A Mixed Precision Iterative Method with Fast Quadruple Precision Arithmetic Operation tuned by SSE2

Hisashi Kotakemori (TCAD International, Inc.)
Hidehiko Hasegawa (U. of Tsukuba)
Motivation

• Krylov methods converge at most $n$ iterations ($n$: dimension of Matrix) in exact computation.

• Diverge, stagnation, and many iterations occur because of round-off errors.

• High Accuracy computation is one choice, however it is very costly.
  – Real *16 of Fortran: Double of Memory 20 times of Computation time.
Story

1) Fast Quadruple Arithmetic Operations
   - Not standards
   - double-double: use 2 double floating numbers
   - Accelerate computation with SSE2
   - Computation time: 3.2 times of Double

2) Mixed Precision Iterative method
   - Reduce Computation time with less Quadruple Arithmetic Operations
   - Restart with different precision
   - Computation time: 1.2 times of Double
Story (continued)

3) Auto restart strategy
   – Automatically changes precision

4) Parallelization
   – Almost same Data transfer
   – Much more computations (20 times)
   – Less round-off errors
   – → light preconditioner (easy to parallelize)

5) Provide Library: Lis and GUI: Lis-test
Computation time
Poisson (n=10^6, CRS), Xeon 2.8GHz

Parallel

4 8
*3.74 *6.84

4 8
*3.88 *7.61
Implementation of Fast Quadruple Arithmetic Operations
Problem

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

\[
\begin{pmatrix}
2 & 1 \\
0 & 2 & 1 \\
1.4 & 0 & 2 & 1 \\
\vdots & \vdots & \vdots & \vdots \\
1.4 & 0 & 2 & 1 \\
1.4 & 0 & 2 \\
\end{pmatrix}
\]

\(n: 10^5\)
Answer

• To improve convergence, High-precision arithmetic operation is effective.
• However it is costly. real *16 of Fortran:
  - memory: Double
  - comp. 20 times

Fast Quadruple is necessary
Round-off error free
Double Arithmetic Addition

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

\[ a + b = \text{fl}(a + b) + \text{err}(a + b) \]

- \( a,b \): double floating point variables
- \( \text{fl}(a + b) \): addition of \( a \) and \( b \) in double
- \( \text{err}(a+b) : (a+b) – \text{fl}(a+b) \): error
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 bit less than IEEE standards
  - Effective digits is approx. 31 vs. 33 digits.

| exponent 11 bit | Mantissa 52 bit | + | Exponent 11 bit | Mantissa 52 bit |
|-----------------|-----------------|+|-----------------|-----------------|
| IEEE Standards  |                 |   |                 |                 |
| exponent 15bit  | Mantissa 112bit |   |                 |                 |
Basic Algorithm

- Dekker showed round-off error free addition in double precision.

\[ |a| \geq |b| \quad 3\text{flops.} \quad \text{Others} \quad 6\text{flops.} \]

\[
\begin{align*}
\text{FAST\_TWO\_SUM}(a, b, s, e) & \quad s = a + b \\
& \quad e = b - (s - a)
\end{align*}
\]

\[
\begin{align*}
\text{TWO\_SUM}(a, b, s, e) & \quad s = a + b \\
& \quad v = s - a \\
& \quad e = (a - (s - v)) + (b - v)
\end{align*}
\]
Quadruple Addition of a=b+c

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 $a=b+c$

\[
\begin{align*}
\text{ADD}(a, b, c) & \quad \text{20 flops} \\
\text{TWO\_SUM}(b.\text{hi}, c.\text{hi}, \text{sh}, \text{eh}) \\
\text{TWO\_SUM}(b.\text{lo}, c.\text{lo}, \text{sl}, \text{el}) \\
\text{eh} &= \text{eh} + \text{sl} \\
\text{FAST\_TWO\_SUM}(\text{sh}, \text{eh}, \text{sh}, \text{eh}) \\
\text{eh} &= \text{eh} + \text{el} \\
\text{FAST\_TWO\_SUM}(\text{sh}, \text{eh}, a.\text{hi}, a.\text{lo})
\end{align*}
\]

\[
a = (a.\text{hi}, a.\text{lo}), \quad b = (b.\text{hi}, b.\text{lo}), \quad c = (c.\text{hi}, c.\text{lo})
\]
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
- Fast Quad.: iterative solution \(x\), All working variables
- Fast Quad.: Applying Preconditioner \(Mu=v\)
Implementation

  – Matrix-vector product (matvec)
  – Inner product (dot)
  – Vector operations (axpy)

• Use multiply-and-Add for matvec, dot, axpy
  – Reduce memory access, especially store

• Make two Multiply-and-Add functions
  – FMA: Operation with Fast Quadruple variables
    • Used for Dot and axpy
  – FMAD: Operation with Double and Fast Quadruple
    • Used for matvec
Implementation FMA • FMAD

- FMA $a = a + b \times c$  
  
  33 flops

```c
FMA(a,b,c) {
    TWO_PROD(b.hi,c.hi,p1,p2)
    p2 = p2 + (b.hi * c.lo)
    p2 = p2 + (b.lo * c.hi)
    FAST_TWO_SUM(p1,p2,p1,p2)
    TWO_SUM(a.hi,p1,sh,eh)
    TWO_SUM(a.lo,p2,sl,el)
    eh = eh + sl
    FAST_TWO_SUM(sh,eh,sh,eh)
    eh = eh + el
    FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```

- FMAD $a = a + b \times c$  
  
  29 flops

```c
FMAD(a,b,c) {
    TWO_PROD(b,c.hi,p1,p2)
    p2 = p2 + (b * c.lo)
    FAST_TWO_SUM(p1,p2,p1,p2)
    TWO_SUM(a.hi,p1,sh,eh)
    TWO_SUM(a.lo,p2,sl,el)
    eh = eh + sl
    FAST_TWO_SUM(sh,eh,sh,eh)
    eh = eh + el
    FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```
Acceleration by SSE2

• SSE2 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 to all

• Code tuning
  – Alignment
  – Some Hand optimization
Speed test

• Machine

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Xeon 2.8GHz</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.4.20smp</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel C++ 7.0, Intel Fortran 9.0</td>
</tr>
</tbody>
</table>

• Compiler options
- Optimization is -O3
- -mp option used for source file of FMA
- -xW used for auto-vectorization
Time for 50 BiCG iterations
Poisson (n=10^6, CRS), Xeon 2.8GHz

<table>
<thead>
<tr>
<th></th>
<th>DOUBLE</th>
<th>Lis QUAD</th>
<th>FORTRAN QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix A(CRS)</td>
<td>4(n+nnz)+8nnz</td>
<td>4(n+nnz)+8nnz</td>
<td>4(n+nnz)+16nnz</td>
</tr>
<tr>
<td>Vector b</td>
<td>8n</td>
<td>8n</td>
<td>16n</td>
</tr>
<tr>
<td>Vector x</td>
<td>8n</td>
<td>16n</td>
<td>16n</td>
</tr>
<tr>
<td>Workings</td>
<td>6*8n</td>
<td>6*16n</td>
<td>6*16n</td>
</tr>
<tr>
<td>sum</td>
<td>121.9MB</td>
<td>175.8MB</td>
<td>221.6MB</td>
</tr>
</tbody>
</table>
Time for 50 BiCG iterations
Poisson (n=10^6, CRS), Core2 Duo 2.4GHz

- Double : Fortran Quad is 1 : 9.9.
  - Intel C/C++ 9.1, Intel Fortran 9.1, -O3 -xW
Confirming Convergence

- BiCG
- Right hand size \( b \)
  - Solution \( x = (1,\ldots,1)^T \)
- Initial solution \( x_0 = (0,\ldots,0)^T \)
- Convergence criteria \( \frac{\|r_{k+1}\|_2}{\|r_0\|_2} \leq 10^{-12} \)
- Max iterations = 10^4
Test Problems

• Poisson
  – 2 dimension, FDM
  – \( N=10^6, \text{nnz}=5\times10^6 \)

• rdb2048l (Chemical engineering)
  – MatrixMarket, \( n=2048, \text{nnz}=12032, \text{cond} = 1.8\times10^3 \)

• olm1000 (Hydrodynamics)
  – MatrixMarket, \( n=1000, \text{nnz}=3996, \text{cond} = 3\times10^6 \)

• A4 (Electronic potential)
  – \( n=23,994, \text{nnz}=214,060 \)

• Cryg10000 (CRYSTAL GROWTH EIGENMODES)
  – UF Sparse Matrix Collection, \( n=10000, \text{nnz}=49699 \)

• circuit_3 (Circuit Simulation)
  – \( n=12,127, \text{nnz}=48,137 \)
Comparison of Real *16 vs. Fast Quadruple with BiCG

- same accuracy = same number of iterations (Difference is at most 10%)!
Convergence History of A4 with Preconditioned BiCG
## Result of Problem A4

<table>
<thead>
<tr>
<th>Pre.</th>
<th>Double</th>
<th>Fast Quadruple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec.</td>
<td>iter.</td>
</tr>
<tr>
<td>BiCG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacobi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILU(0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILUC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPBiCG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacobi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILU(0)</td>
<td>2.99</td>
<td>407</td>
</tr>
<tr>
<td>SSOR</td>
<td>42.53</td>
<td>500</td>
</tr>
<tr>
<td>ILUC</td>
<td>11.71</td>
<td>364</td>
</tr>
</tbody>
</table>
cryg10000 BiCG with ILU(0)

![Graph showing relative residual 2-norm versus number of iterations for different ILUC-BiCG and ILUC-GPBiCG methods.](image)
cryg10000 BiCG with Jacobi
Mixed Precision Iterative Methods
Combination of Double and Fast Quadruple
Mixed Precision Iterative methods

Lis QUAD: Fast Quadruple Arithmetic

• Improve convergence! Make robust?!
• Over quality
• Still Costly
  \( \times 3.2 \) on Xeon, \( \times 3.1 \) Core2 Duo

Reduce computation time
  \( \rightarrow \) Reduce Quadruple Operations
Basic idea of restart

• Until Now:

(1) Solve $Ax^* = b$ with some initial value $x_0$

(2) Solve $Ax = b$ 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 precision (combination of Double and Fast Quadruple).
Mixed Precision Iterative method with combination of double and Fast Quadruple

• Reduction of Fast Quadruple operations
  – Reduction of Computation time

• Two Methods
  – SWITCH Algorithm based on “restart”
    \[ D \rightarrow Q \]
    \[ Q \rightarrow D \]
  – PERIODIC Algorithm based on “correction”
SWITCH algorithm
(D → Q, Q → D)

- 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 Precision

```java
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;
}
```
PERIODIC algorithm

• A Fast Quadruple is used each k iterations
  – All values are passed at the change
  – No cost at the change of $Q \rightarrow D$
  – Lower part is cleared at the change of $D \rightarrow Q$

```c
for(k=0;k<maxitr;k++) {
    if( k%interval<num ) {
        Fast Quadruple is used
    } else {
        Lower part is cleared
        Double is used
    }
}
```
Teoplitz $\gamma = 1.3, n=10^5$

<table>
<thead>
<tr>
<th></th>
<th>iter.</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>double</td>
<td>sec.</td>
</tr>
<tr>
<td>FMA2_SSE2</td>
<td></td>
<td>113</td>
<td>0</td>
<td>6.60</td>
</tr>
<tr>
<td>SWITCH $\epsilon = 1.0E-09$</td>
<td></td>
<td>95</td>
<td>74</td>
<td>2.33</td>
</tr>
<tr>
<td>$\epsilon = 1.0E-10$</td>
<td></td>
<td>95</td>
<td>86</td>
<td>1.82</td>
</tr>
<tr>
<td>$\epsilon = 1.0E-11$</td>
<td></td>
<td>103</td>
<td>100</td>
<td>1.67</td>
</tr>
<tr>
<td>PERIODIC num=1</td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>num=2</td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>num=3</td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>num=4</td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>num=5</td>
<td></td>
<td>107</td>
<td>52</td>
<td>3.98</td>
</tr>
<tr>
<td>num=6</td>
<td></td>
<td>118</td>
<td>46</td>
<td>4.89</td>
</tr>
</tbody>
</table>

- Epsilon is restart criterion of DQ-SWITCH
- Num: Quad. Ops. Used num times per 10 iterations
## Teoplitz $\gamma = 1.4, n=10^5$

<table>
<thead>
<tr>
<th></th>
<th>iter.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td><strong>FMA2_SSE2</strong></td>
<td>155</td>
</tr>
<tr>
<td><strong>SWITCH</strong> $\epsilon = 1.0E-07$</td>
<td>119</td>
</tr>
<tr>
<td>$\epsilon = 1.0E-08$</td>
<td>115</td>
</tr>
<tr>
<td>$\epsilon = 1.0E-09$</td>
<td>–</td>
</tr>
<tr>
<td><strong>PERIODIC num=1</strong></td>
<td>–</td>
</tr>
<tr>
<td>num=2</td>
<td>–</td>
</tr>
<tr>
<td>num=3</td>
<td>–</td>
</tr>
<tr>
<td>num=4</td>
<td>–</td>
</tr>
<tr>
<td>num=5</td>
<td>–</td>
</tr>
<tr>
<td>num=6</td>
<td>–</td>
</tr>
</tbody>
</table>
Convergence History

DQ-SWITCH is good convergence

- $\gamma = 1.3$
- $\gamma = 1.4$

- DQ-SWITCH is good convergence
Epsilon dependency

<table>
<thead>
<tr>
<th>ε</th>
<th>γ=1.3 iter.</th>
<th>γ=1.3 sec</th>
<th>γ=1.4 iter.</th>
<th>γ=1.4 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUAD</td>
<td>113</td>
<td>2.88</td>
<td>155</td>
<td>3.94</td>
</tr>
<tr>
<td>DQ-SWITCH 1.00E-03</td>
<td>114</td>
<td>2</td>
<td>112</td>
<td>2.87</td>
</tr>
<tr>
<td>1.00E-04</td>
<td>109</td>
<td>11</td>
<td>98</td>
<td>2.59</td>
</tr>
<tr>
<td>1.00E-05</td>
<td>105</td>
<td>23</td>
<td>82</td>
<td>2.26</td>
</tr>
<tr>
<td>1.00E-06</td>
<td>104</td>
<td>35</td>
<td>69</td>
<td>2.01</td>
</tr>
<tr>
<td>1.00E-07</td>
<td>95</td>
<td>47</td>
<td>48</td>
<td>1.58</td>
</tr>
<tr>
<td>1.00E-08</td>
<td>94</td>
<td>61</td>
<td>33</td>
<td>1.29</td>
</tr>
<tr>
<td>1.00E-09</td>
<td>95</td>
<td>74</td>
<td>21</td>
<td>1.08</td>
</tr>
<tr>
<td>1.00E-10</td>
<td>95</td>
<td>86</td>
<td>9</td>
<td>0.86</td>
</tr>
<tr>
<td>1.00E-11</td>
<td>103</td>
<td>98</td>
<td>5</td>
<td>0.84</td>
</tr>
</tbody>
</table>

- Choice of appropriate epsilon is important
- Small epsilon reduces much computation time
- Smaller epsilon makes divergence
### D → Q vs. Q → D Switch, BiCG

- **Choice of Appropriate epsilon is necessary**
- **DQ-SWITCH has wide convergent area**

<table>
<thead>
<tr>
<th></th>
<th>$\gamma=1.3$</th>
<th>iter.</th>
<th>total</th>
<th>double</th>
<th>quad</th>
<th>sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QUAD</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>iter.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUAD</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ-SWITCH</td>
<td>1.0E-05</td>
<td>105</td>
<td>23</td>
<td>82</td>
<td>2.26</td>
<td>2.88</td>
</tr>
<tr>
<td></td>
<td>1.0E-06</td>
<td>104</td>
<td>35</td>
<td>69</td>
<td>2.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-07</td>
<td>95</td>
<td>47</td>
<td>48</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-08</td>
<td>94</td>
<td>61</td>
<td>33</td>
<td>1.29</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-09</td>
<td>95</td>
<td>74</td>
<td>21</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-10</td>
<td>95</td>
<td>86</td>
<td>9</td>
<td>0.86</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-11</td>
<td>103</td>
<td>98</td>
<td>5</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>QD-SWITCH</td>
<td>1.0E-05</td>
<td>104</td>
<td>69</td>
<td>35</td>
<td>1.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-06</td>
<td>100</td>
<td>39</td>
<td>61</td>
<td>1.85</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-07</td>
<td>95</td>
<td>48</td>
<td>47</td>
<td>1.56</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-08</td>
<td>100</td>
<td>39</td>
<td>61</td>
<td>1.85</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-09</td>
<td>95</td>
<td>21</td>
<td>74</td>
<td>2.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-10</td>
<td>95</td>
<td>9</td>
<td>86</td>
<td>2.27</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0E-11</td>
<td>103</td>
<td>3</td>
<td>100</td>
<td>2.58</td>
<td></td>
</tr>
</tbody>
</table>
Convergence History

- Decision of restart in QD-SWITCH is difficult.
Comparison of Double and DQ-SWITCH

• University of Florida Sparse Matrix Collection

<table>
<thead>
<tr>
<th>Matrix</th>
<th>dimension</th>
<th>nnz</th>
<th>Size of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>airfoil_2d</td>
<td>14,214</td>
<td>259,688</td>
<td>3.9MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.7MB</td>
</tr>
<tr>
<td>wang3</td>
<td>26,064</td>
<td>177,168</td>
<td>3.7MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.1MB</td>
</tr>
<tr>
<td>language</td>
<td>399,130</td>
<td>1,216,334</td>
<td>39.8MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>61.1MB</td>
</tr>
<tr>
<td></td>
<td>iter.</td>
<td>double</td>
<td>sec.</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>airfoil_2d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
<td>4567</td>
<td>4567</td>
</tr>
<tr>
<td>QUAD</td>
<td></td>
<td>3838</td>
<td>69.39</td>
</tr>
<tr>
<td>SWITCH ε =1.0E−10</td>
<td></td>
<td>4402</td>
<td>4091</td>
</tr>
<tr>
<td>ε =1.0E−11</td>
<td></td>
<td>4331</td>
<td>4176</td>
</tr>
<tr>
<td>ε =1.0E−12</td>
<td></td>
<td>4709</td>
<td>4567</td>
</tr>
<tr>
<td>wang3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
<td>476</td>
<td>476</td>
</tr>
<tr>
<td>QUAD</td>
<td></td>
<td>372</td>
<td>7.31</td>
</tr>
<tr>
<td>SWITCH ε =1.0E−10</td>
<td></td>
<td>460</td>
<td>361</td>
</tr>
<tr>
<td>ε =1.0E−11</td>
<td></td>
<td>459</td>
<td>444</td>
</tr>
<tr>
<td>ε =1.0E−12</td>
<td></td>
<td>479</td>
<td>476</td>
</tr>
<tr>
<td>language</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>QUAD</td>
<td></td>
<td>36</td>
<td>10.53</td>
</tr>
<tr>
<td>SWITCH ε =1.0E−10</td>
<td></td>
<td>38</td>
<td>34</td>
</tr>
<tr>
<td>ε =1.0E−11</td>
<td></td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td>ε =1.0E−12</td>
<td></td>
<td>40</td>
<td>39</td>
</tr>
</tbody>
</table>

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

- Double
- Quad
- Switch ($\varepsilon = 1.0 \times 10^{-10}$)
- Switch ($\varepsilon = 1.0 \times 10^{-12}$)
For BiCG with ILU(0)

### Toeplitz $\gamma = 1.4$

| switch_tol | precon. | iter. | double | sec. | $||b-Ax||$ |
|------------|---------|-------|--------|------|-----------|
| DOUBLE     | ILU(0)  | 51    | 23     | 1.45767 | 1.39E-10 |
| QUAD       | ILU(0)  | 49    | 29     | 1.17805 | 2.41E-10 |
| SWITCH     | 1.00E-06| ILU(0) | 51    | 39     | 0.94101  | 2.16E-10 |
|            | 1.00E-07| ILU(0) | 49    | 29     | 1.17805  | 2.41E-10 |
|            | 1.00E-08| ILU(0) | 51    | 39     | 0.94101  | 2.16E-10 |

### language

| switch_tol | precon. | iter. | double | sec. | $||b-Ax||$ |
|------------|---------|-------|--------|------|-----------|
| DOUBLE     | ILU(0)  | 9     | 9      | 0.67325 | 6.53E-10 |
| QUAD       | ILU(0)  | 9     | 1.82325| 4.27E-11|
| SWITCH     | 1.00E-12| ILU(0) | 10   | 9      | 0.92794  | 3.14E-11 |
A4: electronics effect

![Graph showing relative residual 2-norm against number of iterations for DOUBLE, QUAD, and DQ-SWITCH(ε=1E-4).]
Circuit_3, BiCG with ILU(0)
Mixed Precision Iterative Methods

- Difficult problems are solved with Mixed or QUAD.
- Overhead of the mixed precision iterative methods is 20%
- PERIODIC is NO Good
  - Not effective
- SWITCH is Good
  - Maximum speed-up is 3.9 times compared Lis QUAD.
  - Small epsilon makes more speed up, but dangerous
- DQ-SWITH is the best in these
Auto restart
For Auto restart with Different precisions

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

- To detect (D) and (S)
  - Then, restart at the point

![Graph showing convergence history with different precisions: DOUBLE, QUAD, and DQ-SWITCH(ε = 1E-11).]
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) \( v \leq 10^{-1} \)

```c
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;
}
```
## Toeplitz(1.3)

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$\gamma=1.3$ iter.</th>
<th>total</th>
<th>double</th>
<th>quad</th>
<th>sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUAD</td>
<td></td>
<td></td>
<td>113</td>
<td>2.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ-SWITCH</td>
<td>1.0E-11</td>
<td></td>
<td>103</td>
<td>98</td>
<td>5</td>
<td>0.84</td>
</tr>
<tr>
<td>ADS-SWITCH</td>
<td></td>
<td></td>
<td>123</td>
<td>121</td>
<td>2</td>
<td>1.01</td>
</tr>
</tbody>
</table>

![Relative residual 2-norm vs Number of iterations graph]

- DOUBLE
- QUAD
- ADQ-SWITCH
Electronics BiCG with ILU(0)

- Divergence and Stagnation are detected.
- Computation time is reduced.
Summary: Mixed Precision

• PERIODIC: NG

• SWITCH: Good at least 2 digits with 20% more
  – Q → D: timing of restart is difficult
  – D → Q: easy, robust, however depends on timing of restart

• Auto restart of DQ-SWITH
  – Deviation is used for detecting “Diverge” and “Stagnate”
  – Almost fine for these test problems
Parallel Issues
Parallel Issues for Fast Quad.

- Depends on the implementation of $Ax$, $A^Tx$, $M^{-1}x$, $M^{-T}x$, and Matrix Storage Format
- Data transfer is almost same
- Heavy Computation
  → Suitable for Distributed Parallel
- Less round-off errors
  → light preconditioner (easy to parallelize)
## 50 BiCG iterations on Distributed Parallel

<table>
<thead>
<tr>
<th># of PEs</th>
<th>Double</th>
<th>Fast Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.56sec</td>
<td>24.21sec</td>
</tr>
<tr>
<td>2</td>
<td>3.90sec(1.93)</td>
<td>12.22sec(1.98)</td>
</tr>
<tr>
<td>4</td>
<td>2.02sec(3.74)</td>
<td>6.23sec(3.88)</td>
</tr>
<tr>
<td>8</td>
<td>1.11sec(6.87)</td>
<td>3.18sec(7.61)</td>
</tr>
</tbody>
</table>
Parallel Computing

- Performance depends on problems, implementations, and computing environments

- Sorry! Currently No results for “real” problems
Free Library Lis and GUI Lis-test
Lis-test for evaluation

• Consists of two Windows files
• Not necessary to install. Run from USB
• Prepare Matrix data as text file with Matrix Market’ exchange format
• Run in parallel if the PC is multi-core
• To click, solutions, history, etc are computed
Lis-test: GUI for Library Lis
Comparison is done easily!
To get codes Lis-test/Lis

http://ssi.is.s.u-tokyo.ac.jp/lis/

version 1.1.0 β 2
Lis has more than $10 \times 13 \times 11$ combinations

<table>
<thead>
<tr>
<th>Precond.</th>
<th>Solvers</th>
<th>Storage Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>CG</td>
<td>CRS: Compressed Row</td>
</tr>
<tr>
<td>SSOR</td>
<td>BiCG</td>
<td>CCS: Compressed Column</td>
</tr>
<tr>
<td>ILU(k)</td>
<td>CGS</td>
<td>MSR: Modified Compressed</td>
</tr>
<tr>
<td>Hybrid</td>
<td>BiCGSTAB</td>
<td>Sparse Row</td>
</tr>
<tr>
<td>I+S</td>
<td>BiCGSTAB(l)</td>
<td>DIA: Diagonal</td>
</tr>
<tr>
<td>SA-AMG</td>
<td>BiCGSafe</td>
<td>JDS: Jagged Diagonal</td>
</tr>
<tr>
<td>Crout ILU</td>
<td>Orthomin(m)</td>
<td>COO: Coordinate</td>
</tr>
<tr>
<td>additive</td>
<td>GMRES(m)</td>
<td>DNS: Dense</td>
</tr>
<tr>
<td>schwarz User</td>
<td>TFQMR</td>
<td>BSR: Block Sparse Row</td>
</tr>
<tr>
<td>defined</td>
<td>Jacobi</td>
<td>BSC: Block Sparse Column</td>
</tr>
<tr>
<td></td>
<td>Gauss–Seidel</td>
<td>VBR: Variable Block Row</td>
</tr>
<tr>
<td></td>
<td>SOR</td>
<td></td>
</tr>
</tbody>
</table>

CRS: Compressed Row
CCS: Compressed Column
MSR: Modified Compressed Sparse Row
DIA: Diagonal
JDS: Jagged Diagonal
COO: Coordinate
DNS: Dense
BSR: Block Sparse Row
BSC: Block Sparse Column
VBR: Variable Block Row
Other features

• 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
Requirement

• C Compiler (necessary)
  – Intel C/C++ 7.0, 8.0, 9.0, 9.1  IBM XL C 7.0
  – SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  – GCC 3.3 or later

• Fortran Compiler (optional)
  – For FORTRAN API: F77 or later
  – For SAAMG Precond.: F90 or later
  – Intel Fortran 8.1, 9.0, 9.1  IBM XL FORTRAN 9.1
  – SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  – g77 3.3 or later  gfortran 4.1(NG for SAAMG)  g95 0.91
Steps

1. Initialize
2. Make matrix
3. Make vector
4. Define Solver
5. Set Values
6. Set conditions
7. Execute
8. Finalize
Summary

• Fast Quadruple Arithmetic Operations with SSE2
  – Reducing round-off errors
  – Same accuracy with FORTRAN real *16
  – Computation time is about 3.2 times of Double

• A Mixed Precision Iterative method
  – Combination of Double and Fast Quadruple
  – Restart with different precision
  – Select appropriate epsilon is necessary
  – Automatic D-Q SWITCH has proposed.

• Parallel Issues
  – Same data transfer and heavy computation cost

• Library Lis and its GUI Lis-test for free to public
  – More than 1K combinations of precond., solutions, and format.
  – Serial, shared and distributed parallel and Hybrid parallel
  – Fast Quadruple Arithmetic Operations
Conclusion

- DQ-SWITCH is absolutely faster the difficult problems. (Double does not converge!)
- DQ-SWITCH is robust but slower for the ordinary easy problems
- However A Fast Quadruple Arithmetic Operations fits for Parallel computing environments
- Maybe lighter preconditioner are effective for parallel environments and A Fast Quadruple Arithmetic Operations.
- We should test for more problems
Thank you!

ありがとうございました。