NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. 2015; **22**:299–316
Published online 19 August 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nla.1950

An efficient approach to solve very large dense linear systems with verified computing on clusters

Mariana Kolberg[1,*,†], Gerd Bohlender[2] and Luiz Gustavo Fernandes[3]

[1]*Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*
[2]*Fakultät für Mathematik, Karlsruhe Institut of Technology, Karlsruhe, Germany*
[3]*Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil*

SUMMARY

Automatic result verification is an important tool to guarantee that completely inaccurate results cannot be used for decisions without getting remarked during a numerical computation. Mathematical rigor provided by verified computing allows the computation of an enclosure containing the exact solution of a given problem. Particularly, the computation of linear systems can strongly benefit from this technique in terms of reliability of results. However, in order to compute an enclosure of the exact result of a linear system, more floating-point operations are necessary, consequently increasing the execution time. In this context, parallelism appears as a good alternative to improve the solver performance. In this paper, we present an approach to solve very large dense linear systems with verified computing on clusters. This approach enabled our parallel solver to compute huge linear systems with point or interval input matrices with dimensions up to 100,000. Numerical experiments show that the new version of our parallel solver introduced in this paper provides good relative speedups and delivers a reliable enclosure of the exact results. Copyright © 2014 John Wiley & Sons, Ltd.

Received 6 May 2013; Revised 11 July 2014; Accepted 22 July 2014

KEY WORDS: interval arithmetic; verified computing; linear systems; parallel algorithms

1. INTRODUCTION AND MOTIVATION

Many real-life problems need numerical methods for their simulation and modeling. A large number of these problems can be solved through a dense linear system of equations. Therefore, the solution of systems like

$$Ax = b \tag{1}$$

with a $n \times n$ matrix $A \in \mathbb{R}^{n \times n}$ and a right-hand side $b \in \mathbb{R}^n$ is very common in numerical analysis. Many different numerical algorithms contain this kind of task as a subproblem [1–3].

There are numerous methods and algorithms that compute approximations to the solution x in floating-point arithmetic. However, usually, it is not clear how good these approximations are. In general, it is not possible to answer this question with mathematical rigor if only floating-point approximations are used.

The use of verified computing appears as a possible solution to find bounds of the exact result. An important property of verified computing is that for a given problem, it is proved, with the aid of a computer, that there exists a (unique) solution of a problem within computed bounds [4]. This makes a qualitative difference in scientific computations, as it provides reliable results.

*Correspondence to: Mariana Kolberg, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Caixa Postal 15064, Cep 91501-970, Porto Alegre, Brazil.
†E-mail: mariana.kolberg@inf.ufrgs.br

Verified computing is based on interval arithmetic. The advantage of using interval arithmetic lies in the possibility of achieving verified results on computers. In particular, rounded interval arithmetic allows rigorous enclosures for the ranges of operations and functions [5]. An enclosure can be defined as an interval that contains the exact result. The two most frequently used arithmetic representations for intervals over real numbers are infimum–supremum and midpoint–radius.

However, finding a verified result is a more costly task than finding an approximation of the exact result using traditional methods. In order to compute an enclosure of the exact result of a linear system, more floating-point operations are necessary, consequently increasing the execution time.

To compensate for the lack of performance of such verified algorithms, some works suggest the use of midpoint–radius arithmetic to achieve better performance. This is possible because case distinctions in interval multiplication and switching of rounding mode in inner loops are not necessary. Therefore, only pure floating-point matrix multiplications are performed, and for those, the fastest algorithms available might be used (e.g., Basic Linear Algebra Subprograms (BLAS)) [6, 7].

In previous works [8–12], a verified algorithm for solving linear systems was parallelized for clusters of computers using the midpoint–radius arithmetic (Section 4 introduces more details on the choice of using midpoint–radius arithmetic) along with the numerical libraries Scalable Linear Algebra Package (ScaLAPACK) and Parallel Basic Linear Algebra Subprograms (PBLAS). Clusters of computers are considered as a good option to achieve better performance without using parallel programming models oriented to very expensive machines [13]. This solution presented significant speedup in the verified solution of linear systems. However, because of memory issues, it only managed to solve linear systems up to dimension 10,000.

To overcome this limitation, an approach to solve very large dense linear systems is proposed in this paper. This approach enabled our parallel solver to compute huge linear systems with point or interval input matrices with a dimension up to 100,000.

A formal definition of the problem solved in this paper is given as follows. Consider a set $\mathbb{F}$ of real floating-point numbers being $\mathbb{F} \subseteq \mathbb{R}$. Define the set of interval floating-point numbers $\mathbb{IF} := \{< \tilde{a}, \tilde{\alpha} >:= \tilde{a}, \tilde{\alpha} \in \mathbb{F}, \tilde{\alpha} \geqslant 0\}$. Set $< \tilde{a}, \tilde{\alpha} >:= \{x \in \mathbb{R} : \tilde{a} - \tilde{\alpha} \leqslant x \leqslant \tilde{a} + \tilde{\alpha}\}$. Then, $\mathbb{IF} \subseteq \mathbb{IR}$. In this case, a pair of floating-point numbers $\tilde{a}, \tilde{\alpha} \in \mathbb{F}$ describes an infinite set of real numbers for $\tilde{\alpha} \geqslant 0$ ($\tilde{a}$ is the midpoint, and $\tilde{\alpha}$ is the radius of the interval). An implementation of the verified solution of systems like $[A]x = [b]$ with a very large matrix $[A] \in \mathbb{IF}^{n \times n}$ and a right-hand side $[b] \in \mathbb{IF}^{n}$ is presented in this paper. It is important to highlight that solving an interval system $[A]x = [b]$ means to compute bounds for $\Sigma([A], [b]) := \{x \in \mathbb{R} | \exists A \in [A], b \in [b] \text{ with } Ax = b\}$ [6].

In a bigger scenario, the major contribution of our research is to enable the solution of very large linear systems with verified computing without using interval libraries. This goal was achieved through the use of the midpoint–radius interval arithmetic, directed roundings and parallel computing techniques. Furthermore, this research intends to provide a free, fast, reliable and accurate solver for very large dense linear systems.

Section 2 describes the related work about solvers for linear systems using verified computing and/or high-performance techniques. Section 3 describes the verified computing background, and Section 4 discusses the interval arithmetic behind this implementation. The main aspects of the previous implementation are detailed in Section 5. The discussion of our solution for very large systems is described in Section 6. The case studies and experimental results are presented in Sections 7 and 8, respectively. Finally, Section 9 concludes this work.

2. RELATED WORK

The solution of large (dense or sparse) linear systems is considered as an important part of numerical analysis and requires a large amount of computations [14, 15]. More specifically, large dense problems arise from physics, for example, in problems such as modeling effect of radio frequency heating of plasmas in fusion applications and modeling high-resolution three-dimensional wave scattering problems using the boundary element formulation [16]. Verification methods for linear systems are frequently based on Brouwer's fixed-point theorem; this leads to iterative solution methods in a

DOI: 10.1002/nla

natural way [17, 18]. The most time-consuming operations in iterative methods for solving linear equations are inner products, successive vector updates, matrix–vector products and also iterative refinements [19, 20].

Aiming at better performance of numerical methods, many libraries were developed. Among them, a widely used one is Linear Algebra Package (LAPACK) [21], a FORTRAN library for numerical linear algebra. The numerical algorithms in LAPACK are based on BLAS routines. The BLAS [22] are routines that provide standard building blocks for performing basic vector and matrix operations. BLAS routines are efficient, portable and widely available [23].

A parallel version of LAPACK is called ScaLAPACK library and includes a subset of LAPACK routines redesigned for distributed memory computers. Like LAPACK, ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of ScaLAPACK library are distributed memory versions of the Levels 1, 2 and 3 PBLAS and a set of Basic Linear Algebra Communication Subprograms for communication tasks that frequently arise in parallel linear algebra computations. In the ScaLAPACK routines, all inter-processor communications occur within PBLAS and Basic Linear Algebra Communication Subprograms.

These libraries present very good performance for linear algebra problems. However, they do not deal with verified computing, delivering only a floating-point approximation of the result.

There is a multitude of tools and algorithms that provide verified computing. Among them, an option is C for eXtended Scientific Computing (C-XSC) [17]. C-XSC is a free and portable programming environment for C++ programming language, offering high accuracy and automatically verified results. This programming tool allows many standard problems to be solved with reliable results. The MATLAB toolbox [24] for self-verified algorithms, INTLAB [25], also provides verified routines.

Both libraries present advantages and drawbacks. C-XSC and INTLAB present enclosures of the exact result. However, to use INTLAB solvers, it is necessary to have the MATLAB environment installed on the machine. Because MATLAB is a commercial mathematical tool, to use it in a large cluster would represent an extra cost. Another issue, as far as we know, is that there is no parallel version of INTLAB routines. On the other hand, C-XSC is a free library that can be easily installed in clusters.

Because INTLAB is based on BLAS, it presents a very good performance. C-XSC used to present a worse performance owing to its overhead of providing a full environment for interval computations. A new version of C-XSC, also based on BLAS, was released, and its performance has improved much compared with that of the older version [26].

Several works have been developed to increase the accuracy of the verified computed results when solving a linear system using INTLAB. In [4, 27, 28], Rump discusses methods and presents possibilities to trade overestimation against computational effort. Revol and Nguyen propose in [29–31] the use of more floating-point matrix products for obtaining better accuracy in the computed results.

Some works related to numerical analysis of parallel implementations using verified computing were found. There are some works that focus on verified computing [32] and both verified computing and parallel implementations [33, 34], but these implement other numerical problems or use a different parallel approach. The ideas presented in [6] and implemented in [10] were also implemented as a solver for C-XSC [35, 36].

Finally, it is worthy to mention that a working group has been discussing since 2008 the interval standardization and definition of an IEEE standard [37]. The successful application of interval arithmetic appears to be somewhat difficult for nonexpert users. Indeed, the proper use of interval arithmetic requires attention to details beyond those of usual numerical computations because otherwise the results may be useless, either being invalid (i.e., not enclosing the true results) or overly pessimistic [38]. The interval standardization is an important step toward a guaranteed use of verified computing in hardware and software, as it will establish the basis for interval computations.

DOI: 10.1002/nla

3. VERIFIED LINEAR SYSTEM SOLVER

There are numerous methods and algorithms that compute approximations to the solution of a linear system using floating-point arithmetic. However, it is not possible to answer how good an approximation delivered by a conventional numerical algorithm is. These problems become especially difficult if the matrix A is ill conditioned. In general, it is not possible to answer these questions with mathematical rigor if only floating-point approximations are used.

The use of verified computing appears as an alternative to achieve more reliable results [39]. Verified computing is based on interval arithmetic and directed roundings associated with suitable algorithms [40]. If the solution is not found after a few iterations, for example, if the matrix is singular, the algorithm will let the user know. Some authors suggest that verified computing can be implemented without the use of directed roundings [41, 42]. In that case, only the rounding-to-nearest mode is used, and intervals are inflated a little. This however may lead to less accurate results.

Algorithm 1 presents a verified method for solving linear systems that can be found in [43] based on the enclosure theory described in [18].

Algorithm 1 Enclosure of a square linear system.

1: {STEP 1—compute an approximate inverse using LU decomposition algorithm}
2: $R \approx mid(A)^{-1}$
3: {STEP 2—compute the approximation of the solution}
4: $\tilde{x} \approx R \cdot mid(b)$
5: {STEP 3—compute enclosure for the residuum}
6: $[z] \supseteq R([b] - [A]\tilde{x})$
7: {STEP 4—compute enclosure for the iteration matrix}
8: $[C] \supseteq (I - R[A])$
9: {STEP 5—verification step}
10: $[w] := [z]$, $k := 0$, $y := [0.0, 0.0]$ {initialize machine interval vector}
11: **while not** ($[w] \overset{\circ}{\subset} [y]$ or $k > 10$) **do**
12: $[y] := [w]$
13: $[w] := [z] + [C][y]$
14: $k++$
15: **end while**
16: **if** $[w] \overset{\circ}{\subset} [y]$ **then**
17: $\Sigma(A, b) \subseteq \tilde{x} + [w]$ {the solution set (Σ) is contained in the solution found by the method}
18: **else**
19: no verification
20: **end if**
{$\overset{\circ}{\subset}$ denotes the interior set}

This enclosure method applies the following interval Newton-like iteration:

$$x_{k+1} = Rb + (I - RA)x_k, k = 0, 1, \ldots \tag{2}$$

to find a zero of $f(x) = Ax - b$ with an arbitrary starting value x_0 and an approximate inverse $R \approx A^{-1}$ of A. If there is an index k with $[x]_{k+1} \overset{\circ}{\subset} [x]_k$ (the $\overset{\circ}{\subset}$ operator denotes that $[x]_{k+1}$ is included in the interior of $[x]_k$), then the matrices R and A are regular [18], and there is a unique solution x of the system $Ax = b$ with $x \in [x]_{k+1}$. In this research, it is assumed that $Ax = b$ is a dense square linear system.

One important advantage of the presented algorithm is the ability to find a solution even for ill-conditioned problems, while most algorithms may lead to an incorrect result when it is too ill conditioned (above condition number 10^8). The accuracy of the results in many cases depends on the condition number. However, the result of this method is always an inclusion of the exact result.

DOI: 10.1002/nla

It is important to notice that this method is based on interval operations for vectors and matrices. Naturally, these operations have a high computational cost because they deal with a larger amount of data than ordinary operations. Furthermore, this kind of operations is not supported in hardware (neither for floating-point approximation nor for interval arithmetic). In this scenario, it is necessary to find the most suitable interval representation to use along with highly optimized software libraries (e.g., PBLAS and ScaLAPACK) in order to speed up the algorithm implementation employing high-performance computing platforms.

4. INTERVAL ARITHMETIC FOR HIGH-PERFORMANCE COMPUTING

The two most frequently used representations for intervals over $\mathbb{R}$ (the set of real numbers) are the infimum–supremum and midpoint–radius representations [44]. Infimum–supremum representation is defined as follows:

$$[a_1, a_2] := \{x \in \mathbb{R} : a_1 \leqslant x \leqslant a_2\} \tag{3}$$

for some $a_1, a_2 \in \mathbb{R}$, $a_1 \leqslant a_2$, where $\leqslant$ is the partial ordering $x \leqslant y$. The midpoint–radius representation is defined as follows:

$$\langle a, \alpha \rangle := \{x \in \mathbb{R} : |x - a| \leqslant \alpha\} \tag{4}$$

for some $a \in \mathbb{R}$, $0 \leqslant \alpha \in \mathbb{R}$. Although the two representations are identical when dealing with real intervals, it is important to mention that when dealing with floating-point intervals, different representations are not always identical because of rounding issues.

Infimum–supremum appears today as the mostly used representation for intervals on computers. There are two main reasons for that. First, the standard definition of midpoint–radius arithmetic causes overestimation for multiplication and division, and second, the computed midpoint of the floating-point result of an operation is, in general, not exactly representable in floating point, thus again causing overestimation and additional computational effort. However, in [6], Rump shows that the overestimation of operations using midpoint–radius representation compared with the result of the corresponding power set operation is limited by at most a factor of 1.5 in the radius.

Despite these reasons, both infimum–supremum and midpoint–radius representations have advantages and disadvantages. Some intervals are better represented in one arithmetic and have an overestimation in the other. Therefore, additional studies must be conducted to decide which is the best arithmetic to use in the context of solving linear systems with verified computing.

A detailed description on how to implement midpoint–radius interval arithmetic using pure floating-point operations and high-performance libraries can be found in [6]. As introduced in Section 1, the great advantage of using midpoint–radius arithmetic is that case distinctions in interval multiplications and switching of rounding mode in inner loops are not necessary. Only pure floating-point matrix multiplication operations are enough to perform the matrix interval multiplications [6]. This advantage is reinforced because for those multiplications, the fastest algorithms available may be used. Considering that, we could adopt the PBLAS library to speed up the midpoint–radius interval operations necessary to carry out some steps of Algorithm 1.

It is important to highlight that behind every midpoint–radius interval operation, more than one floating-point operation is performed using the appropriate rounding mode. For instance, in Algorithm 2, it is possible to see that four operations are executed to perform the midpoint–radius interval matrix multiplication defined in [6]. For those operation, the routine PDGEMM of PBLAS would be executed three times in Algorithm 2: one for Step 1, one for Step 2, and one for Step 4. In Algorithm 2, $\tilde{c}$ is the midpoint, and $\tilde{\gamma}$ is the radius of the output matrix. Moreover, in order to represent the directed rounding modes, the symbol ∇ denotes rounding downwards and Δ denotes rounding upwards [40]. The matrix multiplication $R * [A]$ from Step 4 (Algorithm 1) needs the operations of Algorithm 2 to be completed.

To check which is the best arithmetic to be used in this method, a sequential algorithm of a verified linear system solver was implemented using each arithmetic. As input data, the algorithm accepts

DOI: 10.1002/nla

Algorithm 2 Midpoint–radius matrix multiplication

1: $\tilde{c_1} = \nabla(R \cdot mid(A))$
2: $\tilde{c_2} = \Delta(R \cdot mid(A))$
3: $\tilde{c} = \Delta(\tilde{c_1} + 0.5(\tilde{c_2} - \tilde{c_1}))$
4: $\tilde{\gamma} = \Delta((\tilde{c} - \tilde{c_1}) + |R| \cdot rad(A))$

point and interval vectors and matrices. Performance tests showed that the midpoint–radius algorithm needs approximately the same time to solve a point linear system and an interval linear system. On the other hand, when using infimum–supremum arithmetic, the algorithm needs much more time for solving interval linear systems. This effect can be explained because the infimum–supremum interval multiplication must be implemented with case distinctions and optimized functions like BLAS cannot be used. Based on the lack of optimized interval libraries, midpoint–radius arithmetic seems to be the best choice for achieving better performance through the use of highly optimized numerical libraries (e.g., BLAS) [10].

The implementation of Algorithm 1 using midpoint–radius interval arithmetic follows the proposal presented by Rump [6] and uses different rounding modes depending on the operation being performed . Steps 1 and 2 are carried out using LAPACK routines with rounding to nearest. In case we have interval input data, only the midpoint matrix will be used for those steps. Steps 3–5 deal with interval matrix/vector multiplication. These operations are implemented as explained in Algorithm 2. We have used BLAS routines to perform the floating-point vector/matrix operations. Some of those operations are performed with rounding upwards, others with rounding downwards, as proposed in [6]. Technically speaking, the rounding mode is set to ∇ or Δ, and this rounding mode must be applied to all operations until it is changed again.

For the correct behavior of our method, we require that the rounding mode be set (e.g., by a call of the function fesetround) and that this rounding mode be used in all subsequent operations, until it is set differently. If an architecture does not supply this property, it does not conform to the IEEE 754 standard, and we cannot use it. Based on the problem of not having an interval standard implemented in IEEE 754, a work has been performed to develop the IEEE 1788 interval arithmetic standard. This standard is fundamentally based on rounding control and thus excludes any 'half implementation' of IEEE 754 without rounding control or with insecure rounding control. In case directed roundings are not available in a certain environment, one can use the idea proposed by Ogita *et al*. [41] and Revol *et al*. [42].

The possibility of using highly optimized libraries like BLAS and LAPACK appears as a good choice because it is available for almost every computer hardware and it is individually adapted and tuned for specific hardware and compiler configurations. This is specially important when aimed at high-performance computing. Using midpoint–radius arithmetic and directed roundings, we were able to use the proper implementation of highly optimized libraries for the following high-performance hardware: ScaLAPACK for clusters [8–12, 45] and Parallel Linear Algebra Software for Multicore Architectures [46] for multicore [47]. Further studies could investigate how to implement our verified solution for graphics processing units (GPUs) using Matrix Algebra on GPU and Multicore Architectures [48]. The use of these optimized implementations will lead to a better performance in the generation of verified results obtained using interval arithmetic instead of pure floating-point operations over real numbers. This gives an advantage in computational speed, which is difficult to achieve by other implementations.

5. VERIFIED LINEAR SYSTEM SOLVER FOR CLUSTERS

A previous work [10] presented a parallel version of Algorithm 1 for cluster architectures with message passing programming model (MPI) [49] using the highly optimized libraries PBLAS and ScaLAPACK.

The self-verified method presented in Algorithm 1 is divided in several steps. By tests, when dealing with a point linear system with dimension 1000, the computation of R (approximate inverse of

DOI: 10.1002/nla

matrix A) takes more than 50% of the total processing time. Similarly, the computation of the interval matrix $[C]$ that contains the exact value of $I - RA$ (iterative refinement) takes more than 40% of the total time. Both operations deal with matrix multiplication, which requires $O(n^3)$ floating-point operations, and other operations in the algorithm are mostly vector or matrix–vector operations, which require at most $O(n^2)$.

When using midpoint–radius arithmetic, both operations could be implemented using ScaLAPACK (R calculation) and PBLAS (C calculation). Based on that, a parallel version of the self-verified method for solving linear systems was presented in [10]. The solution presented in that paper proposes the following improvements aiming at a better performance:

- Calculate R using just floating-point operations.
- Use the fast and highly optimized libraries: BLAS and LAPACK (sequential version) and PBLAS and ScaLAPACK (parallel version).
- Use midpoint–radius arithmetic (as proposed by Rump [6]).

The parallel implementation presented in [10] has used the following ScaLAPACK/PBLAS routines in Algorithm 1:

- ScaLAPACK
  - *pdgetrf*: for the LU decomposition (matrix R on Step 1)
  - *pdgetri*: to find the approximation of the inverse matrix (matrix R on Step 1)
- PBLAS
  - *pdgemm*: matrix–matrix multiplication (matrix C on Step 4)
  - *pdgemv*: matrix–vector multiplication (Steps 2, 3 and 8 to find the vectors x, z and w)

It is important to point out that this algorithm needs additional memory to store the intermediate matrices and vectors used during the computation. This approach also allows us to preserve the original data. On the other hand, solving a linear system with floating-point computations can be performed with no additional memory allocation. However, the matrix and right-hand side will be destroyed during the computation, and the algorithm will only find an approximation of the exact result.

The experimental results presented in [10] indicate an important gain of performance. The parallel implementation leads to a nearly perfect speedup in a wide range of processor numbers for large dimensions (up to dimension 10,000), which is a very significant result for clusters of computers.

6. APPROACH FOR VERY LARGE SYSTEMS

The good performance achieved by the parallel solution presented in [10] and described in Section 5 can be obtained only with input matrices with dimension up to 10,000. Experiments with matrices with higher dimensions could not be executed because of a lack of memory to allocate them and all the intermediate matrices and vectors needed to compute the verified result.

This limitation is a result of the input matrix memory allocation strategy adopted in the previous work, which concentrates the creation of the input matrix in one node, leading very quickly to memory allocation problems. The natural option would be to allocate only the submatrix of the input matrix on each available node, allowing a better use of the global memory because the original matrix would be partitioned using several local memories.

The main goal of this section is to present the strategy we have used to modify our parallel solver in such a way it is now able to solve very large linear systems using verified computing. The key idea is to optimize the previous algorithm to allocate a portion of matrix A directly in each computing node according to the distribution scheme adopted by the numerical libraries already in use.

PBLAS/ScaLAPACK routines are implemented by supposing the matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [50]. These routines solve the data distribution problem transparently when the whole matrix is available (replicated) in all computing nodes.

DOI: 10.1002/nla

Unfortunately, when the matrix is not available on all computing nodes, PBLAS/ScaLAPACK routines are not able to distribute the data following their internal distribution scheme. Therefore, in order to use the same PBLAS/ScaLAPACK routines but without having a copy of the original matrix in each computing node, the data must be distributed according to the two-dimensional block cyclic distribution.

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 carried out by considering the vector as a column of the matrix. Suppose we have the following 10×10 matrix, a vector of length 10, MB and NB equal to 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 four processors. The process grid would be a 2×2 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 [51]. 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

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

Considerations about choosing the block and grid size can be found in [52]. The grid size is determined by aiming at a square grid, that is, for nine processors, we use a 3×3 grid, as suggested in [53].

To be able to distribute the input data of matrix A to the computing nodes, we had to develop a way to extract the correct portion of matrix A observing the rules of the previously mentioned distribution. This is basically a problem of finding the right indices, use them to collect the data stored in the correspondent position of the input matrix and then assign the values to the right position in the distributed submatrices.

A mapping function was used to establish a correspondence between the global position of matrix A and its local position in matrix MyA (MyA is the name of the local matrix allocated in each computational node; it stores a portion of matrix A).

In the new implementation, matrix elements $A[i, j]$, which are accessible by a given process, are not accessed as elements of the global matrix A, but rather accessed as elements of the submatrix

DOI: 10.1002/nla

MyA of that specific process. Thus, the submatrix MyA is not addressed using the global $[i, j]$ indices. Instead, the global address is converted to a local address in the submatrix MyA (which is stored as a vector). This local address is used in order to access global positions from a given node of the distributed matrix, and *vice versa*. The address conversion is accomplished via the mapping function (Algorithm 3).

Algorithm 3 Mapping global position of A to local positions in MyA.

```
1: mapping(int i, int j, int MB, int NB, int LLD, int NPROW, int NPCOL, int RESULT)
2: int lauxi, lauxj
3: lauxi = i/MB/NPROW
4: lauxj = j/NB/NPCOL
5: RESULT = (LLD*((NB*lauxj) + (j%NB))) + (MB*lauxi) + (i%MB)
```

This function requires, in addition to i (row in the global matrix) and j (column in the global matrix), the number of columns of a block (NB), the number of lines of a block (MB), the number of rows of the submatrix (LLD), the number of rows in the processor grid ($NPROW$) and the number of columns in the processor grid ($NPCOL$). The output result variable will contain the local address in the submatrix MyA and represents the local vector position of the respective global matrix element.

Parallel algorithms that must perform computations with such large matrices usually store data in some kind of secondary memory (e.g., hard disks), and the matrix is loaded to memory in parts on each computing node [54]. Our mapping function is perfectly able to distribute the input data following that strategy. Indeed, this would be the only way to compute real-life problems with our parallel solver. However, real dense input matrices of very large dimensions, for example 100,000, were not available for our experiments. A well-known alternative is to use toy matrices as input data.

To deal with toy matrices, we had two possibilities: (i) generate the entire input matrix, store it in a file and bring parts of the matrix to the computing nodes following a ScaLAPACK distribution scheme during the computation; or (ii) generate directly the portion of the matrix in each node respecting a ScaLAPACK distribution scheme. In both alternatives, performance results would be the same because the operation of loading parts of the input matrix from some kind of secondary memory (e.g., a hard disk) would be necessary only once at the beginning of the computation, having a non-representative cost in the whole computation. Therefore, for the sake of simplicity in the implementation, we have generated parts of matrix A directly in each node, respecting ScaLAPACK's distribution scheme and using the mapping function to correctly match the indices. Nevertheless, it is important to stress out that the first option (the whole input matrix in a file) could also be easily implemented (and it has to be used when dealing with real input matrices) using the mapping function previously described.

7. CASE STUDY—SCENARIO OVERVIEW

This section introduces a case study that is used to evaluate the approach previously presented in Section 6. The main goal of our experiments is to verify whether our solution for clusters is suitable for very large dense linear systems in terms of both performance and accuracy. Secondly, we want to investigate the portability and scalability of our solution by running our experiments over two substantially different cluster environments.

In order to give to the reader a clear view of our test scenario, it is important to highlight that we did not have access to huge real-life dense linear systems data. To overcome this limitation and carry out experiments with very large dense systems, we had to use toy input matrices (as explained in Section 6).

In this section, we also describe the test environment detailing the software and hardware platforms.

DOI: 10.1002/nla

7.1. Input data

As previously mentioned, very large real-life dense input matrices data were not available for our experiments. Therefore, we have used well-known formulas, which are indicated in the literature [55], together with the mapping function to generate the portions of the input matrix $mid(A)$ in each node (Algorithm 3):

- Matrix 1: $A = [a_{ij}]$, $i, j = 1, 2, 3, \ldots, n$, where

$$a_{ij} = \begin{cases} \frac{i}{j} & \text{if } i \leqslant j \\ \frac{j}{i} & \text{if } i > j \end{cases}$$

- Matrix 2: $A = [a_{ij}]$, $i, j = 0, 1, 2, 3, \ldots, (n-1)$, where

$$a_{ij} = \max(i, j)$$

The reader should keep in mind that in our work, each node will contain a part of the original input matrix. In the absence of real-life input data, the simpler way to do that is to allow each node to generate its own $mid(A)$ matrix, called $mid(MyA)$ (which will contain the elements of $mid(A)$ for the specific node). The mapping function, described in Section 6, is responsible for coordinating the data distribution through the nodes, allowing a local generation of $mid(MyA)$ respecting the global view of $mid(A)$. For performance measurements, the midpoint vector $mid(b)$ is generated with random numbers. For accuracy experiments, midpoint vector was generated as follows: $mid(Myb)[i] = 1$.

Based on the way the algorithm is implemented, there is no difference in the performance using interval or point input data. Our implementation allocates the midpoint and radius matrices in both cases. The only difference relies on the fact that for point input data, the full radius matrix $rad(A)$ and the full radius vector $rad(b)$ are initialized with 0.0. Therefore, we have the same amount of operations to solve both interval and point linear systems. Based on that, for the performance tests we have used only point input data.

The performance experiments were executed using four different matrix dimensions: 25,000, 50,000, 75,000 and 100,000. In terms of memory allocation, a matrix with dimension 100,000 composed by double floating-point elements (8 bytes each) needs 80 GB of memory to be stored. This magnitude of input matrices is used in what we call very large or even huge dense linear systems.

Accuracy tests were executed using point and interval input data. First, some tests were performed using Matrix 1 with dimension 15,000 and Matrix 2 with dimension 100,000 with point input data; that is, all elements of the radius matrix were initialized with 0.0. A final experiment was performed to analyze the accuracy of the computed results when using interval input data. In that case, we used a radius of 10^{-12} and tested again Matrix 1 with dimension 15,000.

7.2. Test environments

This section describes the cluster environments we have used to carry out our experiments over very large dense linear systems. Tables I and II illustrate the software and hardware configurations of two clusters located at the University of Karlsruhe (Germany). Many of these features will be key aspects to a better understanding of the results achieved with our implementation.

Table I. Clusters—software.

	XC1	IC1
OS	HP XC Linux	Suse Linux Enterprise 10 SP
BLAS Library	Intel MKL 10.0.011	Intel MKL 10.0.2
Compiler	Intel C/C++ v. 10.0	Intel C/C++ v. 10.1
MPI	MPICH (version 1.2)	OPENMPI (version 1.2.8)

DOI: 10.1002/nla

Table II. Clusters—hardware.

	XC1	IC1
CPU	Intel Itanium2	Intel Xeon Quad Core
Clock frequency	1.5 GHz	2.66 GHz
Number of nodes	108	200
Number of cores	2	2×4
RAM per node	12 GB	16 GB
Cache per node	6 Mb of Level 3 cache	2×4 MB of Level 2 cache
Hard disk	146 GB	4 x 250 GB
Network	Quadrics QsNet II interconnect	InfiniBand 4× DDR
Point-to-point bandwidth	800 MB/s	1300 MB/s
Latency	Low	Very low (below 2 ms)
Theoretical peak performance	1.9 TFLOPS	17.57 TFLOPS

In terms of software (Table I), some differences between the clusters can be identified. The most significant being the different versions of the Math Kernel Library, which is an Intel implementation of the optimized libraries BLAS, LAPACK, PBLAS and ScaLAPACK. Naturally, this is a factor that will affect the performance of our parallel solver for very large dense systems.

Table II shows the hardware configuration of each cluster. For the remainder of this paper, clusters will be referred by the following acronyms: XC1 (which stands for HP XC600) and IC1 (which stands for Instituts Cluster 1). As can be easily noticed, XC1 has a significantly smaller computational power than IC1. The theoretical peak performance that could be achieved by XC1 is 1.9 TFLOPS, while IC1 could achieve 17.57 TFLOPS (almost 10 times faster).

8. EXPERIMENTAL RESULTS

In this section, we present the experimental results achieved by our parallel solver for very large dense linear systems using two sets of input data (Section 7.1) over two different cluster environments (Section 7.2). The results are discussed in the subsequent sections, first in terms of performance and then in terms of accuracy.

8.1. Performance evaluation

As previously described (Section 7.1), the performance evaluation of our parallel solver was carried out by varying the order of input matrix $mid(A)$. Four matrices with different large dimensions (25,000, 50,000, 75,000 and 100,000) were used as input data. For each of those matrices, we ran experiments varying the number of nodes from 8 to 256 (8, 16, 32, 64, 128 and 256). The following performance analysis will be carried out by considering three aspects: memory allocation, portability and scalability.

8.1.1. Memory allocation. Memory allocation is a key aspect to understanding some constraints we have faced. Table III shows the approximate amount of memory needed on each node to store the correspondent part of the input matrix for a single execution of the parallel solver according to the matrix dimension. These values represent the maximum amount of memory needed to compute a part of the input matrix on a single node.

Table III. Memory used per node.

	8	16	32	64	128	256
25,000	4.37 GB	2.18 GB	1.09 GB	546 MB	273 MB	136.5 MB
50,000	17.5 GB	8.75 GB	4.47 GB	2.18 GB	1.09 GB	546 MB
75,000	39.4 GB	19.7 GB	9.84 GB	4.9 GB	2.45 GB	1.23 GB
100,000	70.0 GB	35 GB	17.5 GB	8.75 GB	4.37 GB	2.18 GB

DOI: 10.1002/nla

Many matrices are simultaneously allocated ($NbMat$ stands for the number of matrices simultaneously allocated) in our implementation during the computation of a verified result. For each execution, our method needs to maintain in memory at most seven matrices: matrix A midpoint, matrix A radius, matrix A identity, matrix A inverse and three other temporary/auxiliary matrices to be able to use PBLAS routines. It is necessary to maintain these matrices during the execution because we need them all to compute Step 4 of Algorithm 1 and we also want to preserve the input data. Thus, for a matrix with dimension $MatDim$ running over $NbNodes$ with the basic elements being double floating-point values ($SizeBytes$ equals 8 bytes), we have

$$\text{MemAlloc} = \frac{MatDim^2 \times NbMat \times SizeBytes}{NbNodes} \tag{5}$$

Taking a look at the figures of Table III and considering that clusters XC1 and IC1 have respectively 12 and 8 GB of available memory on each node, clearly, some executions with larger input matrices over a small number of nodes are not possible to be carried out. For instance, matrix dimension 50,000 cannot be partitioned over 8 or 16 IC1 nodes, or matrix dimension 75,000 cannot be stored in executions over 8, 16 or 32 XC1 nodes, and so on. This is the reason why for some of the following tables and graphs, some measurements are missing.

8.1.2. Portability. For the portability analysis, we chose to show the results obtained using as input data Matrix 1 (Section 7.1). Experiments were carried out over the two cluster environments previously described (Section 7.2). The idea here is to analyze the execution times obtained with our parallel solver over two substantially different platforms and identify the factors that directly affect its performance.

Figure 1 shows the execution times (in seconds) obtained to compute Matrix 1 using all four dimensions over a different number of XC1 nodes. In this cluster, the maximum number of nodes we could use was 128 because it has only 216 nodes (108 dual cores). As can be noticed in this figure, the execution times drop consistently each time we double the number of nodes. Crossing these data with those from Table III, we can confirm that all experiments in which more than 12 GB of memory was needed could not be carried out. In these cases, the memory allocation clearly exceeded the available memory on an XC1 single node. There is, however, one exception: matrix dimension 75,000 should be possible to run over 32 nodes because for this experiment, we would need at most 9.84 GB. This unique exception can be explained by the fact that it is a borderline situation. Our memory allocation estimation (Equation 5) offers a quite good approximation of the amount of memory needed in each node to run an experiment. However, it does not consider the following aspects that may be determinant in a borderline case:

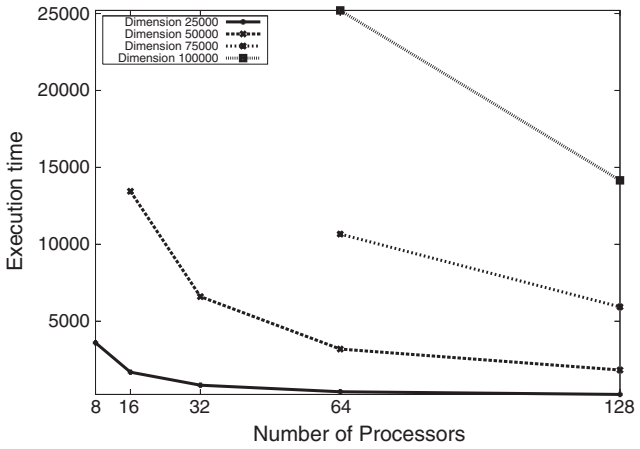

Figure 1. Execution time (s) for solving Matrix 1 on XC1.

DOI: 10.1002/nla

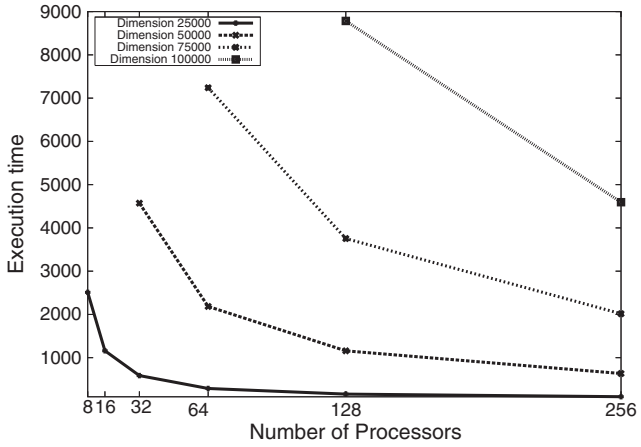

Figure 2. Execution time (s) for solving Matrix 1 on IC1.

- the memory allocated by vectors and other variables needed by the solver;
- the memory that MPI needs for its buffers and internal structures; and
- the memory used by the operating system.

Figure 2 shows the execution times (in seconds) for all dimensions and different numbers of nodes (up to 256 in this case) over cluster IC1. This figure shows a very uniform behavior; there are no borderline exceptions. Note that all experiments that needed more than 8 GB per node could not be carried out.

In order to offer the reader an idea about the differences between the execution times obtained over both clusters, Table IV introduces the execution times (in seconds) to compute each step of Algorithm 1 for Matrix 1 with dimension 100,000 using 128 processors on both XC1 and IC1. As can be noticed in this table, the most expensive operations are Steps 1 and 4, which have complexity $O(n^3)$ and represent around 99.4% of the total execution time. As mentioned before (Section 6), these steps were the ones implemented in parallel with the support of PBLAS and ScaLAPACK library routines.

Another observation that can be extracted from Table IV is related to the better performance achieved by our parallel solver over the IC1 cluster in comparison with XC1. The total execution time over IC1 is around three times faster than over XC1. We believe that these results are due to the following reasons:

- IC1 has a much faster network interconnection.
- The frequency of each individual core is much higher on IC1 than on XC1 (2.66 GHz against 1.5 GHz).
- The Math Kernel Library version is more recent on IC1 than on XC1, offering more optimized routines.

We believe that the portability experiments described earlier strongly indicate that our solution is portable for different cluster environments. Performance gains were possible on both platforms,

Table IV. Execution times (s) for each step of Algorithm 1.

	XC1	IC1
Step 1 (R)	3,973.52	1,469.73
Step 2 ($\tilde{x}$)	1.13	1.33
Step 3 ($[z]$)	1.13	1.33
Step 4 ($[C]$)	10,109.63	3,097.65
Step 5 (verified)	52.79	15.37
Total time	14,159.78	4,594.09

DOI: 10.1002/nla

and they are compatible with the hardware and software differences presented on both clusters. We also could confirm the memory allocation factor (described in Section 8.1.1), which constrains the utilization of our solution over certain platforms depending on the available memory per node.

8.1.3. Scalability. For the scalability evaluation, our option was to carry out experiments with input matrices generated following the formation rules Matrix 1 and Matrix 2 (Section 7.1) over both clusters (IC1 and XC1; Section 7.2). For each cluster, experiments were performed using all matrices dimensions already used in previous sections. Our main goal here was to more closely investigate how our parallel solver behaves when running over a large number of processing nodes.

Naturally, the main metric to evaluate scalability is the speedup factor. However, we cannot solve such large linear systems using just one node because of memory constraints. Therefore, we cannot obtain the sequential execution time for our solver, and consequently, it is not possible to obtain speedups. Thus, the following results were obtained through the computation of relative speedups between the executions with the most number of nodes for each cluster. This means that, for cluster XC1, we have computed relative speedups using 64 and 128 nodes and, for IC1, we have obtained relative speedups using 128 and 256 nodes. It is worth to remember that the maximum relative speedup between two sets of nodes is two considering we are using in both cases exactly the double of nodes for each execution on each different cluster.

In Figure 3, we show the speedup related to the executions using 64 and 128 processors on XC1. As can be seen, the relative speedup is near two for both input matrices. In fact, the speedups are always between 1.6 and 1.8 for each input matrix dimension. Also, for both matrices (1 and 2), results are very close with no significant variation. It is important to point out though that the relative speedup starts to decrease for matrices with dimension 100,000, which indicates that the problem is not scaling well for larger matrices on XC1.

Regarding the results for cluster IC1, we can see in Figure 4 that the relative speedups are scaling very well, even getting closer to the ideal speedup for the experiments with matrices with larger dimensions.

In a quick comparison between the results obtained on both clusters, the obvious conclusion is that our parallel solver obtained better speedups over IC1. This behavior can be explained by the differences between the interconnection networks of both environments. IC1 has a very fast interconnection network using an Infiniband 4× DDR, while XC1 uses a Quadrics QsNet II interconnection. IC1 interconnection has almost 1.5 times the bandwidth of XC1 (1300 MB/s against 800 MB/s), with a very low latency. On XC1, as the grain of the tasks becomes smaller, the communication time represents a larger slice of the execution time, resulting in lower speedups. On the other hand, this fact does not happen on IC1 because its network is very fast and suitable to communication-intensive applications.

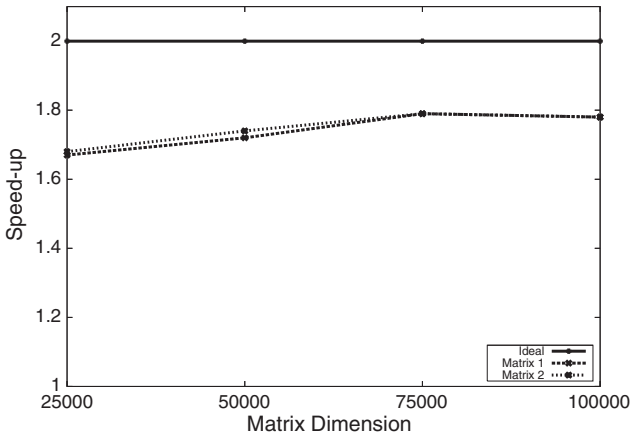

Figure 3. Relative speedup using 64 and 128 processors on XC1.

DOI: 10.1002/nla

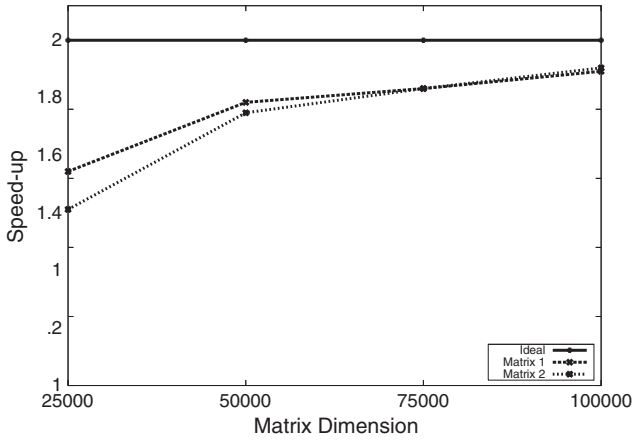

Figure 4. Relative speedup using 128 and 256 processors on IC1.

We had close to linear relative speedup on IC1 over 256 nodes for the highest dimension input matrix. Even for cluster XC1, experiments presented increasing speedups up to dimension 75,000. After that point, the scalability of our parallel solver was limited by the interconnection network capacity. The conclusion on these experiments is that our parallel solver scales very well but it is directly affected by the quality of the interconnection network because communication is a significant factor in our solution.

8.2. Accuracy analysis

In this section, we will do a qualitative comparison regarding the accuracy of the results. In the first experiment, we have used Matrix 1 as the midpoint matrix (which will be generated in each node according to the mapping function). The local midpoint vector ($mid(Myb)$) was generated as follows: $mid(Myb)[i] = 1$. The dimension of the matrix is 15,000, and the condition number is $6.14649 \times 10^{+08}$. The numerical results are presented in Table V.

We also performed an experiment using input Matrix 2 on IC1 cluster using dimension 100,000. The local midpoint vector ($mid(Myb)$) was generated as follows: $mid(Myb)[i] = 1$. Table VI presents the first four elements of the solution vector found for solving Matrix 2, which has a condition number of $3.99996 \times 10^{+10}$.

A third experiment was performed using an interval input matrix. Let $mid(A)$ be a matrix with dimension 15,000 generated by Matrix 1 formula and $mid(b)$ a 15,000-element vector, generated as follows: $mid(Myb)[i] = 1$. Both $rad(A)$ and $rad(b)$ were filled with the value 10^{-12}. The solution for the first 10 positions of vector $[x]$ is presented in Table VII. As expected, the verified

Table V. Numerical results found for Matrix 1.

Res	Midpoint	Radius	Infimum–supremum representation
$x[0] =$	0.666666666673327968	$6.246669 \cdot 10^{-09}$	[0.666666660426658009, 0.666666672919997927]
$x[1] =$	0.266666673443532731	$1.249741 \cdot 10^{-08}$	[0.266666660946117573, 0.266666685940947890]
$x[2] =$	0.171428542393218419	$1.875048 \cdot 10^{-08}$	[0.171428523642730396, 0.171428561143706443]
$x[3] =$	0.126984150895482345	$2.500335 \cdot 10^{-08}$	[0.126984125892125488, 0.126984175898839202]
$x[4] =$	0.101010086536853047	$3.125822 \cdot 10^{-08}$	[0.101010055278631855, 0.101010117795074239]
$x[5] =$	0.083916092680731496	$3.751576 \cdot 10^{-08}$	[0.083916055164965267, 0.083916130196497726]
$x[6] =$	0.071794839102344798	$4.377458 \cdot 10^{-08}$	[0.071794795327757565, 0.071794882876932031]
$x[7] =$	0.062745135782950711	$5.003910 \cdot 10^{-08}$	[0.062745085743848322, 0.062745185822053101]
$x[8] =$	0.055727541840479261	$5.630827 \cdot 10^{-08}$	[0.055727485532206209, 0.055727598148752312]
$x[9] =$	0.050125327081073652	$6.257111 \cdot 10^{-08}$	[0.050125264509957842, 0.050125389652189461]

DOI: 10.1002/nla

Table VI. Numerical results found for Matrix 2.

Res	Midpoint	Radius	Infimum–supremum representation
$x[0] =$	0.00	$1.666933 \cdot 10^{-11}$	$[-1.666933 \cdot 10^{-11}, 1.666933 \cdot 10^{-11}]$
$x[1] =$	0.00	$3.333866 \cdot 10^{-11}$	$[-3.333866 \cdot 10^{-11}, 3.333866 \cdot 10^{-11}]$
$x[2] =$	0.00	$3.333866 \cdot 10^{-11}$	$[-3.333866 \cdot 10^{-11}, 3.333866 \cdot 10^{-11}]$
$x[3] =$	0.00	$3.333866 \cdot 10^{-11}$	$[-3.333866 \cdot 10^{-11}, 3.333866 \cdot 10^{-11}]$

Table VII. Results found for Matrix 1 with radius 10^{-12}.

Res	Midpoint	Radius	Infimum–supremum representation
$x[0] =$	0.666666666673327968	$1.081470 \cdot 10^{-07}$	[0.666666558526246456, 0.666666774820409480]
$x[1] =$	0.266666673443532731	$2.163952 \cdot 10^{-07}$	[0.266666457048279015, 0.266666457048279015]
$x[2] =$	0.171428542393218419	$3.247011 \cdot 10^{-07}$	[0.171428217692048385, 0.171428867094388454]
$x[3] =$	0.126984150895482345	$4.330025 \cdot 10^{-07}$	[0.126983717892928755, 0.126984583898035935]
$x[4] =$	0.101010086536853047	$5.413480 \cdot 10^{-07}$	[0.101009545188830957, 0.101010627884875137]
$x[5] =$	0.083916092680731496	$6.497704 \cdot 10^{-07}$	[0.083915442910309681, 0.083916742451153311]
$x[6] =$	0.071794839102344798	$7.582337 \cdot 10^{-07}$	[0.071794080868570822, 0.071795597336118774]
$x[7] =$	0.062745135782950711	$8.667374 \cdot 10^{-07}$	[0.062744269045465928, 0.062746002520435495]
$x[8] =$	0.055727541840479261	$9.752456 \cdot 10^{-07}$	[0.055726566594805203, 0.055728517086153318]
$x[9] =$	0.050125327081073652	$1.083865 \cdot 10^{-06}$	[0.050124243215723222, 0.050126410946424081]

result found for the solution of an interval linear system is not as accurate as the one found when using point input data. When solving very large interval linear systems, it is more likely that even a small radius may have a terrible impact on the accuracy of computed results.

9. CONCLUSIONS AND FUTURE WORK

In this work, we presented an efficient approach to solve very large dense linear systems with verified computing on cluster. Indeed, this new approach makes it possible to overcome the previous input matrix dimension limit reached in [10] from 10,000 to 100,000.

We have evaluated three aspects of our solution: portability, scalability and accuracy of the results. Experiments were carried out over two cluster environments with substantially distinct configurations. The results indicated that the proposed approach allowed our parallel solver to run correctly and with good performance on both environments. Running our solver on different cluster environments also helped us to identify memory allocation constraints for executions over a few number of nodes (typically less than 32 nodes). This fact only reinforces the importance of a scalable solution, as with the use of a large number of processors, the memory allocation problem disappears. For that matter, using different input matrix dimensions, we could evaluate the scalability of our solver on both clusters. IC1 and XC1 presented good results in terms of scalability, although the second one showed some limitation for the highest dimension input matrix (100,000). Over IC1, on the other hand, our parallel solver was able to scale well up to 256 nodes with a relative speedup close to the ideal. Our solver also delivered accurate results for input matrices with different condition numbers. The experiment using interval input data delivered, as expected, a verified interval result with a larger radius. Nevertheless, these results are similar with the ones found when running sequentially the same algorithm for smaller input matrix dimensions.

Finally, our future works point to two directions: (i) migrate our solution to multiprocessor platforms (multicore architectures) using libraries like Parallel Linear Algebra Software for Multicore Architectures and Matrix Algebra on GPU and Multicore Architectures; (ii) propose a new version of our parallel solver for sparse linear systems. Regarding the multicore version of our solver, preliminary studies were already presented in [45] and [47].

DOI: 10.1002/nla

ACKNOWLEDGEMENTS

The authors would like to thank Rudi Klatte, Walter Kraemer and Michael Zimmer for many fruitful discussions on the topic of this paper. We also would like to thank Viviane Lara for her support. This research was partially supported by CAPES and PROBRAL project 298/08 "High Performance Verified Computing (HPVC)".

REFERENCES

1. Baboulin M, Giraud L, Gratton S. A parallel distributed solver for large dense symmetric systems: applications to geodesy and electromagnetism problems. *International Journal of High Speed Computing* 2005; **19**(4):353–363.
2. Stpiczynski P, Paprzycki M. Numerical software for solving dense linear algebra problems on high performance computers. *APLIMAT 2005: Proceedings of the 4th International Conference on Applied Mathematics*, Bratislava, Slovak Republic, 2005; 207–218.
3. Zhang J, Maple C. Parallel solutions of large dense linear systems using MPI. In *PARELEC '02: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*. IEEE Computer Society Press: Warsaw, Poland, 2002; 312–317.
4. Rump SM. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica* 2010; **19**:287–449.
5. Kearfott RB. Interval Computations: Introduction, Uses and Resources. *Euromath Bulletin* 1996; **2**(1).
6. Rump SM. Fast and Parallel Interval Arithmetic. *Bit Numerical Mathematics* 1999; **39**(3):534–554.
7. Rump SM. INTLAB—INTerval LABoratory. In *Developments in Reliable Computing*, Csendes T (ed.). Kluwer Academic Publisher: Dordrecht, 1999; 77–104.
8. Kolberg M, Baldo L, Velho P, Fernandes LG, Claudio D. Optimizing a parallel self-verified method for solving linear systems. In *PARA 2006: Applied Parallel Computing. State of the Art in Scientific Computing*, vol. 4699, Lecture Notes in Computer Science. Springer Berlin/Heidelberg: Germany, 2008; 949–955.
9. Kolberg M, Baldo L, Velho P, Webber T, Fernandes LG, Fernandes P, Claudio D. Parallel self-verified method for solving linear systems. In *VECPAR 2006: 7th International Meeting on High Performance Computing for Computational Science*. Rio de Janeiro: Brazil, 2006; 179–190.
10. Kolberg M, Bohlender G, Claudio D. Improving the performance of a verified linear system solver using optimized libraries and parallel computation. In *VECPAR 2008: High Performance Computing for Computational Science*, vol. 5336, Lecture Notes in Computer Science. Springer Berlin/Heidelberg: Germany, 2008; 13–26.
11. Kolberg M, Dorn M, Fernandes LG, Bohlender G. Parallel verified linear system solver for uncertain input data. *20th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2008,* IEEE Computer Society, Campo Grande, Brazil, 2008; 89–96.
12. Kolberg M, Fernandes LG, Claudio D. Dense linear system: a parallel self-verified solver. *International Journal of Parallel Programming* 2008; **36**:412–425.
13. Baker M, Buyya R. Cluster computing: the commodity supercomputer. *Software-Practice and Experience* 1999; **29**:551–576.
14. Hedayat GA. Numerical linear algebra and computer architecture: an evolving interaction. *Technical Report UMCS-93-1-5*, University of Manchester: Manchester, England, 1993.
15. Saad Y. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company: Boston, 1995.
16. D'Azevedo E, Dongarra J. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurrency—Practice and Experience* 2000; **12**(15):1481–1493.
17. Klatte R, Kulisch U, Lawo C, Rauch R, Wiethoff A. *C-XSC- A C++ Class Library for Extended Scientific Computing*. Springer-Verlag : Berlin, 1993.
18. Rump SM. Kleine Fehlerschranken bei Matrixproblemen. *PhD thesis*, University of Karlsruhe, Germany, 1980.
19. Feng T, Flueck AJ. A Message-Passing Distributed-Memory Newton-GMRES Parallel Power Flow Algorithm. *Proceedings of the IEEE Power Engineering Society Summer Meeting,* IEEE Press, Vol. 3, Chicago, USA, 2002; 1477–1482.
20. Demmel JW, Heath MT, van der Vorst HA. Parallel Numerical Linear Algebra. *Technical Report UCB CSD-92-703*, Cambridge University, 1993.
21. LAPACK. LAPACK Users' Guide, 2013. (Available from: http://www.netlib.org/lapack/lug/.visited) [accessed on 19th April 2013].
22. Dongarra J, Du Croz J, Duff IS, Hammarling S. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 1990; **16**:1–17.
23. Dongarra J, Pozo R, Walker DW. LAPACK++: a design overview of object-oriented extensions for high performance linear algebra. In *SUPERCOMPUTING '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press: Portland, USA, 1993; 162–171.
24. MathWorks. *MATLAB, The Language of Technical Computing*. The MathWorks, Inc.: Natick, 2001.
25. INTLAB. INTerval LABoratory, 2013. (Available from: http://www.ti3.tu-harburg.de/rump/intlab/.visited) [19th April 2013].
26. Zimmer M, Krämer W, Bohlender G, Hofschuster W. Extension of the C-XSC library with scalar products with selectable accuracy. *Technical Report BUW-WRSWT 2009/4*, Universität Wuppertal, Wuppertal: Germany, 2009.

DOI: 10.1002/nla

27. Rump SM. Accurate and reliable computing in floating-point arithmetic. In *Proceedings of the Third International Congress on Mathematical Software, Kobe, Japan, September 13–17, 2010 (ICMS 2010)*, vol. 6327, Lecture Notes in Computer Science. Springer Berlin / Heidelberg: Germany, 2010; 105–108.
28. Rump SM. Fast interval matrix multiplication, *to be published in Numerical Algorithms*, 2012.
29. Nguyen HD, Revol N. Accuracy issues in linear algebra using interval arithmetic. *SCAN 2010: 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Lyon, France; 2010.
30. Nguyen HD, Revol N. High performance linear algebra using interval arithmetic. *PASCO 2010*, Grenoble, France, 2010; 171–172.
31. Nguyen HD, Revol N. Refining and verifying efficiently the solution of a linear system. *Dagstuhl Seminar 11371: Uncertainty Modeling and Analysis with Intervals: Foundations, Tools, Applications*, Dagstuhl, Germany, September 2011; 11–16.
32. Facius A. Iterative Solution of linear systems with improved arithmetic and result verification. *PhD thesis*, University of Karlsruhe, Germany, 2000.
33. Kersten T. Verifizierende Rechnerinvariante Numerikmodule. *PhD thesis*, University of Karlsruhe, Germany, 1998.
34. Wiethoff A. Verifizierte Globale Optimierung auf Parallelrechnern. *PhD thesis*, University of Karlsruhe, Germany, 1997.
35. Kolberg M, Krämer W, Zimmer M. Efficient parallel solvers for large dense systems of linear interval equations. *Reliable Computing* 2011; **15**(3):193–206.
36. Krämer W, Zimmer M. Fast (parallel) dense linear system solvers in C-XSC using error free transformations and BLAS. In *Numerical Validation in Current Hardware Architectures*, vol. 5492, Lecture Notes in Computer Science. Springer Berlin/Heidelberg: Germany, 2009; 230–249.
37. P1788. IEEE Standard for Interval Arithmetic. (Available from: http://grouper.ieee.org/groups/1788/PositionPapers/VanSnyderP1788.pdf.visited) [19th April 2013].
38. Neumaier A. Improving interval enclosures., *Reliable Computing*, to appear.
39. Bohlender G. What do we need beyond IEEE arithmetic?. In *Computer Arithmetic and Self-validating Numerical Methods*, Christian Ullrich (ed.). Academic Press Professional, Inc.: San Diego, CA, USA, 1990; 1–32.
40. Kulisch U, Miranker WL. *Computer Arithmetic in Theory and Practice*. Academic Press: New York, 1981.
41. Ogita T, Rump SM, Oishi S. Verified solution of linear systems without directed rounding. *Technical Report 2005-04*, Advanced Research Institute for Science and Engineering: Waseda University, Tokyo, Japan, 2005.
42. Revol N, Makino K, Berz M. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *Technical Report INRIA RR-4737*, 2003.
43. Hammer R, Ratz D, Kulisch U, Hocks M. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York: Secaucus, NJ, USA, 1997.
44. Alefeld G, Herzberger J. *Introduction to Interval Computations*. Academic Press: New York, 1983.
45. Kolberg M, Cordeiro D, Bohlender G, Fernandes LG, Goldman A. A Multithreaded Verified Method for Solving Linear Systems in Dual-core Processors. In *PARA—9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, To be published, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.
46. Agullo E, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, YarKhan A. PLASMA Users Guide, visited 19th, April 2013.
47. Milani C, Kolberg M, Fernandes LG. Solving dense interval linear systems with verified computing on multicore architectures. In *VECPAR 2010: High Performance Computing for Computational Science*, vol. 6449, Lecture Notes in Computer Science. Springer Berlin / Heidelberg: Germany, 2010; 435–448.
48. Tomov S, Nath R, Du P, Dongarra J. MAGMA users guide. (Available from: http://icl.cs.utk.edu/projectsfiles/magma/docs/magma-v02.pdf.visited) [accessed on 19th April 2013].
49. Snir M, Otto S, Huss-Lederman S, Walker DW, Dongarra J. *MPI: The Complete Reference*. MIT Press: Cambridge, MA, 1996.
50. Dongarra J, Walker D. LAPACK working note 58: the design of linear algebra libraries for high performance computers. *Technical Report UT-CS-93-188*: Knoxville, TN, USA, 1993.
51. Blackford LS, Choi J, Cleary A, Demmel J, Dhillon I, Dongarra J, Ham- marling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. ScaLAPACK: a portable linear algebra library for distributed memory computers design issues and performance. *SUPERCOMPUTING '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing,* IEEE Computer Society Press, Pittsburgh, Pennsylvania, USA, 1996.
52. Choi J, Demmel J, Dhillon I, Dongarra J, Ostrouchov S, Petitet A, Stanley K, Walker D, Whaley LAPACK working note 95 RC. ScaLAPACK: a portable linear algebra library for distributed memory computers—design issues and Performance. *Technical Report UT-CS-95-283*, University of Tennessee: Knoxville, TN, USA, 1995.
53. Kolberg M, Rocker B, Heuveline V. The impact of data distribution in accuracy and performance of parallel linear algebra subroutines. In *VECPAR 2010: High Performance Computing for Computational Science*, vol. 6449, Lecture Notes in Computer Science. Springer Berlin / Heidelberg: Germany, 2010; 394–407.
54. Toledo S. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms*, American Mathematical Society Boston, USA, 1999; 161–179.
55. Gregory RT, Karney DL. *A Collection of Matrices for Testing Computational Algorithms*. Wiley-Interscience: New York, 1969.

DOI: 10.1002/nla