Parallel Verified Linear System Solver for Uncertain Input Data

Mariana Kolberg
Márcio Dorn
Luiz Gustavo Fernandes
PUCRS - PPGCC
Av. Ipiranga, 6681
Porto Alegre, Brazil
{mkolberg,mdorn,gustavo}@inf.pucrs.br

Gerd Bohlender
Universitaet Karlsruhe
Kaiserstr. 12 – 76128
Karlsruhe, Germany
bohlender@math.uka.de

Abstract

This paper presents a new parallel implementation for solving dense interval linear systems with verified computing. The use of intervals appears as one possible way to handle the uncertainty of input data in real problems. A verified method using midpoint-radius arithmetic and directed roundings was combined with optimized libraries such as SCALAPACK and PBLAS to provide a free, fast, reliable and accurate solver. Accuracy and performance results for executing this implementation in a cluster are shown. It is the authors opinion that the combination of verified and parallel computing is a powerful tool that could be used for several other mathematical problems.

1 Introduction

Many real problems are simulated and modeled using dense linear systems of equations. In many problems the input data is obtained from measurement that are usually done using tools that are not always precise. This fact, among others reason that generate uncertainties (see Section 2), can cause errors in the evaluation of the system, and this can be specially bad for problems that need an accurate result. One possible way to handle the uncertainty of input data is using intervals.

To solve an interval linear systems means that an infinite number of matrices contained in the interval should be solved. The exact solution set of an interval linear system has a complicated non-convex shape which is difficult to compute [20]. The only possible way to find a solution is to compute a narrow interval that contains this set.

Each of these matrices has a different condition number, and the larger the dimension of the matrix and the radius of the matrix are, the larger is the probability that it will contain an ill-conditioned or a singular matrix.

Many different numerical algorithms contain this kind of task as a sub-problem [2, 25, 24]. A large number of these problems can be solved through a dense interval linear system of equations like

$$[A]x = [b] \tag{1}$$

with a $n \times n$ interval matrix $[A] \in \mathbb{IR}^{n \times n}$ and a right hand side interval vector $[b] \in \mathbb{IR}^n$.

When such a system is solved using computers, several rounding errors can occur during its computation. Even well conditioned systems can lead to completely wrong results and this effect can be worse for ill-conditioned matrices. The classical solutions for linear systems of equations using floating-point arithmetic can only deliver an approximation. Since the correct result is unknown, it is not clear how good these approximations are, or if there exists a unique solution at all.

One possible way to obtain reliable results is using verified computing [9]. Verified computing provides an interval result that surely contains the correct result [16, 12]. In this case, it will either deliver the enclosure of the correct solution or do not deliver any result at all.

The verified method for solving linear systems of equation is composed by the computation of the approximate solution and an interval Newton-like iteration. This increases the computational cost to solve such a system, specially when dealing with large matrices. The research already developed shows that the execution times of verified algorithms using e.g. C-XSC (C for eXtended Scientific Computing) [12] are much larger than the execution times of algorithms that do not use this concept even for parallel implementations [10, 11, 13, 15]. Additionally, if the input data are an interval matrix and an interval vector, this cost is two times larger since every interval is defined with two numbers: the midpoint and the radius of the interval (see

1550-6533/08 $25.00 © 2008 IEEE
DOI 10.1109/SBAC-PAD.2008.10

section 3).

There are some advantages of using midpoint-radius representation instead of the traditional infimum-supremum representation. The main advantage is that no case distinctions or switching of rounding mode in inner loops of vector and matrix multiplication are needed. In this case, the multiplication of a vector or matrix could be implemented using pure floating point operations and highly optimizes libraries like BLAS (Basic Linear Algebra Subprograms) [4] could be used to implement it in a much faster way. However, when using infimum-supremum arithmetic, this is not possible. In this case, each interval multiplication has to be executed differently depending if the interval is positive, negative or if it contains zero. Therefore, optimized libraries cannot be used. Even with this advantage, midpoint-radius arithmetic combined with these libraries still represents a considerable computational cost when dealing with large dense interval systems.

In this context, the use of high performance computation techniques appears as a useful tool to drop down the time needed to solve interval systems using verified computing. With the advent of large clusters composed by hundreds or thousands of nodes connected through a gigabit network, huge linear systems can be computed in parallel [8]. The well-known libraries such as PBLAS (Parallel Basic Linear Algebra Subprograms) and SCALAPACK (SCAlable Linear Algebra PACKage) [3] alone are not suited for interval systems based on verified methods once these methods introduce several steps to deliver a guaranteed result. The present work proposes a new cluster solution for solving interval linear systems using high performance computing with MPI [23] and combining the previous mentioned libraries with verified computing techniques in order to treat uncertain data represented by intervals.

The authors have presented a parallel verified method for solving linear systems of equations with point input data, listing also additional references on previous work on the subject [14]. The present research proposes a new parallelization of the verified method for interval input data. The idea is to use popular and highly optimized libraries to gain performance and verified methods to have guaranteed results. In other words, the new implementations try to join the best aspects of each library:

- SCALAPACK and PBLAS: performance.
- Verified method like the one used in C-XSC toolbox: guaranteed results.

Moreover, the major goal of this research is to provide a free, fast, reliable and accurate solver for dense interval linear systems.

This text is organized as follows. Section 2 presents some aspects of uncertain data that can be handled using interval mathematics, Section 3 describes the verification of linear systems, Section 4 presents the implementation issues, Section 5 shows some experimental results, and finally Section 6 concludes the work and present some future works.

2 Uncertain Data

Engineering and scientific problems are frequently modeled by a numerical simulation on a computer. Such simulations have many advantages:

- usually, a computer simulation is much less expensive; e.g. new vehicles are often designed and tested in a "numerical wind tunnel" and in simulated crash tests instead of building a prototype and performing real tests (which may even lead to the destruction of the prototype);
- this is also less time consuming and leads to a much faster design process;
- a simulation does not have the risks which may be caused by a true experiment, e.g. in nuclear energy or aviation and space engineering;
- many experiments are difficult to perform or they even cannot be performed at all; e.g. in astronomy, particle physics, biology, geology.

However, the question arises, how reliable are the results of such numerical simulations. A reliable solution of an engineering or scientific problem should not only give an approximate numerical result but also rigorous error bounds and a proof that the solution is correct within these bounds; e.g. that a solution exists (and possibly is unique) in a mathematical sense.

There are many sources of uncertainty or error which have to be taken care of in a numerical simulation. These include

- measurement errors, unknown or imprecise data;
- errors introduced by the mathematical model of the problem, frequently described by differential equations;
- mathematical errors introduced by the discretization / linearization of the model, leading to linear systems;
- rounding errors in the floating-point evaluation of the numerical approximation.

Interval arithmetic methods can handle most of these sources of error:

- imprecise measured data can be represented by intervals,

- errors in the model or in the mathematical treatment of the model can be represented by an additional interval term,
- rounding errors may be controlled by interval arithmetic operations.

Another important aspect is that intervals represent the continuum of numbers: a floating-point interval $[a, b]$ is the set of all real numbers between a and b, not just the floating-point numbers. Therefore, in contrast to floating-point computations, interval methods can give rigorous mathematical results; e.g. let $F(X)$ be an interval extension of a mathematical function $f(x)$, and let $F([a, b]) > F([c, d])$, then there exists no global maximum of the function f(x) in the interval $[c, d]$, and no global minimum in $[a, b]$, resp.

In the current paper, we concentrate on a special aspect of this numerical simulation. We consider linear equations with interval coefficients. These interval coefficients may be the result of an imprecise model or of measurement errors. We compute an interval enclosure of the result set.

3 Verification of a Linear System of Equations

Finding the solution of a linear system of equations is one of the basic problems in numerical algebra [9]. A verified method for solving this problem for dense square systems is based on a Newton-like method for an equivalent fixed-point problem. Subsection 3.1 will present the basic concepts of verified computing and subsection 3.2 presents how this concepts can be applied to solve a linear system of equations.

3.1 Verification

Verified computing provides an interval result that surely contains the correct result [16]. The algorithm will, in general, succeed in finding an enclosure of the solution. If the solution is not found, the algorithm will let the user know. An enclosure can be defined as an interval that contains the correct result. This makes a qualitative difference in scientific computations, since the results are now intervals in which the exact result must lie. Numerical applications providing automatic result verification may be useful in many fields like Simulation and Modeling [6]. Specially, when accuracy is mandatory.

The use of verified computing guarantees the mathematical rigor of the result. This is the most important advantage of such algorithms compared with ordinary methods.

One possibility to implement these ideas is using interval arithmetic and directed rounding combined with 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. This ensures that the result is an enclosure, verifying the result.

The most frequently used representation for intervals over the set of real numbers ($\mathbb{R}$), is the infimum-supremum representation [1, 18]

$$[a_1, a_2] := \{x \in \mathbb{R} : a_1 \leq x \leq a_2\} \tag{2}$$

for some $a_1, a_2 \in \mathbb{R}$, $a_1 \leq a_2$, where $\leq$ is the partial ordering $x \leq y$. Another possible representation is the midpoint-radius representation defined as

$$\langle a, \alpha \rangle := \{x \in \mathbb{R} : |x - a| \leq \alpha\} \tag{3}$$

for some $a \in \mathbb{R},\ 0 \leq \alpha \in \mathbb{R}$.

Both arithmetics are equivalent for the theoretical operations in $\mathbb{R}$ where no final rounding of the results is necessary. However, it changes when looking at operations in floating-point arithmetic [22].

The midpoint-radius arithmetic is defined for floating point operations in [22]. Considering a set $\mathbb{F} \subseteq \mathbb{R}$ of real floating point numbers, it is possible to define $\mathbb{I}^+\mathbb{F}$ as follows:

$$\mathbb{I}^+\mathbb{F} := \{\langle \tilde{a}, \tilde{\alpha} \rangle : \tilde{a}, \tilde{\alpha} \in \mathbb{F},\ \tilde{\alpha} \geq 0\}$$

where

$$\langle \tilde{a}, \tilde{\alpha} \rangle := \{x \in \mathbb{R} : \tilde{a} - \tilde{\alpha} \leq x \leq \tilde{a} + \tilde{\alpha}\}$$

Then, $\mathbb{I}^+\mathbb{F} \subseteq \mathbb{IR}$. Note that the pair of floating point numbers $\tilde{a}, \tilde{\alpha}$ describes an infinite set of real numbers for $\tilde{\alpha} \neq 0$.

When implementing this arithmetic on computers, special care has to be taken for the rounding. A rounding error will be generated in the midpoint evaluation. This error should be compensated using the relative error unit. Rump denotes the relative rounding error unit by ϵ, sets $\epsilon' = \frac{1}{2}\epsilon$, and denotes the smallest representable (unnormalized) positive floating point number by η. In IEEE 754 double precision, $\epsilon = 2^{-52}$ and $\eta = 2^{-1074}$.

Each of these representations has advantages and disadvantages. It is known that the standard definition of midpoint-radius arithmetic causes overestimation for multiplication and division. Another possible cause of overestimation is that sometimes the computed midpoint is not exactly representable in floating point. However, in [22], Rump shows that the overestimation of operations using midpoint-radius representation compared to the result of the corresponding power set operation is limited by at most a factor 1.5 in radius. There are many examples of intervals that can be better represented in one arithmetic and have an overestimation in the other.

Besides the overestimation, another important point is how the basic operations $(+, -, \cdot, \div)$ must be implemented in a computer. The operations for infimum-supremum, specially the multiplication, must be carefully implemented,

since depending on the sign of the number, a different case will be used to compute the operation. For midpoint-radius operations, it does not happen. All operations are directly defined and no case distinction is needed.

The main point in using midpoint-radius arithmetic is that no case distinctions, switching of rounding mode in inner loops, etc. are necessary, only pure floating point operations. This is specially good for matrix multiplication where the fastest algorithms available may be used, for example the BLAS routines. This gives an advantage in computational speed which is difficult to achieve by other implementations.

3.2 Linear Systems

A suitable algorithm to solve a dense square linear system of equations ensuring a verified result is presented in Algorithm 1. This algorithm is based on the verified method that is fully described in [9, 21]. This algorithm gives an enclosure solution for the problem. The user can be sure that the exact result is contained in this interval. If the algorithm fails in finding an enclosure, it means that the problem is very ill-conditioned or that the matrix A is singular.

Algorithm 1 Enclosure of a square interval linear system

1: $R \approx mid([A])^{-1}$ {Compute an approximate inverse using LU-Decomposition algorithm}
2: $\tilde{x} \approx R \cdot mid([b])$ {compute the approximation of the solution}
3: $[z] \supseteq R([b] - [A]\tilde{x})$ {compute enclosure for the residuum}
4: $[C] \supseteq (I - R[A])$ {compute enclosure for the iteration matrix}
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**

This enclosure method for finding the result of a system like $[A]x = [b]$ with an $n \times n$ interval matrix $[A] \in \mathbb{IR}^{n \times n}$ and a right hand side interval vector $[b] \in \mathbb{IR}^n$ is based on the Newton-like iteration

$$x_{k+1} = R[b] + (I - R[A])x_k, k = 0, 1, ... \qquad (4)$$

An approximate solution $\tilde{x}$ of $[A]x = [b]$ may be improved if we try to enclose the error of the approximate solution.

If R is a sufficiently good approximation of $(mid[A])^{-1}$, then an iteration can be expected to converge since $(I - R[A])$ will have a small spectral radius. These results remain valid if we replace the exact expression by interval extensions.

To compute our approximate solution $\tilde{x}$ and the approximate inverse R, we used the LU- decomposition. In principle, there are 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.

4 Parallel Implementation

The idea of this implementation is to increase the performance of a solver for dense linear systems using a verified method (but no special library for verification, since those libraries can increase the computational time significantly [13]), interval arithmetic, directed roundings and optimized numerical libraries like PBLAS and SCALAPACK aiming to provide both self-verification and speed-up at the same time.

To implement the parallel version of Algorithm 1, we used an approach for cluster architectures with message passing programming model (MPI) and wherever possible the highly optimized libraries PBLAS and SCALAPACK. Matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [7] used by SCALAPACK.

The self-verified method presented above is divided in several steps. By tests [15], the computation of R (the approximate inverse of matrix $mid([A])$) takes more than 50% of the total processing time. Similarly, the computation of the interval matrix $[C]$, which contains the exact value of $(I - R[A])$ (iterative refinement) takes more than 40% of the total time, since matrix multiplication requires O(n^3) execution time. Those are the two most computational intensive operations in the method compared to the other operations that are mostly vector or matrix-vector operations which require at most O(n^2). Therefore, these two steps must be carefully parallelized, aiming at a better performance.

4.1 Approximate Solution

The first two steps of the algorithm compute R, the approximate inverse matrix of $mid([A])$, and x, the approximate solution of the linear system.

Since the input matrix is an interval matrix, a decision should be made about how the steps 1 and 2 of the Algorithm 1 will be implemented. In both, the mathematical and performance point of views, the approximate inverse matrix R should be computed using just the midpoint matrix, i.e. a point matrix. In this case, the result will be much more accurate than using the interval input data, because the interval inverse matrix $[A]^{-1}$ would contain huge intervals with almost no information.

Aiming at a good performance, the idea is to find the approximate inverse R and the approximate solution x using just the midpoint matrix and floating point operations. Later on, the computation of the residuum will use interval arithmetic and the original interval matrix $[A]$ and the interval vector $[b]$ to guarantee that we will find the most accurate result possible.

For the computation of the approximation R in step 1 the following SCALAPACK routines are used:

- *pdgetrf* - computes in parallel a LU factorization of a general matrix using partial pivoting with row interchanges) using double precision;
- *pdgetri* - computes in parallel the inverse of a matrix using the LU factorization computed by *pdgetrf* using double precision.

For the computation of the approximative solution x in step 2 the following PBLAS routine is used:

- *pdgemv* - performs the matrix-vector operation in parallel using double precision.

4.2 Enclosures for the Verification Iteration

Steps 3 and 4 from algorithm 1 compute the enclosure needed to start the verification iteration. Step 3 computes the enclosure for the residuum and step 4 the enclosure for the iteration matrix.

Since $[A]$ and $[b]$ are respectively an interval matrix and vector for the evaluation of $[C]$ and $[z]$, it is essential to use the midpoint-radius interval arithmetic.

Like presented in [22], the algorithms 2 and 3 describe the operations using midpoint-radius arithmetic and IEEE 754 arithmetic standard. Let

$$A = \langle \tilde{a}, \tilde{\alpha} \rangle \in \mathbb{I}^+\mathbb{F} \text{ and } B = \langle \tilde{b}, \tilde{\beta} \rangle \in \mathbb{I}^+\mathbb{F}$$

be given. Then, interval addition and subtraction $C := A \circ B \in \mathbb{I}^+\mathbb{F}, \circ \in \{+, -\}$, with $C = \langle \tilde{c}, \tilde{\gamma} \rangle$ are defined by the algorithm 2 and the multiplication is presented in algorithm 3. The symbols $\square$, $\triangle$ and $\bigtriangledown$ mean that the operation will be done respectively with rounding to nearest, rounding up and rounding down.

Algorithm 2 Midpoint-radius addition and subtraction in $\mathbb{F}$

1: $\tilde{c} = \square(\tilde{a} \circ \tilde{b})$
2: $\tilde{\gamma} = \triangle(\epsilon'|\tilde{c}| + \tilde{\alpha} + \tilde{\beta})$

Algorithm 3 Midpoint-radius multiplication in $\mathbb{F}$

1: $\tilde{c} = \square(\tilde{a} \cdot \tilde{b})$
2: $\tilde{\gamma} = \triangle(\eta + \epsilon'|\tilde{c}| + (|\tilde{a}| + \tilde{\alpha})\tilde{\beta} + \tilde{\alpha}\tilde{\beta})$

As discussed in Subsection 3.1, the major advantage of midpoint-radius arithmetic is that the matrix operations use exclusively pure floating point matrix operations and there is no need for case distinctions. Therefore, Step 4 (the calculation of $Z = R(b - A \cdot x)$) is implemented using the PBLAS routine *pdgemv* and directed roundings.

The calculation of $C = (I - R \cdot [A])$ uses the PBLAS routine *pdgemm* (for double precision matrix multiplication) twice: once rounded up and once rounded down to find lower and upper bounds of the interval matrix $[C]$. Then, floating point operations are performed with rounding up for the evaluation of midpoint of C ($\tilde{c}$) and radius of C ($\tilde{\gamma}$) as presented in algorithm 4.

Algorithm 4 Midpoint-radius matrix multiplication in $\mathbb{F}^n$

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

4.3 Verification Iteration

The verification iteration, is composed by steps 5 to 15. It implements a Newton-like iteration to find the enclosure of the correct solution.

These steps use the midpoint-radius arithmetic with direct roundings. In order to implement it, the PBLAS routine *pdgemv* was used in step 8 to compute the interval vector/matrix multiplication. This iteration does not represent a large amount of computation time, since its operation are of complexity O(n^2). Even so, the parallelization introduced by the PBLAS routine helps to improve the overall performance.

The while statement in step 6 checks if the new result is contained in the interior of the previous result. If it is positive, the while loop is over, and the enclosure was found. If not, it tries for 10 iterations to find the enclosure. It is an empirical fact that the inner inclusion is satisfied nearly after a few steps or never [9]. In this part of the algorithm, the processors have to communicate to check if the enclosure was found (since the result vectors containing the midpoint and the radius are split among the processors).

5 Experimental Results

The experiments performed to test the methods proposed in this paper have been carried out in the same environment using the same input data accordingly to the following descriptions. After that, analysis about the performance and accuracy of the obtained results are presented at the end of this section.

5.1 Platform

The software platform adopted to implement the proposed solution is composed of optimized versions of libraries SCALAPACK and PBLAS (MKL 10.0.011) plus the standard Message Passing Interface (MPI), more specifically the mpich implementation (version 1.2). The basic operating system on each node is HP XC Linux for High Performance Computing (HPC) and the compiler used was the gcc version 3.4.6.

The hardware platform is the HP XC6000 cluster located at the Computing Center of the University of Karlsruhe. The HP XC6000 is a distributed memory parallel computer with three different sets of nodes all in all. In this work, all experiments were carried out over the set composed by 108 2-way HP Integrity rx2620 nodes with 12 Gb memory, each one containing 2 Intel Itanium2 processors (1.5 GHz) with a 6Mb of level 3 cache. The theoretical peak performance of the system is 1.9 TFLOPS. All nodes are connected to the Quadrics QsNet II interconnection which shows a bandwidth over 800 MB/s and a low latency.

5.2 Input Data

The input interval matrix $[A]$ and vector $[b]$ were generated as follows:

- The midpoint matrix $mid([A])$ and midpoint vector $mid([b])$ were generated with random numbers.
- The radius matrix $rad([A])$ and the radius vector $rad([b])$ were filled with the value $0.1 \cdot 10^{-10}$.

The tests were performed using different matrix dimensions from $1,000$ to $9,000$. Larger dimensions were not tested due to the limit of memory. This algorithm generates the matrices and vectors in one processor and distribute it among the others. Therefore no larger matrices could be generated. As a future work, the authors intend to generate the blocks of the matrices and vectors in each processor, making possible to solve even larger linear systems.

5.3 Performance Results

The results presented in this section were obtained through a mean of 10 executions discarding the lowest and the highest times. This procedure is required in order to minimize the influence of environment fluctuations in the final results.

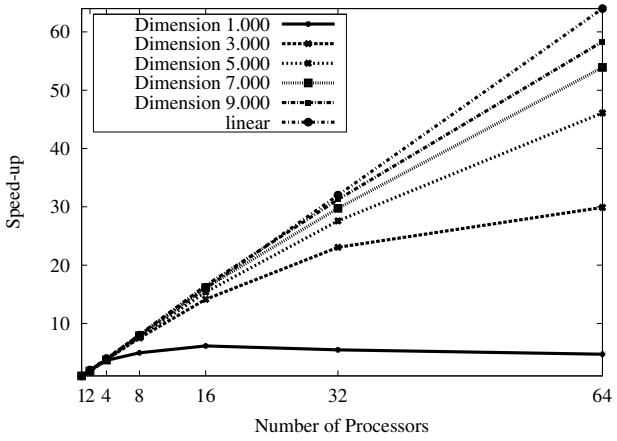

Figure 1. Speed-up

Figure 1 shows the speed-up curves for matrix dimensions varying from dimension $1,000$ to dimension $9,000$. As expected, the speed-up of matrix dimension $1,000$ drops down very quickly after the execution over few processors (8 in this case, since from 8 processors the amount of computation is not so large compared with the cost of communication among the processors). After that dimension, the speed-ups follow the growth of the amount of data to be computed. For matrix dimension $9,000$, the speed-up factor is around 58.3 for 64 processors, indicating that the proposed solution scales well. Experiments with matrices with dimension larger than $9,000$ presented irregular behaviors, certainly due to memory allocation problems since the matrices are not allocated in each node, but generated in one node and then distributed.

Table 1. Summary of the test cases: execution times (T) in seconds and efficiency ($E_n = Sp_n/n$, where $Sp_n = t_1/t_n$).

dimension		number of processors						
		1	2	4	8	16	32	64
1,000	T	2.74	1.34	0.74	0.55	0.44	0.50	0.58
	E	1.00	1.01	0.91	0.62	0.38	0.17	0.07
3,000	T	62.46	31.61	16.35	8.31	4.42	2.70	2.08
	E	1.00	0.98	0,95	0.93	0.88	0.72	0.46
5,000	T	265.72	144.94	71.43	34.46	17.28	9.63	5.76
	E	1.00	0.91	0.93	0.96	0.96	0.86	0.72
7,000	T	700.79	381.85	187.91	88.80	43.44	23.56	13.00
	E	1.00	0.91	0.93	0.98	1.00	0.92	0.84
9,000	T	1467.71	803.37	375.27	182.34	88.92	46.95	25.15
	E	1.00	0.91	0.97	1.00	1.03	0.97	0.91

Looking closer to the results (see Table 1), it is possible

to identify four points where superlinear speed-ups appear: i) matrix dimension 1,000 over 2 processors; ii) matrix dimension 7,000 over 16 processors; iii) matrix dimension $9,000$ over 8 processors and iv) matrix dimension $9,000$ over 16 processors.

The first case can be easily explained by the fact that the matrix is too small and both MPI processes were allocated to processors sharing the same node and cache memory causing a cache effect. The communication time needed to exchange data between the MPI processes decreases dramatically when compared with the situation in which the processors are in different nodes.

For the remaining cases, the superlinear speed-ups can be explained by the memory needed to run the sequential program. The amount of memory in that case is more than ten times larger than the memory needed to run experiments over 8 or 16 processors. For instance, it is necessary to use approximately 5.18 Gb[1] of one processor's memory to run the sequential program against 324 Mb per processor when executing over 16 processors [2]. This situation leads to a much smaller number of page faults for the executions with 8 or 16 processors. In the case of a larger number of processors (e.g., 32 or 64), this effect fades down because the communication cost becomes comparatively more significant when the amount of data to be processed decreases.

5.4 Accuracy Results

The accuracy depends on the condition number of all possible matrices in $[A]$. For well conditioned problems, the new algorithm may deliver a very accurate result with up to 16 correct digits.

For example, let $[A]$ be an $1,000$ x $1,000$ interval matrix and $[b]$ an $1,000$ interval vector, both generated as described in subsection 5.2. The generated midpoint matrix $mid([A])$ has a condition number $5.77 \cdot 10^{+04}$. The developed parallel algorithm with 4 processors will deliver the solution for the first 10 positions of vector $[x]$ presented in table 2.

These parallel result are in fact more accurate than the sequential result presented in table 3 in almost all cases.

As can be seen in tables 2 and 3, the radius of the parallel solution is smaller than the sequential version (possibly by changes in the sequence of operations). Since we divided the problem in smaller parts, it is possible that the condition of the scalar product of smaller parts is better than the condition of the complete scalar product. This effect was already studied in [5, 17]. In this paper, it was shown that if the summation is done in pairs, the final result is more accurate than doing the whole summation at once. This effect can be a reason why we found a more accurate result using the parallel algorithm. Theoretically, it would also be possible to have a worse conditioned scalar product when dividing it into parts. In our test, we did not find any case where the parallel solution was less accurate than the sequential. Despite of this small difference, there was no loss of accuracy in the results. It is important to mention that as required by the algorithm, both results contain the exact result.

[1] 8 bytes (double) * 9,000 (rows) * 9,000 (columns) * 8 (number of matrices needed to solve the problem)

[2] 5.18 Gb divided by 16

Table 2. Parallel Midpoint-radius Result

res	Midpoint	Radius
x[0] =	$-1.3287445 \cdot 10^{+00}$	$6.63489228 \cdot 10^{-07}$
x[1] =	$1.48876460 \cdot 10^{+00}$	$9.50282020 \cdot 10^{-07}$
x[2] =	$-1.4043083 \cdot 10^{+00}$	$5.29588854 \cdot 10^{-07}$
x[3] =	$3.77801931 \cdot 10^{+00}$	$1.19457841 \cdot 10^{-06}$
x[4] =	$1.51834157 \cdot 10^{+00}$	$7.76385827 \cdot 10^{-07}$
x[5] =	$-1.0618383 \cdot 10^{+00}$	$5.61832108 \cdot 10^{-07}$
x[6] =	$4.75233990 \cdot 10^{-01}$	$5.98872205 \cdot 10^{-07}$
x[7] =	$-2.0933523 \cdot 10^{+00}$	$7.28515143 \cdot 10^{-07}$
x[8] =	$1.08921106 \cdot 10^{-01}$	$4.62543062 \cdot 10^{-07}$
x[9] =	$-9.1790106 \cdot 10^{-02}$	$4.58689372 \cdot 10^{-07}$

Table 3. Sequential Midpoint-radius Result

x[0] =	$-1.3287445 \cdot 10^{+00}$	$1.34971880 \cdot 10^{-06}$
x[1] =	$1.48876460 \cdot 10^{+00}$	$1.93288065 \cdot 10^{-06}$
x[2] =	$-1.4043083 \cdot 10^{+00}$	$1.07750578 \cdot 10^{-06}$
x[3] =	$3.77801931 \cdot 10^{+00}$	$2.42986973 \cdot 10^{-06}$
x[4] =	$1.51834157 \cdot 10^{+00}$	$1.57890684 \cdot 10^{-06}$
x[5] =	$-1.0618383 \cdot 10^{+00}$	$1.14296727 \cdot 10^{-06}$
x[6] =	$4.75233990 \cdot 10^{-01}$	$1.21834560 \cdot 10^{-06}$
x[7] =	$-2.0933523 \cdot 10^{+00}$	$1.48170529 \cdot 10^{-06}$
x[8] =	$1.08921106 \cdot 10^{-01}$	$9.41392800 \cdot 10^{-07}$
x[9] =	$-9.1790106 \cdot 10^{-02}$	$9.33610885 \cdot 10^{-07}$

6 Conclusions

This work presents a parallel implementation of a verified method for solving linear systems of equations for uncertain data. The implementation was able to deliver an enclosure of the correct solution for interval input data with considerable accuracy.

The performance tests show that the algorithm scales well specially for larger interval input matrices. The speed-up found for dimension 9,000 using 64 processors was around 58, which is close to linear speed-up.

The following points are considered for future work:

- Matrices and vectors are presently generated by one processor, and after that, distributed among the other processors. This can be a problem since the memory of the processor that contains all the matrix [A] is not enough to store large matrices (larger than dimension 10,000 on the available cluster). A natural future work is to generate the blocks of data in each processor, so that larger matrices could be solved.

- Another future work is to use special routines to improve the accuracy in particularly ill-conditioned cases. This could be done using, for example, the *dotk* routines presented in [19].

- There are also plans to implement a parallel verified method for solving sparse interval linear systems. The most important cases of sparse matrices are band-shaped and general sparse matrices.

The authors believe that combining verified and parallel computing is a powerful tool for having fast, reliable and accurate solvers for several important mathematical problems.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computation*. New York: Academic Press, 1983.

[2] M. Baboulin, L. Giraud, and S. Gratton. A parallel distributed solver for large dense symmetric systems: applications to geodesy and electromagnetism problems. Technical Report TR/PA/04/16, CERFACS, Toulouse, France, 2004. Preliminary version of an article published in Int. J. High Speed Computing, vol. 19, nber 4, pp 353-363, 2005.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A.Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*.

[5] O. Caprani. Implementation of a low round-off summation method. *BIT Numerical Mathematics*, 11(3):271–275, 1971.

[6] D. M. Claudio and J. M. Marins. *Cálculo Numérico Computacional: Teoria e Prática*. Editora Atlas S. A., São Paulo, 2000.

[7] J. Dongarra and D. Walker. Lapack working note 58: The design of linear algebra libraries for high performance computers. Technical report, Knoxville, TN, USA, 1993.

[8] 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.

[9] 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.

[10] C. A. Hölbig, P. S. M. Júnior, B. F. K. Alcalde, and T. A. Diverio. Selfverifying Solvers for Linear Systems of Equations in C-XSC. In *Proceedings of Parallel and Distributed Programming (PPAM)*, volume 3019, pages 292–297, 2004.

[11] C. A. Hölbig, W. Krämer, and T. A. Diverio. An Accurate and Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix. In *Proceedings of Parallel Computing (PARCO)*, pages 283–290, Germany, September 2003.

[12] 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.

[13] M. Kolberg, L. Baldo, P. Velho, L. F. Fernandes, and D. Claudio. Optimizing a Parallel Self-verified Method for Solving Linear Systems. *Lecture Notes in Computer Science: Applied Parallel Computing. State of the Art in Scientific Computing 8th International Workshop, PARA 2006, Umea, Sweden, June 18-21, 2006, Revised Selected Papers*, 3:949–955, 4699/2007 2007.

[14] M. Kolberg, G. Bohlender, and D. Claudio. Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation. In 8^{th} *VECPAR - International Meeting on High Performance Computing for Computational Science*, Toulouse, France, 2008. To appear.

[15] M. Kolberg, L. F. Fernandes, and D. Claudio. Dense Linear System: A Parallel Self-verified Solver . *International Journal of Parallel Programming*, 2007.

[16] U. W. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, Inc., Orlando, FL, USA, 1981.

[17] P. Linz. Accurate floating-point summation. *Commun. ACM*, 13(6):361–362, 1970.

[18] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, 1966.

[19] T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum and Dot Product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[20] J. Rohn. Systems of linear interval equations. *Linear Algebra and its Applications*, 126:39–78, 1989.

[21] S. Rump. Solving algebraic problems with high accuracy. In *Proc. of the symposium on A new approach to scientific computation*, pages 51–120, San Diego, CA, USA, 1983. Academic Press Professional, Inc.

[22] S. Rump. Fast and Parallel Interval Arithmetic. *Bit Numerical Mathematics*, 39(3):534–554, 1999.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[24] P. Stpiczynski and M. Paprzycki. Numerical Software for Solving Dense Linear Algebra Problems on High Performance Computers. In *Proceedings of the 4th International Conference APLIMAT 2005*, pages 207–218, Slovak University of Technology, Bratislava, May 2005.

[25] J. Zhang and C. Maple. Parallel solutions of large dense linear systems using mpi. *parelec*, 00:312, 2002.