The Design and Implementation of a Dense Parallel Linear System Solver

Bogdan Oancea[1] and Razvan Zota[2]

[1] Artifex University, Bucharest, 060754 Romania,
Emai: obogdan@xnet.ro
[2] Academy for Economic Studies, Bucharest, 010552 Romania,
Email: zota@ase.ro

Abstract. Recent developments in high-performance computer architecture has a significant effect on all fields of scientific computing. Linear algebra and especially the solution of linear systems of equations lies at the heart of many applications in scientific computing. Parallel implementation of the dense linear algebra operations is a well understood process but the availability of high performance, general purpose parallel dense linear algebra libraries is limited by the complexity of implementation. This paper describes parallel versions of dense matrix factorization algorithms used in linear system solving and their implementation in PLSS - a library which provides routines used to solve linear systems with an interface easy to use, close to the natural description of sequential algorithms. PLSS uses direct methods for linear systems, based on parallel versions of Cholesky and LU factorizations. PLSS was developed using C, the MPI library for communication and the BLAS library for local computations.

1 Introduction

The importance of designing high performance algorithms for linear systems is motivated by several scientific and engineering applications that made use of linear systems. In many cases the matrix of the linear systems which arises in scientific computing is dense and very large. Solving these systems efficiently requires parallel computers not only because of the amount of computations involved but also because of the limitations in the capacity of computer memory. Matrices that are very large must be split and distributed to local memories of a parallel computer.

We can find dense linear systems of equations in many applications involving the solutions of boundary integral equations: diffusion of solid bodies in liquid, light diffusion, noise reduction, supercomputer benchmarking. Electrodynamics applications which need to compute the Helmholtz equations also works with very large and dense linear systems.

In order to solve a linear system $Ax = b$ one can use two different methods:

- direct methods based on matrix factorization (LU factorization for general matrices and Cholesky factorization for symmetric positive definite matrices);
- iterative methods: stationary methods (Jacobi, Gauss-Seidel) or nonstationary methods (e.g. Krylov methods);

This paper presents and analyzes high performance algorithms based on the direct methods for solving a linear system. Starting with the classical right-looking algorithm for the matrix factorization used to solve dense linear systems, we show how we can obtain a block variant of the factorization algorithm and how we can obtain a parallel version of it.

Software packages for solving dense linear systems have known many generations of evolution in the past 30 years. In '70, LINPACK was the first portable linear system solver package. At the end of '80 the next software package for linear algebra problems was LAPACK [10] which, few years later, was adapted for parallel computation resulting the ScaLAPACK [6] library.

Although parallel algorithms for linear systems are well understood, the availability of general purpose, high performance parallel dense linear algebra libraries is limited by the complexity of implementation. In this paper we describe a parallel implementation of a linear system solver: PLSS - Parallel Linear System Solver [14]. PLSS is a library which provides routines used to solve linear systems. The library was designed with an easy to use interface, which is almost identical with the serial algorithms interface. This goal was obtained by means of data encapsulation in opaque objects that hide the complexity of data distribution and communication operations. The PLSS library was developed in C and for the communication between processors we have used the MPI library [4, 5] which is a *de facto* standard in message passing environments.

The rest of the paper is organized as follows. In section 2 we present the serial right-looking LU factorization algorithm and it's block version. In section 3 we derive a block parallel version of the LU factorization algorithm and we analyze the performance of this algorithm using different data mapping strategies.In section 4 we describe the implementation of parallel algorithms for linear systems in the PLSS library.

2 Direct Methods for Linear Systems

Solving a linear system

$$Ax = b$$

by a direct method consists in two steps:

- in the first step we compute the matrix factorization $A = LU$ where L is unit upper triangular and U is lower triangular if A is a general matrix or $A = LL^t$ if A is a symmetric positive definite matrix;
- in the second step we solve two triangular systems $Ly = b$ and $Ux = y$;

We will focus on the LU factorization. For the LU factorization the classical method is the gaussian elimination. Using the well known Matlab notation, the right-looking LU factorization algorithm is presented below (algorithm 1) [3].

There are some problems with this algorithm. When $A(k, k)$ is zero this LU factorization algorithm cannot proceed. Another problem appears when we encounter small pivots during the computations - in this case this LU factorization algorithm is numerical instable. One way to deal with the problem of small pivots is partial pivoting which means reordering the matrix rows such that $A(k, k)$ is the largest element on

Algorithm 1 *LU* factorization - right-looking variant

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

Algorithm 2 *LU* factorization with partial pivoting

for $k = 1 : n-1$ **do**
  find ν with $k \leq \nu \leq n$ such that $|A(\nu,k)| = \| A(k:n,k) \|_\infty$
  $A(k,k:n) \leftrightarrow A(\nu,k:n)$
  $p(k) = \nu$
  if $A(k,k) \neq 0$ **then**
    $A(k+1:n,k) = A(k+1:n,k)/A(k,k)$
    $A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)A(k,k+1:n)$
  end if
end for

the column k at each step of the algorithm. The partial pivoting LU factorization is presented in algorithm 2, where $p(1:n-1)$ defines the interchange permutations.

The complexity of this algorithm is $\Theta(2n^3/2)$.

After we have obtained the matrix factors L and U we have to solve two triangular systems: $Ly = b$ and $Ux = y$. These systems are solved using forward and backward substitution with algorithms 3 and 4 [3].

Algorithm 3 Forward substitution - b is overwritten with the solution of $Ly = b$

$b(1) = b(1)/L(1,1)$
for $i = 2$ to n **do**
  $b(i) = (b(i) - L(i,1:i-1)b(1:i-1))/L(i,i)$
end for

The complexity of forward and backward substitution is $\Theta(n^2)$.

In practice, with actual computers based on memory hierarchies, the above algorithms are not efficient because they use only level 1 and level 2 BLAS operations [9, 8]: pivot division and rank-one update. As we know, level 3 BLAS operations [7] like matrix-matrix multiplication have a better efficiency than level 1 and level 2 BLAS operations. The standard way to change a level 2 BLAS operation in a level 3 BLAS operation is delayed updating. In our case, we will replace k rank-1 updates (BLAS 2 operations) with a single rank-k update (a BLAS 3 operation).

Algorithm 4 Backward substitution - y is overwritten with the solution of $Ux = y$

$y(n) = y(n)/U(n,n)$
for $i = n-1$ to 1 step -1 **do**
  $y(i) = (y(i) - U(i, i+1:n)y(i+1:n))/U(i,i)$
end for

We will derive a block algorithm for LU factorization which uses level 3 BLAS operations. Suppose that the $n \times n$ matrix A is partitioned as presented in figure 1 where A_{00} is $b \times b$.

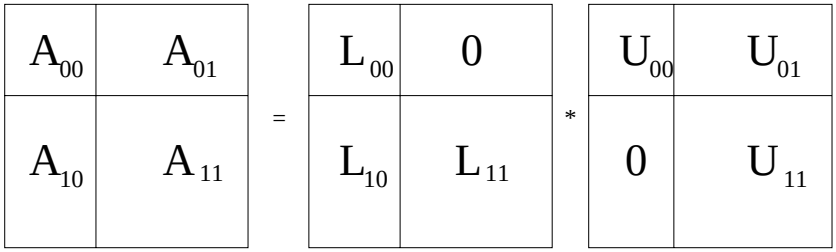

Fig. 1. Block LU factorization

From $A = LU$ we can write the following equations:

$$L_{00}U_{00} = A_{00} \tag{1}$$

$$L_{10}U_{00} = A_{10} \tag{2}$$

$$L_{00}U_{01} = A_{01} \tag{3}$$

$$L_{10}U_{01} + L_{11}U_{11} = A_{11} \tag{4}$$

Equations 1 and 2 performs the LU factorization of the first b columns of the matrix A (A_{00} and A_{10}). From this factorization we obtain L_{00}, L_{10} and U_{00} and now we can solve the lower triangular system defined by the equation 3. This system gives us U_{01}. Equation 4 can be rearranged as

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \tag{5}$$

From equation 5 it is obvious that the problem of computing L_{11} and U_{11} reduces to compute the factorization of the submatrix A'_{11} which can be done by applying the same procedure as above with $A_{11}{}'$ instead of A.

Taking into account these considerations, we can derive now the block LU factorization. Suppose that we have divided the matrix A in column blocks with b columns

in each block. The factorization of the current column block is done with the usual BLAS 2 operations and the active part of the matrix A will be updated with b rank-one updates simultaneously which, in fact, is a matrix-matrix multiplication (a BLAS 3 operation). The performance of the block LU factorization is strongly influenced by the value of b - this value has to be small enough so that the current column block fit into the cache memory and large enough to obtain an efficient matrix-matrix multiplication. Algorithm 5 [14] represents the block LU factorization of the nonsingular matrix A with partial pivoting.

Algorithm 5 Block LU factorization of the nonsingular matrix A

for k_b=1 to $n-1$ step b **do**
  $b_f = \min(k_b + b - 1, n)$
  { LU factorize $A(k_b : n, k_b : b_f)$ with BLAS 2 }
  for $i = k_b$ to b_f **do**
    find k such that $|A(k,i)| = \| A(i:n,i) \|_\infty$
    if $i \neq k$ **then**
      swap rows i and k
    end if
    $A(i+1:n,i) = A(i+1:n,i)/A(i,i)$
    $A(i+1:n, i+1:b_f) = A(i+1:n, i+1:b_f) - A(i+1:n,i) * A(i, i+1:b_f)$
  end for
  { Let $\tilde{L}$ be the unit lower triangular matrix $b \times b$ stored in $A(k_b : b_f, k_b : b_f$ }
  Solve triangular systems: $\tilde{L}Z = A(k_b : b_f, b_f + 1 : n)$
  Update $A(k_b : b_f, b_f + 1 : n) \leftarrow Z$
  {Do *delayed updating*}
  $A(b_f+1:n, b_f+1:n) = A(b_f+1:n, b_f+1:n) - A(b_f+1:n, k_b:b_f) * A(k_b:b_f, b_f+1:n)$
end for

The process of factorization is shown in figure 2. The factorization of $A(k_b : n, k_b : b_f)$ block is done with normal BLAS 2 algorithm and triangular systems with matrix $\tilde{L}$ are solved with a single call of a BLAS 3 routine.

If $n \gg b$ then almost all the floating point operations are done in the last row of algorithm 5 which is a BLAS 3 (GEMM) operation.

3 Parallel Algorithms for Linear Systems

Because the complexity of the matrix factorization, $\Theta(n^3)$, is greater than that of the forward or backward substitution we will focus our attention on parallelization of the LU factorization algorithm. The block LU factorization algorithm (5) can be easily implemented on a distributed memory computer. In order to obtain a good efficiency of the parallel algorithm, we will try to find the best way to distribute the matrix over processors. There are two problems in choosing a data layout for LU factorization: load balancing and the ability to use level 3 BLAS operations. In the following, the matrix columns are numbered with $0 \ldots n-1$ and processors with $0 \ldots p-1$. Figure 3 shows

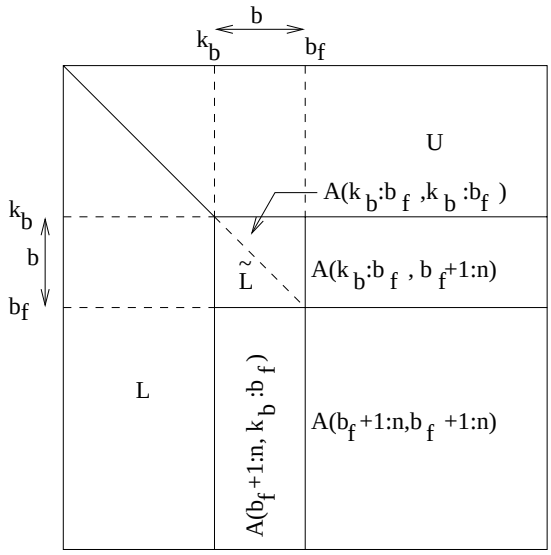

Fig. 2. *LU* factorization with BLAS 3 operations

four different data layouts for $n = 16$ and $p = 4$, each block of matrix A being labeled with the processor's index that stores it.

In figure 3 a) each processor stores n/p adjacent columns of matrix A. This layout does not allow a good load balancing because, for example, after the first n/p columns have been factorized, processor number 0 will be idle. The second data layout (figure 3 b), column cyclic, assigns column i to processor $i(\,mod\, p)$. Thus, the load balancing improves, but we cannot use level 2 and level 3 BLAS operations because each processor have single columns and not contiguous blocks of columns. The third data layout (figure 3 c), column block cyclic, is a combination of the first two. The matrix columns are divided into groups of size b and these groups are distributed to processors in a cyclic fashion: column i is assigned to processor $(i/b)(mod p)$. In our example we have choosen $b = 2$. This layout has a worse load balancing than the previous one but it can use level 2 and level 3 BLAS operations. The major drawback of this layout is the fact that the factorization of $A(k_b : n, k_b : b_f)$ will be computed by one single processor if each processor stores a block of b columns, which is a serialization of computations.

The fourth data layout (figre 3 d), row and column block cyclic (2D), prevents the serialization of computations. In this case, processors are placed in a two dimensional logical grid $p_r \times p_c$. The matrix element $A(i, j)$ is assigned to processor (P_i, P_j) where

$$P_i = (i/b_r)(mod\, p_r)$$

$$P_j = (j/b_c)(mod\, p_c)$$

b_r and b_c being the dimensions of a matrix block. Figure 3 d) shows this layout for $p_r = p_c = 2$ and $b_r = b_c = 2$. This layout allows a parallelism of p_c degree on each column and also allows us two use BLAS 2 and BLAS 3 operations.

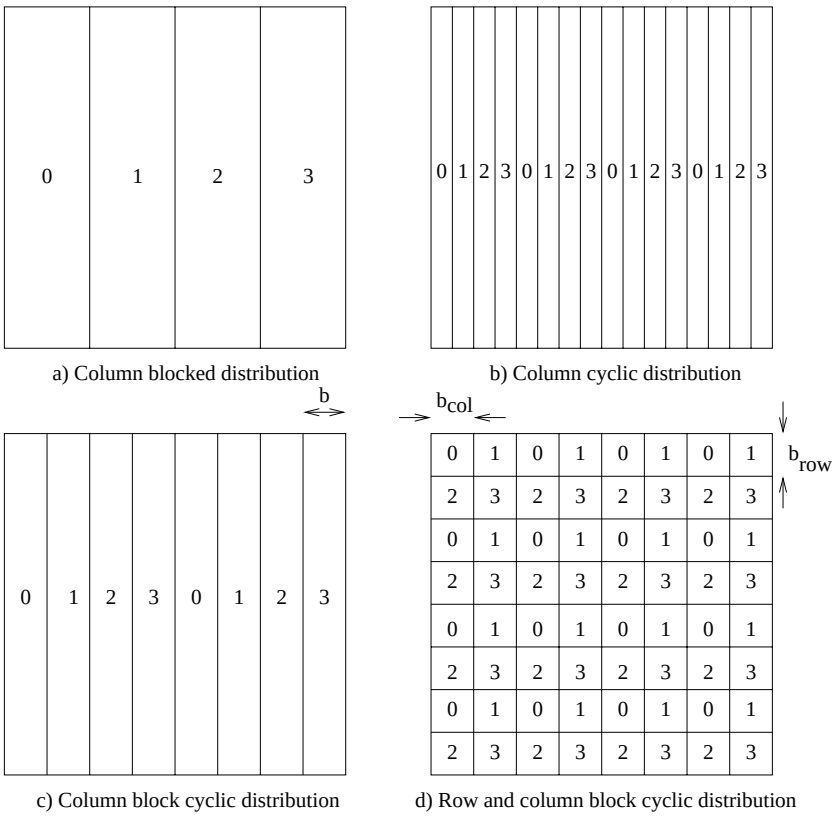

Fig. 3. Data layout for LU factorization

We will analyze the performance of the LU factorization using a 2D cyclic data layout with $b_r = b_c = b$. Algorithm 6 is the parallel version of the serial LU factorization algorithm 5. In algorithm 6 we have presented only the basic operations, including the communication operations.

We will consider an asynchronous communication between processors and we will analyze lines 14, 15 and 16 of the algorithm 6 where the majority of floating point and communication operations take place. As soon as a processor receives the top and the right block, it can compute the local multiplication needed by the local updating of its portion of $A(b_f + 1 : n, b_f + 1 : n)$. After the first b columns from the left of submatrix $A(b_f + 1 : n, b_f + 1 : n)$ have been updated the factorization of these columns can be started while the rest of the processors update the remaining portion of the matrix. Thus, we can obtain an overlapping of computations and communication (lines 15 and 16 from the algorithm 6).

In order to obtain an analytical form of the time need by the LU factorization algorithm, we consider a bidimensional mesh of processors with a cut-through routing

Algorithm 6 Parallel block LU factorization

1: **for** k_b=1 to $n-1$ step b **do**
2: $b_f = \min(k_b + b - 1, n)$
{ Factorization of $A(k_b : n, k_b : b_f)$ - BLAS 2 version}
3: **for** $i = k_b$ to b_f **do**
4: find pivot index k - column broadcast operation
5: swap rows i and k in the current column block - broadcast of row k
6: $A(i+1 : n, i) = A(i+1 : n, i)/A(i, i)$
7: $A(i+1 : n, i+1 : b_f) = A(i+1 : n, i+1 : b_f) - A(i+1 : n, i) * A(i, i+1 : b_f)$
8: **end for**
9: Broadcast all swap information to the left and to the right of the current column block
10: apply all row swaps to other columns
11: Broadcast $\tilde{L}$ to the right
12: Solve triangular systems $\tilde{L}Z = A(k_b : b_f, b_f + 1 : n)$
13: Update $A(k_b : b_f, b_f + 1 : n) \leftarrow Z$
14: Broadcast $A(k_b : b_f, b_f + 1 : n)$ down
15: Broadcast $A(b_f + 1 : n, k_b : b_f)$ right
16: $A(b_f + 1 : n, b_f + 1 : n) = A(b_f + 1 : n, b_f + 1 : n) - A(b_f + 1 : n, k_b : b_f) * A(k_b : b_f, b_f + 1 : n)$
17: **end for**

technique [2]. In line 14 we have a column broadcast operation of a submatrix of $b(n - b_f)/p_c$ dimension. The execution time of this operation is [14]:

$$T_{14} = \left(t_s + t_w \frac{b(n - b_f)}{p_c} \right) \log p_r \tag{6}$$

units of time, where t_s is the message start-up time and t_w is per word time.

In line 15 we also have a broadcast operation, but we use a broadcast in a ring of processors, each processor sending the message to its neighbor on the right. A total number of p_c steps are needed to complete this operation. The execution time is [14]:

$$T_{15} = p_c \left(t_s + t_w \frac{b(n - b_f)}{p_r} \right) \tag{7}$$

Note that only the leftmost column of processors which stores $A(b_f+1 : n, b_f+1 : n)$ needs to receive these submatrices, send them and update their portion of matrix before proceeding to LU factorization of the next block column. Thus, T_{15} can be rewritten as:

$$T_{15} = 2 \left(t_s + t_w \frac{b(n - b_f)}{p_r} \right) \tag{8}$$

In line 16 each processor computes a local matrix-matrix multiplication and the time needed for this operation is:

$$T_{16} = 2b(n - b_f)^2/p \tag{9}$$

Summing up the time for all steps of algorithm 6 we obtain [14]:

$$T_p = t_s(6n + n\log p_r) + t_w(2n^2(p_r - 1)/p + n^2(p_c - 1)/p + n^2\frac{\log p_r}{2p_c}) + 2n^3/3p \quad (10)$$

The efficiency of this algorithm is:

$$E = \frac{T_s}{pT_p} = \frac{2n^3/3}{pt_s(6n + n\log p_r) + pt_w(\frac{2n^2(p_r-1)}{p} + \frac{n^2(p_c-1)}{p} + n^2\frac{\log p_r}{2p_c}) + 2n^3/3} \quad (11)$$

$$E = \frac{1}{1 + \frac{3}{2}\frac{p(6+\log p_r)}{n^2}t_s + \frac{3}{2}\frac{2p_r+p_c+(p_r\log p_r)/2-3}{n}t_w} \quad (12)$$

In order to maximize the efficiency we have to minimize the expression $2p_r + p_c + (p_r \log p_r)/2 = 2p_r + p/p_r + (p_r \log p_r)/2$. This expression is minimized when $p_r \approx p_c \approx \sqrt{p}$. In this case, ignoring the log terms, the efficiency is:

$$E \approx \frac{1}{O(\frac{p}{n^2}) + O(\frac{\sqrt{p}}{n})} \quad (13)$$

To maintain the efficiency at a constant value it is necessary that $n^2 \propto p$. From this relationship we can easily derive the isoefficiency function [2] which is $O(p\sqrt{p})$ [14] - this means that the implementation of the LU factorization algorithm is highly scalable.

After the matrix factorization we have to solve two triangular systems: $Ly = b$ and $Ux = y$. Because the complexity of solving a triangular system is less than those of matrix factorization, we use the same data layout as we used for factorization. The total parallel execution time for solving the triangular system is $\Theta(n^2/\sqrt{p})$. Under these conditions, the cost of the algorithm is $\Theta(n^2\sqrt{p})$. Although, this is not a cost-optimal implementation, the whole process of factorization and triangular systems solving is cost-optimal [14].

4 Implementation of the Parallel Algorithms for Linear Systems in the PLSS Library

We have implemented the parallel algorithms for linear system solving in a library - PLSS (Parallel Linear System Solver). The main purpose of this library is to present a simple and easy to use interface, close to the serial algorithms description. This goal was achieved using an object oriented technique - vectors and matrices are encapsulated in opaque objects, hiding the difficulty of data distribution and communication operations from the user. The PLSS library was developed in C and for the communication operations we have used the MPI library.

The structure of the PLSS library is depicted in figure 4.

The first level is contains the standard BLAS, MPI and C libraries. This level is architecture dependent. The second level provides the architecture independence and implements the interface between the base level and the rest of the PLSS package. This interface has the following components:

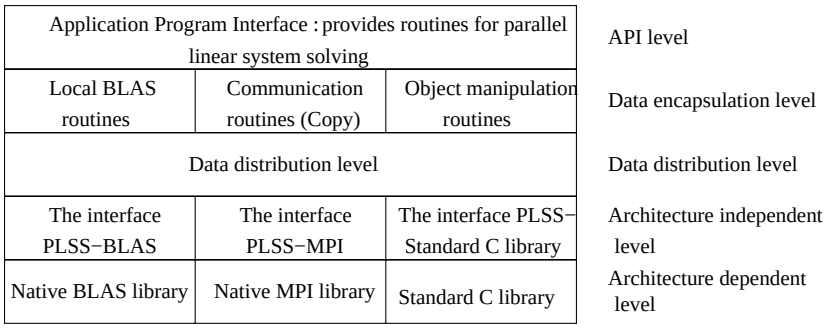

Fig. 4. The structure of the PLSS library

- *BLAS-PLSS interface.* Each processor uses the BLAS routines for local computations. Because the standard BLAS library is written in FORTRAN an interface is needed to call FORTRAN routines from C programs.
- *MPI-PLSS interface.* PLSS uses the following communication operations: MPI_Bcast, MPI_gatherv, MPI_scaterv, MPI_Allgatherv, MPI_Allscatterv, MPI_Reduce,
  MPI_Allreduce, MPI_Send, MPI_Receive, MPI_Wait. All this MPI operations are encapsulated in PLSS functions in order to decouple the PLSS from MPI.
- *PLSS-Standard C library interface.* This interface encapsulates the standard C library functions (e.g. malloc, calloc, free) in PLSS functions.

The next level implements the data distribution model. All details regarding distribution of vectors and matrices on processors are localized at this level. The fourth level realizes data encapsulation in objects that are opaque to users, hiding thus the complexity of communication operations. This level defines:

- Objects that describe vectors and matrices;
- Object manipulation routines: object creation, initializing, destroying, and object addressing routines.
- Local BLAS routines. Because matrices and vectors are encapsulated in objects, we must extract some information from these objects such as vector/matrix dimension, their localization etc before calling a BLAS routine to perform some computations. Local BLAS routines extract these information and then call the standard BLAS routines.
- Copy functions: these functions implement the communication operations between processors.

The top level of the PLSS library is in fact the application program interface. PLSS API provides a number of routines that implements parallel BLAS operations and parallel linear system solving operations based on LU and Cholesky matrix factorization.

The PLSS library uses a bidimensional mesh of processors. We have chosen this model of processor interconnection based on scalability studies of matrix factorization algorithms. In the current version of the library, for a linear system $Ax = b$, vectors

x and b are distributed on processors in a block column cyclic fashion and the system matrix A is distributed according to the vector distribution: the column $A_{*,j}$ will be assigned to the same processor as x_j.

We conclude this section with an example of parallelization of some basic operations in PLSS library. Matrix-vector multiplication $Ax = y$ is a frequent operation in linear system solving algorithms. Figure 5 shows the necessary steps to implement parallel matrix-vector multiplication. In this example A is 9×9 and we use a 3×3 bidimensional grid of processors.

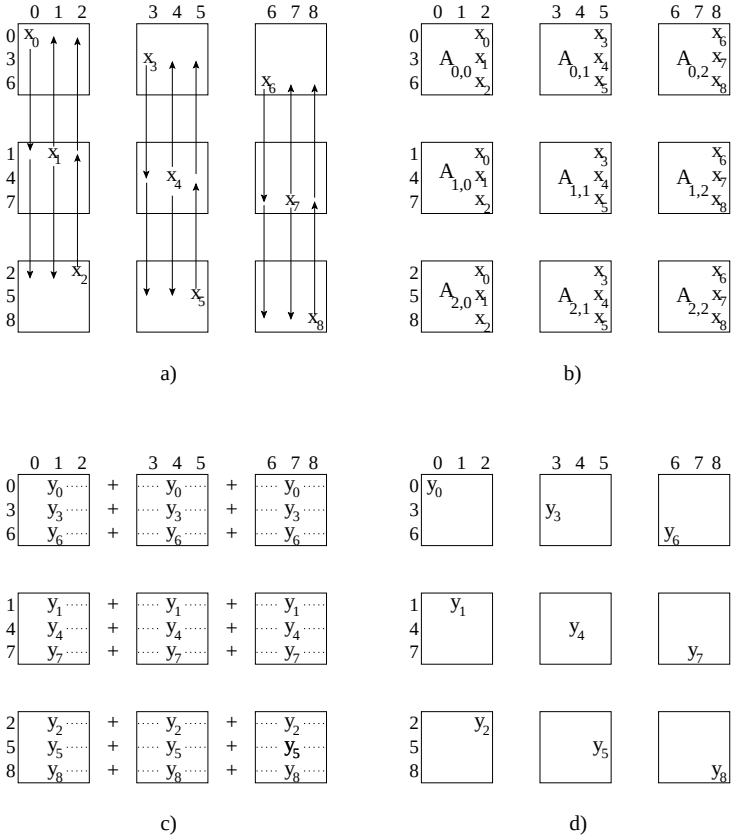

Fig. 5. Matrix-vector multiplication

In the first step (figure 5 a) the vector components are distributed onto the processor columns. After vector distribution, it follows a step consisting of local matrix-vector multiplication (figure 5 b). At this moment each processor owns a part of the final result (figure 5 c); these partial components are summed up along the processor rows giving the final result y (figure 5 d).

Rank-1 update is another basic operation which consists in the following computation: $A = A + yx^t$. Assuming that x and y have identical distributions on processor columns and rows, each processor has the data needed to perform the local computations.

These two basic operations, matrix-vector multiplication and rank-1 update can be used in order to derive a parallel algorithm for matrix-matrix multiplication. It is easy to observe that the product $C = AB$ can be decomposed in a number of rank-1 updates:

$$C = a_0 b_0^t + a_1 b_1^t + \ldots + a_{n-1} b_{n-1}^t$$

where a_i are columns of the matrix A and b_i^t are rows of the matrix B. Parallelization of matrix-matrix multiplication is equivalent with parallelization of a sequence of rank-1 updates. In order to obtain an increase in performance, the rank-1 update can be replaced with rank-k update but in this case x and y will be rectangular matrices

The current version of the PLSS library implements the following BLAS operations.

Level 1 BLAS operations implemented in PLSS:

- *int Axpy(Object alpha, Object x, Object y)* - Computes $y = \alpha x + y$
- *int Dot(Object x, Object y, Object alpha)* - Computes the dot product $\alpha = x^t y$
- *int Nrm2(Object x, Object alpha)* - Computes the Euclidean norm $\alpha = x^t x$
- *int Scal(Object x, Object alpha)* - Scales vector x: $x = \alpha x$
- *int Iamax(Object x, Object k, Object xmax)* - Computes the maximum value (xmax) and the global offset (k) from object x

Level 2 BLAS operations implemented in PLSS:

- *int Ger (Object alpha, Object x, Object A)* - Computes rank-1 update $A = \alpha x y^t + A$
- *int Gemv (int transa, Object alpha, Object A, Object x, Object beta, Object y)* - Computes the matrix-vector multiplication $y = \alpha Ax + \beta b$
- *int Symv (int uplo, Object alpha, Object A, Object x, Object beta, Object y)* - Computes the matrix-vector multiplication for symmetric matrices
- *int Trmv (int uplo, int trans, int diag, Object A, Object x)* - Computes the matrix-vector multiplication for triangular matrices
- *int Trsv (int uplo, int trans, int diag, Object A, object x)* - Solves the linear system $Ax = b$ where A is a triangular matrix

Level 3 BLAS operations implemented in PLSS:

- *int Syrk (int uplo, int trans, Object alpha, Object A, Object beta, Object C)* - Computes rank-k update for symmetric matrices $C = \alpha AA^t + \beta C$
- *int Trsm (int side, int uplo, int trans, int diag, Object alpha, Object A, Object C)* - Solves the triangular system $AX = B$ with multiple right-hand

The name of routines and the significance of parameters *trans*, *side*, *uplo*, *diag*, are the same as in the original BLAS library.

For matrix factorization PLSS library has two routines:

- *Cholesky (Object A)* computes the Cholesky factorization of a symmetric positive definite matrix A;
- *LU (Object A, Object pivots)* computes the LU factorization of the general matrix A with partial pivoting;

5 Conclusions

We have developed a library (PLSS - Parallel Linear System Solver) that implements parallel algorithms for linear system solving. Because of the complexity of parallel algorithms it is difficult to design an easy to use parallel linear system solver. The PLSS infrastructure was designed to provide users a simple interface, close to the description of the serial algorithms. This goal was achieved through data encapsulation, hiding the complexity of data distribution and communication operations from users. PLSS was developed in C using MPI and can be run on many different kinds of parallel computers it can be run on real parallel computers as well as on simple cluster of workstations.

References

1. G.W. Sabot, *High Performance Computing*, Addison-Wesley, (1995)
2. V. Kumar, A. Grama A. Gupta and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings Publishing Company, (1994)
3. G.H. Golub and Ch. Van Loan, *Matrix Computations*, Johns Hopkins University Press, (1996)
4. W. Gropp and E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, (1994)
5. M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker and J. Dongarra, *MPI: the Complete Reference*, MIT Press, Cambridge, MA, (1996)
6. J. Choi, J. Dongarra, R. Pozo and D.W. Walker, ScaLAPACK: a Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, *Proceedings 4th Symposium on the Frontiers of Massively Parallel Computers*, (1992) 120-127
7. J. Dongarra and J. Du Croz and S. Hammarling and I. Duff, A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, 16(1):1-17, (1990)
8. J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *ACM Transaction on Mathematical Software*, 14(1):1-17, (1988)
9. C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, *ACM Transactions on Mathematical Software*, 5(3):308-323, (1979)
10. E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK Users's Guide*, SIAM, Philadelphia, PA, (1992)
11. J. Demmel, M. Heath and H. van der Vorst, Parallel Numerical Linear Algebra, *Acta Numerica*, Cambridge University Press, (1993)
12. F. Desprez, S. Domas and B.Tourancheau, Optimization of an LU Factorization Routine using Communication/computation Overlap, INRIA, 3094, (1997)
13. Gh. Dodescu, B. Oancea and M Raceanu, *Parallel Processing*, Editura Economica, (2003)
14. B. Oancea, Parallel Algorithms for Numerical Methods in Mathematical Models of Economy, Ph.D. Thesis, ASE, Bucharest, (2002)