructions[1]. So reducing the effort for the complex matrix-matrix multiply from $4n^3$ to $3n^3 + O(n^2)$ operations saves approximately 25 % of the required work.

[1] Addition operations require only add instructions and no multiplies, but since the time required for one add is the same as the time for one multiply-add on most of the modern processors the respective numbers of instructions are directly comparable. A difference is, however, when the algorithm is memory bound. In this case the add and multiply instructions are executed faster than the fused multiply-add instructions because they require one memory transfer less (only two reads and one write instead of three reads and one write).

Block Sizes. The aforementioned technique produces an increasing complexity reduction with growing matrix size n. For blocked algorithms this n is the block size. So algorithms having large block sizes are to be preferred.

If the algorithm is memory bound on a specific computer system block sizes must not be too large.

A multiply-add operation on complex data involves four multiplications and four additions on real data whereas the data transfer is only twice as much. So memory bound algorithms may have twice the floating-point performance using complex data than using real data.

If the block size is too large to store all required data in the first level cache the speed-up factor of two for memory bound complex algorithms[2] is lost and only a speed-up factor of 4/3 can be obtained from reduced complexity.

It has to be mentioned that the multiplication technique of this section is numerically less stable than conventional matrix-matrix multiplication (see Higham [15]). Nevertheless, when estimating *normwise* instead of componentwise rounding errors the resulting bounds are comparable to that of conventional matrix-matrix multiplication.

10 Blocking for Parallel Computing

In parallel computing the choice of an appropriate data distribution heavily depends on the data flow features of the involved algorithms. For dense matrix computations, ScaLapack routines (see Blackford et al. [3]) and HPF programs (see High Performance Fortran Forum [13]) achieve their best floating-point performance if a two-dimensional block-cyclic data layout scheme is used.

The two main issues in choosing a data layout for dense matrix computations are (i) load balancing, i. e., splitting the work reasonably well among the processors, and (ii) use of Level 3 Blas for local computation on all processors, to avoid excessive movement of data to and from memory.

Block sizes have to be chosen large enough to enable all the lower levels of blocking to have computations within one block local on one processor or at least on a small subset of the available processors.

[2]A complex multiply-add operation consists of four real multiply-adds, but only twice the memory traffic of a real multiply-add. So a factor of $4/2 = 2$ is won for memory bound problems.

On parallel computers with significant latency, block sizes must not be too small, i. e., communication patterns should not be vector oriented but matrix oriented. This technique is called *message combining*.

In general it can be said that in parallel computing large block sizes are more favorable than small ones.

11 Storage Efficient Blocking

Symmetric and tridiagonal matrices (as arising in Sections 5, 6, and 7) require special storage techniques to avoid significant waste of memory. A tradeoff between floating-point performance and storage efficiency influences the choice of the optimum blocking parameters.

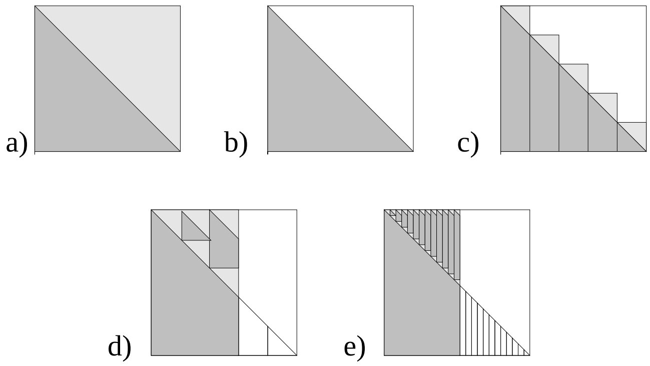

Figure 6: Storage schemes: a) conventional (*unpacked*) storage, b) packed storage, c) block packed storage, d) and e) parallel packed storage scheme (d) $b = n/5$, e) $b = n/25$). The darker shaded blocks indicate matrix elements whose values are stored; the lighter shaded parts indicate redundant storage.

Triangular and symmetric matrices can be stored using only half the memory requirement of standard Lapack routines and at the same time rely on Level 3 Blas. Efficient storage schemes are discussed in Haunschmid, Kvasnicka [12], Kvasnicka [16] (see Fig. 6), and D'Azevedo, Dongarra [4]. The storage overhead of some of these schemes is given in Fig. 7.

Since the storage schemes c), d), and e) may lead to non-contiguous data layouts the respective algorithms become more complicated. Block sizes which are not larger than the block size of the storage scheme have to be used.

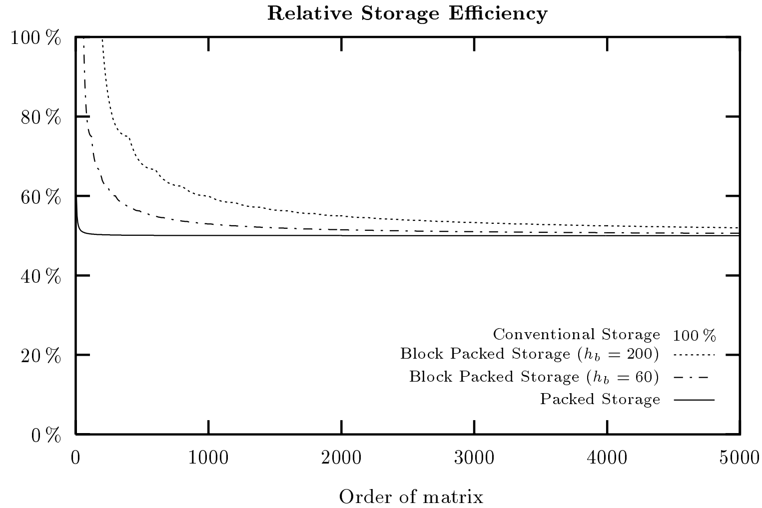

Figure 7: Storage requirement of several storage formats. The ratio of required storage locations (words) to the total number of matrix elements (100 %) is plotted.

12 Blocking Parameter Values

Blocking parameters have to be chosen in a problem and hardware dependent way. Typical values are

$$(32\ldots512) \geq b_{\text{parallel}} \geq b_{\text{storage}} \geq b_{\text{one-sweep}} \geq$$

$$(32\ldots128) \geq b_{\text{two-sweep}} \geq (6\ldots16).$$

b_{parallel} denotes the block size required by data parallel programming on distributed memory computers, b_{storage} the block size of the chosen storage scheme, $b_{\text{one-sweep}}$ the block size of the rank-$2k$ update of the one-sweep tridiagonalization (see Section 6), and $b_{\text{two-sweep}}$ the block size of the two-sweep tridiagonalization (see Sections 6 and 7).

For local computations matrices only need to be blocked in one dimension. The other dimension can be as large as necessary, except for the complexity oriented blocking. Level 3 BLAS routines are often called with rectangular matrices where one dimension is smaller by a factor of up to 100 than the other dimension. The smaller dimension should not be chosen below a certain hardware dependent level in order to avoid performance degradation.

Many block sizes will be chosen to be equal in practice. For instance, numerical experience shows that values of

$$b_{\text{parallel}} = b_{\text{storage}} = b_{\text{lapack}} = 64 \qquad \text{and} \qquad b_{\text{two-sweep}} = 8$$

are a good choice for the tridiagonalization algorithms of Sections 6 and 7.

13 Conclusion

Choosing satisfactory blocking techniques and blocking parameters is not easy and straightforward. There are good reasons for choosing large blocks, for instance to better exploit the memory hierarchy. There are also good reasons for choosing small blocks, for instance, to reduce the number of floating-point operations which are not covered by Level 3 routines, or to reduce the total number of operations in some algorithm, or simply to save memory.

These conflicting reasons lead to totally different optimum block sizes, so that a portable *and* efficient algorithm requires several levels of blocking using different block sizes.

References

[1] E. Anderson et al., LAPACK *Users' Guide, 2nd ed.*, SIAM Press, Philadelphia, 1995.

[2] C. H. Bischof, B. Lang, X. Sun, *Parallel Tridiagonalization through Two-Step Band Reduction*, Proceedings of the Scalable High-Performance Computing Conference, IEEE, Washington, D. C., 1994, pp. 23–27.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[4] E. F. D'Azevedo, J. J. Dongarra, *Packed Storage Extension for ScaLAPACK*, Technical report, LAPACK working note #135, 1998.

[5] J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Press, Philadelphia, 1991.

[6] J. J. Dongarra, S. J. Hammarling, D. C. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, J. Comput. Appl. Math. 27 (1989), pp. 215–227.

[7] J. J. Dongarra, S. J. Hammarling, D. C. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, Technical report, LAPACK working note #02, 1987.

[8] C. C. Douglas, M. Heroux, G. Slishman, R. M. Smith, GEMMW—*A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm*, J. Computational Physics 110 (1994), pp. 1–10.

[9] W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Concepts*, Technical Report to appear, Vienna University of Technology, 1998.

[10] W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Experiments*, Technical Report to appear, Vienna University of Technology, 1998.

[11] G. H. Golub, C. F. Van Loan, *Matrix Computations,* 3rd ed., Johns Hopkins University Press, Baltimore, 1996.

[12] E. Haunschmid, D. F. Kvasnicka, *High Performance Computing in Material Sciences. Maximizing Cache Utilization without Increasing Memory Requirements*, Technical Report to appear, Vienna University of Technology, 1998.

[13] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, 1997.

[14] N. J. Higham, *Exploiting Fast Matrix Multiplication within the Level 3* BLAS, ACM TOMS 16 (1990), pp. 352–368.

[15] N. J. Higham, *Stability of a Method for Multiplying Complex Matrices with Three Real Matrix Multiplications*, SIAM J. Matrix Anal. Appl. 13–3 (1992), pp. 681–687.

[16] D. F. Kvasnicka, *Parallel Packed Storage Scheme (P2S2) for Symmetric and Triangular Matrices*, Technical Report to appear, Vienna University of Technology, 1998.

[17] D. F. Kvasnicka, W. N. Gansterer, C. W. Ueberhuber, *A Level 3 Algorithm for the Symmetric Eigenproblem*, Proceedings of the Third International Meeting on Vector and Parallel Processing (VECPAR '98), Springer-Verlag, Berlin Heidelberg New York Tokio, 1998, to appear.

[18] M. S. Lam, E. E. Rothberg, M. E. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, Computer Architecture News 21 (1993), pp. 63–74.

[19] V. Strassen, *Gaussian Elimination Is not Optimal*, Numer. Math. 13 (1969), pp. 354–356.

[20] C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Heidelberg, 1997.