Int J Parallel Prog (2008) 36:412–425
DOI 10.1007/s10766-007-0058-x

Dense Linear System: A Parallel Self-verified Solver

Mariana Luderitz Kolberg ·
Luiz Gustavo Fernandes · **Dalcidio Moraes Claudio**

Received: 25 August 2006 / Accepted: 20 September 2007 / Published online: 16 October 2007
© Springer Science+Business Media, LLC 2007

Abstract This article presents a parallel self-verified solver for dense linear systems of equations. This kind of solver is commonly used in many different kinds of real applications which deal with large matrices. Nevertheless, two key problems appear to limit the use of linear system solvers to a more extensive range of real applications: solution correctness and high computational cost. In order to solve the first one, verified computing would be an interesting choice. An algorithm that uses this concept is able to find a highly accurate and automatically verified result providing more reliability. However, the performance of these algorithms quickly becomes a drawback. Aiming at a better performance, parallel computing techniques were employed. Two main parts of this method were parallelized: the computation of the approximate inverse of matrix A and the preconditioning step. The results obtained show that these optimizations increase significantly the overall performance.

Keywords Linear systems · Verified computing · Parallel computing

1 Introduction

Many numerical problems can be solved through a dense linear system of equations. Therefore, the solution of systems like $Ax = b$ with an $n \times n$ matrix $A \in \mathbb{R}^{n \times n}$ and a right hand side $b \in \mathbb{R}^n$ is very usual in numerical analysis. This is true for functional

M. L. Kolberg (✉) · L. G. Fernandes · D. M. Claudio
Faculdade de Informática, PUCRS, Avenida Ipiranga, 6681 Prédio 32 sala 507, Porto Alegre, Brazil
e-mail: mkolberg@inf.pucrs.br

L. G. Fernandes
e-mail: gustavo@inf.pucrs.br

D. M. Claudio
e-mail: dalcidio@inf.pucrs.br

linear equations that occur like partial differential equations and integral equations that appear in several problems of Physics and Engineering [1]. Many different numerical algorithms contain this task as a subproblem.

However, even very small systems may present a wrong solution due to lack of accuracy. This problem can be easily seen in the following system:

$$A = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The correct solution would be

$$x_1 = \frac{a_{22}}{a_{11}a_{22} - a_{12}a_{21}} = 205117922, \quad x_2 = \frac{-a_{21}x_1}{a_{22}} = 83739041.$$

However, using IEEE double precision arithmetic and LU decomposition leads us to the following wrong results:

$$\tilde{x}_1 = 102558961 \quad \tilde{x}_2 = 41869520.5.$$

One possible solution to cope with that would be the use of accurate computation through the use of interval arithmetic [2]. Verified computing provides an interval result that surely contains the correct result [3]. The algorithm will, in general, succeed in finding a solution. If the solution is not correct or not found, the algorithm will let the user know. Numerical applications providing automatic result verification may be useful in many fields like Simulation and Modeling [1]. Specially, when accuracy is mandatory. The requirements for verified computing are:

- Interval arithmetic;
- High accuracy arithmetic;
- Well suitable algorithms.

Interval arithmetic defines the operations for interval numbers, such that the result is a new interval that contains the set of all possible solutions. High accuracy arithmetic ensures that the operation is performed without rounding errors, and rounded only once in the end of the computation. The requirements for this arithmetic are: the four basic operations with high accuracy, optimal scalar product, and directed roundings. There are many options on how to implement this optimal scalar product [2,4,5]. One possibility is to accumulate intermediate results in a fixed-point number, that means that the number is stored like an integer, and the computation will be done simulating infinite precision. This guarantees that no rounding error will happen in the accumulation.

There is a multitude of tools that provide verified computing. Among them an attractive option is C-XSC (C for eXtended Scientific Computing) [6]. C-XSC is a free and portable programming environment for C and C++ programming languages, which offers high accuracy. This programming tool allows the solution of several standard problems, including many reliable numerical algorithms. More details will be presented in Sect. 2.2. The use of C-XSC through suitable algorithms to ensure that those properties will be held, can provide automatic verified results.

The use of verified computing makes it possible to find the correct result. However, finding the verified result often increases the execution times dramatically [7]. The research already developed shows that the execution times of verified algorithms are much larger than the execution times of algorithms that do not use this concept [8,9]. In this scenario, a parallel version of a self-verified solver for dense linear systems appears to be essential in order to solve bigger problems.

The advent of parallel computing and its impact in the overall performance of many algorithms on numerical analysis can be seen in the past years [10]. The use of clusters plays an important role in such a scenario as one of the most effective manners to improve the computational power without increasing costs to prohibitive values. The parallel solutions for linear solvers found in the literature explore many aspects and constraints related to the adaptation of the numerical methods to high performance environments [11,12]. Some works present the idea of exploiting 32 bit floating point arithmetic in obtaining full precision (64 bit results) for performance reasons like presented in [13]. However, those works (like many others) do not deal with verified computation. Conventional floating point algorithms can find a solution for a problem with condition number up to 10^8 [13] and deliver no guarantee whether the result is correct or not. In contrast, a verified algorithm will, in general, succeed in finding a solution even for very ill-conditioned problems with condition number up to 2.5×10^{15}. If the solution is not correct or not found, the algorithm will let the user know. It ensures that no wrong result will be delivered from such algorithms. However, the performance of a verified algorithm is not as good as an algorithm based on floating point arithmetic. Such algorithms need some special accumulators and interval arithmetic. Few works related to verified computing [14] and even fewer related to verified computing and parallel implementations were found [15,16], but these authors implement other numerical problems or use a parallel approach for other architectures than clusters.

In this research, we propose a parallel implementation of the current verified algorithm using technologies as MPI communication primitives associated to the C-XSC library to provide both self-verification and speed-up at the same time. Moreover, the major goal of this paper is to point out the advantages and the drawbacks of the parallelization of a self-verifying method for solving linear systems over distributed environments. It is important to mention that readers interested in the authors' previous publications on this subject should read [17,18].

The organization of this paper is as follows: in the next section, an explanation of the self-verified method used for parallelization and its mathematical background are presented. Section 3 shows which parallel and mathematical optimization can be done for the chosen self-verified method. The analysis of the results obtained through this parallelization is presented in Sect. 4. Finally, the conclusion and some future works are given in last section.

2 Self-verified Solver for Dense Linear Systems of Equations

As mentioned before, we have to ensure that the mathematical properties of interval arithmetic as well as high accuracy arithmetic be held if we want to achieve

self-verification. Based on that, we used Algorithm 1 as the starting point of our parallel version. This algorithm is fully described in [19] and will, in general, succeed in finding and enclosing a solution or, if it does not succeed, will let the user know. In the latter case, the user will know that the problem is likely to be very ill-conditioned or that the matrix A is singular.

Algorithm 1 Compute an enclosure for the solution of the square linear system $Ax = b$.

1: $R \approx A^{-1}$ {Compute an approximate inverse using LU-Decomposition algorithm}
2: $\tilde{x} \approx R \cdot b$ {compute the approximation of the solution}
3: $[z] \supseteq R(b - A\tilde{x})$ {compute enclosure for the residuum (without rounding error)}
4: $[C] \supseteq (I - RA)$ {compute enclosure for the iteration matrix (without rounding error)}
5: $[w] := [z]$, $k := 0$ {initialize machine interval vector}
6: **while not** ($[w] \subseteq int[y]$ or $k > 10$) **do**
7: $\quad [y] := [w]$
8: $\quad [w] := [z] + [C][y]$
9: $\quad k{+}{+}$
10: **end while**
11: **if** $[w] \subseteq int[y]$ **then**
12: $\quad \Sigma(A, b) \subseteq \tilde{x} + [w]$ {The solution set (Σ) is contained in the solution found by the method}
13: **else**
14: $\quad$ “no verification”
15: **end if**

However, to implement the operations “without rounding error,” it is mandatory to use a verified computing tool. In the next subsection, we present the mathematical issues behind this implementation. The subsection 2.2 contains some information about the verified tool C-XSC.

2.1 Mathematical Issues

The enclosure method for finding the solution of the system

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

is based on the Newton-like iteration

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

An approximate solution $\tilde{x}$ of $Ax = b$ may be improved if we try to enclose the error of the approximate solution. This can be done by solving the system 3 to find the residual, yielding a much higher accuracy. The error $y = x - \tilde{x}$ of the true solution x satisfies the equation

$$Ay = b - A\tilde{x}, \tag{3}$$

which can be multiplied by R and rewritten in the form

$$y = R(b - A\tilde{x}) + (I - RA)y. \tag{4}$$

Let $f(y) := R(b - A\tilde{x}) + (I - RA)y$. Then Eq. 4 has the form

$$y = f(y). \tag{5}$$

If R is a sufficiently good approximation of A^{-1}, then an iteration based on Eq. 4 can be expected to converge since $(I - RA)$ will have a small spectral radius. These results remain valid if we replace the exact expression by interval extensions. However, to avoid overestimation effects, it is recommended to evaluate it without any intermediate rounding. Therefore, we derive the following iteration from Eq. 4, where we use interval arithmetic and intervals $[y_k]$ for y:

$$[y]_{k+1} = R \diamond (b - A\tilde{x}) + \diamond(I - RA)[y]_k \tag{6}$$

or

$$[y]_{k+1} = F([y]_k), \tag{7}$$

where F is the interval extension of f. Here, $\diamond$ means that the succeeding operations have to be executed exactly and the result is rounded to an enclosing interval (vector or matrix). In the computation of the defect $(b - A\tilde{x})$ and of the iteration matrix $(I - RA)$, serious cancelations of leading digits must be expected. Hence, these should be computed using the exact scalar product. Each component is computed exactly and then rounded to a machine interval. For this purpose, the scalar product expressions of C-XSC are used in the implementations. With $z = R(b - A\tilde{x})$ (line 3 of Algorithm 1) and $C = (I - RA)$ (line 4 of Algorithm 1) Eq. 6 can be rewritten as

$$[y]_{k+1} = z + C[y]_k. \tag{8}$$

To compute our approximate solution $\tilde{x}$ and the approximate inverse R, we used the LU-decomposition. In principle, there is no special requirements about these quantities, we could even just guess them. However, the results of the enclosure algorithm will of course depend on the quality of the approximations. The procedure fails if the computation of an approximate inverse R fails or if the inclusion in the interior cannot be established.

We gave a brief summary of the enclosure methods theory. A more detailed presentation can be found in [20]. In the next section we present some characteristics of C-XSC, the verified library that enables the implementation of this algorithm.

2.2 C-XSC

C-XSC [6], C for eXtended Scientific Computation, is a programming tool for the development of numerical algorithms which provide highly accurate and automatically verified results. The programming language C++, an object-oriented extension of the programming language C, does not provide better facilities than C to programming numerical algorithms, but its new concept of abstract data structures (classes) and the concept of overloading operators and functions provide the possibility to create such a programming tool.

Fig. 1 Fixed-point accumulator

g	2. e_{max}	1	1	2. $\|e_{min}\|$

C-XSC is not an extension of the standard language, but a class library which is written in C++. Therefore, no special compiler is needed. With its abstract data structures, predefined operators and functions, C-XSC provides an interface between scientific computing and the programming language C++. Besides, C-XSC supports the programming of algorithms which automatically enclose the solution of given mathematical problem in verified bounds. Such algorithms deliver a precise mathematical statement about the true solution. The main concepts of C-XSC are:

- Real, complex, interval, and complex interval arithmetic with mathematically defined properties;
- Dynamic vectors and matrices;
- Subarrays of vector and matrices;
- Dot-precision data type;
- Predefined arithmetic operators with highest accuracy;
- Standard functions of high accuracy;
- Dynamic multiple-precision arithmetic and standard functions;
- Rounding control for I/O data;
- Numerical results with mathematical rigor;

One of the most important features of C-XSC is the *dot-precision* data type. C-XSC simulates a fixed-point operator that is essential to solving the optimal scalar product. The fixed-point accumulator provides a simulation of infinite precision according to the model of Fig. 1. However, this accumulator has a defined size of 529 bytes, which is much larger than the data type double (8 bytes). In this figure, l is the number of digits, e_{max} and e_{min} are the maximum and the minimum exponents, and g is the number of guard digits.

3 Parallel Approach

To implement the parallel version of the Algorithm 1, we used an approach for cluster architectures with message passing programming model (MPI) and C-XSC library. Clusters of computers are considered a good option to achieve better performances without using parallel programming models oriented to very expensive (but not frequently used) machines. A parallel version for this algorithm runs on distributed processors, requiring communication among the processors connected by a fast network and the communication library.

Like mentioned in Sect. 2.2, C-XSC is an attractive option among the available tools to achieve verified results. On the other hand, the data types of the C-XSC library, which allow for self-verification and were used on the parallel approach, do not fit in the MPI library data types. In order to solve this problem, the data should be packed and unpacked before sending and receiving. Aiming at enabling this communication, we used a library that extends MPI for the use of C-XSC in parallel environments implementing the pack and unpack procedures in a low level [21].

We carried out some tests to find most time-consuming steps in Algorithm 1. We found out that the steps 1 and 4 consume together more than 90% of the total processing time. The computation of an approximate inverse matrix of A (matrix R on step 1) takes more than 50% of the total time, due to the evaluation with high accuracy. The computation of the interval matrix $[C]$ (parallel preconditioning) takes more than 40% of the total time, since matrix multiplication requires $O(n^3)$ execution time, and the other operations are mostly vector or matrix–vector operations which require at most $O(n^2)$. We chose to optimize these two parts of the algorithm, using two different approaches that will be presented in the following sections.

We assume the coefficient matrix A in Eq. 1 to be dense, i.e., in a C-XSC program, we use a square matrix of type rmatrix to store A, and we do not consider any special structure of the elements of A. Our goal is to make a parallel version of the C-XSC algorithm that verifies the existence of a solution and computes an enclosure for the solution of system the $Ax = b$ for a square $n \times n$ matrix A with a better performance than the sequential version.

3.1 The Inverse Matrix Computation

We may have two possibilities to achieve a better performance at the computation of R (the approximate inverse matrix of A). We can simply evaluate R without high accuracy or parallelize this computation keeping the high accuracy in its evaluation.

Since R is an approximation of the inversion of the matrix A, which does not always need exact results, in several cases it can be computed without high accuracy. In these cases, the approximation obtained with floating point arithmetic is enough to find an accurate inclusion at the end of the algorithm. It is also known that the elimination of the use of high accuracy on R computation could decrease the overall executing time and in most cases would not compromise the results obtained.

On the other hand, in some cases the mathematical high accuracy is needed to find the correct result. In these cases, the non-accurate approximation R would not be enough, and the algorithm would not find the correct result for the linear system. Therefore, we made a parallel version of the computation of the approximate inverse matrix.

Due to this condition, we have implemented the algorithm as follows:

- If the algorithm is able to find the verified solution of the linear system based on the computation of R without high accuracy, it stops.
- If the algorithm does not find the solution, it executes again in parallel, evaluating R with high accuracy.

The approach used in the parallel implementation is explained in the next section.

(1) *Parallel Inverse Matrix Computation:* A deeper analysis of the execution time spent in step 1 was carried out to find the bottleneck in this part of the algorithm. The computation of the inverse matrix is done in two major steps: the LU decomposition and the computation of the inverse column by column through backward/forward substitution. From the 52% of the total time that this step takes, the LU decomposition of matrix A takes 35%, whereas the calculation of the inverse by backward/forward

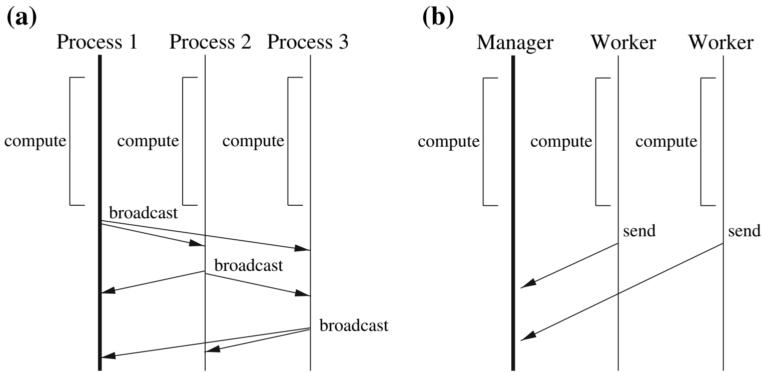

Fig. 2 Communication schemes

substitution takes 65% of step 1, due to C-XSC's special arithmetic and features like dotprecision variable. Through this analysis, it was decided that the parallelization of R computation would be focused only on the backward/forward substitution phase due to the higher computational cost. Moreover, the parallel LU decomposition is well known and many different proposals can be found [22–25].

The algorithm implemented uses a parallel phases scheme to find the approximate inverse matrix (R) through backward/ forward substitution. Every process computes a number of columns (co-named task) of matrix R and after this processing phase, all processes exchange information to construct the overall R. At the end of this process, all nodes must have the inverse matrix R, once R is used in the further parallel computation of the interval matrix C. The Fig. 2(a) shows the communication strategy used in this step.

In Sect. 4, we will present detailed results for the execution time concerning the computation of R with and without maximal accuracy.

3.2 Parallel Preconditioning

Preconditioning is known as a way to speed-up the method convergence based on establishing some algebraic constants used to approximate x at each iteration. The preconditioning in Algorithm 1 can be seen in step 4. In the computation of $[C] = (I - RA)$, I is the identity matrix of the same order as A, and R is a good approximation of A^{-1}. This step was implemented in parallel using the worker/manager approach.

In this approach, each process finds a contiguous block of lines based on its rank, to compute some lines of the result matrix ($[C]$). After computing, the manager receives the computed lines from every process, puts them in the proper places and generates $[C]$. This step is illustrated by Fig. 2(b), where the thicker vertical line means the manager process, and the other vertical lines mean the other processes. After computing, all the worker processes send their result to the manager.

3.3 Load Balancing Approach

The algorithm representing the load balancing strategy for both parallelizations is shown in Algorithm 2. Aiming to decrease the overhead, the load balancing was done in parallel. Since the processes do not have to concern with the load balancing, this approach takes less communication time to exchange information. Another important characteristic of this algorithm is that it should be simple but effective, and above everything, fast. It should not become a bottleneck in the computation process.

Algorithm 2 Workload distribution algorithm.

1: **for** $(pr = 0, i = 1; pr < P; pr++, i++)$ **do**
2: **if** $pr < (N\%P)$ **then**
3: $lb_i = (\frac{N}{P} \times pr) + (pr)$
4: $ub_i = lb_i + \frac{N}{P}$
5: **else**
6: $lb_i = (\frac{N}{P} \times pr) + (N\%P)$
7: $ub_i = lb_i + \frac{N}{P} - 1$
8: **end if**
9: **end for**

Algorithm 2 presents the workload distribution. The identification of a process goes from 0 to $P - 1$ and is called pr, where P is the number of processes involved in the computation and N is the number of rows or columns of a matrix. Also, ub_i and lb_i are the upper and lower bound of the ith process respectively, i.e., the first and the last row/column that a process must compute.

Aiming to decrease the communication cost, the process will receive a continuous block of rows/columns, what can be ensured by this load balancing approach. Since the computational cost to evaluate a row/column of the matrix is the same, other strategies for load balancing could be used.

4 Results Analysis

In order to guarantee the success of these optimizations, two different experiments have been performed. The first test is related to the correctness of the result. Once we make a mathematical optimization, we need to verify that it did not change the accuracy of the result. Another test should be done to evaluate the speed-up achieved in this new implementation. Since the main objective of this paper is to minimize the execution time, achieving good speed-ups is a desired result.

Some test cases were executed varying the number of processes and the size of the input matrix A and vector b. To evaluate the correctness and the performance of the new implementation, matrix A and vector b were generated in two different ways: the Boothroyd/Dekker formula [19] and random numbers, respectively.

The results presented in this paper were obtained over the parallel environment ALiCEnext (Advanced Linux Cluster Engine, next generation) installed at the University of Wuppertal (Germany). This cluster is composed of 1, 024 1.8 GHz AMD

Opteron processors (64 bit architecture) on 512 nodes connected by Gigabit Ethernet. ALiCEnext processors employ Linux as operating system and MPI as communication interface for parallel communication. These experiments are presented in the following sections.

4.1 Correctness

Aiming to verify the accuracy of our parallel solution, many executions were tackled with several matrices. Among them the well-known Boothroyd/Dekker matrices that are defined by the following formula:

$$A_{ij} = \binom{n+i-1}{i-1} \times \binom{n-1}{n-j} \times \frac{n}{i+j-1}, \quad \forall i, j = 1 \ldots N, \quad b = 1, 2, 3, \ldots, N.$$

We ran tests with a 10×10 matrix with different number of processes $(1 \ldots P)$. The tests generated by the Boothroyd/Dekker formula presented the same accuracy for both versions (sequential and parallel) and indicated that the parallelization did not modify the accuracy of the results.

4.2 Performance

Performance analysis of this parallel solver was carried out varying the order of input matrix A. Matrices with five different orders were used as test cases: 500×500, $1,000 \times 1,000$, $2,500 \times 2,500$, $3,000 \times 3,000$, and $4,500 \times 4,500$. For each of those matrices, executions with the number of processors varying from 1 to P were performed. In the experiments presented in the next paragraphs, three different values were assigned to P: 18, 28, and 100. All matrices were specifically generated for these experiments and are composed of random numbers. Since the greater part of systems of equations can be solved evaluating R with floating point precision, the experiments that generated the results presented in Figs. 3 and 5 *did not use high accuracy to compute* R (the approximate inverse of A).

Figure 3 presents a comparison of the speed-ups achieved for small matrices (order 500, 1000, and 2500). The proposed parallel solution presents a good scalability and improves the performance of the application. As expected, the performance gain is proportional to the matrix order (which impacts directly on its computational cost): the larger the input matrix, the better is the speed-up achieved. It is also important to remark that for the input matrix with order 2, 500, the speed-up achieved was around 21 for 28 processors, which is a representative result for the chosen platform. Results for more than 28 processors were not presented for these test cases since beyond this threshold the performance started to drop down.

In Fig. 4, results for larger input matrices are shown. In this experiment, the goal was to verify the behavior, in terms of performance gains, of the proposed parallel solution using a higher number of processors (e.g., one hundred). Parallel solutions developed for multi-computers, such as clusters based on the message passing paradigm, are usually not able to scale properly due to communication issues. Indeed,

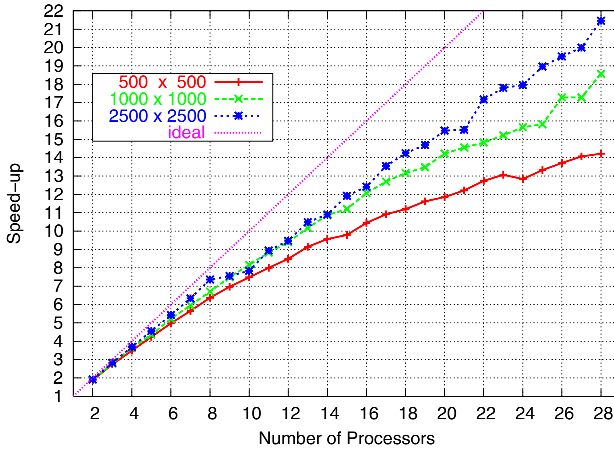

Fig. 3 Experimental results—speed-up

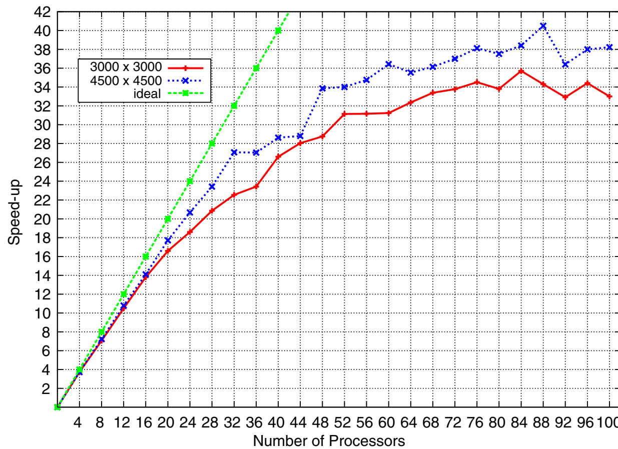

Fig. 4 Experimental results—speed-up with big matrices

the curves show a loss of efficiency at different points for all three input cases, even considering that the proposed parallel solution decreases the execution times up to 92 or 96 processors. However, the best speed-up factor achieved for each test case represents a significant gain of performance in the computation of those matrices.

Considering that in some cases the computation of R with high accuracy is needed, the results of its parallel version are presented in Fig. 5. This figure presents the comparison of the parallel and highly accurate computation of R to the implementation that computes R in serial without high accuracy. The matrix size used in this test was $1,000 \times 1,000$ and only 18 processors were used.

Table 1 summarizes the execution time, speed-up, and efficiency achieved by both implementations. Comparing the sequential time with the execution time for 18 processors, it is possible to notice that a significant decrease is achieved. In this scenario,

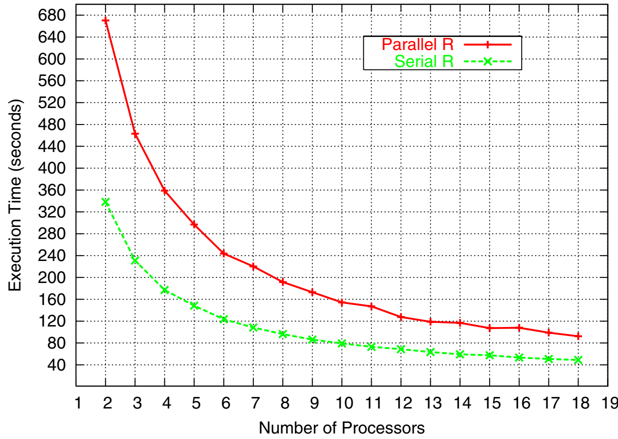

Fig. 5 Experimental results—execution time

Table 1 Experimental results—efficiency

Not highly accurate computation of R				Highly accurate computation of R			
No. of proc	Exec time (s)	Speed-up	Efficiency (%)	No. of proc	Exec time (s)	Speed-up	Efficiency (%)
2	340.8745	1.9053	95	2	670.3673	1.7926	89
3	235.8245	2.7541	91	3	462.9983	2.5955	86
4	180.5205	3.5979	89	4	358.8473	3.3488	83
5	147.1135	4.4149	88	5	296.9023	4.0475	80
6	122.9265	5.2836	88	6	243.6563	4.9320	82
7	106.5875	6.0935	87	7	220.1483	5.4586	77
8	95.0785	6.8311	85	8	191.5113	6.2749	78
9	84.8605	7.6536	85	9	172.9123	6.9498	77
10	78.8815	8.2338	82	10	154.3443	7.7859	77
11	70.7625	9.1785	83	11	147.0943	8.1697	74
12	67.6225	9.6047	80	12	127.4863	9.4262	78
13	61.3335	10.5895	81	13	118.8673	10.1097	77
14	57.8125	11.2345	80	14	116.8573	10.2836	73
15	54.6955	11.8747	79	15	107.2083	11.2092	74
16	52.5655	12.3559	77	16	107.7583	11.1519	69
17	49.5765	13.1008	77	17	98.8683	12.1547	71
18	47.4765	13.6803	76	18	92.2993	13.0198	72

the load balancing strategy adopted is confirmed to be a good choice, since the efficiencies presented vary from 72% up to 95%. Table 1(a) presents the results when the computation of R does not use high accuracy. In the test case presented in Table 1(b), where the highly accurate computation of R was parallelized, the efficiencies are slightly worse than for the test case (a).

5 Conclusion

A parallel implementation for the self-verified method for solving dense linear systems of equations was discussed in this article. The main objective is to allow the use of this new algorithm in real life situations, which deal with large matrices. The self-verification provides reliability but also decreases the performance. Therefore two main parts of this method, which demand a higher computational cost, were studied, parallelized and optimized. The computation of the approximative inverse of A has been optimized in two different approaches: mathematically and with the use of parallel concepts. The preconditioning step was also parallelized to achieve a better performance.

Two different types of input matrices with five different sizes were used in several experiments aiming to evaluate the speed-up achieved in this new implementation. All five granularities increased the performance with significant gain. The correctness tests also point out a good implementation, where the results were obtained without any loss of accuracy. Thus, the gains provided by the self-verified computation could be kept with a significant decrease in the execution time through its parallelization. The load balancing strategy seems to be a good choice according to the results found in all tested input cases.

One important advantage of verified methods is the ability to find a solution even for very ill-conditioned problems (condition number up to 10^{15}) while most algorithms can only find a solution for a problem with condition number up to 10^8 [13] and may lead to an incorrect result when it is too bad-conditioned (above condition number 10^8). Our main contribution is to increase the use of self-verified computation through its parallelization, once without parallel techniques it becomes the bottleneck of an application. We can notice rather interesting speed-ups for the self-verified computation, reinforcing the statement related to the good parallelization choices.

Finally, it is the opinion of the authors that the results obtained are promising and the implementation allowed a quite good understanding of the problem, leading to new directions for further investigations. Also, other possible optimizations in the use of verified operations must be investigated mathematically in order to guarantee that no unnecessary time consuming operations are being executed.

Acknowledgements The authors would like to thank the Brazilian financial support agencies, that made this research possible. The authors thank the M.Sc students Lucas Baldo and Pedro Velho for their support concerning the parallel implementation. Author Mariana Kolberg acknowledges financial support by CAPES through a PhD scholarship. We are also very grateful to Dr. Gerd Bohlender and Dr. Rudi Klatte, both researchers at University of Karlsruhe, for their support during all the research. At last, we would like to thank the referees for the helpful and interesting suggestions.

References

1. Claudio, D.M., Marins, J.M.: Cálculo Numérico Computacional: Teoria e Prática. Editora Atlas S. A., São Paulo (2000)
2. Bohlender, G.: What Do We Need Beyond IEEE Arithmetic? Computer Arithmetic and Self-validating Numerical Methods. Academic Press, Inc., San Diego, CA (1990)
3. Kulisch, U., Miranker, W.L.: Computer Arithmetic in Theory and Practice. Academic Press, New York (1981)

4. Kulisch, U., Miranker, W.L. (eds.): A new approach to scientific computation. In: Proceedings of Symposium held at IBM Research Center. Yorktown Heights, New York (1983)
5. Miranker, W.L., Toupin, R.A.: Accurate Scientific Computations. vol. 235 of Lecture Notes in Computer Science. Berlin, Springer-Verlag (1986)
6. Klatte, R., Kulisch, U., Lawo, C., Rauch, R., Wiethoff, A.: C-XSC- A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Berlin (1993)
7. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955–1988 (2005)
8. Hölbig, C.A., Krämer, W., Diverio, T.A.: An accurate and efficient selfverifying solver for systems with banded coefficient matrix. In: Proceedings of Parallel Computing (PARCO), pp. 283–290. Germany (2003)
9. Hölbig, C.A., Morandi Júnior, P.S., Alcalde, B.F.K., Diverio, T.A.: Selfverifying solvers for linear systems of equations in C-XSC. In: Proceedings of Parallel and Distributed Programming (PPAM), vol. 3019, pp. 292–297. (2004)
10. Duff, I.S., van der Vorst, H.A.: Developments and Trends in the Parallel Solution of Linear Systems. Technical report RAL TR-1999-027, CERFACS, Toulouse, France (1999)
11. Lo, G.C., Saad, Y.: Iterative Solution of General Sparse Linear Systems on Clusters of Workstations. Technical report umsi-96-117, msi, uofmad (1996)
12. Gonzalez, P., Cabaleiro, J.C., Pena, T.F.: Solving sparse triangular systems on distributed memory multicomputers. In: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, pp. 470–478. IEEE Press (1998)
13. Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Dongarra, J.: Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. Technical report, LAPACK Working Note 175 UT-CS-06-574, University of Tennessee Computer Science (2006)
14. Facius, A.: Iterative solution of linear systems with improved arithmetic and result verification. PhD thesis, University of Karlsruhe, Germany (2000)
15. Kersten, T.: Verifizierende rechnerinvariante Numerikmodule. PhD thesis, University of Karlsruhe, Germany (1998)
16. Wiethoff, A.: Verifizierte globale Optimierung auf Parallelrechnern. PhD thesis, University of Karlsruhe, Germany (1997)
17. Kolberg, M., Baldo, L., Velho, P., Webber, T., Fernandes, L.F., Fernandes, P., Claudio, D.: Parallel Selfverified Method for Solving Linear Systems. 7th VECPAR - International Meeting on High Performance Computing for Computational Science, pp. 179–190. Rio de Janeiro, Brazil (2006)
18. Kolberg, M., Baldo, L., Velho, P., Fernandes, L.F., Claudio, D.: Optimizing a Parallel Self-verified Method for Solving Linear Systems. PARA—Workshop on State-of-the-art in Scientific and Parallel Computing, To appear (2006)
19. Hammer, R., Ratz, D., Kulisch, U., Hocks, M.: C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems. Springer-Verlag, New York, Inc., Secaucus, NJ, USA (1997)
20. Rump, S.M.: Solving Algebraic Problems with High Accuracy. IMACS World Congress, pp. 299–300 (1982)
21. Grimmer, M.: An MPI Extension for the Use of C-XSC in Parallel Environments. Technical report, Wuppertal, Germany (2005), http://www.math.uni-wuppertal.de/wrswt/literatur.html
22. Kaya, D., Wright, K.: Parallel algorithms for LU decomposition on a shared memory multiprocessor. Appl. Math. Comput. **163**(1), 179–191 (2005)
23. Liu, Z., Cheung, D.W.: Efficient parallel algorithm for dense matrix LU decomposition with pivoting on hypercubes. Comput. Math. Appl. **33**(8), 39–50 (1997)
24. Stark, S., Beris, A.N.: LU Decomposition Optimized for a parallel computer with a hierarchical distributed memory. Parallel Comput. **18**(9), 959–971 (1992)
25. Tsao, N.K.: The accuracy of a parallel LU decomposition algorithm. Comput. Mathe. Appl. **20**(7), 25–30 (1990)