Department of Mathematics
Faculty of Mechanical Engineering
Slovak University of Technology in Bratislava

4th International Conference
APLIMAT 2005

PLENARY LECTURE

NUMERICAL SOFTWARE FOR SOLVING DENSE LINEAR ALGEBRA PROBLEMS ON HIGH PERFORMANCE COMPUTERS

STPICZYŃSKI Przemysław, (PL), PAPRZYCKI Marcin, (USA, PL)

Abstract. In the paper, fundamentals of numerical software for solving dense linear algebra problems on modern parallel computers are presented. It is intended for readers who are familiar with the general aspects of computing but are new to high performance and parallel computing as it provides a general overview of the field.

1 Introduction

Over the past twenty years, parallel computing has entered the mainstream of computations. While multi-processor machines were once a marvel of technology, today most personal computers arrive with multiple processing units such as the main processor, video processor etc., each with computational power and memory substantially larger than those of the top of the line workstations of just some years ago. More importantly, the increased availability of inexpensive components has made "desktop" systems with two and four processors available for the price ranging from $5,000 to $20,000. Furthermore, recent announcements by both AMD and Intel showed clearly, that the "speed race" consisting of increasing clock speed of individual processors is practically over. Instead, both computer vendors are to offer slower but more powerful dual-core designs and systematically introduce 64-bit processors. This means that, starting in 2005 a typical desktop computer will arrive as a dual-processor "parallel computer" and soon all computers available on the market will be able to run quadruple precision arithmetic. Some claim that this latter development will be necessary to solve really large problems [10]. Finally, we observe continuing progress in development of blade servers, that allow easy and storage efficient assembly of multi-processor systems.

The question thus arrives, why the push towards parallel computing ? The answer is simple: because of a **need to solve large problems** from many areas of science. Some examples

include Earth environment prediction, nuclear weapons testing, quantum chemistry, computational biology, data mining for large and very large datasets, astronomy and cosmology, cryptography, approximate algorithms for $\mathcal{NP}$-complete problems. These problems cannot be solved on single processor machines either due to the limitation of available processor power or insufficiency of available memory.

2 Parallel computers and parallel programming

There exist a number of parallel computer architectures. An old, but still useful, general computer taxonomy was introduced by Flynn in 1966 [18]. By considering the fact that information processing, which takes place inside of a computer, can be conceptualized in terms of interactions between data streams and instructions streams, Flynn classified all existing computer architectures into four types:

- SISD – single instruction stream/single data stream, which includes most of standard von Neumann type computers,
- MISD – multiple instruction streams/single data stream, which describes various special computers but no particular class of machines,
- SIMD – single instruction stream/multiple data streams, the architecture of parallel processor "arrays", and
- MIMD – multiple instruction streams/multiple data streams, which includes most modern parallel computers.

The last two architectures, SIMD and MIMD, are at the center of our interest since most parallel systems that ever existed fell into one of these categories. SIMD-based computers were characterized by a relatively large number of relatively weak processors, each associated with a relatively small memory. These processors were combined into a matrix-like topology, hence the popular name of this category: "processor arrays". This computational matrix was connected to a controller unit (usually a top-of-the-line workstation), where program compilation and array processing management took place. While popular in the early days of parallel computing, they turned out to be much too inflexible to handle general-purpose parallel algorithms. Currently, the concept has migrated "inside" the existing processors. For instance, the MMX extension of the command set of an x86 processor included array processing type instructions.

It can be therefore said, that all existing general purpose parallel computers belong to the MIMD category. They can be divided into two sub-categories: shared memory and distributed memory systems. This division is based on the way that the memory and the processors are interconnected. Shared memory computers consist of a number of processors that are connected to the main (global) memory. The memory connection is facilitated by a bus or a variety of switches (e.g. omega, butterfly, etc.) Distributed memory computers are composed of computational nodes consisting of a processor and a large local memory (there is no global memory) and are interconnected through a structured network. Typical topologies for interprocessor connectivity are meshes, toruses and hypercubes. While code for distributed memory computers is relatively more difficult to write and debug, this architecture can be scaled to a large number of processors. Thus, all of the most powerful computers are currently designed in this way [9], however, their low-level blocks are often shared memory. For instance, the

Earth Simulator, currently the most powerful computer in the world, is build out of shared memory nodes consisting of 8 processors each.

While the largest computers are custom built for the highest performance, they cost tens of millions of dollars as well. Clusters made high performance parallel computing available to those with much smaller budgets. The idea is to combine commodity-off-the-shelf (COTS) components to create a parallel computer. The Beowulf project was the forerunner in this approach [1].

Currently, shared memory computers are usually programmed using OpenMP, which uses the fork-join model of parallel execution. A program starts execution as a single process, called the *master thread* of execution, and executes sequentially until the first parallel construct is encountered. Then the master thread spawns a specified number of "slave" threads and becomes a "master" of the team. All statements enclosed by the parallel construct are executed in parallel by each member of the team. Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct (e.g. shared, private, reduction, etc.).

Parallel Virtual Machine (PVM) has been developed in 1991 by a group of researchers at the University of Tennessee. It is a distributed-memory tool [16] designed to implement parallel applications on distributed memory parallel computers, as well as networks of heterogeneous computers (treated as a single computational resource). *Message Passing Interface* (MPI) has been developed in 1993 [26] by researchers from Argonne National Laboratory. Over time, it has become a de-facto standard for message passing parallel computing (superseding PVM, which is slowly becoming extinct). MPI provides an extensive set of communication subroutines including point-to-point communication, broadcasting and collective communication. It has been implemented on a variety of parallel computers including massively parallel computers, clusters and networks of workstations. Due to its popularity, a number of open source and commercial tools and environments have been developed to support MPI based parallel computing [20].

3 Models for parallel processing

The processing speed of computers involved in scientific calculations is usually expressed in terms of a number of floating point operations completed per second, a measure used also to describe the computational power of the world's largest supercomputers [9]. For a long time, the basic measure was Mflops (10^6 operations) expressed as:

$$r = \frac{N}{t} \quad \text{Mflops}, \tag{1}$$

where N represents a number of floating point operations executed in t microseconds. Obviously, when N floating point operations are executed with an average speed of r Mflops, the execution time of a given algorithm can be expressed as:

$$t = \frac{N}{r}. \tag{2}$$

Due to the increase in computational power of computers Gflops (10^9) and Tflops (10^{12} floating point operations per second) replaced Mflops.

One of the most important methods of analyzing the potential for parallelization of any algorithm is to observe how the algorithm can be divided into parts that can be executed in parallel and into those that have to be executed sequentially. More generally, different parts

of any algorithm are executed with different speeds and use different resources of a computer to a different extent. Therefore, it would be naive to predict the algorithm's performance by dividing the total number of operations by the average speed of the computer. To find a quality performance estimate, one should separate all parts of the algorithm that utilize the underlying computer hardware to a different extent. Significant initial work in this area was done by Amdahl [14, 15]. He established how slower parts of an algorithm influence its overall performance. Assuming that a given program consists of N floating point operations, out of which a fraction f is executed with a speed of V Mflops, while the remaining part of the algorithm is executed with a speed of S Mflops, and assuming further that the speed V is close to the peak performance while the speed S is substantially slower ($V \gg S$), then the total execution time can be expressed using the following formula:

$$t = f\frac{N}{V} + (1-f)\frac{N}{S} = N(\frac{f}{V} + \frac{1-f}{S}) \tag{3}$$

which can be used to establish the total execution speed of the algorithm as

$$r = \frac{N}{t} = \frac{1}{\frac{f}{V} + \frac{(1-f)}{S}} \text{ Mflops.} \tag{4}$$

From formula (3), follows that $t > \frac{(1-f)N}{S}$. If the whole program is executed at the slower speed S, its execution time can be expressed as $t_s = \frac{N}{S}$. If the execution speed of the part f of the program can be increased to V then the performance gain can be represented as

$$\frac{t_s}{t_v} < \frac{N}{S} \cdot \frac{S}{N(1-f)} = \frac{1}{1-f}. \tag{5}$$

In the case of parallel processing, let us assume that fraction f of an algorithm can be divided into p parts and ideally parallelized (executed at exactly the same time t_1/p on p processors), the remaining $1-f$ of operations cannot be parallelized and thus have to be executed on a single processor. The total execution time of this algorithm on p processors can be expressed as:

$$t_p = f\frac{t_1}{p} + (1-f)t_1 = \frac{t_1(f + (1-f)p)}{p}.$$

Therefore the speedup s_p is equal to

$$s_p = \frac{p}{f + (1-f)p}. \tag{6}$$

Obviously, $f < 1$, and therefore the following inequality is true

$$s_p < \frac{1}{1-f}. \tag{7}$$

It should be noted that there exist other interesting models such as the Hockney-Jesshope model of vector processing [23, 31] and Gustafson's model [21] of parallel computing. To predict the behavior of distributed algorithms, one may consider the Bulk Synchronous Parallel Architecture (BSP for short [22]). In this model, a parallel program consists of a number of supersteps. Each superstep comprises local computations, global data exchange and the barrier synchronization. The BSP model is characterized by the following parameters: p, the number of available processors, g, the time (in flop time units) it takes to communicate

(send or receive) a data element and l, the time (in flop time units) it takes all processors to synchronize. The complexity of a superstep is defined as

$$w_{\max} + gh_{\max} + l \tag{8}$$

where $w_{\max}$ is the maximum number of flops performed, and $h_{\max}$ is the number of messages sent or received by any one processor during this superstep. An interesting application of this model can be found in [28], where we improved our earlier algorithm [30, 29] for solving linear recurrence systems with constant coefficients of the following form

$$x_k = \begin{cases} 0 & \text{for } k \leq 0 \\ f_k + \sum_{j=1}^{m} a_j x_{k-j} & \text{for } 1 \leq k \leq n, \end{cases} \tag{9}$$

which is the central part of many numerical algorithms for instance evaluation of orthogonal polynomials [5]. The algorithm is an example of the distributed *divide and conquer* approach for solving linear algebra problems in parallel and using the matrix-vector and matrix-matrix multiplication and its performance is much better than the performance of the simple scalar algorithm based on (9). Moreover, its complexity can be expressed in terms of the BSP parameters and the "internal" parameters of the method, namely the block sizes. In [28] we show that the optimal choices do not depend on the BSP parameters, thus the algorithm is "machine independent" and the choices are valid for all BSP architectures regardless the specific values of the BSP parameters.

4 Shared memory parallel computing: BLAS and LAPACK

Let us now consider the focal point of this paper: writing efficient software to solve linear algebraic problems on parallel computers. Over last 25 years a standard has evolved for doing exactly this. More precisely, there exists a collection of interdependent software libraries that became the standard tool for high performance dense and banded linear software implementation.

The first step in the general direction took place in 1979, when the BLAS (Basic Linear Algebra Subroutines) standard was proposed [25]. Researchers realized that linear algebra software (primarily for dense matrices) consists of a number of basic operations (e.g. vector scaling, vector addition, dot product, etc.) that appear in most algorithms. These fundamental operations have been defined as a collection of Fortran 77 subroutines. The next two steps took place in 1988 and 1990, respectively, when collections of matrix-vector and matrix-matrix operations have been defined [13, 12] (also as libraries of Fortran 77 kernels). These two developments can be traced to the hardware changes occurring at this time. The introduction of hierarchical memory resulted in need of algorithms that would support data locality (i.e. move a *block* of data once, perform all the necessary operations on it and move it back to the main memory and only then proceed with the next block of data). It was established that to achieve this goal one should rewrite linear algebra codes in terms of block operations and such operations can be naturally represented in terms of matrix-vector and matrix-matrix operations.

BLAS routines were used in development of linear algebra libraries that solved a number of standard problems. Level 1 BLAS (vector oriented operations) was used in development of the LINPACK [11] and EISPACK [19] libraries dedicated to the solution of linear algebraic systems and eigenproblems. The main advantages of these libraries were: clarity and readability of

the code, its portability as well as the possibility of hardware oriented optimization of BLAS kernels.

The next step was the development of the LAPACK library [2], which "combined" the functionalities available in the LINPACK and EISPACK libraries. LAPACK utilized level 3 BLAS kernels, while the BLAS 2 and 1 routines were used only when necessary. It was primarily oriented toward single processor high performance computers with vector processors (e.g. Cray, Convex, Fujitsu, NEC) or with hierarchical memory (e.g. SGI Origin, HP Exemplar, DEC Alpha workstations etc.). LAPACK was also designed to work well on shared memory parallel computers, providing parallelization inside the level 3 BLAS routines [15]. Unfortunately, while this performance was very good for the solution of dense symmetric and non-symmetric linear systems (this was also the data used at many conferences to illustrate the success of the approach), the performance of eigenproblem solvers (for both single processor and parallel machines) was highly dependent on the quality of the underlying BLAS implementation and very unsatisfactory in many cases [4, 3].

The BLAS operations have the following form (detailed description of BLAS routines can be found in [2, 14]):
Level 1: vector-vector operations:

- $y \leftarrow \alpha x + y$, $x \leftarrow \alpha x$, $y \leftarrow x$, $y \leftrightarrow x$, $dot \leftarrow x^T y$, $nrm2 \leftarrow \|x\|_2$, $asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$.

Level 2: matrix-vector operations:

- matrix-vector products: $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$
- rank-1 update of a general matrix: $A \leftarrow \alpha x y^T + A$
- rank-1 and rank-2 update of a symmetric matrix: $A \leftarrow \alpha x x^T + A$, $A \leftarrow \alpha x y^T + \alpha y x^T + A$,
- multiplication by a triangular matrix: $x \leftarrow Tx$, $x \leftarrow T^T x$,
- solving a triangular system of equations: $x \leftarrow T^{-1} x$, $x \leftarrow T^{-T} x$.

Level 3: matrix-matrix operations:

- matrix-matrix products: $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha A^T B + \beta C$, $C \leftarrow \alpha AB^T + \beta C$, $C \leftarrow \alpha A^T B^T + \beta C$
- rank-k and rank-$2k$ update of a symmetric matrix: $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$, $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$,
- multiplication by a triangular matrix: $B \leftarrow \alpha TB$, $B \leftarrow \alpha T^T B$, $B \leftarrow \alpha BT$, $B \leftarrow \alpha BT^T$,
- solving a triangular system of equations: $B \leftarrow \alpha T^{-1} B$, $B \leftarrow \alpha T^{-T} B$, $B \leftarrow \alpha BT^{-1}$, $B \leftarrow \alpha BT^{-T}$.

Now, to illustrate the way that BLAS can be applied in implementation of linear algebraic algorithms, let us consider the matrix-matrix multiplication using various levels of BLAS. It is clear that it can be implemented using a single call to the appropriate routine from the Level 3 BLAS, namely

$$C \leftarrow \alpha AB + \beta C. \tag{10}$$

BLAS	loads and stores	flops	ratio
$y \leftarrow y + \alpha x$	$3n$	$2n$	$3:2$
$y \leftarrow \alpha Ax + \beta y$	$mn + n + 2m$	$2m + 2mn$	$1:2$
$C \leftarrow \alpha AB + \beta C$	$2mn + mk + kn$	$2mkn + 2mn$	$2:n$

Table 1: BLAS: memory references and arithmetic operations [14]

	Mflops	sec.
BLAS 1	1162.79	1.72
BLAS 2	1418.43	1.40
BLAS 3	7692.30	0.26

Table 2: Matrix multiplication on the Pentium IV 3GHz

The above operation can be also expressed in terms of matrix-vector multiplications

$$y \leftarrow \alpha Ax + \beta y \tag{11}$$

from the Level 2 BLAS, namely

$$C_{*k} \leftarrow \alpha AB_{*k} + \beta C_{*k}, \quad \text{for } k = 1, ..., n \tag{12}$$

Finally, observe that the operation (11) can be conceptualized in terms of a sequence of dot-products (routine _DOT), vector scalings (_SCAL) or vector updates (_AXPY) from Level 1 BLAS:

$$\begin{array}{ll} z_k \leftarrow A_{k*}x, \text{ for } k = 1, ..., m & (\text{\_DOT}) \\ y \leftarrow \beta y & (\text{\_SCAL}) \\ y \leftarrow y + \alpha z & (\text{\_AXPY}) \end{array} \tag{13}$$

It should be noted that replacing level 1 BLAS operations by level 2 BLAS, the application of (10) instead of (12), while introducing no changes to the total number of arithmetical operations performed, reduces the total amount of processor-memory communication. More precisely, to illustrate the advantages of the application of higher level BLAS, consider the total number of arithmetical operations and the amount of data exchanged between the processor and memory. Table 1 [14] depicts the ratio of the number of processor-memory communications to the number of arithmetical operations for $m = n = k$.

The higher the level of BLAS, the more favorable the ratio becomes. (The number of operations performed on data increases relative to the total amount of data movement.) This has a particularly positive effect in the case of hierarchical memory computers. To illustrate that this course of action plays an important role not only for "supercomputers" but also for more "ordinary" architectures, Table 2 presents processing speed (in Mflops) achieved during the completion of the task $C \leftarrow \alpha AB + \beta C$ using BLAS 1, 2 and 3 kernels (utilizing approaches (13), (12) and (10)) for $m = n = k = 1000$ on a single-processor PC with Intel Pentium IV 3GHz processor and 2GB of RAM.

There are two ways of using BLAS routines in parallel computing. First, very often, BLAS routines are parallelized by the computer hardware vendors. For instance, a call to the level 3 BLAS routine _GEMM may result in parallel execution of matrix-matrix multiplication, and subsequently, any code that utilizes _GEMM would automatically perform this operation in parallel. Vendor provided parallelization can be extremely efficient. For instance on the

Cray, selected BLAS kernels have been written in the Cray Assembly Language and resulted in efficiency of over 90% on an 8-processor machine. It should be stressed, however, that only some of BLAS kernels are parallelized (one of the typical and very important exceptions are routines for symmetric matrices stored in a compact form) and they are typically parallelized for shared memory environments only. In short, parallel performance of BLAS routines cannot be taken for granted (especially since historically they have been primarily optimized for single processor performance in the hierarchical memory environment). Taking this into account, BLAS kernels should be rather used to implement parallel algorithms in such a way that instances of BLAS routines run on separate processors. To illustrate the main idea behind such an approach, consider matrix update procedure [27] based on the formula

$$C \leftarrow \alpha AB + \beta C,$$

which can be rewritten as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{14}$$

Applying the definition of matrix multiplication, the following block algorithm to calculate matrix C is obtained.

$$\begin{array}{ll} C_{11} \leftarrow \alpha A_{11}B_{11} + \beta C_{11} & /1/ \\ C_{11} \leftarrow \alpha A_{12}B_{21} + C_{11} & /2/ \\ C_{12} \leftarrow \alpha A_{11}B_{12} + \beta C_{12} & /3/ \\ C_{12} \leftarrow \alpha A_{12}B_{22} + C_{11} & /4/ \\ C_{21} \leftarrow \alpha A_{22}B_{21} + \beta C_{21} & /5/ \\ C_{21} \leftarrow \alpha A_{21}B_{11} + C_{21} & /6/ \\ C_{22} \leftarrow \alpha A_{22}B_{22} + \beta C_{22} & /7/ \\ C_{22} \leftarrow \alpha A_{21}B_{12} + C_{22} & /8/ \end{array} \tag{15}$$

Observe that this algorithm allows for parallel execution of operations /1/, /3/, /5/, /7/ and in the next phase of operations /2/, /4/, /6/, /8/. Obviously, it is desirable to divide large matrices into a larger number of blocks (e.g. to match the number of available processors). The order of operations may also need to be adjusted to reduce the memory access conflicts. An application of this approach can be illustrated by the block-Cholesky method for solving systems of linear equations for symmetric positive definite matrices. It is well known that there exists a unique decomposition for such matrices

$$A = LL^T, \tag{16}$$

where L is a lower triangular matrix. There exists also a simple algorithm for determining the matrix L with an arithmetical complexity of $O(n^3)$. Its analysis allows one to see immediately that it can be expressed in terms of calls to the level 1 BLAS. However, it can be also translated into block operations expressed in terms of level 3 BLAS. Formula (16) can be rewritten in the following way:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix} \tag{17}$$

Thus

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

After the decomposition $A_{11} = L_{11}L_{11}^T$, which is the same decomposition as the original one but of smaller size, appropriate BLAS 3 kernels can be applied in parallel to calculate matrices L_{21} and L_{31} by applying in parallel equalities $A_{21} = L_{21}L_{11}^T$ and $A_{31} = L_{31}L_{11}^T$. In the next step, equation

$$A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T,$$

can be used. Thus the decomposition $L_{22}L_{22}^T$ for the matrix $A_{22} - L_{21}L_{21}^T$ is used to calculate the matrix L_{22}. Finally, L_{32} can be found from

$$L_{32} = (A_{32} - L_{31}L_{21}^T)(L_{22}^T)^{-1}.$$

In a similar way, subsequent block columns of the decomposition can be calculated. Finally, it should be noted that the parallelization of the matrix multiplication presented here as well as the Cholesky decomposition are examples of the divide-and-conquer method, which is one of the popular approaches to algorithm parallelization [14].

5 Distributed memory scalable computing: BLACS, PBLAS and ScaLAPACK

At the time when the LAPACK project was completed, it became clear that similar software to solve linear algebraic problems on distributed memory architectures should also be developed. Obviously, this could have been done "by hand" using level 3 BLAS kernels and an inter-processor communication library like PVM or MPI. However, this would have made such an approach dependent on their continuous existence and strict backward compatibility. Since PVM is already slowly disappearing, while imposing strict backward compatibility on MPI may be holding it to too high a standard, the decision not to follow this path seems to be very good indeed. It has led in the first place to the development of the BLACS (Basic Linear Algebra Communication Subroutines), a package that defines portable and machine independent collection of communication subroutines for distributed memory linear algebra operations [17, 32]. The essential goals of BLACS are:

- simplifying message passing in order to reduce programming errors,
- providing data structures to simplify at the level of matrices and their subblocks,
- portability across a wide range of parallel computers, including all distributed memory parallel machines and heterogenous clusters.

In BLACS, each process is treated as if it were a processor – it must exists for the life-time of BLACS run and its execution can affect other processes only through use of message passing. Processes involved in BLACS execution are organized in two-dimensional grids and each process is identified by its coordinates in the grid. For example, if a group of consists N_p processes then the grid will have P rows and Q columns, where $P \cdot Q = N_g \leq N_p$. A process can be referenced by its coordinates (p, q), where $0 \leq p < P$ and $0 \leq q < Q$.

BLACS provide structured communication in the process-grid. Processes can communicate using point-to-point paradigm or it is possible to organize communication (broadcasts) within a "scope" which can be a row or a column of a grid, or even the whole grid. Moreover, the performance of communication can be improved by indicating underlying hardware topology [17].

The PBLAS (Parallel Basic Linear Algebra Communication Subprograms) [7] is a set of distributed vector-vector, matrix-vector and matrix-matrix operations (analogous to the

sequential BLAS) with the aim of simplifying the parallelization of linear algebra programs. The basic idea of PBLAS is to distribute matrices among distributed processors (i.e. BLACS processes) and utilize BLACS as the communication infrastructure. The general class of such distributions can be obtained by matrix partitioning like

$$A = \begin{pmatrix} A_{11} & \dots & A_{1m} \\ \vdots & & \vdots \\ A_{m1} & \dots & A_{mm} \end{pmatrix},$$

where each subblock A_{ij} is $n_b \times n_b$. These blocks are mapped to processes by assigning A_{ij} to the process whose coordinates in a grid are

$$((i-1) \mathrm{mod} P, (j-1) \mathrm{mod} Q).$$

Finally, ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of heterogeneous computers [6]. It provides the same functionality as LAPACK for workstations, vector supercomputers, and shared-memory parallel computers. As LAPACK was developed by utilizing calls to the BLAS routines, ScaLAPACK is based on calls to the BLACS and PBLAS kernels. It should be pointed out that there exists a BSP version of ScaLAPACK based on BSP communication routines instead of BLACS, which is faster than the original ScaLAPACK [24].

It is worth mentioning that as of writing of this paper (October, 2004), new releases of LAPACK and ScaLAPACK are planned [8] and expected to contain substantial re-working of the computational kernels to improve their numerical properties as well as parallel performance. This illustrates that the area of development of efficient linear algebraic algorithms for high performance parallel computers remains an important research area and should stay this way for years to come.

References

[1] The Beowulf Project. http://www.beowulf.org.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTRUCHOV, AND D. SORENSEN. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.

[3] I. BAR-ON AND M. PAPRZYCKI. A fast solver for the complex symmetric eigenproblem. *Computer Assisted Mechanics and Engineering Sciences*, 5:85–92, 1998.

[4] I. BAR-ON AND M. PAPRZYCKI. High performance solution of complex symmetric eigenproblem. *Numerical Algorithms*, 18:195–208, 1998.

[5] R. BARIO, B. MELENDO, and S. SERRANO. On the numerical evaluation of linear recurrences. *J. Comput. Appl. Math.*, 150:71–86, 2003.

[6] L. BLACKFORD et al. *ScaLAPACK User's Guide*. SIAM, Philadelphia, 1997.

[7] J. CHOI, J. DONGARRA, S. OSTROUCHOV, A. PETITET, D. WALKER, and R. WHALEY. LAPACK working note 100: A proposal for a set of parallel basic linear algebra subprograms. http://www.netlib.org/lapack/lawns, May 1995.

[8] J. DEMMEL AND J. DONGARRA. St-hec: Reliable and scalable software for linear algebra computations on high end computers. http://www.cs.berkeley.edu/ demmel/ScaLAPACK-Proposal.pdf.

[9] J. DONGARRA. Performance of various computer using standard linear algebra software. http://www.netlib.org/benchmark/performance.ps.

[10] J. DONGARRA. personal communication.
[11] J. DONGARRA, J. BUNSCH, C. MOLER, AND G. STEWARD. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.
[12] J. DONGARRA, J. DUCROZ, I. DUFF, AND S. HAMMARLING. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.
[13] J. DONGARRA, J. DUCROZ, S. HAMMARLING, AND R. HANSON. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14:1–17, 1988.
[14] J. DONGARRA, I. DUFF, D. SORENSEN, AND H. VAN DER VORST. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.
[15] J. DONGARRA, I. DUFF, D. SORENSEN, AND H. VAN DER VORST. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, 1998.
[16] J. DONGARRA et al. *PVM: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994.
[17] J. J. DONGARRA and R. C. WHALEY. LAPACK working note 94: A user's guide to the BLACS v1.1. http://www.netlib.org/blacs, May 1997.
[18] M. FLYNN. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C–21:94, 1972.
[19] B. GARBOW, J. BOYLE, J. DONGARRA, and C. MOLER. *Matrix Eiigensystems Routines – EISPACK Guide Extension*. Lecture Notes in Computer Science. Springer-Verlag, New York, 1977.
[20] W. GROPP and E. LUSK. *A User's Guide for mpich, a Portable Implementation of MPI version 1.2.0*.
[21] J. GUSTAFSON, G. MONTRY, and R. BENNER. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9:609–638, 1988.
[22] J. HILL, S. DONALDSON, and D. SKILLICORN. Portability of Performance with the *BSPlib* Communications Library. In *Programming Models for Massively Parallel Computers, (MPPM'97)*, London, Nov. 1997. IEEE Computer Society Press.
[23] R. HOCKNEY and C. JESSHOPE. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger Ltd., Bristol, 1981.
[24] G. HORVITZ and R. H. BISSELING. Designing a BSP version of ScaLAPACK. In B. Hendrickson et al., editors, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1999. SIAM.
[25] C. LAWSON, R. HANSON, D. KINCAID, and F. KROGH. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–329, 1979.
[26] P. PACHECO. *Parallel Programming with MPI*. Morgan Kaufmann, San Frncisco, 1996.
[27] M. PAPRZYCKI and P. STPICZYŃSKI. A brief introduction to parallel computing. In E. Kontoghiorghes, editor, *Parallel Computing and Statistics*. Marcel Dekker, 2004. (to appear).
[28] P. STPICZYŃSKI. Numerical evaluation of linear recurrences on high performance computers and clusters of workstations. In *Proceedings of PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering*, pages 200–205. IEEE Computer Society Press, 2004.
[29] P. STPICZYŃSKI. Numerical evaluation of linear recurrences on various parallel computers. In M. Kovacova, editor, *Proceedings of Aplimat 2004, 3rd International Conference, Bratislava, Slovakia, February 4–6, 2004*, pages 889–894. Technical Univerity of Bratislava, 2004.

[30] P. STPICZYŃSKI. Solving linear recurrence systems using level 2 and 3 BLAS routines. *Lecture Notes in Computer Science*, 3019:1059–1066, 2004.

[31] P. STPICZYŃSKI and M. PAPRZYCKI. Fully vectorized solver for linear recurrence systems with constant coefficients. In *Proceedings of VECPAR 2000 – 4th International Meeting on Vector and Parallel Processing, Porto, June 2000*, pages 541–551. Facultade de Engerharia do Universidade do Porto, 2000.

[32] R. C. WHALEY. LAPACK working note 73: Basic linear communication algebra subprograms: Analysis and implementation across multiple parallel architectures. http://www.netlib.org/blacs, June 1994.

Current address

Marcin Paprzycki, Computer Science Department, Oklahoma State University, Tulsa, OK 74106, USA and Computer Science Faculty, SWPS, ul. Chodakowska 19/31, 03-815 Warszawa, Poland, e-mail: `marcin@cs.okstate.edu`

Przemysław Stpiczyński, Department of Computer Science, Maria Curie–Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland,
e-mail: `przem@hektor.umcs.lublin.pl`