Efficient Parallel Solvers for Large Dense Systems of Linear Interval Equations*

Mariana Kolberg
Área de Tecnologia e Computaccão, Universidade Luterana do Brasil, Av. Farroupilha 8001 Prédio 14, sala 122, Canoas, Brasil
mariana.kolberg@ulbra.br

Walter Krämer and Michael Zimmer
Wissenschaftliches Rechnen/Softwaretechnologie, Bergische Universität Wuppertal, Gaussstr. 20, 42097 Wuppertal, Germany
{walter.kraemer,michael.zimmer}@math.uni-wuppertal.de

Abstract

Verified solvers for dense linear (interval-)systems require a lot of resources, both in terms of computing power and memory usage. Computing a verified solution of large dense linear systems (dimension $n > 10000$) on a single machine quickly approaches the limits of today's hardware. Therefore, an efficient parallel verified solver for distributed memory systems is needed.

In this work we present such a solver, implemented in C++ and using the C-XSC library for scientific computing [10, 8]. The solver utilizes MPI [27] for communication between the nodes in the parallel environment and, where applicable, high performance ScaLAPACK [4] and BLAS [3] routines for fast computing times. High precision dot products [5, 18, 19, 21] are used to compute narrow enclosures of the solution of the system.

We present test results on several high performance distributed memory systems with different architectures, which show that our solver achieves good results, both in terms of numerical accuracy as well as computing time, and is highly portable. Furthermore, even very large systems ($n \geq 100000$) can be solved given a cluster with sufficient resources.

Keywords: self-verifying methods, large linear interval systems, DotK methods, parallelization, block cyclic distribution, C-XSC, interval computations
AMS subject classifications: 65H10, 15-04, 65G99, 65G10, 65-04, 68W15

*Submitted: January 20, 2009; Revised: February 10, 2010; Accepted: March 1, 2010.

1 Introduction

In this paper, we are interested in finding a verified solution of

$$Ax = b,\ A \in \mathbb{K}^{n\times n},\ x \in \mathbb{L}^n,\ b \in \mathbb{K}^n$$

where $\mathbb{K}$ can be either $\mathbb{R}$, $\mathbb{IR}$, $\mathbb{C}$ or $\mathbb{IC}$ and $\mathbb{L}$ is either $\mathbb{IR}$ or $\mathbb{IC}$, with $\mathbb{IR}$ denoting the set of all real intervals and $\mathbb{IC}$ denoting the set of all complex intervals. Verified solution means that x should be guaranteed to enclose the true solution of the linear system. We do not make any restrictions on the structure of A (symmetric positive definite, diagonally dominant,...) and A, x and b are supposed to be dense.

To compute such a solution, we use a widely known algorithm by Rump [25] based on the Krawczyk operator [20], which is the fastest known algorithm for general dense linear systems. The algorithm is described in Section 2. In a practical implementation, this algorithm is about 6 to 8 times slower than a non-verified floating point solver based on a LU-decomposition and requires several times more memory since not only the matrix A must be stored, but also an approximate inverse R of A and an enclosure $[C] = \diamond(I - RA)$ (here we highlight interval quantities by using brackets, $\diamond$ means that an interval enclosure of the following expression is to be computed).

Because of this, most serial solvers are limited to systems with only a few thousand unknowns, especially when solving interval systems. Therefore, if one wants to compute verified solutions of large dense systems of linear (interval-) equations, a parallel solver for distributed memory systems is needed.

A parallel solver for dense linear interval systems using C-XSC already exists [6, 11], but uses a master-slave approach that requires complete matrices to be stored by one process, thus the problem size is still limited by the memory of the master process instead of the memory of the whole cluster. This solver also uses high precision dot products using the long accumulator (see Section 3.2) for all computations, which leads to slow computing times. Recently, Intel added solvers for interval systems to its Math Kernel Library. However, in our tests these didn't perform very well and Intel is already considering to remove these solvers from the MKL [2].

Also, there are several serial software solutions available. Intlab [24] provides a fast and easy to use solver. However, due to memory constraints because of the matlab overhead, the maximum problem size is even more limited than with a serial C++ solver. C-XSC provides two older solvers for real [7] and interval systems [9]. However, these also use exact dot products throughout and thus are very slow. We recently developed [16] a serial solver using C-XSC which is a lot faster (about as fast as Intlab). It uses the same basic ideas as the parallel solver presented here and was the basis for our development of the new parallel solver.

This paper is organized as follows: First, in Section 2, we give a short summary of the mathematical background of computing a verified solution of a linear dense (interval-)system. In section 3 we give a short overview of some of the libraries utilized by our solver and of the algorithms used to compute exact or higher precision dot products. Then, in Section 4 we discuss the steps taken for the parallelization, while in Section 5 we give some test results on a few different cluster computers. Finally, Section 6 contains some final remarks and discusses further work that needs to be done.

2 Verified solution of linear (interval-)systems

The basic algorithm used in our solver is a well known algorithm by Rump, which is based on the Krawczyk-Operator [25, 20]. The Krawczyk-Operator is defined as

$$K(x) = R \diamond (b - A\tilde{x}) + \diamond(I - RA)x,$$

where R is an approximate inverse of A (or of mid(A), the midpoint matrix of A, if A is an interval matrix) computed by some floating point algorithm and $\tilde{x}$ is an approximate solution (normally computed by $\tilde{x} = Rb$). If in an iteration based on this operator a new iterate lies in the interior of the previous iterate, it can be shown [25, 20] that A is regular and the new iterate contains the unique solution of the linear system. Algorithm 1 shows Rump's algorithm based on this operator.

Input: Square matrix A and right hand side b
Output: An interval vector enclosing the solution of $Ax = b$
Compute approximate inverse R of A
Compute approximate solution $\widetilde{x} := Rb$
repeat
  $\widetilde{x} := \widetilde{x} + R(b - A\widetilde{x})$
until $\widetilde{x}$ *accurate enough or max. iterations reached*
$Z := R \diamond (b - A\widetilde{x})$
$C := \diamond(I - RA)$
$Y := Z$
repeat
  $Y_A := blow(Y, \epsilon)$
  $Y := Z + C \cdot Y_A$
until $Y \subseteq int(Y_A)$ *or max. iterations reached*
if $Y \subseteq int(Y_A)$ **then**
  Unique solution in $x \in \widetilde{x} + Y$
else
  Algorithm failed, A is singular or $cond(A)$ is too large

Algorithm 1: Rump's algorithm for the verified solution of dense linear systems

If A is an interval matrix and b is an interval vector, the midpoint matrix of A and the midpoint vector of b are used for the computation of R and $\tilde{x}$. The *blow* function in the above algorithm is a so called epsilon inflation, meaning that the interval is inflated a little in order to "catch" a nearby fixed point.

This algorithm will work up to a condition number of A of about 10^{15} (in this paper, we define the condition number of a matrix A as $cond(A) = ||A||_\infty ||A^{-1}||_\infty$). For higher condition numbers, the approximate inverse R of A will be too inaccurate, meaning that the spectral radius of $|I - RA|$ will be greater than 1 and an inclusion in the interior can not be obtained during the verification step.

For such badly conditioned systems, a second stage of the algorithm using an inverse of double length can be used. Here, an approximate inverse R_S of $S := RA$ is computed. Since

$$A^{-1} = (RA)^{-1}R$$

and since S will in general have better condition than A, $R_S R$ should be a better approximation of A^{-1}. $R_S R$ should be computed as the sum of two matrices $R_1 + R_2$

of double precision using higher precision dot products. Recently, an extension of this approach for matrices with extremely bad condition numbers has been published [23].

3 Tools

In this section we discuss the libraries used in the implementation of our solver and give a short overview of how dot products in higher precision are computed if needed.

3.1 Libraries

The solver itself is based on the C-XSC (eXtended Scientific Computing) library, a C++ class library for scientific computing developed at the Universities of Karlsruhe and Wuppertal. It provides basic datatypes for computations using interval arithmetic, as well as the most important standard mathematical functions. Corresponding matrix and vector classes are also available. These datatypes are used throughout the solver. C-XSC is capable of computing dot products in maximum precision using the long accumulator described in Section 3.2.

For the more costly parts of the solver in terms of computing time, optimized routines from the ScaLAPACK (Scalable LAPACK) [4] library are used. ScaLAPACK is a parallel version of LAPACK, providing highly optimized and efficient parallelizations of the routines from the serial LAPACK library. A parallel version of the BLAS [3] called PBLAS is also included. These routines are used for the computation of the approximate inverse R and of the interval matrix $[C] = \diamond(I - RA)$, as proposed in [12]. For the computation of $[C]$, manipulation of the rounding mode is used to achieve verified enclosures of the result, similar to the approach described in [24, 12]. More details on ScaLAPACK and its usage follow in Section 4.

3.2 Algorithms for high precision dot products

C-XSC itself uses a so called long accumulator [18, 19] to compute dot products. A long accumulator is essentially a fixed-point register of sufficient length which can store the results of a dot product or summation exactly. The result can then be rounded to the nearest floating point number, leading to a floating point result with maximum accuracy.

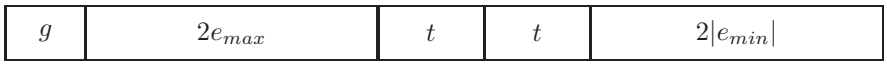

Figure 1: Long accumulator

The accumulator has to be large enough so that it can even store the result of a constant summation of the largest representable number without overflowing during the lifetime of a modern PC. Figure 1 shows the needed length of the accumulator. Here, t is the length of the mantissa, e_{min} and e_{max} are the smallest and largest exponent, respectively, and g is a certain number of guard digits to prevent overflow. A hardware implementation of this method would be even faster than normal floating point computations [17], but is unfortunately not available in today's hardware.

In C-XSC, the accumulator is implicitly used for all matrix and vector products. It can also be used directly through the datatype `dotprecision`, allowing to store the intermediate results of a longer calculation exactly without rounding.

Recently, Oishi et al. presented a fast algorithm for dot products in (simulated) higher precision, called the DotK algorithm [21, 22]. This algorithm is based on error-free transformations and allows to compute a dot product in K-fold double precision, $K \geq 2$. In most cases $K = 2$ will deliver results of sufficient quality and be a lot faster than computation using the software accumulator [1]. Since it is also possible to compute a rigorous bound on the remaining error, the DotK algorithm can be used to compute interval enclosures of the correct result of any dot product in K-fold precision.

In [16], we presented our implementation of the DotK algorithm in several C++ classes using C-XSC. With these classes, the DotK algorithm can be used as a faster, more flexible (but less accurate) alternative to the accumulator in C-XSC. Table 1 shows some results comparing the performance of our implementation of the DotK algorithm and the accumulator on a Pentium 4 with 2.8 GHz. Since C-XSC version 2.3.0, the DotK algorithms are directly included in the C-XSC library [28].

n	**Computed with...**	**real**	**interval**	**complex**	**cinterval**
1000	Accumulator	0.19	0.40	0.72	1.64
	DotK, K=2	0.03	0.12	0.20	0.47
	DotK, K=3	0.09	0.21	0.37	0.87
	DotK, K=4	0.12	0.27	0.47	1.07
	DotK, K=5	0.14	0.32	0.57	1.28
10000	Accumulator	1.85	4.02	7.05	16.22
	DotK, K=2	0.35	1.16	2.03	4.64
	DotK, K=3	0.98	2.27	4.02	9.56
	DotK, K=4	1.24	2.79	5.04	11.62
	DotK, K=5	1.49	3.30	6.07	13.70
100000	Accumulator	18.65	40.32	70.73	161.82
	DotK, K=2	3.53	11.66	20.41	46.46
	DotK, K=3	9.86	24.57	41.48	97.83
	DotK, K=4	12.44	29.76	51.91	118.75
	DotK, K=5	15.02	34.95	62.34	139.64

Table 1: Timings for dot products, $cond = 10^{30}$, repeated 1000 times

In our tests, the numerical quality of the results using the DotK algorithm were indeed of K-fold precision, while always computing an enclosure of the correct result.

[1] The computation of exact dot products of floating point vectors using hardware support would be much faster than the fastest DotK algorithm [17, 5].

4 Parallelization

A former implementation of a parallel verified solver [6, 11] using the same algorithm was based on a master-slave concept, where one node was administrating the whole computation. This approach had several drawbacks, especially concerning load balancing and memory usage. Since the matrices A, R and $[C]$ all had to be stored by the master node, the maximum possible dimension of the system did not increase compared to the serial solver.

In our new implementation, we don't use a master-slave concept. Instead, every step of the solver is divided as equally as possible among the involved nodes, leading to a better load balancing and far better memory usage. Furthermore, highly optimized ScaLAPACK-routines are used to compute the approximate inverse and the matrix [C], thus guaranteeing that the $O(n^3)$ parts of the algorithm are computed as fast as possible.

ScaLAPACK requires a special distribution scheme of the involved matrices, called two dimensional block cyclic distribution. This scheme is used for the whole solver, also for the parts not using ScaLAPACK. In a two dimensional block cyclic distribution, all involved processes are logically ordered (meaning this ordering is only used in software, the true physical ordering still depends on the network) in a process grid as seen in Figure 2. The number of rows and columns is selectable by the user but should normally be the same or nearly the same for best performance.

P_0	P_1	P_2
P_3	P_4	P_5
P_6	P_7	P_8
P_9	P_{10}	P_{11}

Figure 2: Process grid for the two dimensional block cyclic distribution, 12 processes P_i

The matrix is then divided into blocks of size $n_b \times n_b$, where n_b is a choosable parameter. The best value for n_b depends on the used hardware, especially cache sizes and the speed of communication between the processes, and can have a significant impact on the overall performance. A good starting point is $n_b = 256$, from where the best value for the machine used has to be found through trial and error. The process grid is then used to determine which block is stored by which process, as shown in the example in Figure 3.

As stated, the basic elements of the solver are essentially the computation of an approximate inverse, the computation of $[C]$ and the computation of a few matrix-vector products. The approximate inverse can be computed directly by the appropriate ScaLAPACK function. For the computation of $[C]$, the ScaLAPACK routine for matrix-matrix products is used in combination with manipulation of the rounding mode to achieve a verified enclosure. In the most simple case of a real point system, the computation $I - RA$ is performed twice, once with the rounding mode set to downwards and once with the rounding mode set to upwards, giving a lower and an upper bound for $[C]$. The appropriate algorithms for the other cases are given in [24, 29].

The remaining computations are basically matrix-vector products, which are computed using the DotK algorithm (the accumulator can also be used by selecting precision $K = 0$ for our DotK classes, which will in most cases be unnecessary). Since

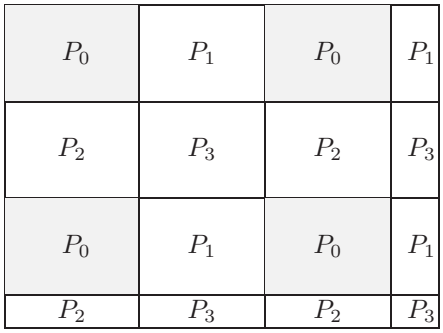

Figure 3: Distribution of a 13×13 matrix in two dimensional block cyclic distribution using a 2×2 process grid based on 4 processes P_i and block size $n_b = 4$

these use no ScaLAPACK routines, the parallelization has to be performed manually.

For this, so called MPI-communicators are introduced for the rows and columns of the process grid. An MPI-communicator basically bundles a number of processes together into a communication group, enabling the programmer to perform a broadcast only inside the communicator (in our case for all processes in the respective row/column). Vectors are always stored completely in every process, since their memory usage is negligible. To compute a matrix-vector product, each process computes the parts of every single dot product for which it stores the corresponding data (the necessary parts of the respective matrix row), so that all processes in one row of the process grid compute a part of the respective dot products. These results are then broadcasted in the respective row of the process grid in form of a long accumulator to prevent rounding errors. Every process in the same row of the process grid can then compute the final result of the respective dot product. The result is then broadcast to all processes in the same column of the process grid, so that finally every process stores the complete result vector of the matrix-vector-product. Algorithm 2 shows this procedure in simplified way (*myrows* is a set containing all indices of the rows of which the respective process stores elements according to the storage scheme explained above).

Input: A distributed matrix A and vector x
Output: The result of A times x
for *all* $i \in myrows$ **do**
    compute own parts of dot product
    broadcast intermediate results in own row
    compute final result for row i
    broadcast final result in own column

Algorithm 2: Parallel matrix-vector product

The second stage of the solver, using an inverse of double length, is intended for badly conditioned system matrices (condition number $\geq 10^{15}$). In this stage, only

the approximate inverse can be computed using ScaLAPACK, the computation of $[C]$ uses the DotK algorithm to achieve results of better quality (without a higher precision dot product, the verification step will also most likely fail in these cases). Especially because of this, the second stage takes a lot more time than the first stage. A parallel version can be very helpful in this case, even for lower dimension like $n = 1000$ (see timings in Section 5).

The matrix-matrix products in the computation of $[C]$ for Stage 2 now also have to be parallelized manually. For this the process grid is set to only have one column, so that all processes store complete rows of the matrix. This leads to significant advantages for Stage 2, since the previous distribution would require a lot of communication of intermediate results which for accuracy reasons have to be communicated in form of long accumulators, leading to a lot of communication overhead. With this new data distribution, the parallelization can be done in a pretty straightforward way that we don't explain here (see [16, 29] for more details).

5 Test results

In this section, we present a few results of our tests on different platforms, demonstrating the efficiency of our solvers. The results presented here are intended to give a broad overview of the performace of the solver and to show that it runs and performs well on platforms with vastly different architecture. ScaLAPACK 1.8.0 and C-XSC 2.2.3 are used for all tests.

The first system is a cluster of 24 standard PCs with Core2Duo processor clocked at 2.33 GHz, 2GB RAM and a standard gigabit ethernet. We used the GNU Compiler, Version 4.2.1 and ATLAS BLAS Version 3.8.1.

On this system, we solve a real, interval, complex, and complex interval system respectively, each with dimension $n = 5000$ and precision $K = 2$ for dot products. The Tables 2, 3 and 4 show timings, speed up (if $t(p)$ is the needed time using p processes, the speed up using p processes is $\frac{t(p)}{t(1)}$) and numerical quality for Stage 1 of the solver. For all tests using only one process the serial version of the solver described in [16, 29] has been used. Apparently, the solver scales quite nicely, though for 4 and especially 8 processors, the problem size here is so small that the communication overhead is too large for optimal speed ups. The quality of the numerical results is very good overall, giving a tight enclosure of the actual solution.

P	**real**	**interval**	**complex**	**cinterval**
1	124.5	180.8	589.9	690.3
2	78.7	103.3	295.7	346.5
4	53.7	69.7	187.4	212.3
8	39.2	51.4	119.1	133.9

Table 2: Time in s, condition 10^{10}, $n = 5000$, $K = 2$

P	**real**	**interval**	**complex**	**cinterval**
1	1.00	1.00	1.00	1.00
2	1.58	1.75	1.99	1.99
4	2.32	2.59	3.15	3.25
8	3.18	3.52	4.95	5.16

Table 3: Speed up, condition 10^{10}, $n = 5000$, $K = 2$

P	**real**	**interval**	**complex**	**cinterval**
1	14.6	5.0	(13.9, 14.0)	(3.8, 4.0)
2	14.6	5.0	(13.9, 14.0)	(3.8, 4.0)
4	15.3	5.0	(14.7, 14.8)	(3.8, 4.0)
8	15.3	5.0	(14.7, 14.8)	(3.8, 4.0)

Table 4: Average number of exact digits, condition 10^{10}, $n = 5000$, $K = 2$

Tables 5, 6 and 7 show the corresponding results for the second stage of the solver. In these tests, the matrix had a condition number of about 10^{17}, for which in the first stage the approximate inverse will be too bad, meaning that the spectral radius of $|I - RA|$ will be greater than one and thus no verification will be possible. Using the second stage, a verified enclosure of the solution can be found. Since the computing times are a lot higher for the second stage, only dimension $n = 1000$ is used.

P	**real**	**interval**	**complex**	**cinterval**
1	173.5	281.7	578.7	1047.3
2	87.6	138.1	284.9	516.0
4	42.5	69.5	147.7	260.5
8	25.1	39.0	75.2	133.8

Table 5: Time in sec., condition 10^{17}, $n = 1000$, $K = 3$

Since the second stage requires a lot of computations and the communication overhead thus becomes less important, the speed up is very good also for 4 and 8 processors. The numerical quality of the results is very good, although for the interval systems no verified solution could be found, which is a typical behavior for such badly conditioned systems. Since the quality of the results stayed basically the same for all systems we tested our solvers on, we only give timing information for the following systems.

The next machine is AliceNext, the supercomputer of the University of Wuppertal. It consists of 2×512 AMD Opteron processors with 1.8GHz and 1GB RAM per

P	**real**	**interval**	**complex**	**cinterval**
1	1.00	1.00	1.00	1.00
2	1.98	2.04	2.03	2.03
4	4.08	4.05	3.92	4.02
8	6.91	7.22	7.70	7.83

Table 6: Speed up, condition 10^{17}, $n = 1000$, $K = 3$

P	**real**	**interval**	**complex**	**cinterval**
1	15.8	–	(15.8, 15.8)	–
2	15.8	–	(15.8, 15.8)	–
4	15.8	–	(15.8, 15.8)	–
8	15.8	–	(15.8, 15.8)	–

Table 7: Average number of exact digits, condition 10^{17}, $n = 1000$, $K = 3$

processor. The network is a Gigabit-Ethernet ordered in a 2D-Torus. On this machine, we used the GNU compiler version 3.3.1 and the AMD Core Math Library. Since this system is out of date and will be out of service soon, we only show a few timings in Table 8 to compare with the above results.

P	**real**	**interval**	**complex**	**cinterval**
1	298.1	465.5	–	–
2	191.8	278.0	789.5	902.7
4	120.8	166.3	461.7	483.3
8	86.7	95.6	250.0	289.9

Table 8: Time in s, condition 10^{10}, $n = 5000$, $K = 2$

Now we want to focus on tests involving large dense linear systems. First we take a look at the XC6000 cluster at the University of Karlsruhe. This cluster consists of 128 Intel Itanium2 processors with 1.5GHz, 6GB RAM per processor and a Quadrics WsNet II interconnect network. We used the Intel Compiler 10.0 and the Intel Math Kernel Library 10.0 on this machine. Results for real systems are shown in Table 9 and 10, results for interval systems are shown in Table 11 and 12.

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	108.1	52.0	35.3
$n = 25000$	1188.0	532.2	299.7
$n = 50000$	-	-	1978.6

Table 9: $K = 2$, well conditioned real system

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	80.1%	59.0%
$n = 25000$	-	89.3%	79.3%
$n = 50000$	-	-	-

Table 10: $K = 2$, well conditioned real system, speed up given as percentage of theoretical optimum

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	136.0	64.6	42.2
$n = 25000$	1571.7	687.3	385.3
$n = 50000$	-	-	2561.1

Table 11: $K = 2$, well conditioned interval system

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	84.2%	64.5%
$n = 25000$	-	91.5%	81.6%
$n = 50000$	-	-	-

Table 12: $K = 2$, well conditioned interval system, speed up given as percentage of theoretical optimum

In the Tables 9 and 11 the symbol − indicates that the system could not be solved due to memory limitations. These results show that the accumulated memory of all nodes is the only limiting factor for the dimension of the system to solve. So using 100 processors with 6GB each we were able to solve a dense system of dimension $n = 50000$. The speed up for such large systems is very good as well. Using the full cluster (all 128 processors), we were even able to solve a real point system of dimension $n = 100000$ in about 200 minutes.

Finally, we take a short look at some timings for the JUMP cluster from the Forschungszentrum Jülich. It consists of 14 nodes with 32 IBM Power6 processors clocked 4.7GHz each and 128GB RAM per node. It uses an Infiniband network. On this machine, we used the IBM XLC compiler. The timings are shown in Table 13 and 14.

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	95.8	44.4	30.7
$n = 25000$	1265.9	552.8	289.8
$n = 50000$	-	-	2130.0

Table 13: $K = 2$, well conditioned real system

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	86.3%	62.4%
$n = 25000$	-	91.6%	87.4%
$n = 50000$	-	-	-

Table 14: $K = 2$, well conditioned real system, speed up given as percentage of theoretical optimum

The results are comparable to that of the XC6000 cluster. Keep in mind however, that the JUMP and XC6000 clusters have a vastly different architecture. This shows that our solvers are highly portable and perform well on a wide range of platforms due to the use of optimized libraries for each system.

6 Conclusion

Our verified solvers for dense linear (interval-)systems using C-XSC are fast and flexible. The parallel version for distributed memory systems presented in this paper can be used to solve large dense systems in an acceptable time with good numeric results. The parallelization is also helpful for the second stage of the solver for badly conditioned systems, which is very demanding in terms of computing time because of the use of high precision dot product algorithms.

Since this solver is intended for general dense linear systems, it is inefficient when using systems with a special structure and especially sparse systems. While it can solve such system in general, it doesn't take advantage of the sparsity, neither in terms of computations nor in terms of memory usage. Because of this, specialized sparse solvers are needed. For sparse systems, Algorithm 1 can not be used since the approximate inverse R will in general be dense even for sparse A. Thus, a modified version of this algorithm (or a completely different approach) has to be used (see for example [15, 26]). This will be a focus of our future work.

Acknowledgments

We would like to thank Prof Dr. Lippert and Wolfgang Frings of the Forschungszentrum Jülich for granting us access to the JUMP-cluster.

References

[1] Downloads:
C-XSC library: http://www.math.uni-wuppertal.de/~xsc/
Solvers: http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html

[2] Intel Software Network Community:
http://software.intel.com/en-us/forums/intel-math-kernel-library/topic/57435/

[3] Blackford, L. S.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; Heroux, M.; Kaufman, L.; Lumsdaine, A.; Petitet, A.; Pozo, R.; Remington, K.; Whaley, R. C.: An Updated Set of Basic Linear Algebra Subprograms (BLAS) *ACM Trans. Math. Softw.*, **28(2)** (2002), pp. 135–151.

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

[5] Bohlender, G.: What Do We Need Beyond IEEE Arithmetic?, *Computer Arithmetic and Self-Validating Numerical Methods* (Ch. Ullrich, ed.), Academic Press, San Diego, 1990, pp. 1–32.

[6] Grimmer, M.: *Selbstverifizierende Mathematische Softwarewerkzeuge im High-Performance Computing. Konzeption, Entwicklung und Analyse am Beispiel der parallelen verifizierten Lösung linearer Fredholmscher Integralgleichungen zweiter Art*, Logos Verlag, 2007.

[7] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*, Springer Verlag, 1993.

[8] Hofschuster, W.; Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing, in *Numerical Software with Result Verification*, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, 2004, pp. 15–35.

[9] Hölbig, C.; Krämer, W.: Selfverifying solvers for dense systems of linear equations realized in C-XSC. Technical Report BUW-WRSWT 2003/1, 2003.

[10] Klatte, Kulisch, Wiethoff, Lawo, Rauch: *C-XSC — A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, Heidelberg, 1993.

[11] Kolberg, M., Fernandes, L. G., Claudio, D.: Dense Linear System: A Parallel Self-verified Solver. *International Journal of Parallel Programming*, **36(4)** (2008), pp. 412–425.

[12] Kolberg, M., Bohlender, G., Claudio, D.: Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation, in *Lecture Notes in Computer Science: 8th VECPAR - International Meeting on High Performance Computing for Computational Science 5336/2008, Toulouse, France, 2008, Revised Selected Papers*, Springer Verlag, 2008.

[13] Kolberg, M., Krämer, W., Zimmer, M.: A Note on Solving Problem 7 of the SIAM 100-Digit Challenge Using C-XSC, in *Springer Lecture Notes in Computer Science: Cuyt et al. (Editors), Numerical Validation in Current Hardware Architectures*, Springer Verlag, 2008

[14] Kolberg, M.: *Parallel Self-Verified Solver for Dense Linear Systems*. Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2009.

[15] Krämer, W., Kulisch, U., Lohner, R.: *Numerical Toolbox for Verified Computing II, Advanced Numerical Problems*. Draft, about 400 pages. Available on the web: `http://www.uni-karlsruhe.de/~Rudolf.Lohner/papers/tb2.ps.gz`

[16] Krämer, W., Zimmer, M.: Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS, *Lecture Notes in Computer Science: Cuyt et al. (Editors), Numerical Validation in Current Hardware Architectures*, Springer Verlag, 2008.

[17] Kulisch, U.: *Computer Arithmetic and Validity - Theory, Implementation and Applications*. De Gruyter, Studies in Mathematics 33, 2008.

[18] Kulisch, U.; Miranker, W.: The arithmetic of the digital computer: A new approach, *SIAM Rev.* **28(1)**:1–40, 1986.

[19] Kulisch, U.: Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation, 1997.

[20] Krawczyk, R.: Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing* **4** (1969), pp. 187–201.

[21] Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product, *SIAM Journal on Scientific Computing* **26(6)** (2005), pp. 1955–1988.

[22] Oishi, S., Tanabe, K., Ogita, T., Rump, S.M., Yamanaka, N.: A Parallel Algorithm of Accurate Dot Product, *Parallel Computing* **34(6–8)** (2008), pp. 392–410.

[23] Oishi, S., Ogita, T., Rump, S.M.: Iterative Refinement for Ill-conditioned Linear Equations, *2008 International Symposium on Nonlinear Theory and its Applications, NOLTA'08, Budapest, Hungary, September 7-10*, pp. 516–519, 2008.

[24] S.M. Rump. INTLAB - INTerval LABoratory, *Developments in Reliable Computing* (T. Scendes ed.), Kluwer Academic Publishers, Dordrecht, 1999, pp. 77–104.

[25] Rump, S.M.: *Kleine Fehlerschranken bei Matrixproblemen*. Dissertation, University of Karlsruhe, 1980.

[26] Rump, S.M.: Validated Solution of Large Linear Systems, *Validation Numerics: Theory and Applications* (R. Albrecht, G. Alefeld, and H.J. Stetter eds.), volume 9 of Computing Supplementum, pages 191-212. Springer, 1993.

[27] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: *MPI: The Complete Reference*, MIT Press, 1995

[28] Zimmer, M., Krämer, W., Bohlender, G., Hofschuster, W.: Extension of the C-XSC Library with Scalar Products with Selectable Precision, *Serdica Journal of Computing* **4(3)** (2010), pp. 349–370.

[29] Zimmer, M.: *Laufzeiteffiziente, parallele Löser für lineare Intervallgleichungssysteme in C-XSC*. Master's Thesis, University of Wuppertal, 2007.