Efficient parallel solvers for large dense systems of linear interval equations

Mariana Kolberg[1], Walter Krämer[2], Michael Zimmer[2]

[1]Pontifícia Universidade Católica do Rio Grande do Sul, Brazil
[2]University of Wuppertal, Germany

October 2, 2008

Outline

Outline

Task

Find a verified enclosure $[x]$ for the solution of the dense linear system $[A][x] = [b]$, $A \in \mathbb{IR}^{n \times n}$, $b \in \mathbb{IR}^n$.

Also allowed:

- Real point systems
- Complex point systems
- Complex interval systems

Task

Find a verified enclosure $[x]$ for the solution of the dense linear system $[A][x] = [b]$, $A \in \mathbb{IR}^{n \times n}$, $b \in \mathbb{IR}^n$.

Also allowed:

- Real point systems
- Complex point systems
- Complex interval systems

Task

Find a verified enclosure $[x]$ for the solution of the dense linear system $[A][x] = [b]$, $A \in \mathbb{IR}^{n \times n}$, $b \in \mathbb{IR}^n$.

Also allowed:

- Real point systems
- Complex point systems
- Complex interval systems

Task

Find a verified enclosure $[x]$ for the solution of the dense linear system $[A][x] = [b]$, $A \in \mathbb{IR}^{n \times n}$, $b \in \mathbb{IR}^n$.

Also allowed:

- Real point systems
- Complex point systems
- Complex interval systems

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Problem

Problems of higher dimension require a lot of processing power and especially a lot of memory.

- Verified solution using Rump's algorithm is 6-8 times slower than normal floating point solution using LU-decomposition.
- Memory requirements: Matrices A, R and $[C]$
  - Real point system, n=20000: about 12GB
  - Real point system, n=50000: about 75GB
  - Real interval system, n=50000: about 93GB
  - Complex point system, n=50000: about 150GB
  - Complex interval system, n=50000: about 186GB

Solution: Parallelization for distributed memory systems.

Algorithm: Verified solution of dense linear (interval-)systems

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 $\widetilde{x}$ *exact enough or max. iterations reached*
$Z := R \diamond (b - A\widetilde{x})$
$C := \diamond(I - RA)$
$Y := Z$
repeat
  $Y_A := \text{blow}(Y, \epsilon)$
  $Y := Z + C \cdot Y_A$
until $Y \subset int(Y_A)$ *or max. iterations reached*
if $Y \subset int(Y_A)$ **then**
  Unique solution in $x \in \widetilde{x} + Y$
else
  Algorithm failed, A is singular or condition is too bad

For badly conditioned systems: Second stage using inverse of double length and extended precision dot products.

- Intlab: Fast, easy to use
- Serial C-XSC solvers: Fast, high accuracy
- Intel MKL and CMKL: Krawczyk-Solver is about 20 times slower, discontinued(?)

- Intlab: Fast, easy to use
- Serial C-XSC solvers: Fast, high accuracy
- Intel MKL and CMKL: Krawczyk-Solver is about 20 times slower, discontinued(?)

- Intlab: Fast, easy to use
- Serial C-XSC solvers: Fast, high accuracy
- Intel MKL and CMKL: Krawczyk-Solver is about 20 times slower, discontinued(?)

Outline

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

- C++ library for eXtended Scientific Computing
- Basic datatypes: `real`, `interval`, `complex`, `cinterval`
- Datatypes for vectors and matrices
- Many built in functions
- Computation of sums and dot products with maximum precision using fixed-point accumulator
- Toolbox with algorithms for many problems (linear- and nonlinear systems, optimization, ...)

In C-XSC

Use a fix-point accumulator of sufficient length

g	$2e_{max}$	t	t	$2\lvert e_{min}\rvert$

- Dot products can be computed with *maximum* accuracy
- Slow! (when realized in software)

In C-XSC

Use a fix-point accumulator of sufficient length

g	$2e_{max}$	t	t	$2\lvert e_{min}\rvert$

- Dot products can be computed with *maximum* accuracy
- Slow! (when realized in software)

In C-XSC

Use a fix-point accumulator of sufficient length

g	$2e_{max}$	t	t	$2\lvert e_{min}\rvert$

- Dot products can be computed with *maximum* accuracy
- Slow! (when realized in software)

DotK algorithm (Ogita, Rump, Oishi)

Compute dot product in K-fold working precision by using error free transformations.

Error free transformations

For all $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$ there exists a $y \in \mathbb{F}$ with

$$a \circ b = x + y$$

and $x = fl(a \circ b)$.

DotK algorithm (Ogita, Rump, Oishi)

Compute dot product in K-fold working precision by using error free transformations.

Error free transformations

For all $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$ there exists a $y \in \mathbb{F}$ with

$$a \circ b = x + y$$

and $x = fl(a \circ b)$.

DotK algorithm (Ogita, Rump, Oishi)

Compute dot product in K-fold working precision by using error free transformations.

Error free transformations

For all $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$ there exists a $y \in \mathbb{F}$ with

$$a \circ b = x + y$$

and $x = fl(a \circ b)$.

- Dot product in K-fold working precision
- Uses pure floating point operations
- Reliable error bound can be computed with floating point operations

- Dot product in K-fold working precision
- Uses pure floating point operations
- Reliable error bound can be computed with floating point operations

- Dot product in K-fold working precision
- Uses pure floating point operations
- Reliable error bound can be computed with floating point operations

```
Exact result: 9.9999999999999995E-021 Condition: 1E+020

Direct computation using C-XSC operators:
-3.4944599247537239E-012
Time used: 0.003654s

Computation using accumulator:
9.9999999999999995E-021
Time used: 0.148152s

Computation using DotK:
k=2:
[3.7130847403408012E-021,1.6286857271626398E-020]
Time used: 0.0236309s

k=3:
[9.9999999999999979E-021,1.0000000000000001E-020]
Time used: 0.0542479s

k=4:
[9.9999999999999994E-021,9.9999999999999995E-021]
Time used: 0.0600731s

k=5:
[9.9999999999999994E-021,9.9999999999999995E-021]
Time used: 0.0686409s
```

- BLAS and LAPACK: Highly optimized routines for numerical algebra
- ScaLAPACK: Special version for distributed memory parallelization
- Manipulate rounding mode to get verified enclosures.

Example: Product of two real matrices

Input: Two real matrices A and B
Output: An interval matrix enclosing $C = AB$

```
SetRound(-1);
SetInf(C, A*B);
SetRound(1);
SetSup(C, A*B);
```

- BLAS and LAPACK: Highly optimized routines for numerical algebra
- ScaLAPACK: Special version for distributed memory parallelization
- Manipulate rounding mode to get verified enclosures.

Example: Product of two real matrices

Input: Two real matrices A and B
Output: An interval matrix enclosing $C = AB$

```
SetRound(-1);
SetInf(C, A*B);
SetRound(1);
SetSup(C, A*B);
```

- BLAS and LAPACK: Highly optimized routines for numerical algebra
- ScaLAPACK: Special version for distributed memory parallelization
- Manipulate rounding mode to get verified enclosures.

Example: Product of two real matrices

Input: Two real matrices A and B
Output: An interval matrix enclosing $C = AB$

```
SetRound(-1);
SetInf(C, A*B);
SetRound(1);
SetSup(C, A*B);
```

- BLAS and LAPACK: Highly optimized routines for numerical algebra
- ScaLAPACK: Special version for distributed memory parallelization
- Manipulate rounding mode to get verified enclosures.

Example: Product of two real matrices

Input: Two real matrices A and B
Output: An interval matrix enclosing $C = AB$

```
SetRound(-1);
SetInf(C, A*B);
SetRound(1);
SetSup(C, A*B);
```

Outline

Basic concept

All matrices are distributed equally among processes, every step of the algorithm is computed by all processes

- Uses ScaLAPACK instead of LAPACK
- Uses two dimensional block cyclic distribution for matrices

Basic concept

All matrices are distributed equally among processes, every step of the algorithm is computed by all processes

- Uses ScaLAPACK instead of LAPACK
- Uses two dimensional block cyclic distribution for matrices

Basic concept

All matrices are distributed equally among processes, every step of the algorithm is computed by all processes

- Uses ScaLAPACK instead of LAPACK
- Uses two dimensional block cyclic distribution for matrices

Process grid:

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

Number of rows and columns should be the same or nearly the same.

Matrix is distributed accordingly:

Optimal size of blocks is hardware dependent.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

MPI communicators are introduced for every row and column of process grid.
→ Broadcasts limited to a row or column are possible.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

MPI communicators are introduced for every row and column of process grid.
→ Broadcasts limited to a row or column are possible.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

MPI communicators are introduced for every row and column of process grid.
→ Broadcasts limited to a row or column are possible.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

MPI communicators are introduced for every row and column of process grid.
→ Broadcasts limited to a row or column are possible.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

MPI communicators are introduced for every row and column of process grid.
→ Broadcasts limited to a row or column are possible.

- Matrix-matrix products and approximate inverse are computed with ScaLAPACK.
- Matrix-vector products are computed using high precision dot products.
- These dot products must be split among processes in a row.
- Intermediate results are stored in accumulators.

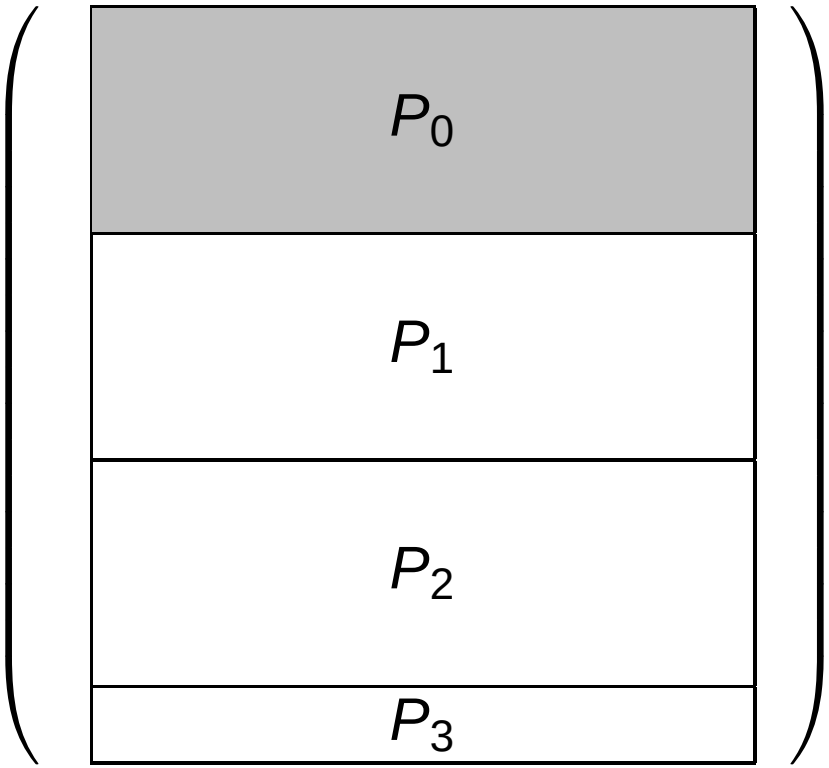
MPI communicators are introduced for every row and column of process grid.
$\rightarrow$ Broadcasts limited to a row or column are possible.

Algorithm: Parallel matrix-vector product

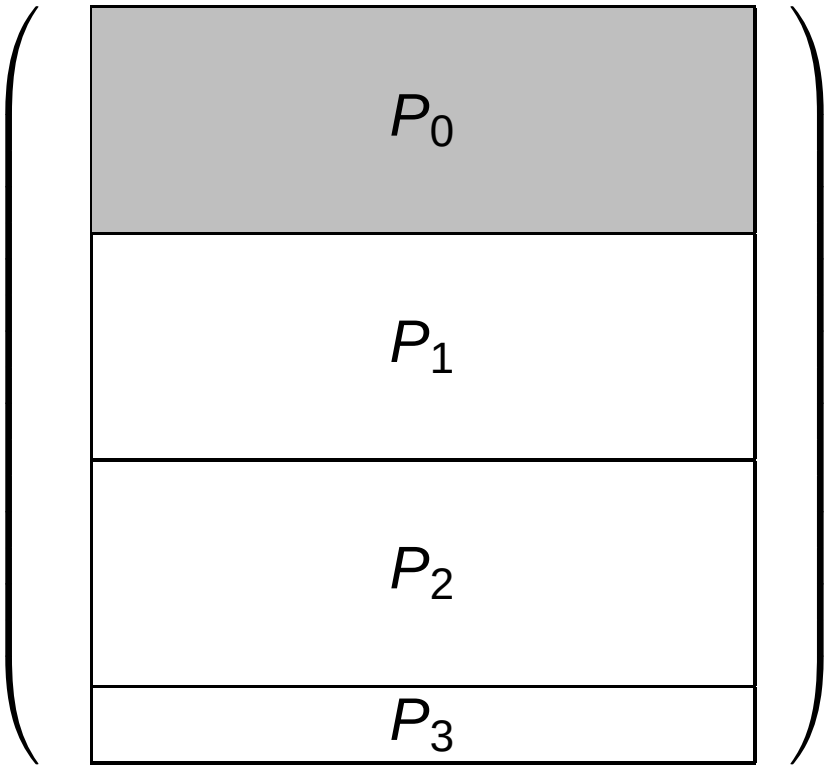
Input: A matrix A and vector x
Output: The result of A times x
for *all* $i \in$ *myrows* **do**
    compute own parts of dot product
    broadcast intermediate results in own row
    compute final result for row i
    broadcast final result in own column

Now all processes know the result of Ax. (Necessary to check break conditions etc.)

- Use of higher precision dot products, BLAS/LAPACK only used for inversion
- Switch to a special case of the two-dimensional block cyclic distribution



- Use of higher precision dot products, BLAS/LAPACK only used for inversion
- Switch to a special case of the two-dimensional block cyclic distribution



Parallel matrix-matrix product $R = AB$ in stage two:

- R is stored in the same way as A and B.
- Every process needs the vertical blocks of size $n \times nb$ of A.
- Every process broadcasts his part of this block.
- Computations for the corresponding vertical $n \times nb$ block can then be performed in parallel.
- This is repeated for all P vertical blocks.

Parallel matrix-matrix product $R = AB$ in stage two:

- R is stored in the same way as A and B.
- Every process needs the vertical blocks of size $n \times nb$ of A.
- Every process broadcasts his part of this block.
- Computations for the corresponding vertical $n \times nb$ block can then be performed in parallel.
- This is repeated for all P vertical blocks.

Parallel matrix-matrix product $R = AB$ in stage two:

- R is stored in the same way as A and B.
- Every process needs the vertical blocks of size $n \times nb$ of A.
- Every process broadcasts his part of this block.
- Computations for the corresponding vertical $n \times nb$ block can then be performed in parallel.
- This is repeated for all P vertical blocks.

Parallel matrix-matrix product $R = AB$ in stage two:

- R is stored in the same way as A and B.
- Every process needs the vertical blocks of size $n \times nb$ of A.
- Every process broadcasts his part of this block.
- Computations for the corresponding vertical $n \times nb$ block can then be performed in parallel.
- This is repeated for all P vertical blocks.

Parallel matrix-matrix product $R = AB$ in stage two:

- R is stored in the same way as A and B.
- Every process needs the vertical blocks of size $n \times nb$ of A.
- Every process broadcasts his part of this block.
- Computations for the corresponding vertical $n \times nb$ block can then be performed in parallel.
- This is repeated for all P vertical blocks.

How is the system matrix distributed in the beginning?

1. Process 0 stores the complete matrix in the beginning and distributes it.
2. A function pointer to a function like `void getA(int i, int j, real &r)` is used, every process fills his part of the matrix with this function.

How is the system matrix distributed in the beginning?

1. Process 0 stores the complete matrix in the beginning and distributes it.
2. A function pointer to a function like `void getA(int i, int j, real &r)` is used, every process fills his part of the matrix with this function.

Outline

Some Remarks concerning compilation:

- DotK algorithm does not work if processor uses higher precision registers. Appropriate compiler switches must be set (use of SSE floating point registers,...).
- Inlining has a very high impact on the performance. Appropriate compiler switches must be used (Inlining limits may have to be extended).
- OpenMP instructions require an OpenMP capable compiler, preferrably the latest version.

Some Remarks concerning compilation:

- DotK algorithm does not work if processor uses higher precision registers. Appropriate compiler switches must be set (use of SSE floating point registers,...).
- Inlining has a very high impact on the performance. Appropriate compiler switches must be used (Inlining limits may have to be extended).
- OpenMP instructions require an OpenMP capable compiler, preferrably the latest version.

Some Remarks concerning compilation:

- DotK algorithm does not work if processor uses higher precision registers. Appropriate compiler switches must be set (use of SSE floating point registers,...).
- Inlining has a very high impact on the performance. Appropriate compiler switches must be used (Inlining limits may have to be extended).
- OpenMP instructions require an OpenMP capable compiler, preferrably the latest version.

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

Hardware:

- 24 standard PCs
- CPU: Core2Duo 2.33GHz
- 2GB RAM
- Standard Gigabit Ethernet

Software:

- GNU compiler 4.2.1
- LAM MPI
- ATLAS BLAS 3.8.1

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

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

Speed up, 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

Average number of exact digits, 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	$(14.7, 14.8)$	$(3.8, 4.0)$

Time in s, condition 10^{17}, $n = 1000$, $K = 3$

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

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

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

Average number of exact digits, 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	–	$(15.8, 15.8)$	–

Time in s, OpenMP version, random real matrix, $n = 1000$, $K = 2$

	real	**interval**	**complex**	**cinterval**
$P = 1$	1.38	1.82	5.06	6.02
$P = 2$	0.87	1.04	2.86	3.38
Speed Up	1.59	1.75	1.77	1.78

Time in s, OpenMP version, random real matrix, $n = 2000$, $K = 2$

	real	**interval**	**complex**	**cinterval**
$P = 1$	8.88	11.42	33.56	38.43
$P = 2$	5.12	6.26	17.90	20.69
Speed Up	1.73	1.82	1.87	1.86

Time in s, OpenMP version, random real matrix, $n = 3000$, $K = 2$

	real	**interval**	**complex**	**cinterval**
$P = 1$	27.41	35.57	105.71	119.87
$P = 2$	15.27	19.21	55.98	63.61
Speed Up	1.80	1.85	1.89	1.88

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

Hardware:

- 512 $\times$ 2 AMD Opteron 1.8GHz
- 1GB RAM per processor
- Gigabit-Ethernet + 2D-Torus

Software:

- OS: Linux
- GNU Compiler 3.3.1
- AMD Core Math Library

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

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

Hardware:

- 128 Intel Itanium 2 1.5GHz
- 6GB per processor
- Quadrics QsNet II interconnect

Software:

- Intel Compiler 10.0
- Intel Math Kernel Library 10.0

Hardware:

- 128 Intel Itanium 2 1.5GHz
- 6GB per processor
- Quadrics QsNet II interconnect

Software:

- Intel Compiler 10.0
- Intel Math Kernel Library 10.0

Hardware:

- 128 Intel Itanium 2 1.5GHz
- 6GB per processor
- Quadrics QsNet II interconnect

Software:

- Intel Compiler 10.0
- Intel Math Kernel Library 10.0

Hardware:

- 128 Intel Itanium 2 1.5GHz
- 6GB per processor
- Quadrics QsNet II interconnect

Software:

- Intel Compiler 10.0
- Intel Math Kernel Library 10.0

Hardware:

- 128 Intel Itanium 2 1.5GHz
- 6GB per processor
- Quadrics QsNet II interconnect

Software:

- Intel Compiler 10.0
- Intel Math Kernel Library 10.0

$K = 2$, well conditioned real system

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

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

Speed Up is given as percentage of theoretical optimum

$K = 2$, well conditioned interval system

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

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

Speed Up is given as percentage of theoretical optimum

Since data is distributed equally among processors, huge dense systems can be solved.

Using the full XC6000 cluster (128 Itanium 2 processors, 6GB per processor), we could solve a real system of dimension 100000×100000 in 12118 seconds.

Since data is distributed equally among processors, huge dense systems can be solved.

Using the full XC6000 cluster (128 Itanium 2 processors, 6GB per processor), we could solve a real system of dimension 100000×100000 in 12118 seconds.

Hardware:

- 14 nodes with 32 IBM Power6 4.7GHz processors
- Memory: 14×128 Gigabyte
- Infiniband network

Software:

- OS: AIX 5.3
- IBM Compiler XLC

Hardware:

- 14 nodes with 32 IBM Power6 4.7GHz processors
- Memory: 14 $\times$ 128 Gigabyte
- Infiniband network

Software:

- OS: AIX 5.3
- IBM Compiler XLC

Hardware:

- 14 nodes with 32 IBM Power6 4.7GHz processors
- Memory: 14×128 Gigabyte
- Infiniband network

Software:

- OS: AIX 5.3
- IBM Compiler XLC

Hardware:

- 14 nodes with 32 IBM Power6 4.7GHz processors
- Memory: 14×128 Gigabyte
- Infiniband network

Software:

- OS: AIX 5.3
- IBM Compiler XLC

Hardware:

- 14 nodes with 32 IBM Power6 4.7GHz processors
- Memory: 14×128 Gigabyte
- Infiniband network

Software:

- OS: AIX 5.3
- IBM Compiler XLC

$K = 2$, well conditioned real system

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

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

Speed Up is given as percentage of theoretical optimum

Outline

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

- Fast and very accurate verified solvers in C-XSC.
- Efficient parallelization.
- Parallel solvers can solve very large dense systems.
- Tested with satisfying results (performance and accuracy) on very different architectures.

- Outlook
  - Solvers for sparse systems.
  - Use of OpenMP for dot products in parallel solvers(?)

Thank you