Solving Dense Interval Linear Systems with Verified Computing on Multicore Architectures

Cleber Roberto Milani[1], Mariana Kolberg[1,2] and Luiz Gustavo Fernandes[1]

[1] GMAP - PPGCC - PUCRS
Av. Ipiranga, 6681 - Prédio 32
CEP 90619-900 Porto Alegre, Brazil
{cleber.milani, mariana.kolberg, luiz.fernandes}@pucrs.br
[2] Universidade Luterana do Brasil
Av. Farroupilha 8001 Prédio 14, sala 122
Canoas/RS, 92425-900 - Brasil
mariana.kolberg@ulbra.br

Abstract. Automatic result verification is an important tool to reduce the impact of floating-point errors in numerical computation and to guarantee the mathematical rigor of results. One fundamental problem in Verified Computing is to find an enclosure that surely contains the exact result of a linear system. Many works have been developed for optimizing Verified Computing algorithms using parallel programming techniques and message passing paradigm on clusters of computers. However, the High Performance Computing scenario changed considerably since the emergence of multicore architectures in the past few years. This paper presents an ongoing research project which has the purpose of developing a self-verified solver for dense interval linear systems optimized for parallel execution on these new architectures. The current version has obtained up to 85% of reduction at execution time and a speedup of 6.70 when solving a 15,000 x 15,000 interval linear system on an eight core computer.

1 Introduction and Motivation

In numerical algorithms, the correct implementation of a method does not guarantee that the computed result will be correct. Floating point arithmetic uses finite fractions to represent the real numbers, which are originally defined in Mathematics as infinite fractions. The difference between the true value and the approximation is the *roundoff error*. Floating point operations on computers are considered of *maximum accuracy* if the rounded result differs at most by one unit in the last place from the exact result. Automatic result verification is an important technique to reduce the impact of arithmetic errors in Numerical Computation [1, 2]. Verified Computing guarantees the mathematical rigor of the results of a computation by providing an interval result that surely contains the correct result. This interval result is called an *enclosure* [3].

Interval Arithmetic provides the mathematical basis for the Verified Computing. This arithmetic is based on sets of intervals, rather than sets of real

numbers. Typically, the interval evaluation of an arithmetic expression such as a polynomial costs about twice as much as the evaluation of the expression in simple floating point arithmetic. However, using interval function evaluation with directed roundings, the algorithm may provide a guarantee of the computed result which could not be achieved even by thousands of floating point evaluations [1–3].

Usually the input of a numerical method are point numbers. Engineering and scientific problems, however, are frequently modeled by numerical simulations on computers which are based on real measures that sometimes may be unprecise. To deal with uncertain data, the computer would need to be able to support interval input data as input data instead of point numbers and to do computations using Interval Arithmetic [3]. Dealing with uncertain data in the context of linear systems means that an interval linear systems must be solved. The solution of such a system is not trivial, since an infinite number of matrices contained in the interval should be solved. However, the computation of this solution set is a NP-complete problem [17]. Thus, the only possible way to find a solution is to compute a narrow interval that contains the solution set, that is, interval vectors that bound $\sum(A, b)$, and whose overestimation decreases as the widths of the entries in A and b decrease [3, 4].

There are many libraries that compute approximate solutions to point linear systems. Widely used for that purpose are the optimized software libraries LAPACK (Linear Algebra PACKage) [5] and SCALAPACK (SCAlable Linear Algebra PACKage) [6]. These libraries present a great performance and can manage to find an approximation of the correct solution, which is needed to compute the error bounds, faster than verified libraries [3, 18, 19]. However, even when using highly optimized libraries to solve part of the verified method, the task of solving an interval linear systems with verified results still presents a very high computational cost when dealing with large dense interval systems. Thus, the use of High Performance Computing (HPC) techniques appears as a useful tool to drop down the time needed to solve interval linear systems with Verified Computing [3, 4].

2 Parallel and Verified Computing

Many works have been developed for implement self-validated (SV) linear systems solvers using parallel computing on clusters of computers. These works try to combine message passing paradigm programming with linear algebra libraries like ScaLAPACK. Some of these works like [7, 8] operate with point input data while others like [3] propose solvers to treat uncertain input data represented by intervals matrices and vectors. A first study in the direction of multicore processors was also presented in [18]. However, this work presents an initial approach specifically for dualcores processors.

Today's computers are almost all built with multicore processors which cannot be considered as independently processors. Multicores processors share on-chip resources, which separate processors do not, and therefore cannot be con-

sidered as the new SMP [9]). On the Top500 List [10] released in November 2009, 426 systems are using quadcores processors, 59 systems use dualcores and only four systems still use singlecore processors. Six systems use IBMs advanced Sony PlayStation 3 processor with 9 cores and three systems are using the new 6 cores Shanghai AMD Opteron processor.

Historically, the standard parallelization approach of numerical linear algebra utilized by the LAPACK and ScaLAPACK libraries relied on parallel implementations of BLAS (Basic Linear Algebra Subprograms) [16]. But, although this approach solves numerous complexity problems, it also enforces a very rigid and inflexible software structure, where, at the level of linear Algebra, the algorithms are expressed in a serial way [9].

Recent research efforts are addressing this critical and highly disruptive situation. In [9], the authors present the Parallel Linear Algebra for Scalable Multicores Architectures (PLASMA) which should succeed LAPACK and ScaLAPACK. PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms are represented as Directed Acyclic Graphs (DAG) where nodes represent tasks and edges represent dependencies among them. This programming model enforces asynchronous and out of order scheduling of operations [9, 11]. In [12], it is presented the *SuperMatrix*, a runtime system that parallelizes matrix operations for SMP and multicores architectures. The supermatrix idea is based on a number of insights gained from the FLAME project [13]. Basically, it views matrices hierarchically as blocks that serve as units of data where operations over those blocks are treated as units of computation. Thus, implementation transparently enqueues the required operations (internally tracking dependencies) and then executes the operations using out-of-order execution techniques inspired by superscalar microarchitectures. However, these optimized software libraries do not implement verified computing methods. Additionally, support for uncertain input data and interval linear systems solvers are not provided.

On the other hand, verified computing tools (such as C-XSC [17]) can provide verified results but the execution times for solving the problem are much higher since their are not developed for multicore architectures. Additionally, the use of verified methods in C-XSC usually introduces a loss of performance in the application, since it uses special data structures and operations to implement dot scalar products [8]. This effect is even worse when dealing with large interval systems [3]. In this context, this paper proposes a self-verified solver for dense interval linear systems optimized for parallel execution on multicore processors.

3 Mathematical Background

Previous researches show the Midpoint-Radius approach of Interval Arithmetic as a good choice for implementations using floating point arithmetic [3, 18, 19]. The main point in using Midpoint-Radius arithmetic is that this representation allows to employ optimized algorithms and software libraries to implement operations. The use of such libraries have the striking advantages that they

are available for almost every computer hardware and that they are individually adapted and tuned for specific hardware and compiler configurations. A Midpoint-Radius interval is defined as follows [14]:

$$\langle a, \alpha \rangle := \{x \in \Re / \, |x - a| \leq \alpha\} \quad for \;\; a \in \Re, 0 \leq \alpha \in \Re \tag{1}$$

Interval operations always satisfy the fundamental property of isotonicity. That is, if X is contained in another interval X', and Y is contained in Y', then the combination of X and Y is contained in the interval computed by combining the bigger intervals X' and Y' [2]. Sometimes the standard definition of Midpoint-Radius arithmetic causes overestimation. However, it was proved by Rump [14] that the overestimation of Midpoint-Radius Arithmetic is uniformly bounded by 1.5 for the basic arithmetic operations as well as for vector and matrix operations over $\Re$. In the case of an interval presenting a not too large radius, the factor is quantified to be near 1.

For interval vectors and interval matrices the relations $=$, $\overset{\circ}{\subset}$, and $\subseteq$ are defined component wise. The inner inclusion relation is defined by $[x] \overset{\circ}{\subset} [y] \Leftrightarrow [x]_i \overset{\circ}{\subset} [y]_i$, i $= 1, \ldots, n$ for $[x], [y] \in I\Re^n$. On the other hand, the proper subset relation is defined by $[x] \subset [y] \Leftrightarrow ([x] \subseteq [y] \, and \, [x] \neq [y])$. The *midpoint* and the *diameter* of an interval vector or matrix are also defined component wise. For example, $m([x]) = (m([x]_i))$, and $d([A]) = \left(d\left([a]_{ij}\right)\right)$, for $[x] \in I\Re^n, [A] \in I\Re^{nxn}$.

Many algorithms for numerical verification are based on the application of well known fixed point theorems to interval sets. As an example, the Brouwer's Fixed Point Theorem is used to guarantee the convergence. Let $X = [x] \in I\Re^n$ be a machine interval vector. As a box in n-dimensional space, $[x]$ satisfies the conditions of Brouwer's Fixed Point Theorem. Supposing its possible to find a box with $f([x]) \subseteq [x]$, then $[x]$ is proved to be an enclosure of at least one fixed point x^* of f. The assertion remains valid replacing f by its floating point interval evaluation $f_\diamond$ because $f_\diamond([x]) \subseteq [x]$ implies $f([x]) \subseteq [x]$ since $f_\diamond([x])$ is a superset of $f([x])$ [2, 4].

In order to achieve validated enclosures, the algorithm must enclose all sources of error that can be generated during the computation. The basic approach of many SV-methods is computation of an approximate solution, local linearization and estimation of linearization and numerical errors by means of suitable theorems the assumptions of which are verified on the computer [21]. A simple mechanism for implementing these idea follows the principle of *iterative refinement*. However, it is important to mention that an interval algorithm differs significantly from the corresponding point algorithm.

The method we chose is based on the *Residual Iteration Scheme* of the *Newton-like Method*. The main reason for this choice is that besides being a well-established SV-method, it also allows use of optimized libraries because can be implemented with Midpoint-Radius arithmetic. The description of the method, fully given by [2], is summarized on the following. Let $Ax = b$ be a real system of equations, finding a solution of the system $Ax = b$ is equivalent to

finding a zero of $f(x) = Ax - b$. Hence, Newton's method gives the following fixed point iteration scheme, where $x^{(0)}$ is some arbitrary starting value [2]:

$$x^{(k+1)} = x^{(k)} - A^{-1}\left(Ax^{(k)} - b\right), \quad k = 0, 1, \ldots \tag{2}$$

In general, the inverse of A is not exactly known. Thus, instead of A^{-1}, an approximate inverse $R \approx A^{-1}$ of A is used. Replacing the real iterates $x^{(k)}$ by interval vectors $[x]^{(k)} \in I\Re^n$, if there exists an index k with $[x]^{(k+1)} \subset [x]^{(k)}$, then, by Brouwer's Fixed point Theorem, the equation has at least one fixed point $x \in [x]^{(k)}$. Supposing R is regular, then this fixed point is also a solution of $Ax = b$. However, considering the diameter of $[x]^{(k+1)}$ the following is obtained: $d\left([x]^{(k+1)}\right) = d\left([x]^{(k)}\right) + d\left(R\left(A[x]^{(k+1)} - b\right)\right) \geq d\left([x]^{(k)}\right)$. Thus, in general, the subset relation will not be satisfied. For this reason, the right-hand side is modified to the following equation, where I denotes the n x n identity matrix:

$$x^{(k+1)} = Rb + (I - RA)\, x^{(k)}, \quad k = 0, 1, \ldots, \tag{3}$$

It was proved that if there exists and index k with $[x]^{(k+1)} \overset{\circ}{\subset} [x]^{(k)}$, then the matrices R and A are regular, and there is a unique solution x of the system $Ax = b$ with $x \in [x]^{(k+1)}$. This result remains valid for any matrix R. However, it is an empirical fact that the better R approximates the inverse of A, the faster the inclusion relation will be satisfied. Additionally, it is a well-known numerical principle that an approximate solution $\tilde{x}$ of $Ax = b$ may be improved by solving the system $Ay = d$, where $d = b - A\tilde{x}$ is the residual of $A\tilde{x}$. Since $y = A^{-1}(b - A\tilde{x}) = x - \tilde{x}$, the exact solution of $Ax = b$ is given by $x = \tilde{x} + y$. Therefore, the Residual Iteration Scheme is presented on Equation 4.

$$y^{(k+1)} = \underbrace{R(b - A\tilde{x})}_{=:z} + \underbrace{(I - RA\tilde{x})}_{=:C} y^{(k)}, \quad k = 0, 1, \ldots \tag{4}$$

The residual equation $Ay = d$ has a unique solution $y \in [y]^{(k+1)} \overset{\circ}{\subset} [y]^{(k)}$ for the corresponding interval iteration scheme. Moreover, since $y = x - \tilde{x} \in [y]^{(k+1)}$, a verified solution of the unique solution of $Ax = b$ is given by $\tilde{x} + [y]^{(k+1)}$. These results remain valid if replace the exact expressions for z and C in (4) by interval extensions. However, to avoid overestimation effects, it is highly recommended to evaluate $b - A\tilde{x}$ and $I - RA$ without any intermediate rounding [2].

4 Proposed Approach

The Residual Iteration Scheme adaptation to solve interval linear systems using Verified Computing led to Algorithm 1, proposed on [17]. Its result is a high accuracy interval vector that surely contains the correct result. Verification process is composed by steps 5 to 15. These steps use the Midpoint-Radius arithmetic with direct roundings [2, 3].

Algorithm 1 Enclosure of a square interval linear system

1: $R \approx mid\,([A])^{-1}$ {Compute an approximate inverse using LU-Decomposition algorithm}
2: $\tilde{x} \approx R.mid\,([b])$ {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: $\quad \sum([A]\,, [b]) \subseteq \tilde{x} + [w]$ {The solution set $(\sum)$ is contained in the solution found by the method}
13: **else**
14: $\quad$ No Verification
15: **end if**

4.1 Initial Implementation

An initial version of Algorithm 1 using Midpoint-Radius arithmetic was implemented and used to obtain the computational costs of each step. This implementation was developed in C++ using the Intel MKL 10.2.1.017 [15] library for optimized LAPACK and BLAS routines for Intel processors. In order to achieve better performance, the approximate inverse R and approximate solution x are calculated using only traditional floating point operations using only the midpoint matrix. Later, for computation of the residuum, interval arithmetic is applied with original interval matrix $[A]$ and interval vector $[\boldsymbol{b}]$ to ensure the accuracy of the result [3].

The Step 1 (approximate inverse calculation using LU-Decomposition) uses the following LAPACK routines: *dgetrf*, *dlange*, *dgecon* and *dgetri*. Step 2 (approximation of the solution) is implemented by BLAS *dgemv* routine. Steps 3 and 4 compute respectively the enclosure for the residuum and enclosure for the iteration matrix. Since $[A]$ and $[\boldsymbol{b}]$ as well as $[C]$ and $[\boldsymbol{z}]$ are interval matrices and vectors, the computation of enclosures must employ interval algorithms as defined on [14]. Let $A = \langle \tilde{a}, \alpha \rangle \in I^+F$ and $B = \left\langle \tilde{b}, \beta \right\rangle \in I^+F$ be two Midpoint-Radius intervals, the operations of addition and subtraction $C := A \circ B \in I^+F$, with $\circ \in \{+, -\}$ and $C = \langle \tilde{c}, \gamma \rangle$ are implemented in IEEE 754 Standard for Binary Floating point Arithmetic [22] by Algorithm 2. Similarly, the multiplication is defined by Algorithm 3. The symbols $\square$, ∇ and Δ indicate respectively the directed roundings for nearest, downward and upward.

As previously mentioned, the major advantage of Midpoint-Radius Arithmetic is to allow calculation with pure floating point operations without making any changes in the rounding mode on interim operations. Therefore, although $[C]$ and $[\boldsymbol{z}]$ are intervals, they are calculated with *dgemv* and *dgemm* BLAS

Algorithm 2 IEEE 754 Midpoint-Radius Interval Addition and Subtraction.

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

Algorithm 3 IEEE 754 Midpoint-Radius Interval Multiplication.

1: $\tilde{c} = \square\left(\tilde{a}.\tilde{b}\right)$
2: $\tilde{\gamma} = \Delta\left(\eta + \epsilon' \left|\tilde{c}\right| + \left(\left|\tilde{a}\right| + \tilde{\alpha}\right)\tilde{\beta} + \tilde{\alpha}\tilde{\beta}\right)$

routines with directed roundings. The rounding mode is manipulated by *fesetround* C++ function from *fenv.h* header which supports four rounding modes: *FE_UPWARD, FE_DOWNWARD, FE_TONEAREST*, and *FE_TOWARDZERO*.

An error will be generated in the midpoint evaluation. This error should be compensated using the relative error unit. According to [14], denote the relative rounding error unit by ϵ, set $\epsilon' = \frac{1}{2}\epsilon$, and denote the smallest representable (unnormalized) positive floating point number by η. In IEEE 754 double precision $epsilon = 2^{-52}$ and $\eta = 2^{-1074}$. Therefore, the evaluation of C midpoint ($\tilde{c}$) and radius ($\tilde{\gamma}$) is given by Algorithm 4.

Algorithm 4 IEEE 754 Matrix-matrix Midpoint-Radius Interval Multiplication.

1: $\tilde{c}_1 = \nabla\left(R.mid\left(A\right)\right)$
2: $\tilde{c}_2 = \Delta\left(R.mid\left(A\right)\right)$
3: $\tilde{c} = \Delta\left(\tilde{c}_1 + 0.5\left(\tilde{c}_2 - \tilde{c}_1\right)\right)$
4: $\tilde{\gamma} = \Delta\left(\tilde{c} - \tilde{c}_1\right) + \left|R\right|.rad\left(A\right)$

At last, steps from 5 to 15 implement the iteration to obtain the enclosure. Again, Midpoint-Radius Arithmetic and direct roundings are employed. Step 8 ($[C]$ and $[y]$ multiplication) uses BLAS *dgemv* with directed roundings. The while loop verifies if the new result are contained in the interior of the previous result. If it is true, the while loop is finished, 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 [2].

4.2 Initial Approach Evaluation

Two kinds of evaluations were considered around the initial implementation: accuracy and performance. Aiming at verifying the accuracy, we used an ill-conditioned matrix generated by the well known Boothroyd/Dekker formula (Equation 5) with dimension 10, which has a condition number $1.09.10^{+15}$. The radius for both matrix A and vector b were defined as $0.1.10^{-10}$.

$$A_{ij} = \binom{n+i-1}{i-1} \text{ x } \binom{n-1}{n-j} \text{ x } \frac{n}{i+j-1}, b_i = i, \forall (i,j) = 1, 2, \ldots, N \quad (5)$$

The results found by the solver are presented in Table 1. It is important to highlight that despite the implemented solver uses Midpoint-Radius Arithmetic to do the computation, the results in Table 1 were converted to Infimum-Supremum notation to facilitate the visualization. *Exact Result* column indicates the known exact point result, *Infimum* and *Supremum* columns contain interval bounds for enclosure.

Table 1. Results found by implemented solver for a 10x10 Boothroyd/Dekker formula interval linear system

	Exact Result	Infimum	Supremum
0	0.0	-0.0000119	0.0000108
1	1.0	0.9998992	1.0001113
2	-2.0	-2.0005827	-1.9994736
3	3.0	2.9979758	3.0022444
4	-4.0	-4.0070747	-3.9936270
5	5.0	4.9826141	5.0193190
6	-6.0	-6.0472924	-5.9574730
7	7.0	6.9045776	7.1061843
8	-8.0	-8.2221804	-7.8004473
9	9.0	8.6064275	9.4384110

As expected, the exact result is contained in the interior of the solution set our solver computed. The interval diameter varies between 2.27×10^{-5} and 8.319835×10^{-1}. This was expected, since the Boothroyd/Dekker formula creates a very ill-conditioned system. Experiments of well-conditioned systems randomly generated with values between 0 and 1 were also performed. In these cases, the diameter was between 0 and 1×10^{-7}. The average condition number of these systems was around 6.12×10^{1}.

Performance experiments were carried out over a Intel Core 2 Duo T6400 2.00 GHz processor with 2MB L2 and 3GB of DDR2 667MHz RAM operating in dual channel. The operating system is Linux Ubuntu 9.04 (32 bits version, kernel 2.6.28-13-generic). The compiler used was gcc v. 4.3.3 with the MKL library v.10.2.1.017. The input for these experiments were linear systems randomly generated with values between 0 and 1 for A and b and a radius of 0.1×10^{-10} on both cases. The execution times of each step of the algorithm was computed. For simplicity reasons, steps from 6 to 15 were joined into 1 step. Table 2 presents the average execution times for each step for solving a system with dimension $n = 5,000$.

Table 2 shows that the computation of the approximate inverse R and the computation of the interval matrix C (steps 1 and 4 respectively) are the two

Table 2. Average exec. times (sec) for a randomly generated system with $n = 5,000$.

Task Description	Execution times
Computation of approximate inverse R (Step 1)	144.652161
Computation of approximate solution x (Step 2)	0.535467
Computation of enclosure for the residuum z of x (Step 3)	1.949728
Compute enclosure for the iteration matrix C (Step 4)	109.452704
Machine interval vector initialization (Step 5)	0.000117
Iterative refinement and inner inclusion verification (Steps 6 to 15)	4.635784
Total execution time including E/S operations	262.710470

most computational intensive operations in this algorithm. Step 1 takes more then 55% (144.65 seconds) of the total time while Step 4 takes 42% (109.45 seconds). These two steps correspond to 97% of total processing time, and therefore, they must be carefully parallelized aiming at a better performance.

4.3 Optimized Parallel Approach

As presented in the previous subsection, steps 1 and 4 are the most computational intensive operations in the algorithm. Thus, the proposed parallelization focused on these two steps as follows.

Optimization of the Approximate Inverse Calculation: since the Newton Like Iteration requires only an approximation of the inverse matrix of A and once our approach employs Midpoint-Radius Interval Arithmetic, R can be computed using highly optimized software libraries. In [3], the *pdgetri* routine of ScaLAPACK was employed for R calculation. Our initial approach was implemented using analogous LAPACK routine *dgetri*. However, although MKL implementation of LAPACK is highly optimized for Intel processors, LAPACK algebra algorithms are not efficient on multicore. Hence, as expected LAPACK routines had no performance gain when increasing the number of cores.

Therefore, our strategy for Step 1 is to explore fine granularity parallelism as well as asynchronous and out of order scheduling of operations by employing the PLASMA library. However, the most actual version of PLASMA does not provide yet a matrix inversion routine. In fact, when dealing with multicore processors there are no libraries available that can be directly employed for optimized matrix inversion. Thus, the idea is exploit PLASMA *dgesv* routine.

The *dgesv* was developed to compute the solution of a system of linear equations. However, it is possible to operate the right hand side b of *dgesv* as a matrix and it is a well-known mathematical property that multiplying a matrix by its inverse results the identity matrix. Considering that, we employed PLASMA *dgesv* routine passing to A and b parameters, respectively, the A and its identity matrices. This way, the result computed by *dgesv* is the approximate inverse R.

It is important to mention that while packages like LAPACK and ScaLAPACK exploit parallelism within multithreading BLAS, PLASMA uses BLAS

only for high performance implementations of single core operations (often referred to as kernels). PLASMA exploits parallelism at the algorithmic level above the level of BLAS. For that reason, PLASMA must be linked with a sequential BLAS library or a multithreaded BLAS library with multithreading disabled. PLASMA must not be used in conjunction with a multithreaded BLAS, as this is likely to create more threads than actual cores, which annihilates PLASMA′s performance [23]. Since our approach takes advantage of multithreaded BLAS in operations executed by other steps (like matrices multiplication) we used multithreaded MKL. To avoid affecting PLASMA performance, the function *mkl_set_num_threads* is used to dynamically control the number of threads.

Optimization of the Iteration Matrix Computation: Concerning Step 4, the computation of the enclosure for the iteration matrix $[C]$, the adopted strategy is to use half of the available processors to compute the interval upper bound and the other half to compute the interval lower bound. A similar strategy was successfully employed in [18] where threads were used to compute the interval bounds on a dual core processor. In that case, however, synchronization is simpler and it was not necessary to deal with load balancing.

The idea is to utilize different threads to execute the operations in each rounding mode. This strategy avoids the constant rounding mode changing which is a time expensive operation. Additionally, since the cache is shared between cores, computing distinct bounds over the same data in parallel optimizes data locality. Threads are created and managed using the standard POSIX threads library [20] and inter-thread synchronization is done using shared memory and POSIX semaphores primitives.

Initially, a routine verifies the number of available cores and distributes the number of each bound threads among them. Cores identified by odd numbers are assigned to upper bound computation and the even numbers to lower bound. If the number of total cores available is odd, then upper bound will be computed with one more thread than lower bound. The *cpu_set_t* variables of *sched.h* header are used to create the core pools. Threads are then statically attributed to cores by calling *sched_setaffinity* function. It is important to highlight, that defining the processor affinity instructs the operating system kernel scheduler to not change the processor used by one particular thread.

After threads are assigned to processors they start setting their rounding modes and get blocked by semaphores until the main flow releases them all at once. On the sequence, each thread calls the *dgemm* BLAS routine for the matrix-matrix multiplication. The main flow blocks itself with a semaphore until the computation of upper and lower bounds ends. Once the computation of $[C]$ is completed, threads send signals to unblock main flow semaphore, which then follows to next step.

4.4 Optimized Approach Evaluation

In order to verify the benefits of employed optimizations, two kind of experiments were performed. The first concerns the correctness of the result. The

second experiment was done to evaluate the speedup improvement brought by the proposed method. The evaluations were executed in a 2 processors quad-core Intel Xeon E5520 2.27 GHz with 128 KB L1, 1MB L2, 8MB L3 shared and 16 GB of DDR3 1066 MHz RAM. The operating system is Linux Ubuntu 9.04 (kernel 2.6.28-11-server). The compiler used was gcc v. 4.3.3 together with the libraries MKL 10.2.2.025 and PLASMA 2.1.0.

Once modifications were done in the algorithm, we conducted some experiments with the same well-conditioned and ill-conditioned matrices solved by our initial approach to confirm that there were no accuracy loss on the result. The tests generated by the Boothroyd/Dekker formula presented almost the same accuracy on both versions (initial and optimized). Actually, for this example, the result of the initial version is minimally better than the result of the optimized version. As required by the algorithm, both results contain the exact result. For well-conditioned matrices, both implementations give exactly the same results.

We carried out performance experiments for matrices dimensions from 1,000 to 15,000. Table 3 presents the execution times in seconds for each algorithm step when solving a random 15,000 x 15,000 interval linear system varying the number of cores. Column *Imp.* refers to the approach where *In.* is the initial implementation and *Op.* is the optimized version. *Cores* column indicates the number of cores employed in that execution and columns *Step 1..15* refer to the algorithms steps in the same way as in Table 2. As we had a small standard deviation, we just run the solver 10 times for each situation. The upper and lower bounds, i.e., highest and lowest execution times, were removed and the final times were obtained by calculating the arithmetic mean of remaining times.

Table 3. Execution times in seconds to solve a 15,000 x 15,000 interval linear system.

Imp.	Cores	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6–15	Total
In.	1	1,905.4969	8.3916	23.6316	2,204.0620	0.0002	73.0728	4,488.7467
Op.	1	1,147.8716	5.6718	19.1374	2,218.5130	0.0002	70.4101	3,461.6043
Op.	2	575.8929	5.7024	19.3800	1,169.3131	0.0002	64.8068	1,835.0956
Op.	3	387.9773	5.6246	18.1846	1,058.4973	0.0002	68.9738	1,539.2580
Op.	4	292.6839	5.6923	19.4538	646.0218	0.0002	32.4533	996.3056
Op.	5	249.2505	5.5699	18.1954	626.2732	0.0002	34.9536	934.2430
Op.	6	209.5168	5.7400	19.3016	493.2460	0.0002	33.6801	761.4850
Op.	7	182.3047	5.5987	17.8858	474.7029	0.0002	34.1734	714.6659
Op.	8	160.8892	5.6867	18.9293	451.5206	0.0002	32.9934	670.0196

Figure 1 shows the speedups obtained from Table 3. Line *Sp T.T.Seq.* is the speedup of total execution time comparing optimized implementation running in n cores to the initial approach in 1 core (i.e., $\frac{T\_Op.(n)}{T\_In.(1)}$). *Sp T.T.Par.* concerns to optimized total time in n cores compared to optimized algorithm executing in 1 core (i.e., $\frac{T\_Op.(n)}{T\_Op.(1)}$). *Sp Inv. Seq.* and *Sp Inv. Par.* illustrate speedups obtained in an analogous manner considering only the Step 1 execution time. *Sp S4. Par.* presents the speedup for Step 4 of algorithm.

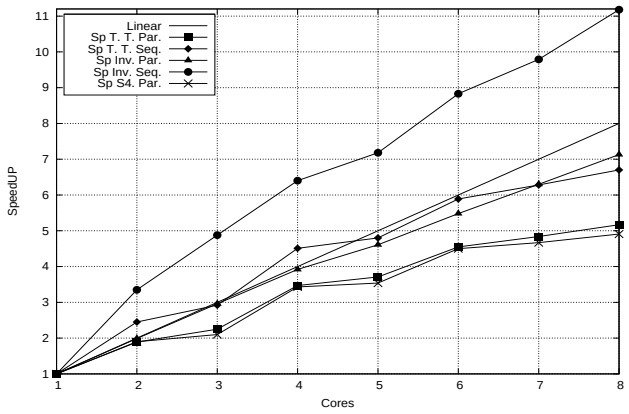

Fig. 1. *Speedups* obtained solving an interval linear system of size 15,000 x 15,000.

In Table 3 and Figure 1 it is possible to see a significant reduction in the execution time. *Sp T.T.Seq.* initially presented super linear speed up and slowly decreased until 6.70 for 8 cores, which is a expected result due to scalability issues like as the influence of sequential portions of code. *Sp T.T.Par.* also presented high speedups and a similar behavior. The main reason for this difference is the Step 1 of the algorithm. The optimized implementation running in 1 core spent only 60% of the time spent by the initial approach. This is because PLASMA optimizations not boil down only to the parallelism but also to new algorithmic approaches for data management and tasks scheduling which are more suitable for multicore architectures.

Sp Inv. Seq. and *Sp Inv. Par.* can be explained by these same reasons. It is important to note that Step 1 computed with LAPACK *dgetri* routine on 8 cores spent 1,864.168661 seconds, which means a speedup of 1.02 and confirms that this is not suitable for multicore.

Sp S4. Par. presented good speedups too. We suppose that this is due to cache effects. In the sequential version, all matrix elements must be loaded in the cache to compute $[C]$ with rounding-up, and after that, again, to compute it with rounding-down. If the entire matrix does not fit in the cache, there will be many cache misses for each rounding mode. Since more threads use the same data at the same time, the multithreaded version allows a more effective utilization of the available cache memory, resulting in a better speedup as expected.

At last, verification steps (6–15) although not explicit parallelized showed performance gains too. The reason is that the use of *dgemm* routine benefits from multithreaded MKL.

5 Considerations and Future work

This paper presented the current version of a self-verified solver for dense interval linear systems optimized for parallel execution on multicore architectures.

The implementation delivered enclosures of the correct solutions for interval input data with considerable accuracy. The computational costs of each of its intermediate steps were computed and the main time expensive oh them were optimized aiming at obtaining performance gain on multicore processors. The proposed solution led to a scalable implementation which has achieved up to 85% of reduction at execution time when solving a 15,000 x 15,000 interval linear system over an eight core computer.

Its important to mention that the presented solver was written for dense systems. However, sparse systems are also supported although they will be treated as a dense system. No special method or data storage is used concerning the sparsity of these systems. Many performance related issues still remain under investigation. There is a clear tradeoff between the overhead incurred by thread synchronization and the performance gain, which affects the solver scalability. Therefore, future directions includes the investigation of how to optimize the parallelized steps, the identification other parts of the algorithm to parallelize and the exploitation of new architectures as the hybrid computers that mix GPUs and multicore processing.

The ability of finding verified results for dense linear systems of equations increases the result accuracy. The possibility to perform this computation in multicore architectures reduces the computational time that verified computing need through the benefits of high performance computing. Therefore, the use of verified and high performance computing together appears as a suitable way to increase the reliability and performance of many applications, specially when those applications deal with uncertain data.

References

1. Hayes, B.: A Lucid Interval. American Scientist **v. 91** n.6 (2003) p. 484–488
2. Hammer, R., Ratz, D., Kulisch, U., Hocks, M.: C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems. Springer-Verlag New York, Inc. (1997)
3. Kolberg, M., Dorn, M., Bohlender, G., Fernandes, L. G.: Parallel Verified Linear System Solver for Uncertain Input Data. Proceedings of 20th SBAC-PAD - International Symposium on Computer Architecture and High Performance Computing (2008) 89–96
4. Kearfott, R.: Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin **v. 2** n.1 (1996) p. 95–112
5. Demmel, J.: LAPACK: A Portable Linear Algebra Library for Supercomputers. Proceedings of IEEE Control Systems Society Workshop on Computer-Aided Control System Design (1989) p. 1–7
6. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.: ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance. Computer Physics Communications **v. 97** n.1–2 (1996) p. 1–15
7. Kolberg, M., Baldo, L., Velho, P., Fernandes, L. G., Claudio, D.: Optimizing a Parallel Self-verified Method for Solving Linear Systems. In: 8th PARA - International Workshop on State-of-the-Art in Scientific and Parallel Computing, (2006), Umea - Sweden. PARA 2006 - Revised Selected Papers. Springer - Lecture Notes in Computer Science **v. 4699** (2007) p. 949–955

8. Kolberg, M., Fernandes, L. G., Claudio, D.: Dense Linear System: A Parallel Self-verified Solver. International Journal of Parallel Programming **36** n.4 (2008) p. 412–425
9. Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., Tomov, S.: The Impact of Multicore on Math Software. In: 8th PARA - International Workshop on State-of-the-Art in Scientific and Parallel Computing, (2006), Umea - Sweden. PARA 2006 - Revised Selected Papers. Springer - Lecture Notes in Computer Science **v. 4699** (2007) p. 1–10
10. TOP 500 Supercomputing Home Page. Available at http://www.top500.org/. Accessed on December, 11th (2009)
11. Agullo, E., Hadri, B., Ltaief, H., Dongarra, J.: Comparative Study of One-Sided Factorizations with Multiple Software Packages on Multi-Core Hardware. LAPACK Working Note 217, ICL, UTK. (2009)
12. Chan, E., Van Zee, F., Bientinesi, P., Quintana-Orti, E., Quintana-Orti, G., van de Geijn, R.: SuperMatrix: a Multithreaded Runtime Scheduling System for Algorithms-by-blocks. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008) p. 123–132
13. Gunnels, J., Gustavson, F., Henry, G., van de Geijn, R.: FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software **v. 27** n.4 (2001) p. 422–455
14. Rump, S. M. Fast and Parallel Interval Arithmetic. BIT Numerical Mathematics **v. 39** n.3 (1999) p. 534–554
15. Intel Math Kernel Library Home Page. Available at http://software.intel.com/en-us/intel-mkl/. Accessed on December, 11th (2009).
16. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software **v. 5** n.3 (1979) 308–323
17. Klatte, R., Kulisch, U., Lawo, C., Rauch, R., Wiethoff, A.: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag (1993)
18. Kolberg, M., Cordeiro, D., Bohlender, G., Fernandes, L. G. and Goldman, A.: A Multithreaded Verified Method for Solving Linear Systems in Dual-Core Processors. In: 9th PARA - International Workshop on State-of-the-Art in Scientific and Parallel Computing, (2008), Trondheim - Noruega. PARA 2008 - Revised Selected Papers. Springer - Lecture Notes in Computer Science (to appear).
19. Kolberg, M. L. ; Bohlender, G. ; Claudio, D. M. . Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computing. In: 8th VECPAR - International Meeting on High Performance Computing for Computational Science, (2008), Toulouse - France. High Performance Computing for Computational Science - VECPAR'08. Springer - Lecture Notes in Computer Science, **v. 5336** (2008) p. 13–26.
20. D. R. Butenhof. Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1997).
21. Rump, S. M. Self-validating Methods. Linear Algebra and Its Applications. **v. 324** n.1–3 (2001) p. 3–13
22. ANSI/IEEE. A Standard for Binary Floating-point Arithmetic, Std.754-1985. American National Standards Institute / Institute of Electrical and Eletronics Engineers. USA, (1985).
23. PLASMA README. Available at http://icl.cs.utk.edu/projectsfiles/plasma/html. Accessed on April, 8th (2010)