Reliable Computing 1 (2) (1995), pp. 189–199

Parallel accurate linear algebra subroutines

Jürgen Wolff von Gudenberg

In this paper we present a set of linear algebra subroutines which serve as building blocks for numerical software, and develop algorithms to implement these subroutines as a portable library for parallel computers. We consider these routines as a part of the standard arithmetic of a computer. Therefore they have to deliver a validated result of high accuracy.

Параллельные высокоточные линейно-алгебраические подпрограммы

Ю. Вольфф фон Гуденберг

Представлен набор линейно-алгебраических подпрограмм, которые могут служить строительными блоками для численного программного обеспечения, а также алгоритмы для реализации этих подпрограмм в виде переносимой библиотеки для параллельных компьютеров. Эти процедуры рассматриваются как часть стандартной арифметики компьютера, поэтому они должны возвращать проверенный результат высокой точности.

1. Basic linear algebra subprograms

Arithmetic operations for matrices and vectors play a vital role in the basic building blocks for numeric software. Therefore Basic Linear Algebra Subprograms (BLAS) have been designed for many languages, especially for FORTRAN. The construction of efficient, portable software is strongly supported by using these routines. The BLAS routines can be divided into three levels: vector/vector operations (Level 1), matrix/vector operations (Level 2), and matrix/matrix operations (Level 3).

Levels 2 and 3 are important for parallel and vector computers which are becoming more and more popular, and which are used for large scale numerical problems. As a parallel computer, we consider a loosely coupled network of processors with local memory. The parallel computer (or *processing network*) is driven by a host computer which distributes the data and controls the communication between the various processors. A fixed network topology, e.g. a grid, a tree, or a hypercube, is assumed. All processing elements execute the same code on different data. Communication is by message passing (and not by access to common memory). This concept is called SPMD (single program—multiple data) even if the code must be replicated to every processing element. This programming model lies between the strict SIMD mode (where every instruction is identical on all processors) and the MIMD mode (where each processor may perform different code).

We consider the following BLAS routines. In our description, α, β denote scalars, x, y denote vectors, A, B, C denote matrices of real numbers, and T denotes a lower triangular real matrix.

Level 1

Scalar multiplication with update

$$\begin{aligned} x &:= \beta x + \alpha y; \\ A &:= \beta A + \alpha B. \end{aligned}$$

Dot product with update

$$\alpha := \beta\alpha + x^T y.$$

For efficiency, the special case $\beta = 0$ may be implemented separately.

The first two routines compute scalar products of length 2, while the third one is a general scalar product with n or $n+1$ components.

Level 2

Matrix vector product with update

$$\begin{aligned} x &:= \beta x + \alpha A y; & x &:= \beta x + \alpha A^T y; \\ z &:= Tx; & z &:= T^T x. \end{aligned}$$

Solution of triangular system

$$x := T^{-1}x; \quad x := (T^{-1})^T x.$$

The matrix vector product computes dot products multiplied by α.

The solution of a triangular system may be implemented using dot products.

The rank-one update $A := \alpha x y^T + A$ is not considered in this paper, because its computation with 1 ulp (*u*nit in *l*ast *p*lace) accuracy is not important for the overall accuracy of an algorithm.

Level 3

Matrix products

$$\begin{aligned} C &:= \beta C + \alpha AB; & C &:= \beta C + x\alpha A^T B; \\ C &:= \beta C + \alpha AB^T; & C &:= \beta C + \alpha A^T B^T. \end{aligned}$$

Solution of triangular systems with multiple right-hand sides

$$B := T^{-1}B; \quad B := (T^{-1})^T B.$$

2. Accurate reliable arithmetic

We want not only to increase efficiency, but to increase accuracy as well. Since we consider the BLAS routines as a (secondary) standard, we require that they shall be as accurate as usual standard functions. In other words, our matrix multiplication, e.g. is as accurate as usual real multiplication, i.e. the error is less than 1 ulp for every matrix component.

An optimal scalar product algorithm suffices to compute all the products in which $\alpha \in \{-1, 0, 1\}$ with guaranteed accuracy of one ulp. This scalar product algorithm either uses a

long accumulator or the Bohlender algorithm [4]. In both cases, the exact scalar product can be represented as a sum of real numbers

$$s = \sum_{i=1}^{p} s_i$$

so the multiplication with α is just another scalar product. Details concerning overflow and efficient implementation are given in Section 2.2.

An algorithm for the solution of triangular systems with the same accuracy requirement is proposed in [3].

The final result of a complete algorithm, e.g., of the solution of a linear system, will either be computed with guaranteed high accuracy, or enclosed into a sharp interval. For this purpose, several additional BLAS routines are provided.

2.1. Additional subroutines

Level A

Generalized matrix-vector products

$$[z] := \beta x + \alpha \sum_{\nu=1}^{r} A_\nu \sum_{\mu=1}^{s} y_\mu;$$

$$[z] := \beta x + \alpha T \sum_{\mu=1}^{s} y_\mu.$$

Generalized matrix product

$$[D] := \beta C + \alpha \sum_{\nu=1}^{r} A_\nu \sum_{\mu=1}^{s} B_\mu.$$

Here, enclosures $[z], [D]$ are computed, and in the general case, $\alpha = \pm 1$ and $r = 1$.

Interval matrix-vector multiplication

$$[x] := [A][y].$$

Interval matrix multiplication

$$[C] := [A][B].$$

The solution of triangular systems with interval right hand side

$$[x] := T^{-1}[x]$$

is given in two versions: one for a relatively coarse enclosure, and one which computes the best possible interval vector.

2.2. Long and short sequential dot products

Usually optimally accurate dot products are calculated using a so-called long fixed-point accumulator, but for short vector length the Bohlender algorithm using addition with remainder may be more efficient.

The breakpoint between the two algorithms has to be calculated for each floating-point format.

If a long accumulator is used, the multiplication of a dot product by a floating-point number should be implemented directly without building an intermediate real vector representing the dot product. If the exponent of this product exceeds the exponent range of the accumulator, an overflow exception is appropriate, since the addition of one twofold product cannot adjust the final value of the operation in such a way that it would belong to the usual floating-point format. The underflow information (whether there are significant digits beyond the least accumulator digit) may be relevant for the final result and thus has to be kept.

3. Data distribution

3.1. Distribution patterns

The distribution of matrices or vectors crucially effects the performance of the algorithms. From the algorithmic point of view vectors may be distributed as contiguous segments or scattered cyclically, and matrices may be considered as vectors of rows or vectors of columns. Furthermore matrices may be distributed blockwise or each block may be scattered cyclically forming a grid pattern.

All these patterns can easily be archieved in modern data parallel languages like Modula 2* or High Performance Fortran. In Modula 2* [5] e.g., the allocators SPREAD or CYCLE are applicable for each dimension of an array. SBLOCK or CBLOCK combine various SPREAD or CYCLE allocators, respectively, and thus produce the block and grid pattern. LOCAL and RANDOM are further allocators.

In order to support portability a library shall be designed following the algorithmic distribution of data. An exhaustive library would contain a subroutine for each combination of input/output patterns. But in practive only a few patterns will give a sufficient speed-up. This, of course, also depends on the underlying hardware, its arithmetic performance, communication time, and network topology.

The BLAS routines generally gain time if nearest-neighbor communication is fast, thus favoring the linear segmentation; on the other hand, the parallel solution of a given problem may be better for scattered data. The trade-off between efficient BLAS routines using redistribution vs. the direct use of less efficient routines without communication has to be considered for the overall performance of an algorithm.

It will be an interesting topic to develop strategies for automatic support of data distribution.

3.2. Development of the algorithms

In the following we

1. First, dertermine the optimal distribution for each algorithm, and compute its possible speed-up.

2. Second, describe the communication method for distributing the centralized input data and for collecting the results. It turns out that we need a mainly homogeneous distribution of matrices by rows or columns, and broadcast of vectors and scalars. These routines have to be implemented for each parallel computer. Their performance depends on the hardware topology.

3. We then develop versions of the algorithms which exploit a reasonable initial distribution of data and also consider the required amount of available memory per processor, which determines the largest possible problem size and therefore, crucially limits the choice of the algorithm.

Notations and simplifying assumptions

- Dimension of vectors is n.
- Matrices are $n \times n$. Generalization to rectangular matrices is obvious.
- There are p^2 processors where $n = k \cdot p$.
- $\Theta_s(n)$ denotes the time for the calculation of a dot product of n components.
- Θ_A the time to add two long accumulators.
- Θ_{TR} the transport time, i.e., a time that it takes to transport a real number to an adjacent processor.

4. Parallel accurate BLAS routines

4.1. Level 1

a) $x := \beta x + \alpha y$; $A := \beta A + \alpha B$.

1) Any distribution where α is broadcast and each processor has the same amount of vector or matrix elements with corresponding subscripts is optimal. Afterwards the result follows the same pattern. The speed-up obviously is p^2.

2) Data distribution: Homogeneous partition.

3) A segmented or blockwise distribution of input and output data corresponds to the optimal distribution.

b) $\alpha := \beta\alpha + xy^T$.

1) In the optimal distribution, α is kept on one processor, and x and y are equally partitioned. Parallel partial dot products are computed on each processor. The resulting long accumulators are added in logarithmic time using the fan-in summation algorithm.

Due to the transport of long accumulators speed-up ratios of more than $p^2/2$ require relative long vectors and fast communication [6].

2) It generally does not pay to parallelize a scalar product for centralized data. We therefore sequentally compute all dot products in the level-2 and level-3 routines.

3) In case of a single dot product, if the vector length is large enough proceed like 1), else collect vector and compute sequential dot product.

4.2. Level 2

a) $x := \beta x + \alpha Ay$; $x := \beta x + \alpha A^T y$.

1) Row-wise distribution of A is optimal, if y, α, β are available on all processors. x is distributed by the same pattern as the rows of A. Exchange rows with columns for the second procedure.

- Compute first component of $A_\pi y$, where A_π denotes the π-th processor's partition of A.
- Multiply by α and update.
- Repeat these two steps for the other components.

Since all processors are working without communication or synchronisation the speed-up again is p^2.

2) The optimal distribution can be achieved by a homogeneous partition of A and x, and a broadcast of y, α, β.

3) Since the redundancy of storing one vector is not so high we suggest the optimal algorithm. Note that only one accumulator is needed on each processor.

b) $z := Tx$.

1) The $n(n+1)/2$ elements of a matrix T are distributed so that each processor has the same number of elements, but only complete rows. To achieve that, let us define k_π so that $k_0 = 0$ and

$$\sum_{i=k_\pi+1}^{k_{\pi+1}} i < \frac{n(n+1)}{2p^2} \leq \sum_{i=k_\pi+1}^{k_{\pi+1}+1} i.$$

Then each processor π obtains rows $k_{\pi-1}+1$ through k_π.

The first k_π elements of vector x are available on processor π. The result vector z is distributed like the rows of T.

Although the computation is independent, a speed-up of p^2 can not be reached, because the data can not be distributed equally (the assumption of equally distributable data was very optimistic in the other cases as well).

2) A nearly homogeneous partition of T goes along with a rather inhomogeneous partition and duplication of x. Different numbers of results are collected from different processors.

This inhomogenlity can be decreased, if the rows are distributed in a round robin manner, where the different runs start alternatively either from the end (i.e. from row n, which contains the larest number of elements), or from the beginning rows (which contain few elements).

3) We suggest to implement a version for a row-wise scattered matrix distribution, because this standard pattern is the best approximation for the optimal distribution. $z := T^T x$ analogously.

c) $x := T^{-1}x$; $x := (T^{-1})^T x$.

The solution of a triangular system cannot be implemented without communication, since the newly calculated solution components have to be used right after their computation.

The algorithm is a linear recurrence relation of order 1 to $n-1$.

Various techniques to parallelize the algorithm have been proposed. Reith [6] discusses 3 versions which apply the optimal dot product wherever possible: a cyclic form, a pipelined version of a cyclic form, and a form based on data broadcast.

If these routines yield maximum accuracy of the result, the intermediate solution vector can be kept in staggered correction form

$$x = \sum_{\mu=1}^{s} x_\mu$$

to simulate nearly s-fold working precision. For this purpose the following routines are called

$$\begin{aligned} [z] &:= x - \alpha T \textstyle\sum_{\mu-1}^{s} y_\mu; \\ [z] &:= T^{-1}[x]. \end{aligned}$$

They belong to the additional level A (see below). The first one is a generalization of $z := Tx$.

Since the second routine is used to enclose the global s-th defect of a solution, i.e. the defect of an s-fold multiple precision approximation, high accuracy is not necessary in this case.

4.3. Level 3

a) $C := \beta C + \alpha AB$.

1) If the matrices C and A are distributed row-wise, and the matrix B is broadcast to all processors, the result can be computed componentwise similar to the corresponding level 2 subroutine.

This, however, means an enormous amount of storage per processor and therefore, is not feasible.

A slightly better solution is to partition C and A row-wise, and to partition B columnwise in a divide-and-conquer style, and collect the results after the computation.

2) The optimal distribution requires the p-fold storage capacity. Each of p^2 processors obtains n^2/p elements of each matrix, and computes n^2/p elements of the result.

3) In the following, we discuss algorithms which only store the data once and therefore need communication during the computation. The algorithms are characterized by their internal communication topology. The algorithms have been analyzed setting $\beta = 0$ and $\alpha = 1$ [8].

i) Ring Algorithm.

- Distribute C and A by rows and B by columns, broadcast β and α. Each processor receives n/p^2 full rows of C, A and n/p^2 full columns of B.
- Repeat the following two steps p^2 times.
- Compute n^2/p^4 dot products to obtain the corresponding $(n/p^2) \times (n/p^2)$ block of the result matrix C.
- Rotate matrix B.

After the initial distribution this algorithm requires the following time:

$$\frac{n^2}{p^2} \cdot \Theta_s(n) + n^2 \Theta_{TR}.$$

This yields a real speed-up, if

$$\Theta_{TR} < 1 - \frac{1}{p^2}\Theta_s(n).$$

ii) Torus Algorithm.

- Distribute A, B and C blockwise, broadcast β and α. Each processor receives a $(n/p) \times (n/p)$ block of each matrix.
- Stagger the matrices A and B by appropriate row or column rotations, so that each processor can multiply its blocks.
- The product of the blocks of A and B now contributes to the resulting block of C. Blockwise rotations of A along rows and B along columns bring the next matching blocks together. After p rotations, the result is computed. This, however, requires n^2/p^2 long accumulators per processor, because we have to store all intermediate results with full length. Four versions of this algorithm are discussed in [9]. The one with the least memory requirements rotates parts of the blocks only, so that one dot product is completed per processor before the computation of the next one starts.

After the initial distribution, this algorithms requires the following time:

$$\frac{n^2}{p}\Theta_s\left(\frac{n}{p}\right) + \frac{n^2}{p}\Theta_A + 2\left(\frac{n^3}{p^2} + \frac{n^2}{p}\right)\Theta_{TR}.$$

Speed up is achieved if

$$\Theta_A + 2\frac{n}{p}\Theta_{TR} < p\Theta_s(n).$$

iii) Tree Algorithm.

This is a version of the divide-and-conquer algorithm which only needs storage for $2n + p^2 + 2$ elements on each processor.

- Distribute the first p^2 rows of A, the columns of B, and the upper left $p^2 \times p^2$ block of C.

- Compute the first new $p^2 \times p^2$ block of C by corresponding permutation of the columns of B.
- Store this part.
- Redistribute the next block of C and columns of B and repeat the computation until the first row of the result is complete.
- Repeat these steps for the next p^2 rows of A n/p^2 times.

The algorithm needs approximately

$$\frac{n^2}{p^2}\Theta_s(n) + 1.34n^3\Theta_{TR}$$

time for $\alpha = 1$ and $\beta = 0$. Only for very fast communication, i.e., when

$$1.34n\Theta_{TR} < 1 - \frac{1}{p^2}\Theta_s(n)$$

this will give a speed-up.

It is obvious that $\beta \neq 0$ only changes $\Theta_s(n)$ to $\Theta_s(n+1)$. If $\alpha \neq 1$, then one additional dot product of constant maximal length (depending only on the long accumulator format) is necessary for each component.

The other matrix products are treated similarly.

b) $B := T^{-1}B$; $B := (T^{-1})^T B$.

The corresponding algorithms for a single right hand side may be adapted.

4.4. Level A

The first three algorithms of this level are generalisations of level 2 or level 3 algorithms. The dot products are computed from multiple input matrices and vectors. They can however be arranged in such order that one long accumulator suffices. This accumulator which keeps the full information is then rounded twice. This only adds a constant time.

a) $[z] := \beta x + \alpha \sum_{\nu=1}^{r} A_\nu \sum_{\mu=1}^{s} y_\mu$.

Matrices A_ν and vectors y_μ are distributed as described for the simple matrix-vector product in 4.2.a). The exact dot product expression is determined and then rounded outwardly.

b) $[z] := Bx + \alpha T \sum_{\mu=1}^{s} y_\mu$.

This important special case can be implemented like $z := Tx$.

c) $[D] := \beta C + \alpha \sum_{\nu=1}^{r} A_\nu \sum_{\mu=1}^{s} B_\mu$.

Proceed similarly to the matrix product, but note that the resulting matrix is not an input argument.

d) $[x] := T^{-1}[x]$.

The solution of a triangular system with interval right hand side is an important operation which often is underestimated. Rump [7] reduces this problem to the determination of a validated lower bound for the smallest singular value σ of T. This can be achieved by Cholesky factorzation of $TT^T - \lambda^2 I$ where λ is an approximation of σ. The algorithm uses the generalized dot product expressions of the form c) for triangular matrices.

e) $[x] := \alpha[A][y]$ and $[C] := \alpha[A][B]$

are interval versions of the corresponding real functions. They use the interval dot product which computes the smallest interval containing all dot products of vectors arbitrarily taken out of the two input interval vectors. A short dot product with two summands is computed to determine the relevant bounds for each component. Then two real dot products are calculated with directed rounding. Although there is a little more work to do for each component, the parallelization of interval scalar products itself is not very promissing. We therefore suggest to proceed similarly to the real (non-interval) versions.

Updating versions with interval inputs always increase the width of the intervals and are therefore not feasible.

5. Summary

We have shown that an optimal dot product algorithm suffices to implement all the BLAS routines with the required accuracy. Practical experiences with parallel computers show that the trade-off to usual matrix multiplication is not so bad [2, 10], if floating-point arithmetic and dot product computation are equally supported by hardware.

References

[1] Albrecht, R., Alefeld, G., and Stetter, H. J. (eds) *Validation numerics.* Computing Suppl. **9** (1993).

[2] Bohlender, G. and Wolff von Gudenberg, J. *Accurate matrix multiplication on the array processor AMT–DAP.* In: Kaucher, E., Markov, S. M., and Mayer, G. (eds) "Computer Arithmetic, Scientific Computation and Mathematical Modelling", IMACS Annals on Computing and Applied Mathematics **12** (1992).

[3] Cordes, D. and Kaucher, E. *Self-validating computation for sparse matrix problems.* In: Kaucher, E., Kulisch, U., and Ullrich, C. (eds) "Computerarithmetic", Teubner, Stuttgart, 1987.

[4] Kulisch, U. and Miranker, W. L. *The arithmetic of the digital computer: a new approach.* SIAM Review **28** (1) (March 1986).

[5] Philipsen, M. and Tichy, W. *Compiling for massively parallel machines.* In: "Proc. of Code-Generation-Concepts, Tools, Techniques", Springer Series on Workshops in Computing, 1992.

[6] Reith, R. *Wissenschaftliches Rechnen auf Multicomputern—BLAS-Routinen und die Lösung linearer Gleichungssysteme mit Fehlerkontrolle.* Dissertation, Universität Basel, 1993.

[7] Rump, S. M. *Validated solution of large linear systems.* In: [1].

[8] Wolff von Gudenberg, J. *Modelling SIMD—type parallel arithmetic operations in Ada.* In: Christodoulakis, D. (ed.) "Ada: The Choice for '92", LNCS 499, Springer, Berlin, 1991.

[9] Wolff von Gudenberg, J. *Accurate matrix operations on hypercube computers.* In: Herzberger, J. and Atanassova, L. (eds) "Computer Arithmetic and Enclosure Methods", North-Holland, Amsterdam, 1992.

[10] Wolff von Gudenberg, J. *Implementation of accurate matrix multiplication on the CM−2.* In: [1].

Received: September 8, 1993
Revised version: September 25, 1994

Lehrstuhl für Informatik II
Universität Würzburg
Am Hubland
D-97074 Würzburg
Germany