The Impact of Two-dimensional Block Cyclic Distribution in Accuracy and Performance of Parallel Linear Algebra Subroutines

Mariana Kolberg[1,2], Björn Rocker[3] and Vincent Heuveline[3]

[1] Universidade Luterana do Brasil
Av. Farroupilha 8001 Prédio 14, sala 122
Canoas/RS, 92425-900 - Brasil
mariana.kolberg@ulbra.br
[2] Pontificia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 - Prédio 32
Porto Alegre/RS, 90619-900, Brazil
mariana.kolberg@pucrs.br
[3] Karlsruhe Institute of Technology (KIT)
Institute for Applied and Numerical Mathematics 4
Fritz-Erler-Str. 23
71633 Karlsruhe, Germany
{bjoern.rocker, vincent.heuveline}@kit.edu

Abstract. In parallel computing the data distribution may have a significant impact in the application performance and accuracy. These effects can be observed using the parallel matrix-vector multiplication routine from PBLAS with different grid configuration for the data distributions. Matrix-vector multiplication is an especially important operation once it is widely used in numerical simulation (*e.g.*, iterative solvers for linear systems of equations).
This paper presents a mathematical background of error propagation in elementary operations and proposes benchmarks to show how different grids configuration on two dimensional cyclic block distribution impacts on accuracy and performance using parallel matrix-vector operations. The experimental results validate the theoretical findings.

1 Introduction

In many numerical algorithms, problems are reduced to a linear system of equations. Therefore, solving systems like $Ax = b$ with a matrix $A \in \mathbb{R}^{n \times n}$ and a right hand side $b \in \mathbb{R}^n$ is essential in numerical analysis. There are two major ways of solving those systems: by direct solvers, which are mainly based on the Gaussian algorithm, or by iterative solvers which are often based on projections. The second type usually contains one multiplication of a matrix with a vector in each iteration step. Therefore the precision of such matrix-vector multiplication has a significant impact on the convergence of the iterative solver [3].

In a computer, each mathematical operation is computed using floating point arithmetics. However, the finite floating-point arithmetic can only deliver an approximation of the exact result due to rounding errors. Since the exact result is usually unknown, it is sometimes difficult to measure the quality of these approximations. Besides, as a result of several operations, the accumulation of those errors may have an impact in the accuracy of the results.

There are many papers proposing different solutions to find more accurate results. Some authors concern is to improve the numerical accuracy of the computed results in computers through the use of extra precise iterative refinement [4, 5]. Others try to use mixed-precision algorithms [8, 12] to obtain a good accuracy and improve the performance. Another possible way to deal with this unreliability is to use verified computing [9]. Such techniques provide an interval result that surely contains the correct result [10, 11]. However, the use of such methods may increase the execution time significantly. This effect is even worse for large linear systems, that may need several days or even more to be solved. Based on these researches, it is possible to notice that there is a tradeoff performance versus accuracy. Usually to be able to obtain more accurate results, it is possible to notice a loss of performance.

Parallel computing is a well-known choice for simulating large problems. Since many numerical problems are solved via a large linear system of equations, a parallel algorithm would be a good approach. In this context, the libraries BLAS [2] and LAPACK [13] seem a good choice, since they have a parallel version (PBLAS and SCALAPACK [1]) that could be used in the case of very large systems. However, it is important to remember that these libraries provide an approximation of the correct result and not a verified result.

It is well known that the data distribution has a major impact in the performance of a parallel application [1]. However, the data distribution can also present an important influence on the accuracy of the numerical results. Sometimes a fixed problem can lead to distinct solutions depending on the data distribution or the number of processes used in its solution. This effect can possibly be explained by the rounding error theory [14].

Based on that, this paper investigates the impact of different grids configuration used in the two-dimensional block cyclic distribution on the accuracy and performance of the parallel matrix-vector multiplication implemented by PBLAS. This particular distribution was chosen since it was proved to be a good choice for parallel matrix distribution on parallel environments with distributed memory [6]. Other interesting data distribution was proposed in [7], however it is also based on block distribution and was consider equivalent to the two-dimensional block cyclic distribution [15].

In this paper, the performance of different grids configuration was measured and compared among them. To evaluate the accuracy of the approximations generated by PBLAS, a comparison with the verified solution provided by C-XSC [10] is done. The experimental results indicate how the grids should be configured to find a compromise between accuracy and performance considering the application needs.

This text is organized as follows. To better understand this problem, section 2 presents two important backgrounds: the theory of rounding errors and the two-dimensional block cyclic distribution scheme. Section 3 introduces the platform, input data and results obtained in the numerical experiments. Finally, section 4 present some final remarks and considerations about future work.

2 Background

This section presents the theoretical background concerning rounding errors, based on a paper of Linz [14], and the two-dimensional block cyclic distribution used by PBLAS.

2.1 Theory of rounding errors

Let ϵ be the machine accuracy and $fl(a \circ b)$ the floating point result for an elementary composition of two real numbers a and b. An elementary operation $\circ \in \{+, -, *, /\}$ of a and b can be estimated with $fl(a \circ b) = (a \circ b) + \epsilon(a, b, \circ)$ for the worst case. We assume $A \in \mathbb{R}^{n \times n}, x, y \in \mathbb{R}^n$ and get $y_k = a_k x$ as result for the product $Ax = y$ for every entry $y_k \in y$. Let a_k denote the $k - th$ row of A. For all y_k, the approximation using the floating point arithmetic is $\hat{y}_k$.

Simple approach The simple strategy for computing each $y_k \in y$ is to add the first entry to the next one and then add the following entries one by one to the previous result. Using floating point arithmetic and the abbreviation $fl(a_{k,i} \cdot x_i) = a_{k,i} \cdot x_i + \epsilon(a_{k,i}, x_i, \cdot) =: \hat{b}_{k,i}$, this strategy can be written as follows:

$$\hat{y}_{k_1} := (\hat{b}_{k,1} + \hat{b}_{k,2}) + \epsilon_1$$

$$\hat{y}_{k_i} := \hat{y}_{k_{i-1}} + \hat{b}_{k,i} + \epsilon_i = y_{k_i} + \sum_{j=1}^{i} \epsilon_j \ , i \in \{2, 3, \ldots n-1\}$$

Let the representation be the normalized floating-point with binary exponent and q fraction bits and assume the addition to be done by truncating the exact sum to q bits. Let p_i be the exponent of $\hat{y}_{k_i}$ and $\nu = 2^{-q}$. The error for the ith step is then $|\epsilon_i| \leq \nu 2^{p_i}$ and the global error can be written using the estimates $a_{k,i} x_i \leq b$, $|\hat{y}_{k_i}| \leq ib$ and $2^{p_i} \leq 2ib$ in the following way

$$|y_k - \hat{y}_k| \leq \nu \sum_{i=1}^{n-1} 2^{p_i} \leq 2\nu b \sum_{i=1}^{n} i = \nu b n(n+1).$$

This means the error using this approach grows like $O(n^2)$.

Advanced approach The second strategy for the summation is the so called "Fan-In" algorithm. The values are added to each other in pairs and the algorithm is then executed recursively. Let us define the notation

$$y_k = \underbrace{\underbrace{\underbrace{a_{k,1}x_1 + a_{k,2}x_2}_{\hat{y}_{n_{1,1}}} + \underbrace{a_{k,3}x_3 + a_{k,4}x_4 +}_{\hat{y}_{n_{2,1}}}}_{\hat{y}_{n_{1,2}}} \ldots + a_{k,n}x_n}_{\hat{y}_{n_{1,m}}}$$

and

$$\hat{y}_{k_{i,j}} = \hat{y}_{k_{2i-1,j-1}} + \hat{y}_{k_{2i,j-1}} + \epsilon_{i,j}$$

where $\epsilon_{i,j}$ is the rounding error when computing $\hat{y}_{k_{i,j}}$. For the global error we have

$$|y_k - \hat{y}_k| = \sum_{\sigma_{1,k}} \epsilon_{i,j} \leq \nu \sum_{\sigma_{1,k}} 2^{p_{\sigma_{1,k}}}$$

where p is the exponent of the result and $\sigma_{1,k}$ is the set of all index pairs needed to get $\hat{y}_{k_{i,j}}$. Assuming that $a_{k,i}x_i \leq b$, we have:

$$\hat{y}_{k_{i,j}} \leq 2^j b \text{ and } 2^{p_{i,j}} \leq 2^{j+1} b.$$

Based on that, the error upper bound can be estimated by

$$|y_n - \hat{y}_k| \leq \nu \sum_{\sigma_{1,k}} 2^{p_{\sigma_{1,k}}} \leq 2\nu b \sum_{j=1}^{k} \sum_{i=1}^{n/2^j} 2^j = 2\nu bkn \leq 2\nu b\, n \log_2 n.$$

For the advanced approach the error grows like $O(n \log_2 n)$. The proof can be extended to cases in which more than two entries are added to each other using the "Fan-In"-algorithm. In that case, the error propagations is bounded by the one presented by the strategies above.

The two approaches differ in a factor of $n/(2\log_2 n)$. This study suggests that a finer granularity in the summation leads to lower upper boundaries for rounding errors. The proofs presented above show the impact of rounding errors in scalar products, which are commonly part of matrix-vector multiplications.

2.2 Data distribution in numerical algorithms

On distributed memory platforms, the application programmer is responsible for assigning the data to each processor. How this is done has a major impact on the load balance and communication characteristics of the algorithm, and largely determines its performance and scalability [1].

PBLAS routines are implemented supposing the matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [6]. In this distribution, an M by N matrix is first decomposed into MB by NB blocks starting at its upper left corner. The distribution of a vector is done considering the vector as a column of the matrix. Suppose we have the following 10x10 matrix, a vector of length 10 an MB and NB equal 3. In this case, we would have the following blocks:

$$\left(\begin{array}{ccc|ccc|ccc|c}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} & A_{0,5} & A_{0,6} & A_{0,7} & A_{0,8} & A_{0,9} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} & A_{1,6} & A_{1,7} & A_{1,8} & A_{1,9} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{2,5} & A_{2,6} & A_{2,7} & A_{2,8} & A_{2,9} \\ \hline
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} & A_{3,6} & A_{3,7} & A_{3,8} & A_{3,9} \\
A_{4,0} & A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & A_{4,5} & A_{4,6} & A_{4,7} & A_{4,8} & A_{4,9} \\
A_{5,0} & A_{5,1} & A_{5,2} & A_{5,3} & A_{5,4} & A_{5,5} & A_{5,6} & A_{5,7} & A_{5,8} & A_{5,9} \\ \hline
A_{6,0} & A_{6,1} & A_{6,2} & A_{6,3} & A_{6,4} & A_{6,5} & A_{6,6} & A_{6,7} & A_{6,8} & A_{6,9} \\
A_{7,0} & A_{7,1} & A_{7,2} & A_{7,3} & A_{7,4} & A_{7,5} & A_{7,6} & A_{7,7} & A_{7,8} & A_{7,9} \\
A_{8,0} & A_{8,1} & A_{8,2} & A_{8,3} & A_{8,4} & A_{8,5} & A_{8,6} & A_{8,7} & A_{8,8} & A_{8,9} \\ \hline
A_{9,0} & A_{9,1} & A_{9,2} & A_{9,3} & A_{9,4} & A_{9,5} & A_{9,6} & A_{9,7} & A_{9,8} & A_{9,9}
\end{array}\right)
\left(\begin{array}{c}
b_0 \\ b_1 \\ b_2 \\ \hline b_3 \\ b_4 \\ b_5 \\ \hline b_6 \\ b_7 \\ b_8 \\ \hline b_9
\end{array}\right)$$

Suppose we have 4 processors. The process grid would be a 2x2 grid as follows:

$$\left(\begin{array}{c|c} P^0 & P^1 \\ \hline P^2 & P^3 \end{array}\right)$$

These blocks are then uniformly distributed across the process grid. Thus, every processor owns a collection of blocks [1]. The first row of blocks will be distributed among the first row of the processor grid, that means among P_0 and P_1, while the second row will be distributed among P_2 and P_3, and so on. For this example, we would have:

$$\left(\begin{array}{c|c|c|c}
P^0 & P^1 & P^0 & P^1 \\ \hline
P^2 & P^3 & P^2 & P^3 \\ \hline
P^0 & P^1 & P^0 & P^1 \\ \hline
P^2 & P^3 & P^2 & P^3
\end{array}\right)
\left(\begin{array}{c}
P^0 \\ \hline P^2 \\ \hline P^0 \\ \hline P^2
\end{array}\right)$$

According to this distribution, each processor would have the following data:

$$P^0 : \left(\begin{array}{ccc|ccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,6} & A_{0,7} & A_{0,8} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,6} & A_{1,7} & A_{1,8} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,6} & A_{2,7} & A_{2,8} \\ \hline
A_{6,0} & A_{6,1} & A_{6,2} & A_{6,6} & A_{6,7} & A_{6,8} \\
A_{7,0} & A_{7,1} & A_{7,2} & A_{7,6} & A_{7,7} & A_{7,8} \\
A_{8,0} & A_{8,1} & A_{8,2} & A_{8,6} & A_{8,7} & A_{8,8}
\end{array}\right)
\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_6 \\ b_7 \\ b_8 \end{pmatrix}
\quad
P^1 : \left(\begin{array}{ccc|c}
A_{0,3} & A_{0,4} & A_{0,5} & A_{0,9} \\
A_{1,3} & A_{1,4} & A_{1,5} & A_{1,9} \\
A_{2,3} & A_{2,4} & A_{2,5} & A_{2,9} \\ \hline
A_{6,3} & A_{6,4} & A_{6,5} & A_{6,9} \\
A_{7,3} & A_{7,4} & A_{7,5} & A_{7,9} \\
A_{8,3} & A_{8,4} & A_{8,5} & A_{8,9}
\end{array}\right)$$

$$P^2 : \left(\begin{array}{ccc|ccc}
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,6} & A_{3,7} & A_{3,8} \\
A_{4,0} & A_{4,1} & A_{4,2} & A_{4,6} & A_{4,7} & A_{4,8} \\
A_{5,0} & A_{5,1} & A_{5,2} & A_{5,6} & A_{5,7} & A_{5,8} \\ \hline
A_{9,0} & A_{9,1} & A_{9,2} & A_{9,6} & A_{9,7} & A_{9,8}
\end{array}\right)
\begin{pmatrix} b_3 \\ b_4 \\ b_5 \\ b_9 \end{pmatrix}
\quad
P^3 : \left(\begin{array}{ccc|c}
A_{3,3} & A_{3,4} & A_{3,5} & A_{3,9} \\
A_{4,3} & A_{4,4} & A_{4,5} & A_{4,9} \\
A_{5,3} & A_{5,4} & A_{5,5} & A_{5,9} \\ \hline
A_{9,3} & A_{9,4} & A_{9,5} & A_{9,9}
\end{array}\right)$$

The two dimensional block cyclic distribution is usually not used when the matrix has a sparse data structure. Common storage formats, like CRS (compressed row storage), CCS (compressed column storage) etc., usually lead to distributions where a certain number of rows or columns are given to each process. The CRS presents a row-wise data distribution, that could be seen as a $np \times 1$ grid of processes in the two dimensional block cyclic distribution. In the same way, a $1 \times np$ grid could be interpreted as a column wise distribution, e.g. similar to a strategy taken when the CCS is used.

From the mathematical theory it is clear that the upper bound for the rounding errors occurring in a sparse matrix vector multiplication increases with the number of nonzero elements (nnz) per row. In the proof presented in the last subsection, the variable n should be replaced with nnz, assuming that the matrix data is sparse and the vector data has $O(n)$ entries.

It is important to mention that, for a matrix vector multiplication, no benefit in accuracy can be expected due to parallelization when this is done based on a row-wise data distribution and assuming that the sequential part on each process does not use techniques like "fan-in".

3 Numerical experiments

This section presents experimental results for different grid compositions. The accuracy and performance are the focus of these tests.

For our experiments we compute the results of the considered matrix-vector multiplication using first the sequential BLAS-routines to obtain the sequential time. After this sequential test, we use the PBLAS-routines from the MKL package for different grid compositions. All results have been computed three times to avoid effects caused by hard- and software problems.

Aiming to be able to analyze the accuracy obtained by the parallel implementation, a comparison with a verified result is done to evaluate which grid presents the best accuracy among the tested grids. The library used to obtain the verified result was C-XSC, which stands for "**e**xtension for **s**cientific **c**omputing", and is a free programming tool for the development of numerical algorithms which provides highly accurate and automatically verified results. C-XSC does computations based on interval arithmetics and direct rounding, providing an enclosure of the exact solution, which is represented by an interval. This means that for a matrix-vector multiplication, C-XSC will deliver a vector of intervals, each entry of the vector containing an interval enclosure of the correct solution. The diameter of the intervals is usually very small, since C-XSC implementation uses techniques to iteratively reduce the interval diameter proofing that the interval includes the exact result [10].

The average error from the pblas result to the C-XSC result is computed as follows. First, for each component of the result vector it is checked if it is in the interior of the interval given from C-XSC. If it is inside the interval, it is considered correct. If it is not inside, the distance to the interval is stored. Then the arithmetic mean over all distances is computed, which we denote as average error.

Next section introduces the platform used for the experiments. Section 3.2 illustrates the input data of four different matrices used in the tests. Finally section 3.3 presents the accuracy and performance results with some considerations.

3.1 Platform

The software platform adopted for our numerical experiments is composed of optimized versions of the library PBLAS (Intel CMKL in version 10.0.2.018 for test case M1 and version 10.1.2.024 for the test cases M2, M3 and M4), C-XSC version 2.2.3 and the standard Message Passing Interface (MPI), more specifically the OpenMPI implementation in version 1.2.8. The compiler was the Intel compiler in version 10.1.021.

The hardware platform is the Institutscluster located at the SCC at the Karlsruhe Institut for Technology (KIT). The cluster consists of 200 computing nodes each equipped with two Intel quadcore EM64T Xeon 5355 processors running at 2,667 GHZ, 16 GB of main memory and an Infiniband 4x DDR interconnect. The overall peak performance of the whole system is about 17,57 TFlops and 15,2 TFlops in the Linpack benchmark.

3.2 Input data

The results shown in Section 3.3 refer to four different input matrices and vectors. The first test matrix, called M1, is of dimension 16000 and filled with pseudo random numbers from the interval $[-0,5;0.5]$ and its properties can be seen in table 1. This matrix data needs 2048 MB to be stored.

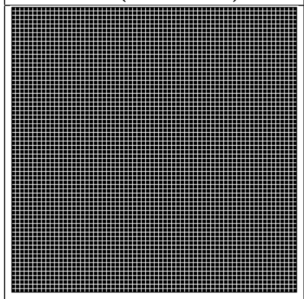
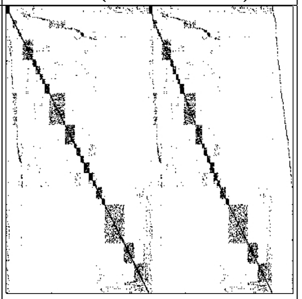
M1 (Random)	**M2 (GEMAT11)**
	
problem: Artificial problem size: $n = 16000$ sparsity: $nnz = 256000000$ cond. number: n.a. Frobenius norm: n.a.	problem: Power flow. problem size: $n = 4929$ sparsity: $nnz = 33185$ cond. number: 3.74e+08 Frobenius norm: 8.2e+02

Table 1. Sparsity plots and properties of the test-matrices M1 and M2

The test cases M2 to M4, and more detailed information about their creation and properties, can be found on the Matrix Market[4].

M2 is a square matrix with dimension 4929, that is used as the initial basis for constrained nonlinear optimization problem represented by GEMAT1 which is the Jacobian matrix for an approximately 2400 bus system in the Western United States. M2 is a sparse matrix with a medium condition number, as presented in Table 1. It is important to mention that no special data storage is used for sparse matrices. They are always stored as dense matrices.

The third matrix tested was M3. As shows Table 2, it is a dense matrix with dimension 66. This matrix is used in the generalized eigenvalue problem $Kx = \lambda Mx$, where M3 is matrix K and matrix M is BCSSTM02 [4], from BCSSTRUC1[4] set. This matrix is used in dynamic analysis in structural engineering.

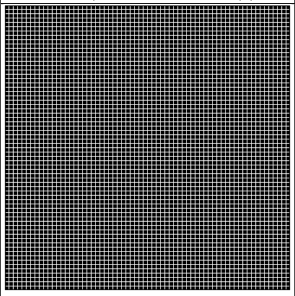
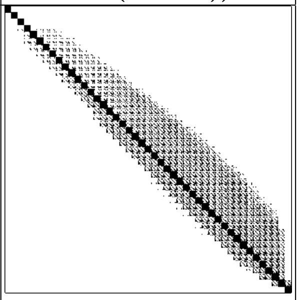
M3 (BCSSTK02))	**M4 (MCFE))**
	
problem: Structural eng. problem size: $n = 66$ sparsity: $nnz = 2211$ cond. number: 1.3e+04 Frobenius norm: 5.3e+04	problem: Astrophysics problem size: $n = 765$ sparsity: $nnz = 24382$ cond. number: 1.7e+14 Frobenius norm: 2e+17

Table 2. Sparsity plots and properties of the test-matrices M3 and M4

Table 2 also presents the properties of matrix M4. It is a sparse matrix with dimension 765, and presents a very high condition number. This matrix is used in the real application of nonlinear radiative transfer and statistical equilibrium in astrophysics.

3.3 Numerical results

This section discusses the results of a set of experiments using the four different matrices presented in the previous section. The first analysis is based on the accuracy obtained using different grid sizes. After that, the results of performance are shown.

[4] http://math.nist.gov/MatrixMarket

Accuracy The accuracy for nine processes and different grids for test case M1 and M2 is presented in Figure 1. The comparison between our PBLAS algorithm and the verified results of the C-XSC-algorithm show that there are constellations of grid processors, namely when the grid is $np \times 1$, where the accuracy of the parallel computation is as precise as the sequential one. In all other cases, that is when the grid of processors is not $np \times 1$, the results present a better accuracy, suggesting that the accuracy depends on the grid. The optimal accuracy for a fixed number of processes can be found by a $1 \times np$ grid. In general we observed that the more columns the processes grid have, the better is the accuracy.

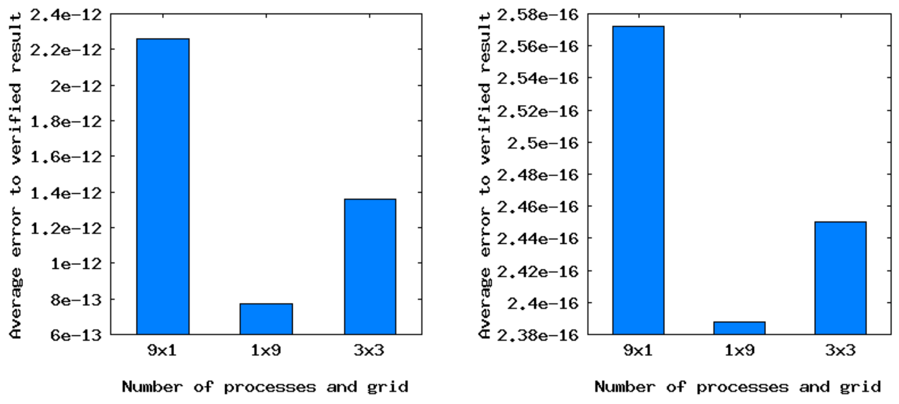


Fig. 1. Average error to the verified result for three different grids of processes and a fixed number of nine processes for the test cases M1 (left plot) and M2 (right plot).

Let us investigate our example in the light of Section 2.1 using, for simplicity, a number of four processes. Let the dimension of the matrix be n, the number of processes $np = 4$, the grid of the processes $nr \times nc$ and the block size $nb := \frac{n}{np}$. Then for the case of an 2×2 grid of processes, the distribution follows the scheme in the example in section 2.2. For a 1×4 and 4×1 grid, let us use the notation P^a_{Bc}, where a (from 1 to np) represents the number of the processes containing the data, B denotes if it is a matrix(M) or a vector(v) and c is the number of the data block related to one processor. The data distribution is as presented in tables 3 and 4.

Analyzing table 3 and 4, it is possible to notice that in the computation based on a $np \times 1$ grid, each entry of the result vector is computed in just one different process, which means that the summation is done like in the simple approach from Section 2.1.

$$\left(\begin{array}{c|c|c|c} P^0_{M0} & P^1_{M0} & P^2_{M0} & P^3_{M0} \\ \hline P^0_{M1} & P^1_{M1} & P^2_{M1} & P^3_{M1} \\ \hline P^0_{M2} & P^1_{M2} & P^2_{M2} & P^3_{M2} \\ \hline P^0_{M3} & P^1_{M3} & P^2_{M3} & P^3_{M3} \end{array}\right) \left(\begin{array}{c} P^0_{v0} \\ \hline P^0_{v1} \\ \hline P^0_{v2} \\ \hline P^0_{v3} \end{array}\right)$$

(a) Grid (1×4)

$$\left(\begin{array}{c|c|c|c} P^0_{M0} & P^0_{M1} & P^0_{M2} & P^0_{M3} \\ \hline P^1_{M0} & P^1_{M1} & P^1_{M2} & P^1_{M3} \\ \hline P^2_{M0} & P^2_{M1} & P^2_{M2} & P^2_{M3} \\ \hline P^3_{M0} & P^3_{M1} & P^3_{M2} & P^3_{M3} \end{array}\right) \left(\begin{array}{c} P^0_{v0} \\ \hline P^1_{v0} \\ \hline P^2_{v0} \\ \hline P^3_{v0} \end{array}\right)$$

(b) Grid (4×1)

Table 3. Data distribution for two different grids of four processes

The structure of the result distribution is shown bellow. The leading entry is the position of the resulting vector, followed by the explanation of which parts were combined to compute the result:

$$\left(\begin{array}{c} P^0(P^0_{M0}P^0_{v0} + P^1_{M0}P^0_{v1} + P^2_{M0}P^0_{v2} + P^3_{M0}P^0_{v3}) \\ \hline P^0(P^0_{M1}P^0_{v0} + P^1_{M1}P^0_{v1} + P^2_{M1}P^0_{v2} + P^3_{M1}P^0_{v3}) \\ \hline P^0(P^0_{M2}P^0_{v0} + P^1_{M2}P^0_{v1} + P^2_{M2}P^0_{v2} + P^3_{M2}P^0_{v3}) \\ \hline P^0(P^0_{M3}P^0_{v0} + P^1_{M3}P^0_{v1} + P^2_{M3}P^0_{v2} + P^3_{M3}P^0_{v3}) \end{array}\right)$$

(a) Grid (1×4)

$$\left(\begin{array}{c} P^0(P^0_{M0}P^0_{v0} + P^0_{M1}P^1_{v0} + P^0_{M2}P^2_{v0} + P^0_{M3}P^3_{v0}) \\ \hline P^1(P^1_{M1}P^0_{v0} + P^1_{M1}P^1_{v0} + P^1_{M1}P^2_{v0} + P^1_{M1}P^3_{v0}) \\ \hline P^2(P^2_{M2}P^0_{v0} + P^2_{M2}P^1_{v0} + P^2_{M2}P^2_{v0} + P^2_{M2}P^3_{v0}) \\ \hline P^3(P^3_{M0}P^0_{v0} + P^3_{M1}P^1_{v0} + P^3_{M2}P^2_{v0} + P^3_{M3}P^3_{v0}) \end{array}\right)$$

(b) Grid (4×1)

Table 4. Processes which contain the final result and parts from which it is computed

The advanced approach can be seen by looking on the $1 \times np$ grid where the final result will be placed on process one, but there are intermediate results on every process leading to a higher quality in the computed result. This suggests that the number of columns of the processor grid is responsible for the granularity of the computation - a higher numbers of columns can lead to better accuracy. So it is not astonishing that a symmetric grid produces results with an intermediate precision (bounded by the other grids).

It is also possible to notice that the results for the application based problem M2 show that for all grid sizes the average error is less than $2.58e-16$ which is excellent considering the double precision format.

Figure 2 presents the average error for matrix M3 and M4. Based on the M3 graphic, we can see that, even for small problems, the data distribution and the inducted computations can have an impact on the result. The average error to the verified result, depending on the grid configuration, differs in about one magnitude.

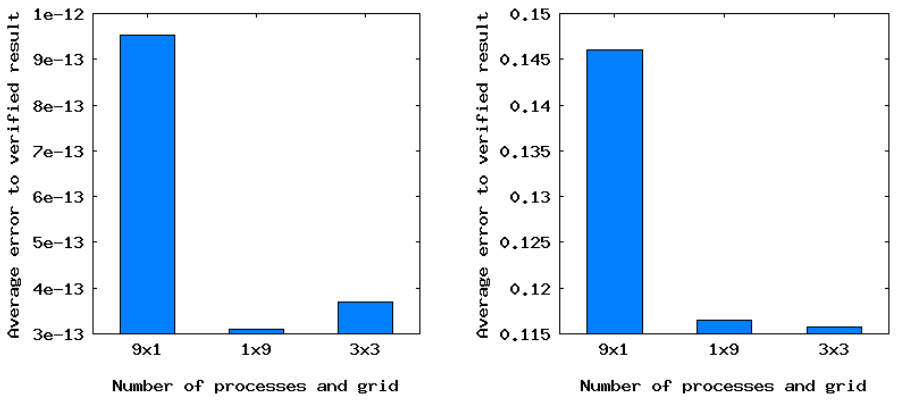


Fig. 2. Average error to the verified result for three different grids of processes and a fixed number of nine processes for the test cases M3 (left plot) and M4 (right plot).

For the test case M4 we observe that the 9×1 grid delivers, analogue to all other experiments, the most inaccurate result. The fact that the 3×3 grid is a little more accurate than the 1×9 grid might be astonishing on the first view but this is possible because the mathematical theory gives only a upper bound for the rounding error propagation.

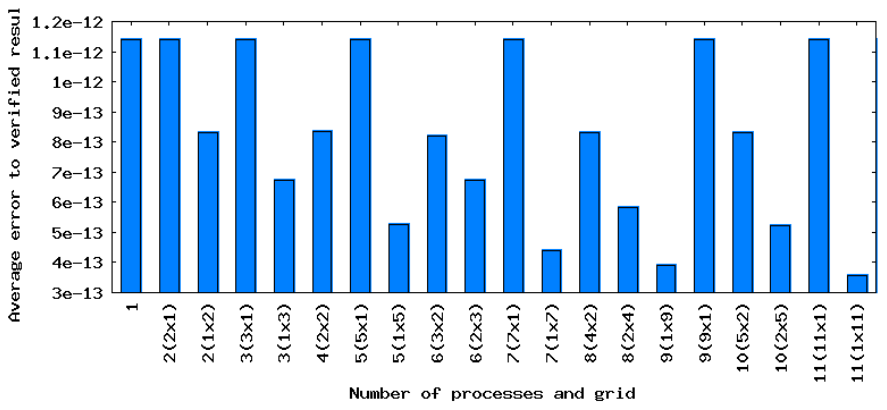


Fig. 3. Average error to the verified result for different grids and different numbers of processes. A matrix similar to M1 but with dimension 8192 was taken for the experiments.

The results in Figure 3 show for different number of processes and grids of processes the average error to the verified result based on a matrix with dimension 8192 and input data generated like in M1. For all grids $np \times 1$ the accuracy is like in the sequential case and independent of the number of processes. The plot shows that the more processes are used the better is the result. It can be seen that the larger the number of processes, the better is the accuracy, following a logarithmic behavior, which corresponds to the theoretical findings.

Performance This section presents the performance analysis considering matrix M1. The performance analysis for matrices 2 to 4 were not discussed since they have small dimensions. In this case it is not worth to parallelize the multiplication, since the program would spend more time communicating among the processors than computing the result. And it would maybe increase the computational time instead of speedup the computation. Since matrix 1 has dimension 16000, it is the natural choice for the performance test.

Figure 4 shows that directly interrelated to the grid is the processing speed. It is possible to notice that the computational time for the same problem size is very different depending on the grid.

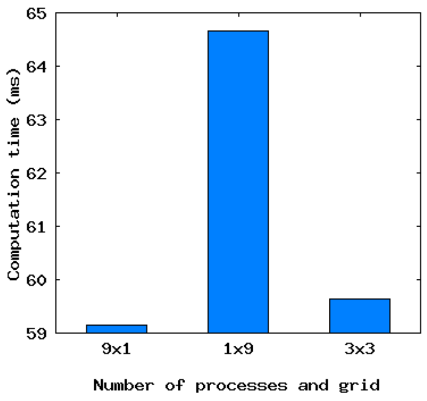


Fig. 4. Commutation time for three different grids of processes and a fixed number of nine processes for the test case M1.

This performance variation can be explained by the fact that different grids communicate differently. The amount, length and topology of such communication have a significant impact on the performance [6]. In Table 3 is possible to notice that some of the data that a processes need may be stored in another processes, and therefore the processes need to communicate before the computation to send/receive parts these data. This occurs also after the computation, when parts of the result have to be collected from each processor and accumulates so that we found the result. This necessity can be seen in Table 4, for the 1×4 grid.

The communication load-balance is optimal if it is equally distributed on all processes. The impact on the performance depends significantly on the underlying hardware (interconnect, memory bandwidth etc.). This means that not a single process is sending or receiving a big bunch of data to all other processes, but that all processes are sending little bunches of data to all other processes. The data has to be divided equally among all processes.

For the grids shown above, the structure of the communication is:

(a) Grid (1×4)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*length*	*sender* $\rightarrow$ *receiver*	*length*
$P^0 \rightarrow P^1$	nb	$P^1 \rightarrow P^0$	n
$P^0 \rightarrow P^2$	nb	$P^2 \rightarrow P^0$	n
$P^0 \rightarrow P^3$	nb	$P^3 \rightarrow P^0$	n

(b) Grid (4×1)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*length*	*sender* $\rightarrow$ *receiver*	*length*
$P^0 \rightarrow P^1$	nb	$-$	
$P^0 \rightarrow P^2$	nb	$-$	
$P^0 \rightarrow P^3$	nb	$-$	
$P^1 \rightarrow P^0$	nb	$-$	
$P^1 \rightarrow P^2$	nb	$-$	
$P^1 \rightarrow P^3$	nb	$-$	
$P^2 \rightarrow P^0$	nb	$-$	
$P^2 \rightarrow P^1$	nb	$-$	
$P^2 \rightarrow P^3$	nb	$-$	
$P^3 \rightarrow P^0$	nb	$-$	
$P^3 \rightarrow P^1$	nb	$-$	
$P^3 \rightarrow P^2$	nb	$-$	

(c) Grid (2×2)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*length*	*sender* $\rightarrow$ *receiver*	*length*
$P^0 \rightarrow P^2$	$2 * nb$	$P^1 \rightarrow P^0$	$n/2$
$P^2 \rightarrow P^1$	$2 * nb$	$P^3 \rightarrow P^2$	$n/2$
$P^2 \rightarrow P^3$	$2 * nb$		

4 Final Remarks and Future Work

This paper presents benchmarks to analyze the influence of the process-grid using the two-dimensional block cyclic distribution on performance and accuracy. Tests show that the process-grid has a significant impact on both, but in a different way. The experiments suggested that the more columns the grid has, the better is the accuracy. However, this is not true for the performance, in which the effect is the opposite: the more columns the grid has, the worse is the performance. For symmetric grids, we found a good performance, due to a better balance in the communication process, with a little less accuracy. These aspects were validated through the theory of rounding error and many experiments.

As a future work, more tests should be performed, considering different libraries and self coded implementations of PBLAS routines. Some testes concerning the hardware platforms (CPUs, GPUs and other accelerators) and compilers are also planned.

In addition, more experiments in the context of complete solvers should be performed. The idea is to present a strategy that use the influence of the load balance among the processes as a guide to find a compromise between accuracy and performance. This compromise will depend on the application needs.

References

1. L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed memory computers Design Issues And Performance. In *SUPERCOMPUTING '96*. IEEE Computer Society, 1996.
2. L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
3. James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numer. Math.*, 108(1):59–91, 2007.
4. James Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2):325–351, 2006.
5. Jim Demmel and Jack Dongarra. LAPACK 2005 prospectus: Reliable and scalable software for linear algebra computations on high end computers. LAPACK Working Note 164, February 2005. UT-CS-05-546, February 2005.
6. J. Dongarra and D. Walker. Lapack working note 58: The design of linear algebra libraries for high performance computers. Technical Report UT-CS-93-188, Knoxville, TN, USA, 1993.
7. Carter Edwards, Po Geng, Abani Patra, and Robert Van De Geijn. Parallel matrix distributions: Have we been doing it all wrong?, 1996.
8. L. Giraud, A. Haidar, and L. T. Watson. Mixed-precision preconditioners in parallel domain decomposition solvers. Technical Report TR/PA/06/84, CERFACS, Toulouse, France, 2006. Also appeared as IRIT Technical report ENSEEIHT-IRIT RT/APO/06/08.
9. R. Hammer, D. Ratz, U. Kulisch, and M. Hocks. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems.* Springer-Verlag New York, Secaucus, NJ, USA, 1997.
10. R. Klatte, U. Kulisch, C. Lawo, R. Rauch, and A. Wiethoff. *C-XSC- A C++ Class Library for Extended Scientific Computing.* Springer-Verlag Berlin, 1993.
11. U. Kulisch. and W. L. Miranker. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.
12. Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113, New York, NY, USA, 2006. ACM.
13. LAPACK. Linear Algebra Package. http://www.cs.colorado.edu/ jessup/lapack/. visited in 09th February 2009.
14. Peter Linz. Accurate floating-point summation. *Commun. ACM*, 13(6):361–362, 1970.
15. Majed Sidani and Bill Harrod. Parallel matrix distributions: Have we been doing it all right? Technical report, Knoxville, TN, USA, 1996.