Parallel Selfverified Method for Solving Linear Systems

Mariana Kolberg, Lucas Baldo, Pedro Velho, Thais Webber,
Luiz Gustavo Fernandes, Paulo Fernandes and Dalcidio Claudio

Faculdade de Informática, PUCRS
Avenida Ipiranga, 6681 Prédio 16 - Porto Alegre, Brazil
{mkolberg, lbaldo, pedro, twebber, paulof,
gustavo, dalcidio }@inf.pucrs.br

Abstract. This paper presents the parallelization of a self-verified method for solving dense linear equations. Verified computing provides an interval result that surely contains the correct result. The advent of parallel computing and its impact in the overall performance of various algorithms on numerical analysis have been increasing in the last decade. Two main points of this method, which demand a higher computational cost, were carried out: the backward/forward substitution of a LU-decomposed matrix A and an iterative refinement step. Our main contribution is to point out the advantages an drawbacks of our approach, in order to popularize the use of self-verified computation.

1 Introduction

The ability to develop mathematical models in Biology, Physics, Geology and other applied areas has pull, and has been pushed by, the advances in High Performance Computing. Moreover, the use of iterative methods have increased substantially in many application areas in the last years [9, 25]. One reason for that, is the advent of parallel computing and its impact in the overall performance of various algorithms on numerical analysis [4].

The use of clusters plays an important role in such scenario as one of the most effective manner to improve the computational power without increasing costs to prohibitive values. However, in some cases, the solution of numerical problems frequently presents accuracy issues increasing the need for computational power.

Verified computing provides an interval result that surely contains the correct result [16]. Numerical applications providing automatic result verification may be useful in many fields like simulation and modelling. Finding the verified result often increases dramatically the execution time [20]. However, in some numerical problems, the accuracy is mandatory. The requirements for achieving this goal are: interval arithmetic, high accuracy combined with well suitable algorithms.

The interval arithmetic defines the operations for interval numbers, such that the result is a new interval that contains the set of all possible solutions. The 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 direct rounding.

These arithmetics should be used in appropriate algorithms to ensure that those properties will be hold. There is a multitude of tools that provide verified computing, among them an attractive option is C-XSC (C for eXtended Scientific Computing) [15]. C-XSC is a free and portable programming environment for C and C++ programming languages, offering high accuracy and automatic verified results. This programming tool allows the solution of several standard problems, including many reliable numerical algorithms.

For example, in the solution of linear systems even very small systems may present accuracy problems. To illustrate this problem [1], we may observe the solution of the system $Ax = b$ with the following values for A and b:

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

The correct solution would obviously 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 the IEEE double precision arithmetic and LU decomposition the absolutely wrong results would be:

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

One possible solution to cope that problem would be the use of correct computation through the use of interval arithmetic [1].

This paper presents a parallel version of the self-verified method for solving linear systems. Our main contribution is to popularize the use of self-verified computation trough its parallelization, once without parallel techniques it becomes the bottleneck of an application.

The organization of this paper is as follows. In the next section, some related works are discussed and compared to the solution proposed. In section 3, an explanation of the self-verified method used for parallelization and its mathematical background are presented. Section 4 shows the parallel solution for the chosen self-verified method. The analysis of the results obtained through this parallelization is presented in section 5. Finally, the conclusion and some future works are given in last section.

2 Related Work

The solution of large (dense or sparse) linear systems is considered an important part of numerical analysis, and often requires a large amount of scientific computations [9, 25]. More specifically, the most time consuming operations in iterative methods for solving linear equations are inner products, vector successively updates, matrix-vector products and also iterative refinements [7, 12]. Tests pointed out that the Newton-like iterative method, presents a iterative refinement step and uses a inverse matrix obtained through the backward/forward substitution (after LU decomposition), which are the most time consuming operations.

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 [19, 21]. However, the proposed solutions are not often realistic, and mostly deal with unsuitable models for high performance environments of distributed memory as clusters of workstations [22]. In many theoretical models (such as the PRAM family) the transmission cost to data exchange is not considered [22], but in distributed memory architectures this issue is crucial to gain performance.

Nevertheless, the difficulty in parallelizing some numerical methods, mainly iterative schemes, in an environment of distributed memory, is the interdependency among data (*e.g.* the LU decomposition) and the consequent overhead needed to perform interprocess communication (IPC) [3, 30]. Due to this, in a first approach some modifications were done in the backward/ forward substitution procedure [8] to allow less communications and independent computations over the matrix. Another possible optimization when implementing for such parallel environments is to reduce communication cost through the use of load balance techniques, as we can see in some recent parallel solutions for linear systems solvers [30]. Anyway, their focus was toward the issues related to MPI implementation through a theoretical performance analysis. Few works were found related to numerical analysis of parallel implementations of iterative solvers, mainly using MPI. Moreover, some interesting papers found present algorithm which allow the use of different parallel environments [2, 5, 17]. However, those papers (like others) does not deal with verified computation. We also found some works which focus on verified computing [6] and both verified computing and parallel implementations [14, 29], but these thesis implement other numerical problems or use a different parallel approach.

Another concern is the implementation of selfverified numerical solvers which allow high accuracy operations. The researches already made, show that the execution time of the algorithms using this kind of routines is much larger than the execution time of the algorithms which do not use it [11, 10]. The C-XSC library was developed to provide functionality and portability, but early researches indicate that more optimizations may be done to provide more efficiency, due to additional computational cost in sequential, and consequently for other environments as Itanium clusters. Some experiments were conducted over Intel clusters to parallelize selfverified numerical solvers that use Newton-based techniques but there are more tests that may be done [11]. Hereby we propose new adaptations of the current algorithms to speedup this data calculation using technologies as MPI communications functions associated to the C-XSC library to improve higher precision [27], selfverification and speedups at the same time. Moreover, the major goal of this paper is point out the advantages and the drawbacks of the parallelization of a self-verifying method for solving linear systems over distributed environments.

3 Background

One of the most frequent tasks in numerical analysis is the solution of systems of linear equations like:

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

With an $n \times n$ matrix $A \in \mathbb{R}^{n \times n}$ and a right hand side $b \in \mathbb{R}^n$. Many different numerical algorithms contain this task as a subproblem.

In equation 1, we assume the coefficient matrix A in equation 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 $Ax = b$ for a square $n \times n$ matrix A with a better performance as the sequential version.

The algorithm 1 used as base of our parallel version is described in [8] and will, in general, succeed in finding and enclosing a solution or, if it does not succeed, will tell the user so. In the latter case, the user will know that the problem is probably 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**

We give now a brief summary of the enclosure methods theory. A more detailed presentation can be found in [23]. A solution of the system $Ax = b$ be found is equivalent to finding a zero of $f(x) = Ax - b$, such that $A \in \mathbb{R}^{n \times n}$ and $b, x \in \mathbb{R}^n$. Using Newton's method we found the fixed-point iteration presented in equation (2):

$$x_{k+1} = x_k - A^{-1}(Ax_k - b) \tag{2}$$

Where x_0 is an arbitrary starting value. The inverse of A is not known, so we use $R \approx A^{-1}$ as presented in (3):

$$x_{k+1} = x_k - R(Ax_k - b) \tag{3}$$

When replacing x_k for the interval $[x_k]$, the fixed-point theorem will not be satisfied. For this reason, we modify the right-hand side of equation 3, using I as the $n \times n$ identity matrix:

$$x_{k+1} = Rb + (I - RA)x_k \tag{4}$$

An approximate solution $\tilde{x}$ of $Ax = b$ may be improved if we try to enclose the error of the approximate solution finding the residual by solving the system 5, yielding a much higher accuracy. The error $y = x - \tilde{x}$ of the true solution x satisfies the equation:

$$Ay = b - A\tilde{x} \tag{5}$$

Which can be multiplied by R and rewritten in the form:

$$y = R(b - A\tilde{x}) + (I - RA)y \tag{6}$$

Let $f(y) := R(b - A\tilde{x}) + (I - RA)y$. Then equation 6 has the form

$$y = f(y) \tag{7}$$

Of a fixed point equation for the error y. If R is a sufficiently good approximation of A^{-1}, then an iteration based on equation 7 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 equation 7, where we use interval arithmetic and intervals $[y_k]$ for y:

$$[y]_{k+1} = R \diamond (b - A\tilde{x}) + \diamond(I - RA)[y]_k \tag{8}$$

or

$$[y]_{k+1} = F([y]_k) \tag{9}$$

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 cancellations 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 XSC-languages are used extensively in the implementations. With $z = R(b - A\tilde{x})$ (line 3 of algorithm 1) and $C = (I - RA)$ (line 4 of algorithm 1). Thus, equation 8 can be rewritten as:

$$[y]_{k+1} = z + C[y]_k \tag{10}$$

In order to prove the existence of a solution of equation 5 and thus of equation 1, we use Brouwer's fixed point theorem, which applies as soon as we have at some iteration index $k + 1$ an inclusion of the form:

$$[y]_{k+1} = F([y]_k) \subset [y]_k^\circ \tag{11}$$

Where $[y]_k^\circ$ means the interior of $[y]_k$. If this inclusion test (equation 11) holds, then the iteration function f maps $[y]_k$ into itself. From Brouwer's fixed point theorem, it follows that f has a fixed point $y*$ which is contained in $[y]_k$ and in $[y]_{k+1}$. The requirement that $[y]_k$ is mapped into its interior ensures that this fixed point is also unique, *i.e.*, equation 5 has an unique solution $y*$, and thus equation 1 also has a unique solution $x* = \tilde{x} + y*$.

According to [23], if the inclusion test (equation 11) is satisfied, the spectral radius of C (and even that of $|C|$, which is the matrix of absolute values of C) is less than 1, ensuring the convergence of the iteration (also in the interval case). Furthermore, this implies also the nonsingularity of R and of A and thus the uniqueness of the fixed point.

A problem which still remains is that we do not know whether we can succeed in achieving condition, because it may be never satisfied. To force equation 11, we therefore introduce the concept of ε-inflation, which blows up the intervals somewhat, in order to "catch" a nearby fixed point. It can be shown (*e.g.* in [24]) that equation 11 will always be satisfied after a finite number of iteration steps, whenever the absolute value $|C|$ of iteration matrix C has spectral radius less than 1.

We have not yet said how we compute our approximate solution $\tilde{x}$ and the approximate Inverse R. In principle, there is no special requirement 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.

We use in our C-XSC program the LU-Decomposition for the computation of R and $\tilde{x}$. We do not use a special algorithm for the computation of the approximate solution, since we must compute an approximate inverse $R \approx A^{-1}$ anyway. Thus, we also have immediately an approximate solution $\tilde{x} = Rb$. The procedure fails if the computation of an approximate inverse R fails or if the inclusion in the interior cannot be established.

4 Parallel Approach

The Parallel implementation for selfverified method for solving linear systems discussed on this section was developed in order to allow the use of this new algorithm in real situations. Thus, it was necessary to achieve better performance without using parallel programming models oriented to very expensive (but not frequently used) machines. Useful parallel versions for this algorithm should run distributed over several processors connected by a fast network. Therefore, the natural choice was a cluster with a message passing programming model.

As seen in algorithm 1, the selfverified method is divided in some steps. By tests, the computation of the inverse of matrix A (matrix `R` on step 1) takes more than 50% of the total processing time. Similarly, the computation of the interval matrix [C] (parallel iterative refinement) takes more than 40% of the total time, once matrices 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). Due to this, both parts of the algorithm were parallelized, using different techniques.

Our parallel approach involves two different techniques to solve the main bottlenecks of the original algorithm. First, we used a parallel phases approach to achieve

speedup in the core of the computation bottleneck: the computation of the inverse matrix. The second technique is a worker/manager approach to achieve parallel iterative refinement. For the sake of clarity we now present each of these two approaches in two different subsections.

4.1 The Inverse Matrix Computation

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. The LU decomposition of matrix A takes 18% of the total time, whereas the calculation of the inverse by backward/forward substitution takes 34% of the total time. Thus, the parallelization of R computation were focused only in the backward/forward substitution phase due to the higher computational cost. Moreover, the parallel LU decomposition is well spread and many different proposals and can be found in [13, 18, 26, 28]. Those proposals may be perfectly used instead of our choice to find the inverse matrix.

Like said before, the parallel computation of the inverse matrix through backward/ forward substitution is based on the parallel phases scheme, where every process computes a number of columns (co-named task) of matrix R. After the processing phase, all processes exchange information aiming the construction of the overall R. It is important to mention that all processes must know the inverse matrix R, once R is used for further parallel computation of interval matrix C. Figure 1 (a) presents the communication strategy used on this part.

The load balancing approach is made using a simple, yet effective, algorithm that accomplishes two main constraints. (i) The load balancing is done in parallel, so one process does not have to concern about the overall load balancing, and hence spend communication time exchanging information; (ii) It must be fast, so it will not become a bottleneck in the computation process. Thus, the algorithm representing the load balancing strategy for both parallelization is shown in algorithm 2.

Algorithm 2 Workload distribution algorithm.

1: **if** $pr \leq (N\%P)^1$ **then**
2: $\quad lb_i = (\frac{N}{P} \times pr) + (pr)$
3: $\quad ub_i = lb + \frac{N}{P} + 1$
4: **else**
5: $\quad lb_i = (\frac{N}{P} \times pr) + (N\%P)$
6: $\quad ub_i = lb + \frac{N}{P}$
7: **end if**

Where N is the number of rows or columns of a matrix and P the number of processes involved on computation. Also, pr represents the identification of a process (starting from 1 to P), lb_i is the lower bound and ub_i is the upper bound of the i^{th} process, *i.e.*, the first and the last row/column that a process must compute. With this load balancing approach, it is clearly that processes receive continuous block of

rows/columns. This choice was adopted to improve the facility on the implementation of communication step. However, others load balancing schemes could be used without lost of performance, once the computational cost to process a row/column is the same.

4.2 Parallel Iterative Refinement

Iterative Refinement is known as a way to speedup the method convergence based in establishing some algebraic constants used to approximate x at each iteration. Focusing in the iteration formula 4, where I is an identity matrix of the same order of A and R is a good approximation of A^{-1}, as said before, we may compute the $C = (I - RA)$ as a iterative refinement step, once it is composed only by numerical constants. This is step is done in parallel using a worker/manager approach.

The task mapping is done as follows: (i) each process find, based on its rank, a contiguous block of lines to compute the result matrix (C); (ii) after computing each process (unless the manager) send their lines to the manager which put in the proper places the lines received. This step is illustrated by the Figure 1 (b), where the thicker vertical line means the manager process, and the others vertical lines mean the other process. After computation, worker processes send their result to manager process.

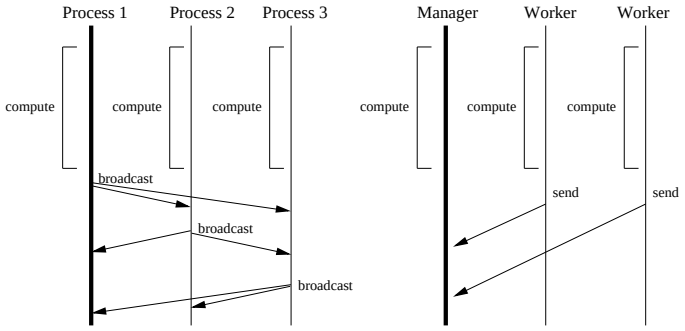


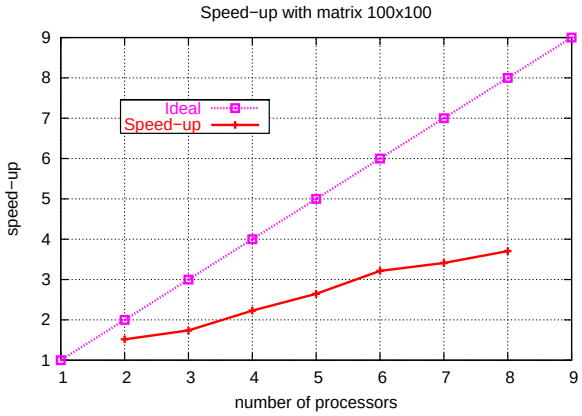
Fig. 1. communication schemes

5 Results Analysis

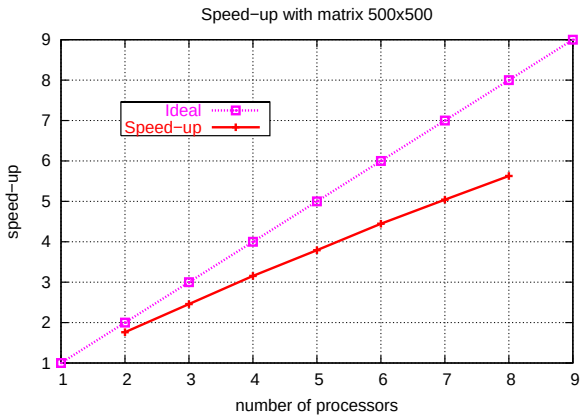
In order to verify the accuracy of our parallel solution, some test cases were made varying on the number of processes and size of the input matrix A and vector b. Matrix A and vector b were generated by two distinct forms: random numbers and using the Boothroyd/Dekker formula [8], where

$$A_{ij} = \binom{\text{n+i-1}}{\text{i-1}} \times \binom{\text{n-1}}{\text{n-j}} \times \frac{n}{i+j-1}, \forall i, j = 1..N$$

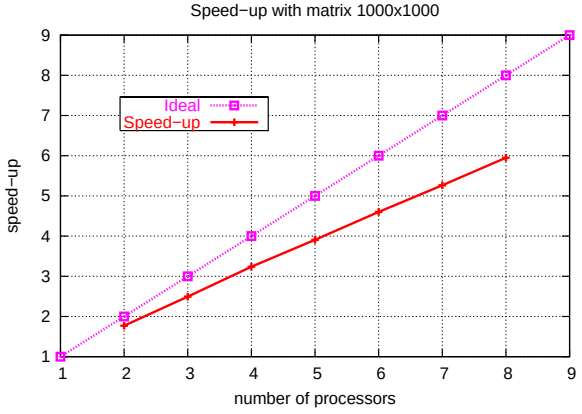
$$b = 1, 2, 3, ..., N.$$



# of processes	2	3	4	5	6	7	8
execution time (sec)	0.7457	0.6506	0.5076	0.4276	0.3520	0.3317	0.3052
efficiency (%)	75.85	57.96	55.71	52.91	53.55	48.72	46.32
sequential time: 1.131445							



# of processes	2	3	4	5	6	7	8
execution time (sec)	74.3002	53.2145	41.4681	34.5165	29.4352	25.9633	23.2476
efficiency (%)	88.06	81.97	78.89	75.83	74.10	72.00	70.36
sequential time: 130.870691							



# of processes	2	3	4	5	6	7	8
execution time (sec)	578.2480	410.7169	316.3659	262.2760	222.6202	194.5636	172.2104
efficiency (%)	88.59	83.15	80.96	78.12	76.70	75.22	74.36
sequential: 1024.563750							

Fig. 2. Results using the random approach.

For the first one, three matrices sizes were carried out: 100×100 (small), 500×500 (medium) and $1,000 \times 1,000$ (large). The second one it was used a matrix 10×10, to verify that the parallelization did not modify the accuracy of the results. For those matrices, many executions were tackled with different number of processes $(1..P)$. All tests were executed over a cluster of workstations environment, with 8 nodes Pentium IV 2.8 Ghz, 1Gb RAM using fast ethernet network for inter-processes communication.

The results using the random approach can be seen in figure 2. The first important remark is that changing the size of matrix, the speedup achieved is better, *i.e.*, as larger is the input matrix (higher computational cost) the better is the performance obtained. Although the speedup for medium and large input matrices does not has significant increases, the results show that the solution proposed improve the performance of the application. For instance, the speedup $\cong 6$ achieved with 8 processes can be considered a good result, since the target architecture is a distributed memory environment.

The execution times presented in of the three tables in figure 2 show a significant decrease, from the sequential time to the parallel time with 8 processes. This analysis reinforces the affirmation related to the good parallelization choices. Moreover, the load balancing strategy adopted has been proven as a good choice, since the efficiencies presented did not show significant processes suballocation. For the first test case (small matrix), the efficiencies are worst than for medium and large matrices. Nevertheless, the efficiencies reached vary from 46.32% up to 88.59%, indicating that tests with higher matrices may tackle important results. The tests using a 10×10 matrix generated by the Boothroyd/Dekker formula presented same accuracy on both versions (sequential and parallel).

6 Conclusion

A parallel implementation of a self-verified method to solve dense linear equations was presented in this paper. Two main points of this method, which demand a higher computational cost, were carried out: the backward/forward substitution of a LU-decomposed matrix A and an iterative refinement step.

Several experiments were conducted in order to verify the strong and weak points of our approach. Three different input matrices were used in these experiments. The parallel implementation presents a significant gain of performance in all three different granularities. The results show that our load balancing strategy was successful for all tested input cases. In all examples, the results were obtained without any lost of accuracy, showing that the gain provided by the self-verified computation could be kept. The exploration of the scalability limits of the proposed parallel solution will permit the establishment of the actual contribution of our work. However, even before the execution of more tests with a higher amount of processors, we can notice rather interesting speedups for the self-verified computation. Finally, it is the authors opinion that the results obtained are interesting and the implementation allowed a quite good understanding of the problem, leading to promising directions for further investigations.

References

1. G. Bohlender. *What Do We Need Beyond IEEE Arithmetic? Computer Arithmetic and Self-validating Numerical Methods* . Academic Press Professional, Inc., San Diego, CA, 1990.
2. R. D. da Cunha and T. Hopkins. The parallel solution of triangular systems of linear equations. Technical Report 86*, University of Kent, Canterbury, UK, June 1991.
3. I. S. Duff. The Impact of High Performance Computing in the Solution of Linear Systems: Trends and Problems. Technical Report RAL TR-1999-072, 1999.
4. I. S. Duff and H. A. van der Vorst. Developments and Trends in the Parallel Solution of Linear Systems. Technical Report RAL TR-1999-027, CERFACS, Toulouse, France, 1999.
5. S. C. Eisenstat, M. T. Heath, C. S. Henkel, and C. H. Romine. Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors. *SIAM J. Sci. Stat. Comput.*, 9(3):589–600, 1988.
6. A. Facius. *Iterative solution of linear systems with improved arithmetic and result verification*. PhD thesis, University of Karlsruhe, Germany, 2000.
7. T. Feng and A. J. Flueck. A Message-Passing Distributed-Memory Newton-GMRES Parallel Power Flow Algorithm. In *Proceedings of the IEEE Power Engineering Society Summer Meeting*, volume 3, pages 1477–1482. IEEE Press, 2002.
8. R. Hammer, D. Ratz, U. Kulisch, and M. Hocks. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
9. G. A. Hedayat. Numerical Linear Algebra and Computer Architecture: An Evolving Interaction. Technical Report UMCS-93-1-5, University of Manchester, Manchester, England, 1993.
10. C. A. Hölbig, P. S. Morandi Júnior, B. F. K. Alcalde, and T. A. Diverio. Selfverifying Solvers for Linear Systems of Equations in C-XSC. In *Proceedings of Parallel and Distributted Programming (PPAM)*, volume 3019, pages 292–297, 2004.
11. C. A. Hölbig, W. Krämer, and T. A. Diverio. An Accurate an Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix. In *Proceedings of Parallel Computing (PARCO)*, pages 283–290, Germany, September 2003.
12. M. T. Heath J. W. Demmel and H. A. van der Vorst. Parallel Numerical Linear Algebra. Technical Report UCB CSD-92-703, Cambridge University, 1993.
13. D. Kaya and K. Wright. Parallel Algorithms for LU Decomposition on a Shared Memory Multiprocessor. *Applied Mathematics and Computation*, 163(1):179–191, 2005.
14. T. Kersten. *Verifizierende rechnerinvariante Numerikmodule*. PhD thesis, University of Karlsruhe, Germany, 1998.
15. R. Klatte, U. Kulisch, C. Lawo, R. Rauch, and A. Wiethoff. *C-XSC- A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, Berlin, 1993.
16. U. Kulisch and H. J. Stetter (Eds.). *Scientific Computation with Automatic Result Verification*. Springer-Verlag, New York, 1988.
17. G. Li and T. F. Coleman. A new method for solving triangular systems on distributed memory message-passing multiprocessors. *SIAM J. Scientific & Statistical Computing*, 10:382–396, 1989.
18. Z. Liu and D. W. Cheung. Efficient Parallel Algorithm for Dense Matrix LU Decomposition with Pivoting on Hypercubes. *Computers & Mathematics with Applications*, 33(8):39–50, 1997.
19. G. C. Lo and Y. Saad. Iterative Solution of General Sparse Linear Systems on Clusters of Workstations. Technical Report umsi-96-117, msi, uofmad, 1996.
20. T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum and Dot Product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

21. J. C. Cabaleiro P. Gonzalez and T. F. Pena. Solving Sparse Triangular Systems on Distributed Memory Multicomputers. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, pages 470–478. IEEE Press, January 1998.
22. V. Pan and J. Reif. Fast and Efficient Parallel Solution of Dense Linear Systems. *Computers & Mathematics with Applications*, 17(11):1481–1491, 1989.
23. S. M. Rump. Solving Algebraic Problems with High Accuracy. In *IMACS World Congress*, pages 299–300, 1982.
24. S.M. Rump. Convergence Properties of Iterations Using Sets. Technical Report 15(6):427-432, TU Leipzig, Wissenschaftliche Zeitschrift, 1991.
25. Y. Saad. *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Company, 1995.
26. S. Stark and A. N. Beris. LU Decomposition Optimized for a Parallel Computer with a Hierarchical Distributed Memory. *Parallel Computing*, 18(9):959–971, 1992.
27. S. M. Rump T. Ogita and S. Oishi. Accurate Sum and Dot Product with Applications. In *2004 IEEE International Symposium on Computer Aided Control Systems Design*, LNCS, pages 152–155, Taipei, Taiwan, September 2004. IEEE Press.
28. N. K. Tsao. The Accuracy of a Parallel LU Decomposition Algorithm. *Computers & Mathematics with Applications*, 20(7):25–30, 1990.
29. A. Wiethoff. *Verifizierte globale Optimierung auf Parallelrechnern*. PhD thesis, University of Karlsruhe, Germany, 1997.
30. J. Zhang and C. Maple. Parallel Solutions of Large Dense Linear Systems Using MPI. In *International Conference on Parallel Computing in Electrical Engineering, PARELEC '02*, pages 312–317. IEEE Computer Society Press, 2002.