Reliable Computing 1 (4) (1995), pp. 411–419

# Design of a parallel linear algebra library for verified computation

J. Wolff von Gudenberg

In this paper we discuss design principles to implement a set of linear algebra subroutines in a portable library for parallel computers. Our design supports reuse of code and easy adaption to new parallel programming paradigms or network configurations.

The routines are supposed to be used in self-verifying algorithms. They therefore have to deliver a validated result of high accuracy.

# Разработка параллельной линейно-алгебраической библиотеки для верифицируемых вычислений

Ю. Вольфф фон Гуденберг

Обсуждаются принципы построения и реализации набора линейно-алгебраических подпрограмм переносимой библиотеки для параллельных компьютеров. Поддерживается повторное использование кода. Подпрограммы могут легко адаптироваться к новым парадигмам параллельного программирования и сетевым конфигурациям.

Подпрограммы предназначены для использования в самоверифицирующих алгоритмах, поэтому они дают достоверные результаты высокой точности.

## 1. Introduction

Basic Linear Algebra Subprograms (BLAS) heavily support the construction of efficient, portable software for scientific computation. Usually three levels are provided, concerning vector/vector, matrix/vector or matrix/matrix operations, respectively. The routines cover generalized products, rank-one and rank-two updates as well as the solution of triangular systems.

In [10] we introduced a fourth level A which is mandatory for verifying computation and consists of:

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 products

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

Interval matrix-vector multiplication

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

Interval matrix multiplication

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

Solution of triangular systems with interval right hand side

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

Here $\alpha$ and $\beta$ denote scalars, $x$, $y_\mu$ vectors, $A, A_\nu, \ldots, D$ matrices and $T$ a triangular matrix. Intervals are indicated by brackets.

All these operations except for the solution of triangular systems can easily be performed with maximum accuracy, i.e. the width of the intervals is less than or equal to 1 ulp (unit last place) in case of point input data, if an updating optimal dotproduct algorithm with directed rounding is available. This also holds for the levels 1 through 3. The corresponding algorithms have been developed in [1, 7–9].

If we look at the typical structure of a self-verifying algorithm, we notice that it starts with a floating-point approximation and then an interval defect iteration with high accuracy is added for the verification. Hence our high accuracy requirements which considerably decrease the performance may seem too severe for the standard levels, since these routines are mainly used to compute approximations serving as input for level A procedures in the validation step. We therefore provide two versions of level 1, 2, 3—one approximative, and the other highly accurate. All level A routines, however, yield sharp, accurate, and guaranteed bounds for the true result.

In a parallel or distributed environment the speed-up increases with the ratio of computation over communication time, hence level 3 routines are preferred. In particular it is not reasonable to distribute a single dotproduct computation. Therefore the following basic arithmetic operations must be provided by each processor:

- interval operations;
- interval dotproduct;
- maximally accurate, accumulating dotproduct with directed rounding facility.

The last item means that a dotproduct computation can be divided into several steps, but the full information has to be kept in a data structure like a "long accumulator".

## 2. Data distribution

The distribution of matrices or vectors crucially effects the performance of the algorithms on parallel computers. 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.

The BLAS routines generally gain, 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. In table 1 we give an overview of the preferable distribution patterns for conventional BLAS routines [7, 10]. It is obvious that the generalisations of level A follow the same patterns.

| | $A/T$ | $B$ | $x$ | $y$ | $C$ |
|---|---|---|---|---|---|
| $x = \beta x + \alpha y$ | – | – | arb | arb | |
| $A = \beta A + \alpha B$ | arb | arb | not | not | |
| $\alpha = \beta\alpha + x^T y$ | – | – | not | not | |
| $x = \beta x + \alpha y$ | row wise | | segment | broadcast | |
| $x = Ty$ | row scattered | | scattered | broadcast | |
| $x = T^{-1}x$ | row scattered | | scattered | | |
| $C = \beta C + \alpha AB$ | row block | column block | row block | | row block |
| $B = T^{-1}B$ | | row | | | |

Here we do not allow matrices to be broadcast. A row-wise scattered distribution of data seems to be appropriate in nearly all cases, but note that the operations using the transposed matrix are not included which in turn favor a columnwise partition.

If rectangular blocks are allowed, the block scattered distribution is a generalisation of all those which are mentioned above. It turned out to be the most promising pattern for non-verifying algorithms [2].

For matrix multiplication, two algorithms are considered, one distributes the blocks consisting of full rows to a ring of processors, and the other deals with square blocks distributed to a toroidal structure of processors.

# 3. Design principles

A library always may be regarded as a link between a user program and the underlying hardware. Hence its design has to support the ease of use as well as the highly efficient implementation of the algorithms. Portability of programs between different platforms and even different structures of parallel or distributed computers shall be achieved by an appropriate library design. In this paper we describe a way to obtain an at least partly portable library itself.

The usual interface between a user program and a library is by call of a subprogram. The parameters may be treated as abstract data types, since components may be accessed differently in the user program and in the library. This means that we want to keep the representation of matrices or vectors secret for this level of (end) user interface.

By a slight extension of the dotprecision expressions known from the XSC languages [5, 6] BLAS routines may also be called as closed expressions thus enlarging the readability of the program considerably.

The top library layer implements the algorithms for the abstract data types using access to rows, columns or blocks of matrices as well as segments or slices of vectors. This layer still

makes no cogent difference between serial and parallel computers, but various versions for each algorithm regarding storing conventions or possible distribution should be provided in order to gain efficiency.

The implementation of the subarray access routines, however, depends on the actual distribution of the data. The abstract data types are instantiated to distributed data types which reflect the algorithmic structure. These data types provide access methods for their corresponding substructures, hence make obvious the communication structure of the algorithm and call for several communication primitives like broadcast or nearest-neighbor.

These routines and relations are finally implemented on top of the hardware.

So, over all, we have the following levels and interfaces:

1. comfortable user level
   - dotprecision expressions
2. user level
   - subroutine call
3. algorithmic layer
   - subarray access
4. distribution layer
   - communication primitives
5. hardware topology

The first two levels describe the user interface whereas the three bottom layers concern the design of the library.

Different design principles may be pursued and shall be compared with respect to

- easy portability
- maximal reuse of code
- high efficiency
- adaptibility to new structures
- necessary language support

## 3.1. Modular design

The transition between the two user levels clearly will be accomplished by a precompiler independent from the lower levels. Its specification will be described in a future paper.

Here we concentrate on the three library levels. By breaking the library design into levels, we opened a way to obtain a portability similar to that of user programs for the algorithmic layer. It usually can be written in high level languages like FORTRAN 90 or C++. For parallel SIMD or SPMD computers High Performance Fortran or C* may be used to gain efficiency.

Some dialects of C++ or related languages supporting message passing may serve to implement this layer for distributed computers.

Subarray access is thus either performed by a compiler or may otherwise be explicitly implemented in standard C according to the data distribution and with the help of the communication primitives which are in turn to be written in a low level language like Occam or C again.

Note, however, that we have mentioned at least three implementations of each algorithm, each using a different language, thus contradicting our goal of portability and easy maintenance.

We therefore try to use inheritance and other object oriented features to improve the reuse of the software.

## 3.2. Object-oriented design

The first approach is to define a hierarchy of matrix and vector structures, i.e. a matrix is a data structure with indexing operator, a row-matrix implements or refines a row access method, a scattered-row-matrix is a row-matrix with a specific distribution information and so on. Furthermore we may define a hierarchy of distribution topologies ranging from the general distinction of serial, SPMD or MIMD models over specific topologies, like hypercubes or rings, to explicitly stating the interconnection network.

The combination of these two hierarchies together with a parametrization with the component type leads to the following structure. We use C++ for the implementation, but here we only quote fragments of the code.

```
template <class Type> class collection
{ first(), last(), next(),  // iterate through collection
  Type value(), put_value (Type v),
  apply (f), reduce (f)
}

template <class Type> class Matrix : collection <Type>
{ int m,n;    // m x n matrix
  nextcol(),nextrow();   // advance to next row resp column
  Matrix (int m, int n);  // construct matrix
  ...
  // General matrix multiply
  void matmul Type alpha, Type beta, Matrix A, Matrix B)
  { //assert matrices match
    A.first(); B.first(); first();
    for (i=1; i<=m; i++)
      for (j=1; j<=B.n; j++)
      { dotprecision D (Type(0.0));
        for (k=1; k<=A.n; k++)
        { D.dotadd (A.value(),B.value());
          A.nextcol(); B.nextrow();
        }   //exact scalar product
        D.dotmul (alpha);
        D.dotadd(beta,value());
```

```
        put_value(D.round());        // component  [i,j]
        next();
      }

   }
...
}

class Node
{ static int p;   //number of processors
  int my_proc_no;
  send (Node to);
  receive (Node from);
...
};

template class Gridnode : Node
{ send_north(); ...
    receive_west();
}

template <class  Type> class DistMatrix :
 Matrix <Type>, Node
```

The library is a class library where matrices and vectors are descendants of a general collection type which provides abstract methods to iterate through all elements, set and retrieve their values, apply operations to all elements with or without combining the results. A similar approach has been described in [4], whereas [3] explicitly declares matrices and vectors. The BLAS routines can be formulated for the abstract types Matrix or Vector and the various specific versions are obtained by instantiating the general methods for different substructures. This design is extensible in the sense that a new distribution strategy can be introduced, and, if it contains specializations of the general matrix methods, the BLAS routines are automatically available.

We have separated the matrix operations from the distribution information. For the latter we defined a general class `Node` which provides necessary attributes for each processor in the network as well as abstract methods to send and receive data. Its subclass `Gridnode` describes the specialization for two-dimensional processor grids—the most relevant for matrix algorithms. The actual implementation of the communication routines is obtained by a call of an appropriate method from the actual hardware configuration which, in turn, is realized as a separate object.

The class `DistMatrix` thus inherits serial matrix operations from `Matrix` and communication primitives and distribution information from `Node`. Hence parallel algorithms can be implemented. Optimized versions for specific structures may be obtained by inheritance or instantiation. Thus a class `RingMatrix` which only uses the `send_east`, `receive_west` communication can be derived.

For the

```
template <class Type> class RingMatrix :
  DistMatrix <Type> , Ring
{
constructor performs scattered row distribution
into local matrices
...}
```

the following multiplication algorithm can be implemented:

- Distribute B by columns, broadcast $\beta$ and $\alpha$. Each processor receives $\frac{n}{p}$ full rows of C, A and $\frac{n}{p}$ full columns of B.
- Repeat the following two steps $p$ times.
- Compute $n^2/p^2$ dotproducts to obtain the corresponding $(\frac{n}{p} \times \frac{n}{p})$ block of the result matrix C.
- Rotate matrix B to east.

```
template <class Type>  void RingMatrix :: matmul
   (Type alpha, Type beta,RingMatrix A, RingMatrix B)
{ B.columnwise();
  broadcast (alpha); broadcast(beta);
  for (i=1; i<=p; i++)
  {  Matrix::matmul(alpha,beta,A,B);
     // serial matrix mult in each node updates local matrix
     B.send_east;
     B.receive_west;     // buffered rotation
  }
}
```

This design has the following advantages:

- Separation of algorithmic distribution and hardware configuration.
- Extensible for new distribution strategies.
- Distribution transparent for the user, but sophisticated users may use specialized versions.
- Encapsulation of hardware characteristics.
- Completely written in C++.
- Trade-offs between efficiency and portability may be solved by the user or system integrator.

# 4. Interface

The modular design provides a conventional library interface by call of subroutines. If all method calls are encapsulated in global functions, also the class library can be furnished with such a traditional interface even for C programmers [6]. But we want to exploit the extensibility and reuse facilities of the class library and therefore provide the full object-oriented interface.

The arithmetic datastructures for vectors and matrices are both derived from the abstract `collection` class. The class for triangular matrices which is derived from `Matrix` provides specialized `next` methods. All structures are parameterized with their component type which is supposed to support the usual arithmetic operations as well as a dotproduct.

For the level A routines interval vectors and matrices are obtained by instantiating each type with an interval template providing two attributes of component type and an interval rounding method.

```
template <class Type> class Interval
{ Type left,right;
  Interval (dotprecision<Type> C); }

template <class Type> class IntMatrix :
   Matrix<Interval<Type>>
```

A further step of inheritance can hide the complex structure of the library from the end user and provide "standard data types" Matrix and Vector.

For these types the usual matrix / vector operations are given as infix operators. The updating BLAS routines are provided as methods with the following interfaces. Note that because of overloading we do not have to distinguish all routines by their names.

| Class | Level | Name | Parameters |
|---|---|---|---|
| Vector | 1 | update | Type $\alpha, \beta$; Vector y |
| | 2 | matvecmul | Type $\alpha, \beta$; Matrix A; Vector y |
| | 2 | operator * | Triangular T; Vector y |
| | 2 | trisolve | Triangular T |
| Matrix | 1 | update | Type $\alpha, \beta$; Matrix B |
| | 3 | matmul | Type $\alpha, \beta$; Matrix A; Matrix B |
| | 3 | operator * | Triangular T; Matrix B |
| | 3 | trisolve | Triangular T |
| Vector <Interval> | A | matvecmul | interval $\alpha, \beta$; Matrix A[]; Vector y[], x |
| | A | trivecmul | interval $\alpha, \beta$; Triangular T; Vector y[] |
| | A | operator * | Matrix <Interval> A; Vector <Interval> y |
| | A | trisolve | Triangular T |
| Matrix <Interval> | A | matmul | interval $\alpha, \beta$; Matrix A[]; Matrix B[] |
| | A | operator * | Matrix <Interval> A, B |


# Acknowledgement

The library is currently being implemented for a workstation cluster under PVM.

# References

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

[2] Choi, J., Dongarra, J., and Walker, D. *LAPACK working note 57: PUMMA: parallel universal matrix multiplication algorithms on distributed memory concurrent computers.* University of Tenessee, TR CS–93–187, 1993.

[3] Dongarra, J., Pozo, R., and Walker, D. *LAPACK working note 61: an object oriented design for high performance linear algebra on distributed memory architectures.* University of Tenessee, TR CS–93–200, 1993.

[4] Jézéquel, J. M., Bergheul, F., and André, F. *Programming massively parallel arcitectures with sequential object oriented languages.* FGCS **10** (1) (1994), pp. 59–70.

[5] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch. *PASCAL–XSC—language reference with examples.* Springer, Berlin, 1992.

[6] Klatte, R., Kulisch, U., Lawo, C., Rauch, M., and Wiethoff, A. *C–XSC, a C++ class library for extended scientific computing.* Springer, Berlin, 1993.

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

[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. *Parallel accurate linear algebra subroutines.* Reliable Computing **1** (2) (1995), pp. 189–199.

Received: June 7, 1994
Revised version: November 10, 1994

Lehrstuhl für Informatik II
Universität Würzburg
Am Hubland
D–97074 Würzburg
Germany
E-mail: wolff@informatik.uni-wuerzburg.de