High Performance Iterative Methods using Accurate Computations

Hidehiko Hasegawa

Graduate School of Library, Information and Media Studies
University of Tsukuba

ICL lunch talk at March 16, 2012

Accelerate Iterative Methods

- Good Algorithms
- Good Preconditioners
- Parallel Algorithms
- Good Implementations
- Accurate Computations

Accelerate Iterative Methods

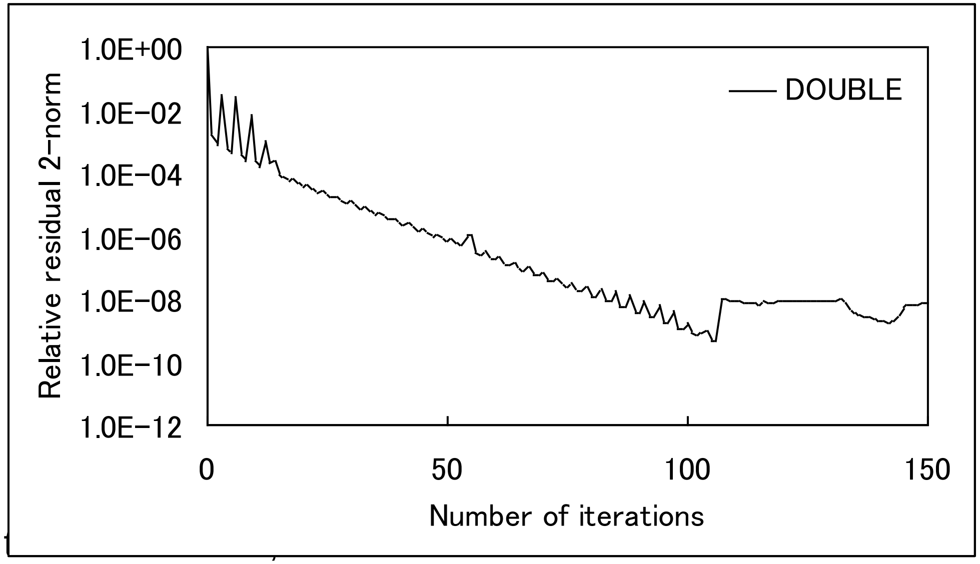
- Good Algorithms
  Need Innovation
- Good Preconditioners
  Need Innovation
- Parallel Algorithms
  More iterations or low performance
- Good Implementations
  Depends on Hardware and Data Structure
- Accurate Computations
  High Cost

Problem

- Theoretically Krylov methods converge at most n iterations
- Divergence or Stagnation occurs because of round-off errors

$$\begin{pmatrix} 2 & 1 & & & & & \\ 0 & 2 & 1 & & & & \\ 1.4 & 0 & 2 & 1 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & & 1.4 & 0 & 2 & 1 \\ & & & & 1.4 & 0 & 2 \end{pmatrix}$$

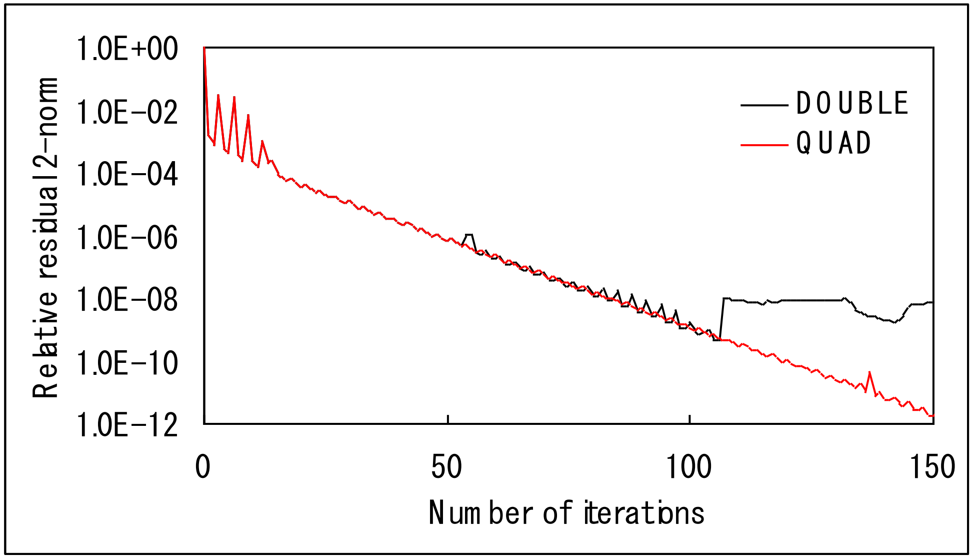
n：10^5



Answer

- To improve convergence, High-precision arithmetic operations are effective.
- However they are costly, real *16 of Fortran:
  - memory: Double
  - comp.: 20 times

Fast Quadruple are necessary



Our Solution:

Utilize Accurate Computations for Iterative Methods

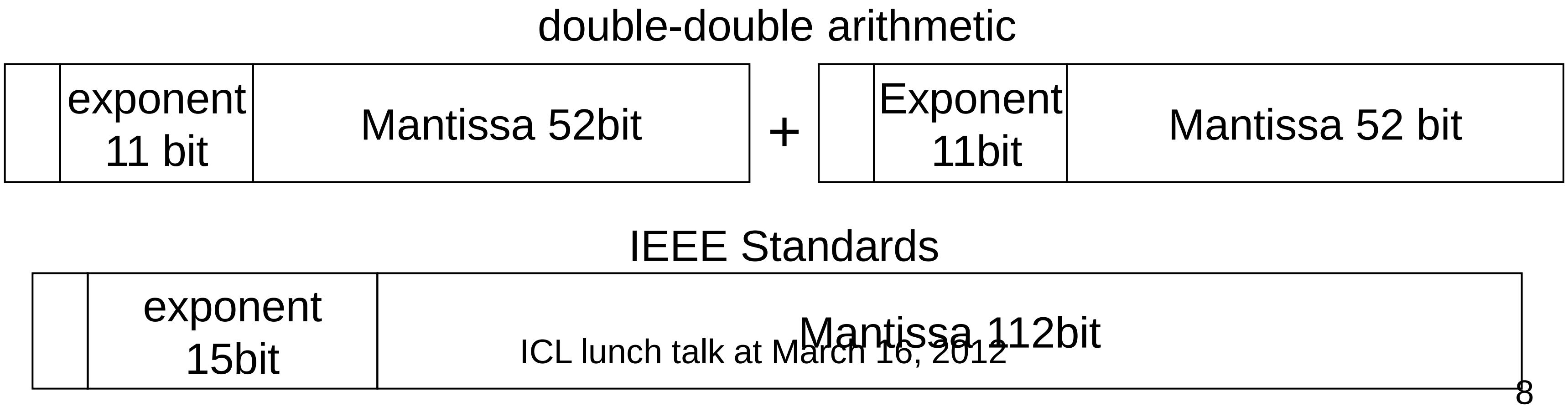
- Use Double-double
- Use D-D vectors and Double Matrices (Mixed Precision Arithmetic Operations)
- Use SSE and AVX
- Restart with different Precision

Advantages

- Tough for round-off errors
- Small Additional Memory
- Small Additional Communications
- Much Computations
- Applicable for **ALL** Iterative Methods (even if serial computation such as ILU)

Implementation of Double-double Arithmetic

- Quadruple value is stored in two double floating point numbers
  - Double-double arithmetic: a = a.hi + a.lo, |a.hi|>|a.lo|
  - 8 bits less than IEEE standards
  - Effective digits are approx. 31 vs. 33 digits.



double-double arithmetic

exponent 11 bit | Mantissa 52bit + Exponent 11bit | Mantissa 52 bit

IEEE Standards

exponent 15bit | Mantissa 112bit

Round-off Error Free Double Arithmetic Addition

- Round-off error free addition can be done with two double precision variables:

$$a + b = fl(a + b) + err(a + b)$$

  - a,b : double floating point variables
  - fl(a + b) : addition of a and b in double
  - err(a+b) : (a+b) – fl(a+b): error

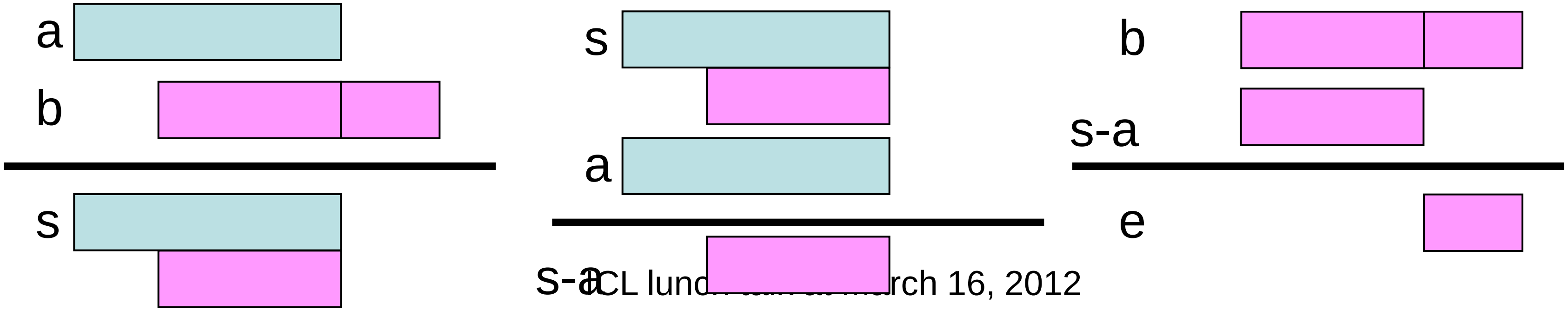
Basic Algorithm

- Dekker showed round-off error free addition in double

|a|>=|b| 3flops. Others 6flops.

```
FAST_TWO_SUM(a,b,s,e)
    s = a + b
    e = b - (s - a)
```

```
TWO_SUM(a,b,s,e)
    s = a + b
    v = s - a
    e = (a - (s - v)) + (b - v)
```



Quadruple Addition of a=b+c

b.hi | b.lo + c.hi | c.lo

A. TWO_SUM for upper parts

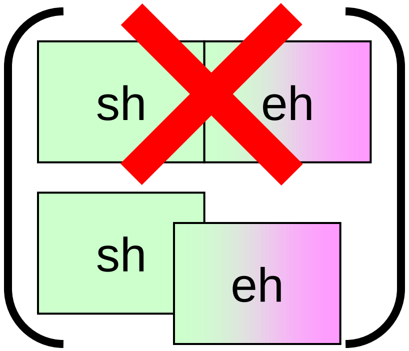
b.hi + c.hi = sh | eh

B. Addition of lower parts

b.lo + c.lo = b.lo+c.lo

C. Addition of result and error of upper part

b.lo+c.lo + eh = eh



D. FAST_TWO_SUM of results A and C

sh + eh = a.hi | a.lo

Quadruple Addition of a=b+c

20 flops

```
ADD(a,b,c)
    TWO_SUM(b.hi,c.hi,sh,eh)
    TWO_SUM(b.lo,c.lo,sl,el)
    eh = eh + sl
    FAST_TWO_SUM(sh,eh,sh,eh)
    eh = eh + el
    FAST_TWO_SUM(sh,eh,a.hi,a.lo)
```

a=(a.hi,a.lo), b=(b.hi,b.lo), c=(c.hi,c.lo)

Design of Fast Quad. Operations for Lis (**a Library of Iterative Solvers for linear systems**)

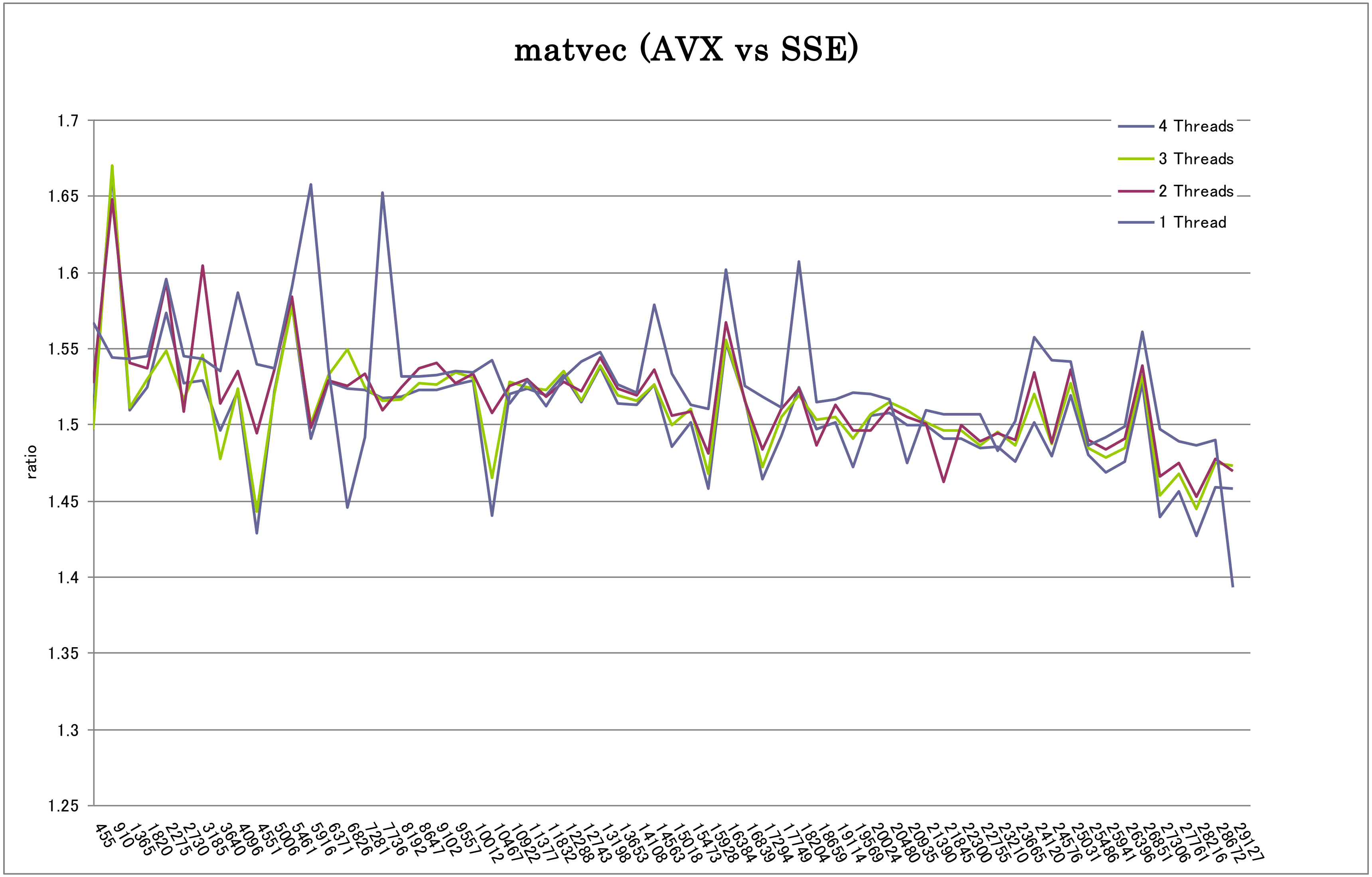
- Same API with Double
- Double: Input（A，b，x_0 ）
- Double: Output
- Double: Creation of Preconditioner M
- Fast Quad.: Iterative solution x
  All working variables
- Fast Quad.: Application of Preconditioner
  Mu=v

Implementation

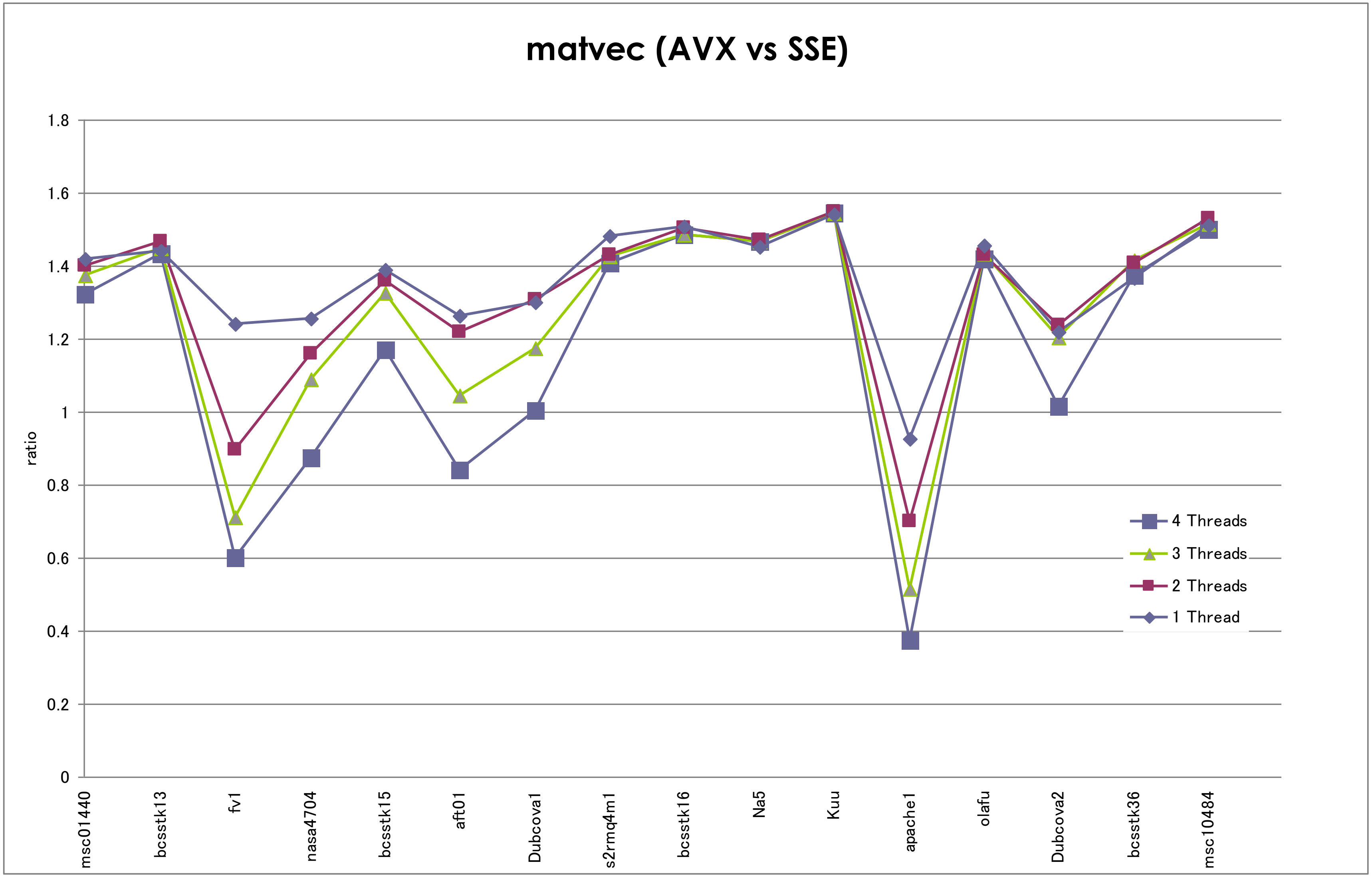
- Replace Double to Fast Quadruple Arith. Op.
  - Matrix-vector product(`matvec`)
  - Inner product(`dot`)
  - Vector operation(`axpy`)
- Use multiply-and-Add for `matvec`, `dot`, `axpy`
  - Reduce memory access, especially store
- Make two Multiply-and-Add functions
  - FMA (Fast Quadruple) for `dot` and `axpy` require 33 double flops
  - FMAD (Double and Fast Quadruple) for `matvec` require 29 double flops

Acceleration by SSE2/AVX

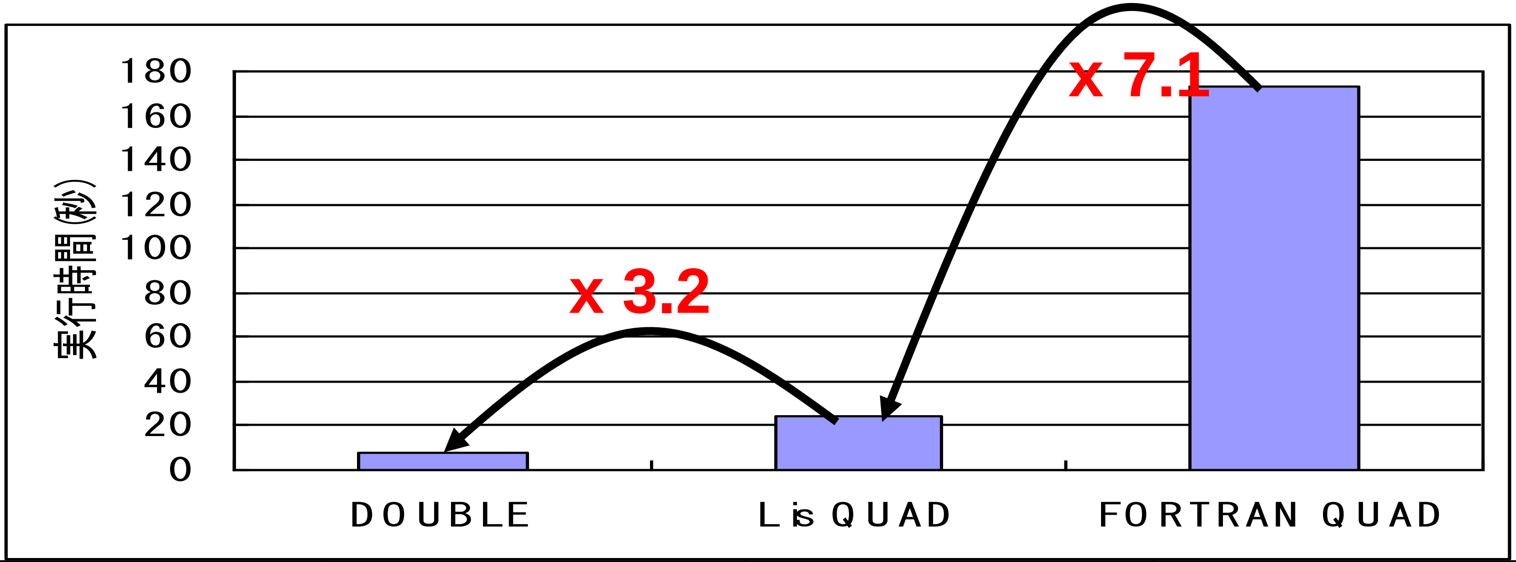
- SSE2 is used for vectors(`dot`, `axpy`, `matvec`)
  - 2 Multiply-and-add in same time
- Two FMA in a loop with loop unrolling
  - pd instruction of SSE2 can be used for all
- Code tuning
  - Alignment
  - Some hand optimization



matvec (AVX vs SSE)
ratio
1.7
1.65
1.6
1.55
1.5
1.45
1.4
1.35
1.3
1.25
4 Threads
3 Threads
2 Threads
1 Thread
455
910
1365
1820
2275
2730
3185
3640
4096
4551
5006
5461
5916
6371
6826
7281
7736
8192
8647
9102
9557
10012
10467
10922
11377
11832
12288
12743
13198
13653
14108
14563
15018
15473
15928
16384
16839
17294
17749
18204
18659
19114
19569
20024
20480
20935
21390
21845
22300
22755
23210
23665
24120
24576
25031
25486
25941
26396
26851
27306
27761
28216
28672
29127

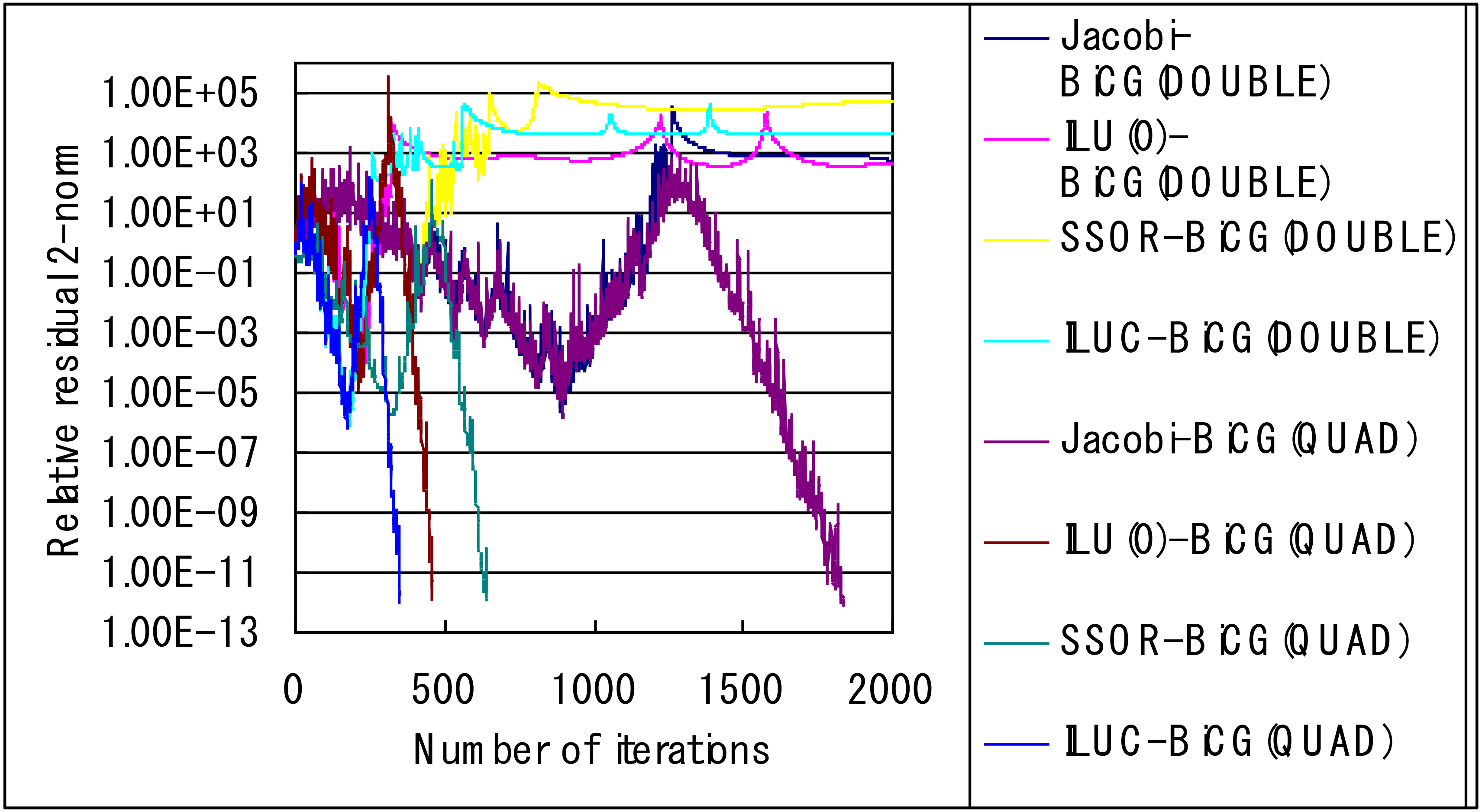

matvec (AVX vs SSE)
ratio
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
msc01440
bcsstk13
fv1
nasa4704
bcsstk15
aft01
Dubcova1
s2rmq4m1
bcsstk16
Na5
Kuu
apache1
olafu
Dubcova2
bcsstk36
msc10484
4 Threads
3 Threads
2 Threads
1 Thread

Time for 50 BiCG Iterations
Poisson（n=10^6, CRS）, Xeon 2.8GHz



	DOUBLE	Lis QUAD	FORTRAN QUAD
Matrix A(CRS)	4(n+nnz)+8nnz	4(n+nnz)+8nnz	4(n+nnz)+16nnz
Vector b	8n	8n	16n
Vector x	8n	16n	16n
Workings	6*8n	6*16n	6*16n
sum	121.9MB	175.8MB	221.6MB

Convergence History of A4 with Preconditioned BiCG



Result of Problem A4

Pre.	Double			Fast Quadruple		
	sec.	iter.	TRR	sec.	iter.	TRR
BiCG						
Jacobi				26.58	1833	7.68E-15
ILU(0)				20.41	460	1.25E-14
SSOR				29.78	642	1.27E-14
ILUC				17.78	350	1.13E-14
GPBiCG						
Jacobi				34.50	1403	6.89E-15
ILU(0)	2.99	407	1.91E-14	18.43	225	1.17E-14
SSOR				42.53	500	1.02E-14
ILUC	11.71	364	1.67E-14	25.95	274	3.05E-15

Mixed Precision Iterative Methods

Combination of Double and Fast Quadruple

Lis QUAD: Fast Quadruple Arithmetic

- Improve convergence! Make robust!
- Excessive quality
- Still Costly

  x 3.2 on Xeon, x 3.1 on Core2 Duo

Reduce computation time

→ Reduce Quadruple Operations

Basic Idea of Restart

- Until Now:

$$(1)\text{Solve } Ax^{*}=b \text{ with some initial value } x_0$$

$$(2)\text{Solve } Ax=b \text{ with an initial value } x^{*}$$

  - In general, (1) and (2) have same spaces, same methods, and same precisions
  - (1) and (2) have same spaces, same methods but **different precisions**
    (combination of Double and Fast Quadruple).

SWITCH Algorithm

- Restart with different precision arithmetic
  - Current solution x_k is passed at the restart
  - Upper and Lower part of Double-Double var. are stored in different arrays
  - Only Upper part is used for Double Precision
  - Two Stages are performed by Different Precisions

```
for(k=0;k<matitr;k++){
    The first step
    if( nrm2<restart_tol ) break;
}
Clear all values except x
for(k=k+1;k<maxtr;k++) {
    The second step
    if( nrm2<tol ) break;
}
```

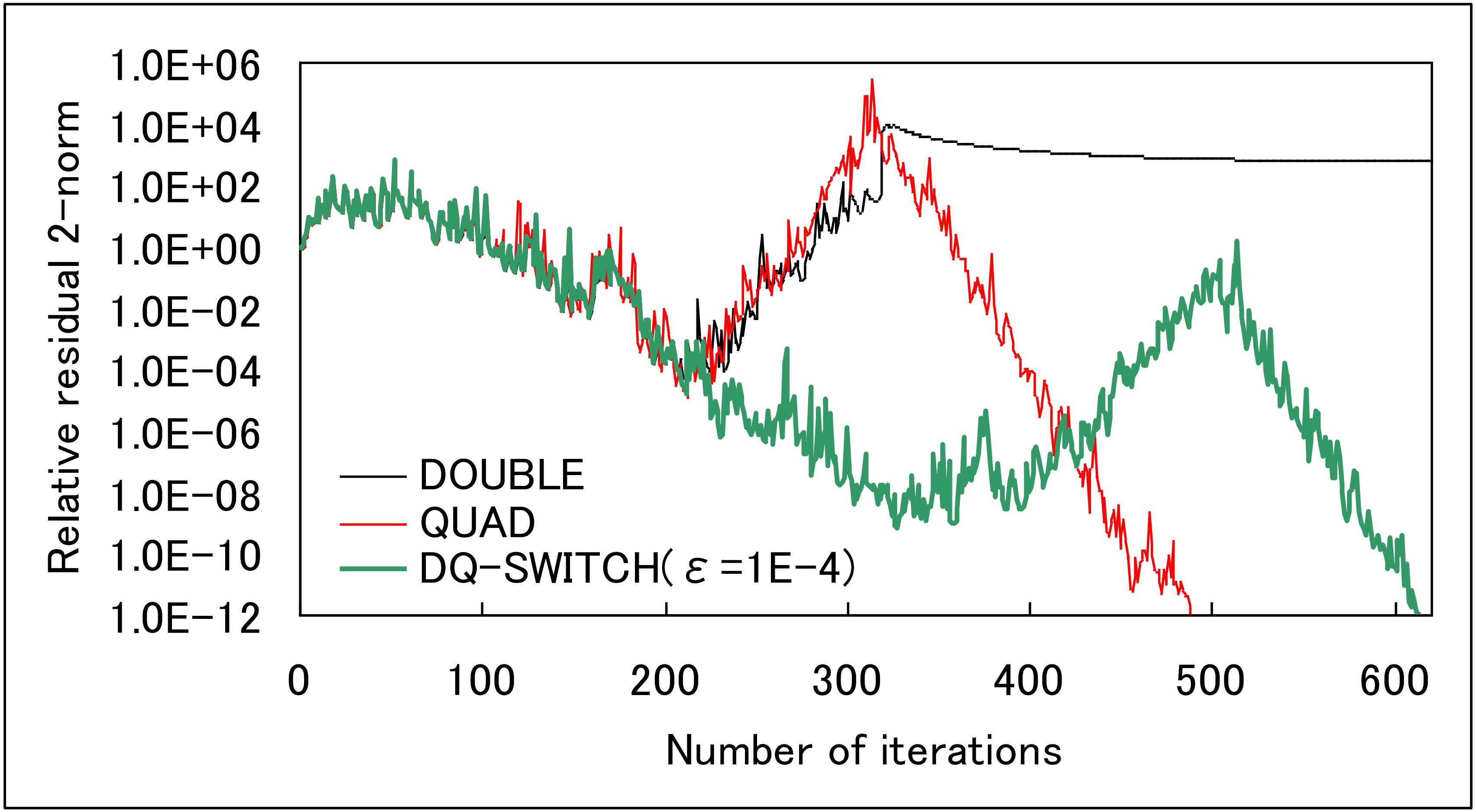
airfoil_2d		iter. total	double	sec.	||b−Ax||
DOUBLE		4567	4567	18.64	3.25E−08
QUAD		3838		69.39	5.36E−10
SWITCH	ε =1.0E−10	4402	4091	24.25	3.15E−10
	ε =1.0E−11	4331	4176	21.66	3.13E−10
	ε =1.0E−12	4709	4567	22.87	3.56E−10

wang3		total	double	sec.	||b−Ax||
DOUBLE		476	476	2.03	3.52E−10
QUAD		372		7.31	1.49E−10
SWITCH	ε =1.0E−10	460	361	3.67	1.59E−10
	ε =1.0E−11	459	444	2.42	9.22E−11
	ε =1.0E−12	479	476	2.32	1.46E−10

language		total	double	sec.	||b−Ax||
DOUBLE		39	39	3.42	2.96E−09
QUAD		36		10.53	4.25E−11
SWITCH	ε =1.0E−10	38	34	4.57	1.71E−10
	ε =1.0E−11	37	35	4.07	4.20E−10
	ε =1.0E−12	40	39	4.18	4.47E−10

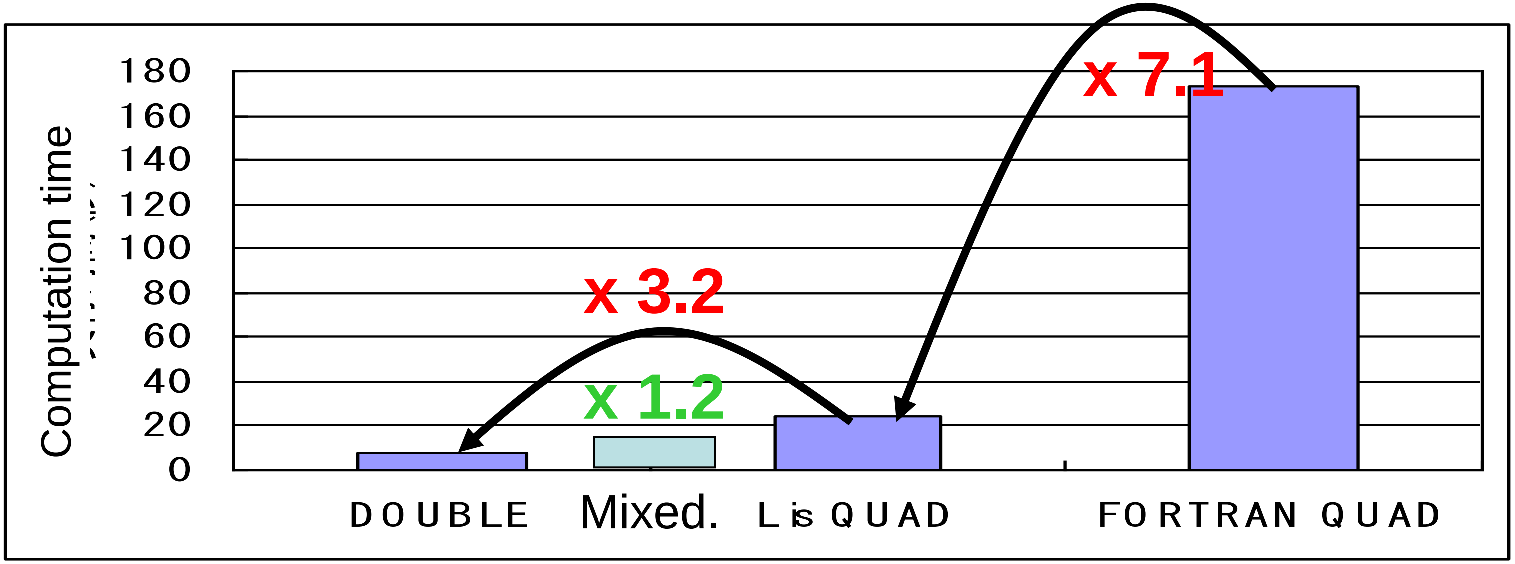
  - ε is restaring criterion of SWITCH
- QUAD and SWITCH improve 2 digits for solution’ quality
- SWITCH is 20% overhead on the double, however robust

Circuit_3, BiCG with ILU(0)



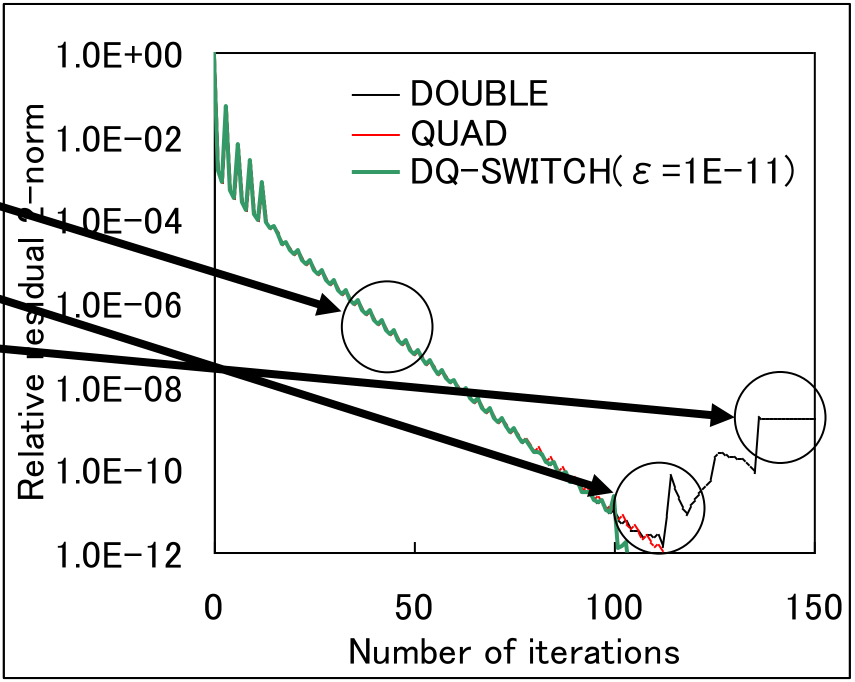
Computation Time

Poisson（n=10^6, CRS）, Xeon 2.8GHz



Auto Restart with Different Precisions

- Convergent history shows three patterns:
  - （C)Converge
  - （D)Diverge
  - （S)Stagnate



- To Detect （D) and (S) restart at the point

Auto Restart of DQ-SWITCH

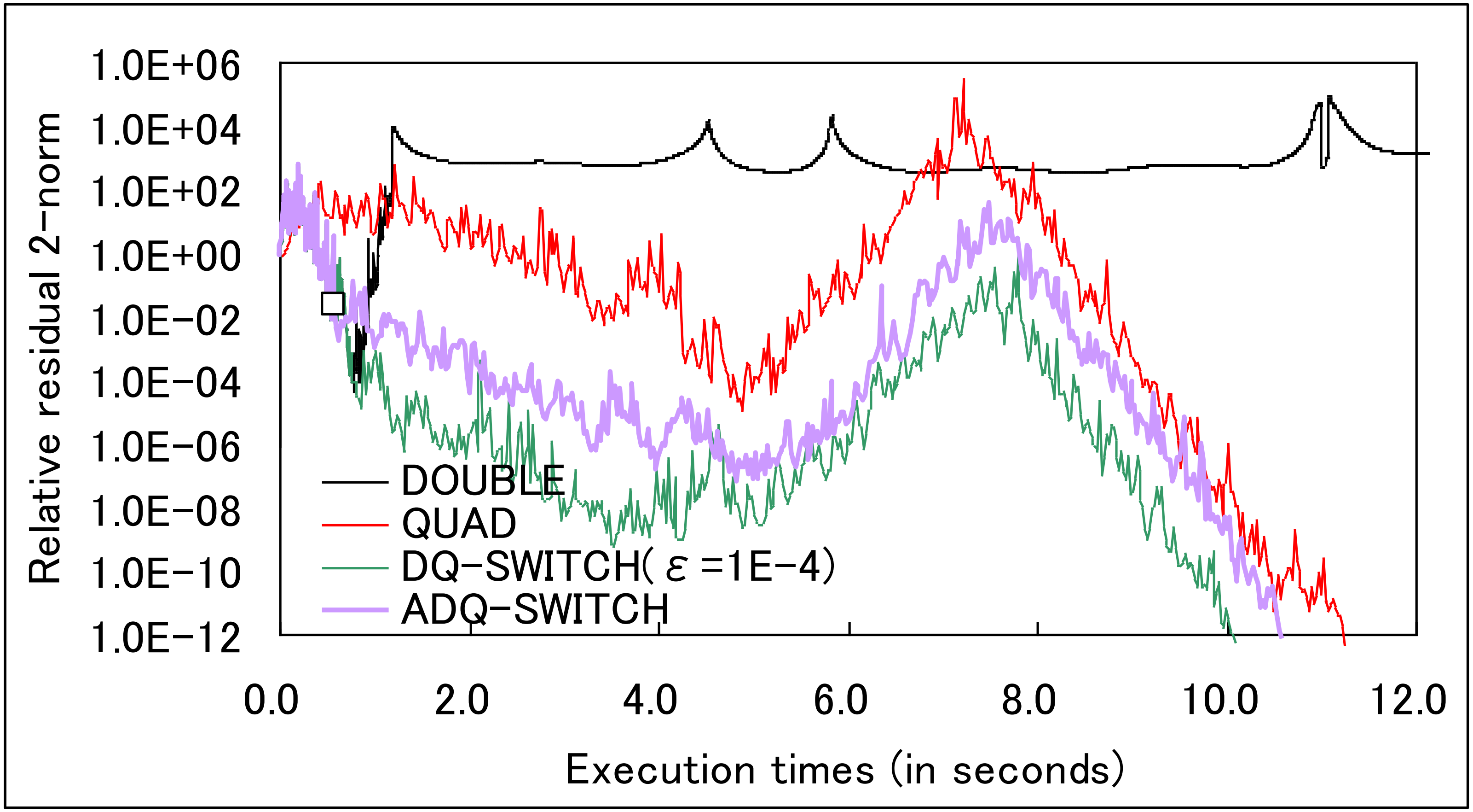
- Compute deviation of residual norm

$$v = \frac{1}{p}\sum_{i=1}^{p}\left(\frac{nrm(i)-nrm(1)}{nrm(1)}\right)^2$$

- (D) $v \geq 10^2$
- (S) $v \leq 10^{-1}$

```
if( nrm2 < nrm2_min )
      nrm2_min = nrm2;   x_bak = x;
nrm_bak[k%10] = nrm2;
if( k>=10 ) {
    v = 0.0;   c = 0;
    for(i=0;i<10;i++) {
      t = nrm_bak[i] - nrm_bak[(k-9)%10];
      t = t / nrm_bak[(k-9)%10];
      v = v + t*t;
      if( nrm_bak[(k-9)%10] <= nrm_bak[i] )
        c = c+1;
    }
    v = v / 10;
    if( v<=0.1 || (c==10 && v>=100) ) break;
    if( nrm2<tol ) break;
}
```

Electoronics BiCG with ILU(0)



- Divergence or Stagnation is detected.
- Computation time is reduced.

Parallel Issues for Fast Quad.

- Depends on the implementation of Ax, A^Tx, $M^{-1}x$, $M^{-T}x$, and Matrix Storage Format
- Data transfered is almost same
- Heavy Computation

→ Suitable for Distributed Parallel

- Less round-off errors

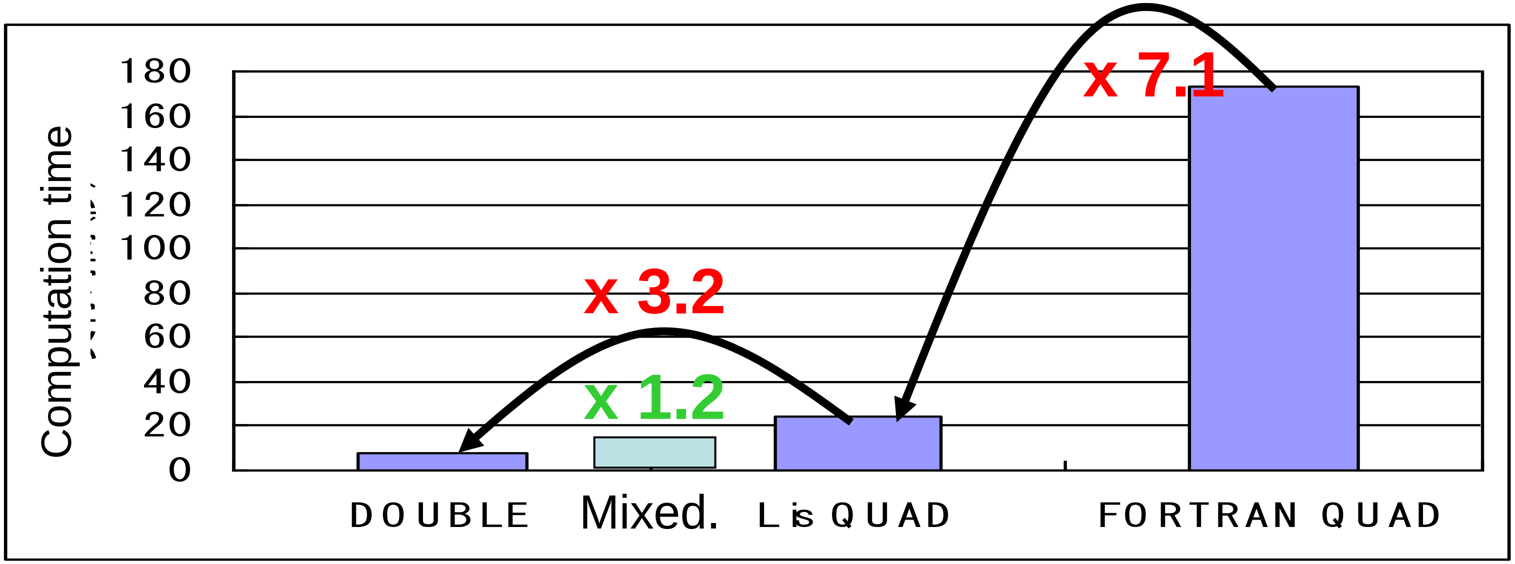
→ lighter preconditioner (easy to parallelize)

50 BiCG Iterations on Distributed Parallel

# of PEs	Double	Fast Quadruple
1	7.56sec	24.21sec
2	3.90sec(1.93)	12.22sec(1.98)
4	2.02sec(3.74)	6.23sec(3.88)
8	1.11sec(6.87)	3.18sec(7.61)

Computation Time

Poisson（n=10^6, CRS）, Xeon 2.8GHz



Parallel

4 8 4 8

***3.74 *6.84 *3.88 *7.61**

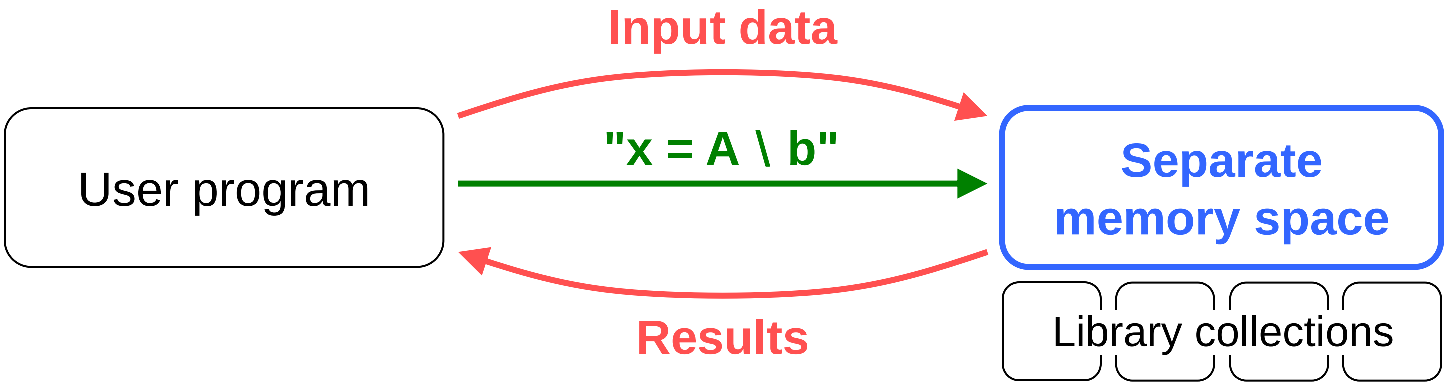
ICL lunch talk at March 16, 2012

Conclusion

- Fast Quadruple Arithmetic Operations using SSE2
  - Reduce round-off errors
  - Computation time is about 3.2 times of Double
- Mixed Precision Iterative Methods
  - Combine Double and Fast Quadruple
  - DQ-SWITCH is faster for complecated problems (Double does not converge!)
  - DQ-SWITCH is robust but costly for ordinary problems
  - Overhead is 20%
  - Automatic Restart
- These methods fit for Parallel computing environments.
- Lighter preconditioners should be effective for parallel environments

SILC: Simple Interface for Library Collections

- Basic ideas
  - **Data transfer** and **a request for computation**
  - **Mathematical expressions** for the request
  - **A separate memory space** for the computation



Solving a system of linear equations

$$A\boldsymbol{x}=\boldsymbol{b}$$

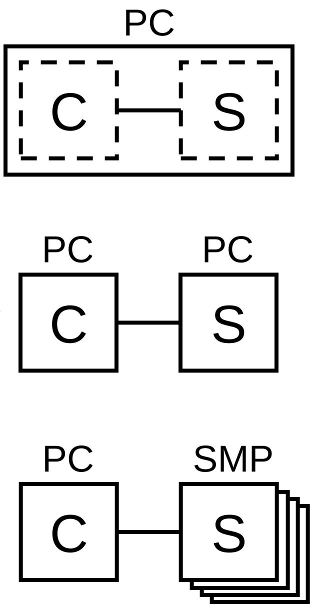
- In the traditional way (using LAPACK in C)

```
double *A, *b;
int kl, ku, lda, ldb, nrhs, info, *ipiv;

dgbtrf (N, N, kl, ku, A, lda, ipiv, &info);  /* LU factorization */
if (info == 0)
    dgbtrs ('N', N, kl, ku, nrhs, A, lda, ipiv, b, ldb, &info);  /* solve */
```

- In SILC

```
silc_envelope_t A, b, x;

SILC_PUT ("A", &A);
SILC_PUT ("b", &b);
SILC_EXEC ("x = A \ b");  /* call a solver (e.g., dgbtrf & dgbtrs) */
SILC_GET (&x, "x");
```

Main benefits of using SILC

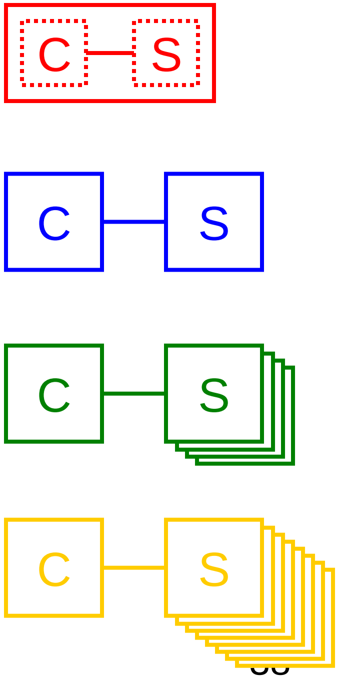


- Source-level independence between user programs and matrix computation libraries
  - Easy access to alternative solvers and matrix storage formats, possibly in other libraries
  - Instant porting to other computing environments without any modification in user programs
- You need to prepare only the smallest amount of data
  - Temporary buffers are automatically allocated
- Language-independent mathematical expressions
  - Applicable in many programming languages (C, Fortran, Python, MATLAB)

SILC servers in different computing environments

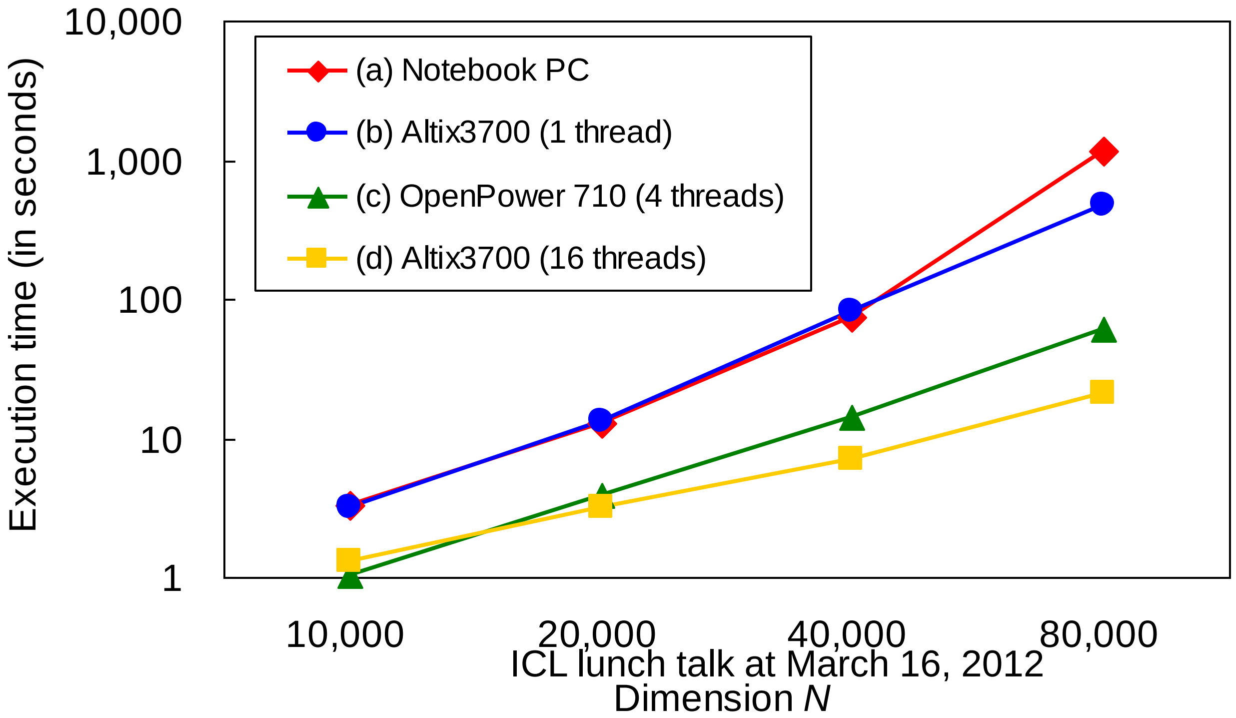
- A user program (client) that solves $\boldsymbol{A x} = \boldsymbol{b}$
  - Where A is a tridiagonal matrix in the CRS format
  - Run in the notebook PC of Environment (a)
  - In a 100-Base TX local-area network

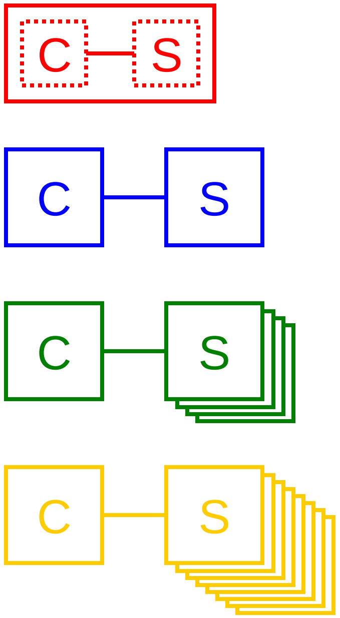
	Environment	Specification	OpenMP
(a)	A notebook PC	Intel Pentium M 733 1.1GHz, 768MB memory, Fedora Core 3	N/A
(b)	SGI Altix3700	Intel Itanium2 1.3GHz × 32, 32GB memory, Red Hat Linux Advanced Server 2.1	1 thread
(c)	IBM eServer OpenPower 710	IBM Power5 1.65GHz × 2 (4 logical CPUs), 1GB memory, SuSE Linux Enterprise Server 9	4 threads
(d)	SGI Altix3700	Same as (b)	16 threads



Experimental results

- About 0.1 second of data communications over the LAN
  - Data size: 0.46MB (N=10,000) to 4.27MB (N=80,000)
- SILC servers in (c) and (d) achieved better performance because of parallel computation





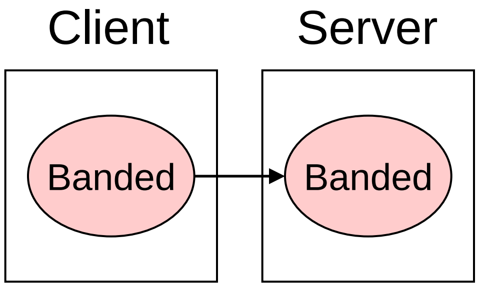
Functionalities of SILC

- Data structures
  - Data types: scalar, vector, matrix, cubic array
  - Precisions: integer, real, complex (single or double)
  - Matrix storage formats: dense, banded, CRS
- Mathematical expressions
  - Binary arithmetic operators (+, −, *, /, %)
  - Solutions of systems of linear equations (A\b)
  - Conjugate transposes (A'), complex conjugates (A~)
  - Built-in functions
    - Ex. "sqrt(b' * b)" is the 2-norm of vector b
  - Subscript
    - Ex. "A[1:5,1:5]" is a 5×5 submatrix of A

Modes for using LAPACK in SILC

Mode (A)

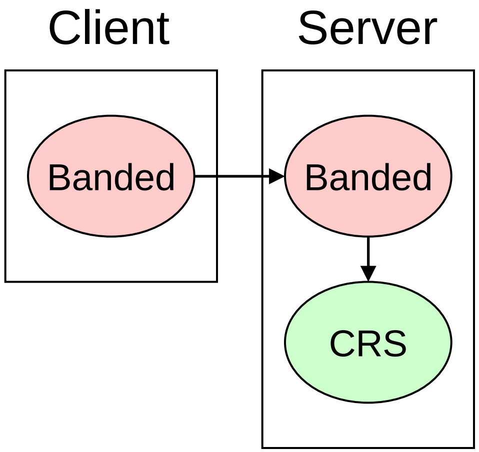
- Both data transfer and computation with **LAPACK**



Solved with
dgbtrf & dgbtrs

Mode (B)

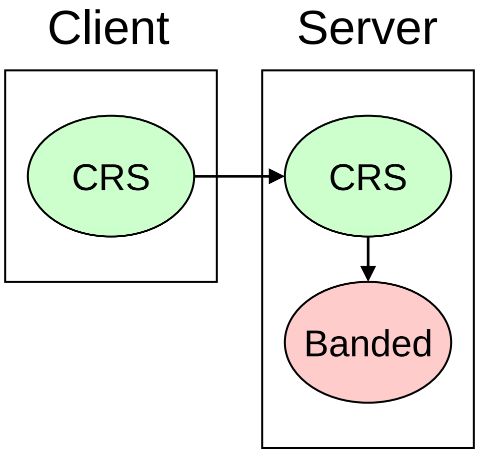
- Data transfer with **LAPACK**
- Computation with **another library**



Solved with
the CG method

Mode (C)

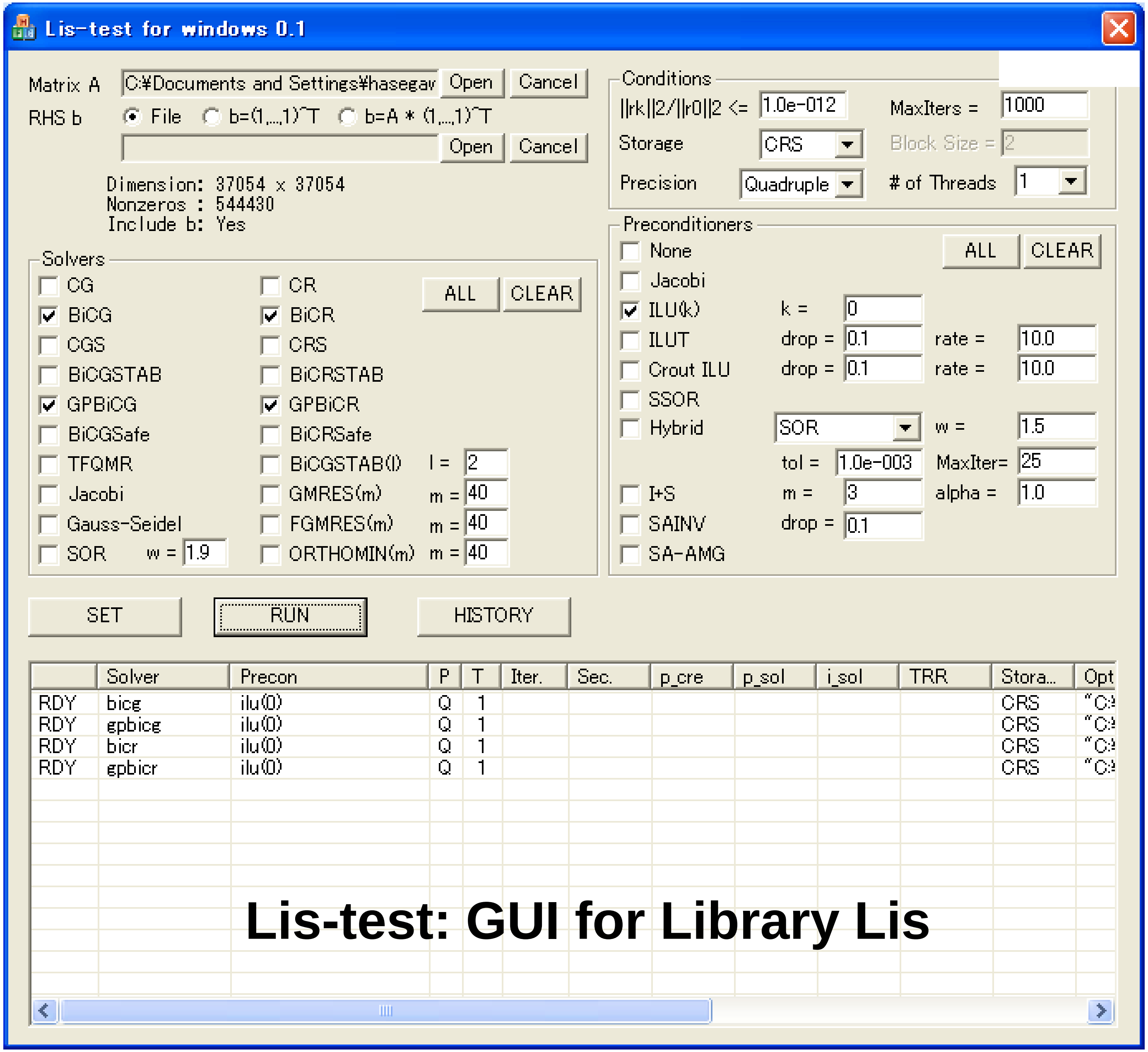
- Data transfer with **another library**
- Computation with **LAPACK**



Solved with
dgbtrf & dgbtrs

Lis-test for evaluation

- Over 2K combinations:
  10 Preconditioners x 13 Solvers x 11 Storage formats x 2 precisions
- Run on Windows from USB not to install.
- Prepare Matrix data as text file with Matrix Market' exchange format
- May use data located in Web page
- Run in parallel if the PC has multi-cores
- To click, solutions, history, etc are computed



Matrix A: C:¥Documents and Settings¥hasegaw

RHS b: ● File ○ b=(1,...,1)^T ○ b=A * (1,...,1)^T

Dimension: 37054 x 37054
Nonzeros : 544430
Include b: Yes

Conditions
||rk||2/||r0||2 <= 1.0e-012
MaxIters = 1000
Storage: CRS
Block Size = 2
Precision: Quadruple
# of Threads: 1

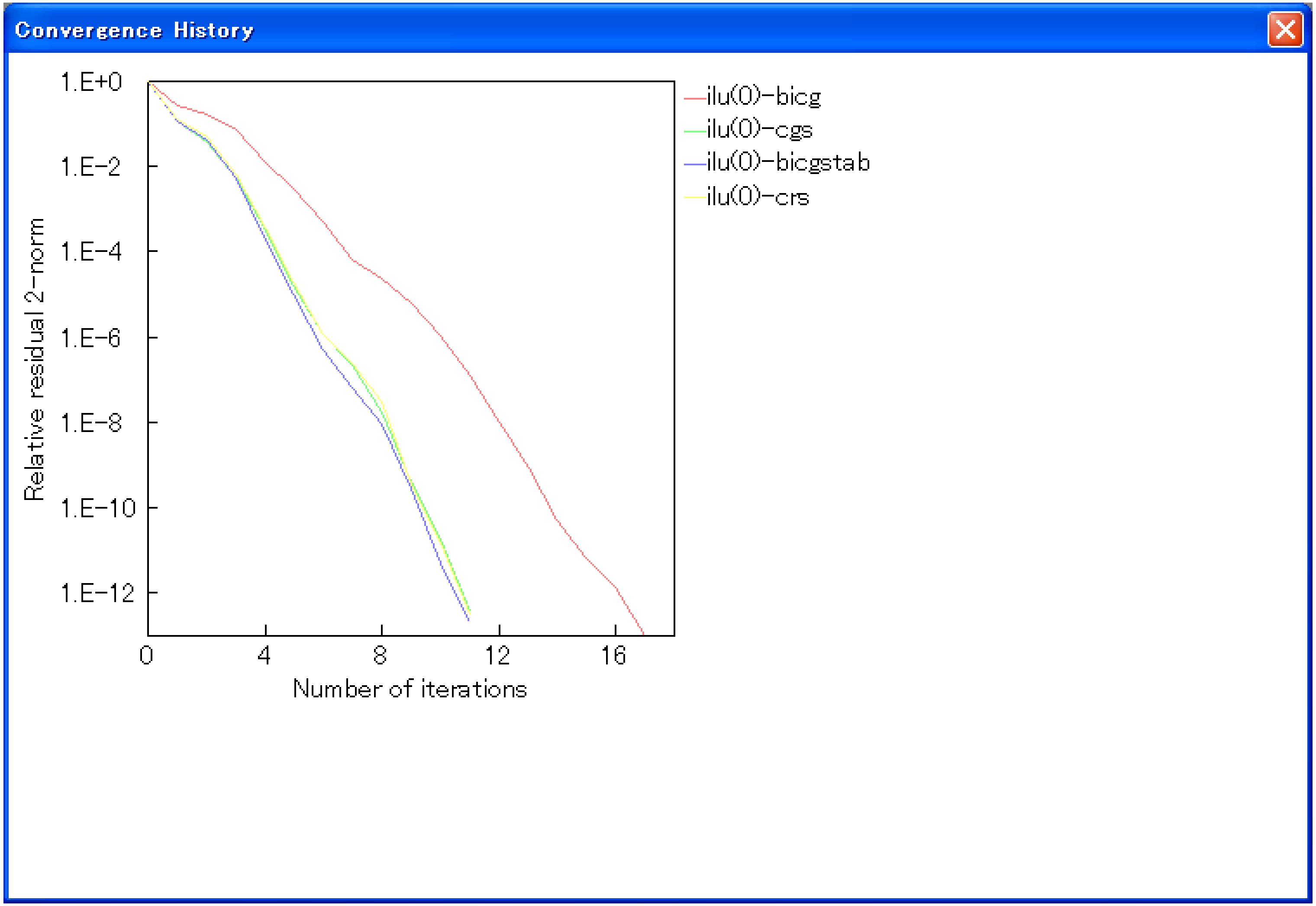
Solvers
☐ CG
☑ BiCG
☐ CGS
☐ BiCGSTAB
☑ GPBiCG
☐ BiCGSafe
☐ TFQMR
☐ Jacobi
☐ Gauss-Seidel
☐ SOR w = 1.9
☐ CR
☑ BiCR
☐ CRS
☐ BiCRSTAB
☑ GPBiCR
☐ BiCRSafe
☐ BiCGSTAB(l) l = 2
☐ GMRES(m) m = 40
☐ FGMRES(m) m = 40
☐ ORTHOMIN(m) m = 40

Preconditioners
☐ None
☐ Jacobi
☑ ILU(k) k = 0
☐ ILUT drop = 0.1 rate = 10.0
☐ Crout ILU drop = 0.1 rate = 10.0
☐ SSOR
☐ Hybrid SOR w = 1.5
tol = 1.0e-003 MaxIter= 25
☐ I+S m = 3 alpha = 1.0
☐ SAINV drop = 0.1
☐ SA-AMG

	Solver	Precon	P	T	Iter.	Sec.	p_cre	p_sol	i_sol	TRR	Stora...	Opt
RDY	bicg	ilu(0)	Q	1							CRS	"C:
RDY	gpbicg	ilu(0)	Q	1							CRS	"C:
RDY	bicr	ilu(0)	Q	1							CRS	"C:
RDY	gpbicr	ilu(0)	Q	1							CRS	"C:

Lis-test: GUI for Library Lis

Comparison is done easily!



Features of Lis:

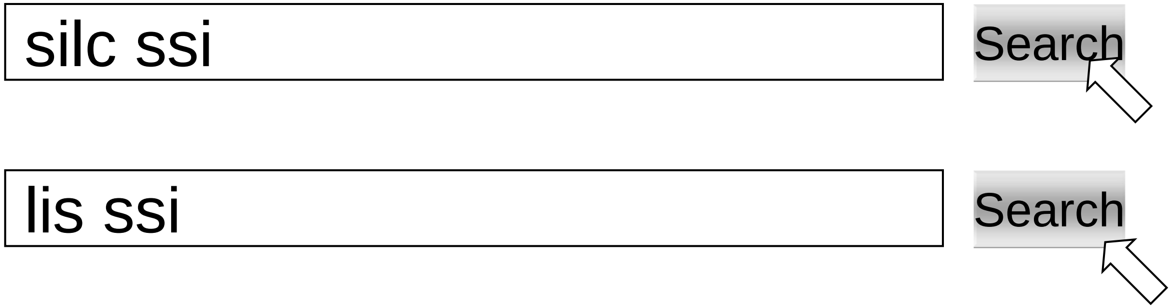
a Library of Iterative Solvers for linear systems

- Over 2K combinations:

  10 Preconditioners x 13 Solvers x 11 Storage formats x 2 precisions
- 4 computing environments
  - Serial
  - OpenMP for Shared memory
  - MPI for Distributed memory
  - Hybrid of OpenMP and MPI
- Fast quadruple arithmetic operations
- Same interface with Double/Quadruple

To get code and more

Visit the SILC and Lis home pages:



silc ssi

lis ssi

Collaborators and Acknowledgement

- Lis & Lis-test
  H. Kotakemori
  A. Fujii (Kogakuin U)
  K. Nakajima (U Tokyo)
  A. Nishida(U Kyushu)

- Tuning on AVX
  K. Asakawa (kogakuin U)
  A. Fujii (Kogakuin U)
  T. Tanaka (Kogakuin U)

- SILC
  T. Kajiyama (Universidade Nova de Lisboa)
  A. Nukada (TITECH)
  R. Suda (U Tokyo)
  A. Nishida (U Kyushu)

- Lis and SILC are parts of SSI project funded by JST/CREST

Thank you!

ありがとうございました。