Intervals on Self-verified Linear Systems Solvers for Multicore Computers

Luiz Gustavo Fernandes[1], Mariana Kolberg[2], Cleber Roberto Milani[3]

[1,3] PUCRS, Grupo de Modelagem de Aplicações Paralelas (GMAP)
Av. Ipiranga, 6681 - PPGCC - Prédio 32 - Sala 619 - 90619-900 - Porto Alegre, RS

[2] ULBRA - Universidade Luterana do Brasil
Av. Farroupilha, 8001 · Prédio 14, sala 122 - 92425-900 - Canoas, RS

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.

Keywords. Verified Computing, Interval Linear Systems, Multicore Architectures.

Resumo. A verificação automática de resultados é uma ferramenta poderosa para reduzir o impacto de erros oriundos de operações com ponto-flutuante em computação numérica pois garante o rigor matemático dos resultados. Um dos problemas fundamentais em Computação Verificada é descobrir um intervalo que asseguradamente contém o resultado exato de um dado sistema linear. Nos últimos anos, muitos trabalhos foram desenvolvidos no sentido de otimizar algoritmos de Computação Verificada usando técnicas de Programação Paralela baseadas no paradigma de troca de mensagens para *clusters* de computadores. Em anos recentes, no entanto, o cenário da Computação de Alto Desempenho evoluiu consideravelmente levando ao surgimento das arquiteturas *multicore*. Este trabalho introduz os resultados iniciais de um projeto de pesquisa em andamento cujo principal objetivo é desenvolver um *solver* auto-verificado para sistemas lineares densos otimizado para essas novas arquiteturas.

Palavras-chave. Computação Verificada, Sistemas Lineares Intervalares, Arquiteturas *Multicore*.

[1]luiz.fernandes@pucrs.br
[2]marianakolberg@gmail.com
[3]cleber.milani@pucrs.br

1. Introduction

Bounding the solution set of systems of linear equations is a major problem in Computer Science. However, traditional methods offer no guarantee of correct solutions and not even of the existence of a solution. The main cause of this is that 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 round off error. Hence, automatic result verification is an important additional tool to guarantee the mathematical rigor of the results [15].

The basis for Verified Computing is provided by Interval Arithmetic, which is defined on sets of intervals, rather than sets of real numbers. However, Verified Computing increases the computational cost and, in some cases, the required resolution time becomes unacceptable. This occurs because, besides the additional verification steps, the interval evaluation of an arithmetic expression costs about twice as much as the evaluation of the expression in simple floating point arithmetic. However, by employing interval function evaluation with directed rounding the algorithm may provide a guarantee of the computed result which cannot be achieved even with millions of floating point evaluations.

Additionally, Interval Arithmetic allows computers to deal with uncertain data. In the context of linear systems, it implies that an interval linear system must be solved. The solution of such a system is not trivial, since the infinite number of matrices contained in the interval should be solved. However, the computation of this solution set is an NP-complete problem [15]. Thus, the only possible way to find a solution is to compute a narrow interval that contains this set [3, 4].

There are two main approaches for Interval Arithmetic: the Infimum-Supremum and Midpoint-Radius representations. These representations are equivalent for the theoretical operations. However, it changes when implemented in floating point systems where each of these representations has advantages and disadvantages. On one hand, the standard definition of Midpoint-Radius Arithmetic causes overestimation for multiplication and division. On the other hand, the operations for Infimum-Supremum approach, 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.

Previous works [3] show the Midpoint-Radius representation as a good choice for implementations using floating-point arithmetic. The main point in using Midpoint-Radius arithmetic is that it is possible to use optimized algorithms and software libraries to implement operations. The latter bear 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. Furthermore, the overestimation is uniformly limited to 1.5, as proved by Rump [13]. Thus, these libraries are used to find the approximation needed to compute the narrow interval that contains the solution [3, 16, 17].

Besides the optimized libraries, the use of high performance computing (HPC) techniques appears as a solution for computational cost problem. Several works

have focused on optimizing Verified Computing performance for computer clusters. However, many changes have been occurring in high performance computing. Given the number of cores on multicore chips expected to reach tens in a few years, efficient implementations of numerical solutions using shared memory programming models is of urgent interest. In this context, we developed a self-verified solver for dense interval linear systems optimized for parallel execution on multicore processors.

Two techniques were employed on the development of the solver. The first one was to optimize the matrix inversion step of algorithm by employing PLASMA [20] routines. The second was to divide the computation of the interval iteration matrix bounds by using different threads to execute the operations in each rounding mode. The adopted strategies have resulted in a scalable solver that obtained up to 85% of reduction at execution time and a speedup of 6.70 with efficiency nearly to 84% when solving a 15,000 $\times$ 15,000 interval linear system on an eight core computer. Additional discussion and details are presented in [21].

2. Initial Solution

The Residual Iteration Scheme [2] adaptation to solve interval linear systems using Verified Computing led to Algorithm 1, proposed on [15]. 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 rounding [3].

Algorithm 1 Enclosure of a square interval linear system

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

An initial version of Algorithm 1 using Midpoint-Radius arithmetic was implemented and used to obtain the computational costs of each step. This implementa-

tion was developed in C++ using the Intel MKL 10.2.1.017 [14] 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 $\left[\vec{b}\right]$ to ensure the accuracy of the result [3]. Further details as well as an accuracy and a performance evaluation of this solution can be found in [21].

Experiments were carried out in order to measure the computational cost of each step of Algorithm 1. They were executed considering 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. For simplicity reasons, steps from 6 to 15 were joined into 1 step. Table 1 presents the amount of time that was consumed for each step solving a linear system with $n = 5000$.

Table 1: Pencentual times for a randomly generated system with $n = 5000$.

Task	Percentage
Computation of approximate inverse R (Step 1)	55.06%
Computation of approximate solution x (Step 2)	0.41%
Computation of enclosure for the residuum z of x (Step 3)	0.94%
Compute enclosure for the iteration matrix C (Step 4)	41.66%
Machine interval vector initialization (Step 5)	0.16%
Iterative refinement and inner inclusion verification (Steps 6 to 15)	1.77%

Table 1 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 most computational intensive operations in this algorithm. Step 1 takes more then 55.06% of the total time while Step 4 takes 41.66%. These two steps correspond to 97% of total processing time, and therefore, they must be carefully parallelized aiming at a better performance.

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.

3.1. 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. Thus, 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 [20]. 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.

3.2. 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 [16] 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 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. Experiments and 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 and 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.

Table 2 presents the execution times in seconds for each algorithm step when solving a random 15,000 $\times$ 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 *Step 1..15* refers to the algorithms steps. 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.

Figure 1 shows the speedups obtained from Table 2. Line *Sp T.T.Seq.* is the speedup of total execution time comparing optimized implementation running in

Table 2: Execution times (sec) to solve a 15,000 × 15,000 interval linear system.

Impl.	Cores	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6–15	Total
In.	1	1,90550	8.39	23.63	2,204.06	0.02	73.07	4,488.75
Op.	1	1,147.87	5.67	19.14	2,218.51	0.02	70.41	3,461.60
Op.	2	575.89	5.70	19.38	1,169.31	0.02	64.81	1,835.10
Op.	3	387.98	5.62	18.18	1,058.50	0.02	68.97	1,539.26
Op.	4	292.68	5.69	19.45	646.02	0.02	32.45	996.31
Op.	5	249.25	5.57	18.19	626.27	0.02	34.95	934.24
Op.	6	209.52	5.74	19.30	493.25	0.02	33.68	761.49
Op.	7	182.30	5.60	17.89	474.70	0.02	34.17	714.67
Op.	8	160.89	5.69	18.93	451.52	0.02	32.99	670.02

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.

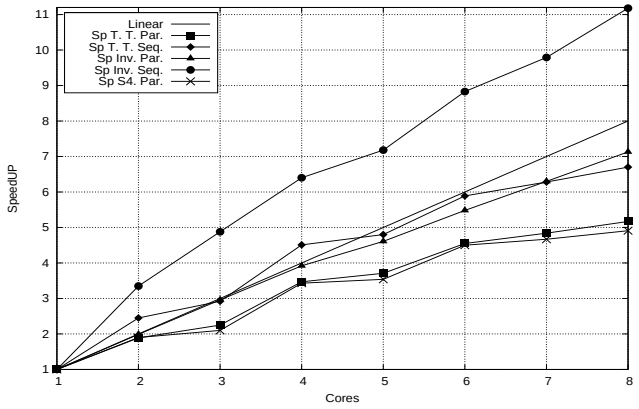

Figure 1: *Speedups* obtained solving an interval linear system of size 15,000 × 15,000.

In Table 2 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 $\times$ 15,000 interval linear system over an eight core computer.

It is 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.

Acknowledgement: Mariana Kolberg thanks to FAPERGS for the financial support through the research project 0905026. The authors would like to thanks to all the members of the research groups working on the joint Brazilian-German cooperation project (PROBRAL - 298/08), funded by CAPES and DAAD. More information about this joint project can be found at the website http://www.math.uni-wuppertal.de/org/WRST/projekte/brasilien/.

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. Berlin Heidelberg : Springer - Lecture Notes in Computer Science **v. 4699** (2006) 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. Applied Parallel Computing. State of the Art in Scientific Computing. Springer Lecture Notes in Computer Science **v. 4699** (2008) p. 1–10

[10] TOP 500 Supercomputing Home Page. Available at http://www.top500.org/. Accessed on August, 8th (2010).

[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] Rump, S. M. Fast and Parallel Interval Arithmetic. BIT Numerical Mathematics **v. 39** n.3 (1999) p. 534–554

[14] Intel Math Kernel Library Home Page. Available at http://software.intel.com/en-us/intel-mkl/. Accessed on August, 8th (2010).

[15] Klatte, R., Kulisch, U., Lawo, C., Rauch, R., Wiethoff, A.: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag (1993)

[16] 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. Berlin Heidelberg: Springer - Lecture Notes in Computer Science (to appear).

[17] 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. Berlin Heidelberg: Springer - Lecture Notes in Computer Science, **v. 5336** (2008) p. 13–26.

[18] Rump, S. M. Self-validating methods. Linear Algebra and Its Applications. **v. 324** n.1–3 (2001) p. 3–13

[19] ANSI/IEEE. A Standard for Binary Floating-point Arithmetic, Std.754-1985. American National Standards Institute / Institute of Electrical and Eletronics Engineers. USA, (1985).

[20] PLASMA README. Available at http://icl.cs.utk.edu/projectsfiles/plasma/. Accessed on August, 8th (2010).

[21] Milani, C. R., Kolberg, M. L., Fernandes, L. G., Solving Dense Interval Linear Systems with Verified Computing on Multicore Architectures. In: 9th VECPAR - International Meeting on High Performance Computing for Computational Science), to appear, Berkeley, USA, 2010.