Iterative methods with mixed-precision preconditioning for ill-conditioned linear systems in multiphase CFD simulations

Takuya Ina\textsuperscript{1}, Yasuhiro Idomura\textsuperscript{2}, Toshiyuki Imamura\textsuperscript{1}, Susumu Yamashita\textsuperscript{3}, Naoyuki Onodera\textsuperscript{2}

\textsuperscript{1}R-CCS, Riken, Kobe, Hyogo 650-0047, Japan
\textsuperscript{2}Japan Atomic Energy Agency, Kashiwa, Chiba 277-0871, Japan
\textsuperscript{3}Japan Atomic Energy Agency, Tokai, Ibaraki 319-1195, Japan

Acknowledgements
- This work was supported by the MEXT (Grant No.19K11992), the Fugaku Preliminary Use Project (hp200172), and the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (jh210003).
- This research used Fugaku@Riken, Oakforest-PACS@JCAHPC, FX1000@Nagoya Univ., and SGI8600@JAEA.
Main contributions in this work

- Iterative methods for sparse linear systems are important for exascale CFD simulations
- Characteristic performance features in exascale supercomputers
  - Large performance gap between computation and communication
  - Communication-avoiding (CA) algorithms
    - CA Krylov methods [Hoemmen, PhD10, Carson, PhD15, Suda, RIMS16]
    - CA preconditioners [Yamazaki, IPDPS14, Mayumi, ScalA16, Idomura, ScalA18]
  - Support for FP16 operations which have 4x FLOPS than FP64 (8x on A100 GPU)
    - FP16-based mixed-precision computing
      - GPUs: Lattice QCD [Clark, CPC10], Earthquake [Ichimura, SC18], Dense matrix [Carson, SIAM17, Haidar, SC18]
      - Fugaku: HPL-AI [Kudo, ScalA20], Fusion plasma [Idomura, SC20]
- Mixed-precision iterative methods for ill-conditioned matrices in multiphase CFD simulations
  - Iterative refinement based (IR) preconditioner is designed using hybrid FP16/32 implementation
  - Robustness of IR preconditioner is examined by scanning precision or bit length of significand
  - P-CG and MGCG solvers with IR preconditioner are tested up to 8,000 CPUs on Fugaku
  - Performance and accuracy issues in FP16-based mixed-precision iterative methods are discussed
Pressure Poisson solver in JUPITER

- JUPITER code [Yamashita,NED17] simulates molten materials in nuclear reactors
  - Incompressible fluid model based on finite difference in structured grids
  - Volume of fluid method for multiphase flows (Solid/Liquid phases of UO₂, Zry, B₄C, SUS, and Air)
  - 3D domain decomposition (MPI+OpenMP)
  - Pressure Poisson solver occupies more than 90% of the total cost

\[ \nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \mathbf{u}^* - \nabla \cdot \left( \frac{\Delta t}{\rho} \nabla p \right) = 0 \]

- 2nd order centered finite difference in structured grids (7-stencils)
- Large contrast ~10⁷ of density \( \rho \) gives an ill-conditioned problem
  - Single node benchmark: 240x150x1,024 ~ 37M DOFs, Cond. number ~ 4x10⁷
  - Large scale benchmark: 3,200x2,000x14,160 ~ 90G DOFs, Cond. number ~ 6x10⁹
- Extreme scale CA Krylov solvers on state-of-the-art supercomputers
  - CACG solver on K-computer (30,000 CPUs) [Mayumi,ScalA16]
  - CAMGCG solver on Oakforest-PACS (8,000 CPUs) [Idomura,ScalA18]
  - CBCG solver on Oakforest-PACS (2,000 CPUs) [Idomura,LNCS18]
  - CBCG solver on Summit (7,689 GPUs) [Ali,ScalA19]
Main features of Oakforest-PACS and Fugaku

Similarity: ~3TFlops peak performance with 50+ cores, wide SIMD operations with 512 bits

Difference: Support for FP16, 2.1x memory BW, 3.3x node injection BW, network topology

JUPITER code (C, ~60k lines)

- OFP: Intel C compiler 19.0.5.281, Intel MPI library 2019
- Fugaku: Fortran and C compiler in Fujitsu Technical Computing Suite V4.5.0, Fujitsu MPI library 4.0.0

→ Mixed-precision IR preconditioner is implemented in Fortran, which optimizes FP16 operations
Preconditioned conjugate gradient (P-CG) solver

- Original P-CG solver computes block Jacobi (BJ) preconditioner with ILU(0) in FP64
  - BJ blocks: 3D MPI domain $x$ $z$-decomposition for OMP ($z$-div)
  - $→$ lack of concurrency for wide SIMD operations ($512\text{bits}=32 \times \text{FP16 variables}$)

- Optimization on Fugaku
  - BJ blocks: 3D MPI domain $x$ $z$-decomposition for OMP $x$ $x$-decomposition for SIMD ($zx$-div)
  - Convergence degradation due to fine BJ blocks is compensated by IR method
  - Memory access in IR method is reduced by hybrid FP16/32 implementation

```
Algorithm 1 Preconditioned Conjugate Gradient (P-CG) method

Input: $Ax = b$, Initial guess $x_1$

Output: Approximate solution $x_i$

1: $r_1 := b - Ax_1$, $z_1 = M^{-1}r_1$, $p_1 := z_1$
2: for $i = 1, 2, ..., \text{until convergence}$ do
3: \hspace{1em} $w := Ap_i$
4: \hspace{1em} $\alpha_i := \langle r_i, z_i \rangle / \langle w, p_i \rangle$
5: \hspace{1em} $x_{i+1} := x_i + \alpha_i p_i$
6: \hspace{1em} $r_{i+1} := r_i - \alpha_i w$
7: \hspace{1em} $z_{i+1} := M^{-1}r_{i+1}$
8: \hspace{1em} $\beta_i := \langle r_{i+1}, z_{i+1} \rangle / \langle r_i, z_i \rangle$
9: \hspace{1em} $p_{i+1} := z_{i+1} + \beta_i p_i$
10: end for
```
Multigrid preconditioned CG (MGCG) method

- CA Geometric MG preconditioner with a V-cycle [Idomura, ScalA18]
- Preconditioned Chebyshev iteration (P-CI) smoother (no All_reduce)
  - Min/max eigenvalues $\lambda_{\text{min}}/\lambda_{\text{max}}$ are computed by CA-Lanczos method
  - Block Jacobi (BJ) preconditioner with ILU(0)
- Mixed precision to reduce computation and halo communication
  - FP64 implementation for CG and CA-Lanczos, which are based on Krylov subspace
  - FP32 implementation for P-CI smoother $\rightarrow$ replace BJ-ILU with mixed-precision IR

Algorithm 2 Preconditioned Chebyshev Iteration (P-CI) method

Input: $Ax = b$, Initial guess $x_1$, Approximate minimum/max-imum eigenvalues of $AM^{-1}$, $\lambda_{\text{min}}, \lambda_{\text{max}}$

Output: Approximate solution $x_i$

1: $d := (\lambda_{\text{max}} + \lambda_{\text{min}})/2, c := (\lambda_{\text{max}} - \lambda_{\text{min}})/2$
2: $r_1 := b - Ax_1, z_1 := M^{-1}r_1, p_1 := z_1, \alpha_1 := 1/d$
3: for $i = 1, 2, \ldots$ until convergence do
4: $x_{i+1} := x_i + \alpha_ip_i$
5: $r_{i+1} := b - Ax_{i+1}$
6: $z_{i+1} := M^{-1}r_{i+1}$
7: $\beta_{i+1} := (\alpha_i c/2)^2$
8: $\alpha_{i+1} := 1/(d - \beta_{i+1}/\alpha_i)$
9: $p_{i+1} := z_{i+1} + \beta_{i+1}p_i$
10: end for
In mixed-precision computing, IR method is used to improve low precision solutions
[Carson,SIAM17,Haidar,SC18,Kudo,ScalA20]

In this work, we use IR method to improve low precision preconditioning
- Solve processes are approximated by BJ-ILU(0) operator M
- To avoid underflow/overflow, residual vectors and linear systems are normalized by D
- To avoid roundoff errors, hybrid FP16/32 implementation is applied to all computation
  load/store FP16 data ⇨ on-cache FP16/32 conversion ⇨ FP32 computation

Algorithm 3 Iterative Refinement (IR) method

\[
\text{Input: } Ax = b \\
\text{Output: } \text{Approximate solution } x_i \\
1: \text{Solve } Ax_1 = b \\
2: \text{for } i = 1, 2, \ldots \text{ until convergence do} \\
3: \quad r_i := b - Ax_i \\
4: \quad \text{Solve } As_i = r_i \\
5: \quad x_{i+1} := x_i + s_i \\
6: \text{end for}
\]

[Algorithm 3: Iterative Refinement (IR) method]

![Algorithm 3: Iterative Refinement (IR) method](image)

Algorithm 4 IR-based preconditioner

\[
\text{Input: } Mz = r, A, \text{Matrix for BJ preconditioning } M, D_{ii} = \max(|M_{i1}|, |M_{i2}|, \ldots, |M_{in}|) \\
\text{Output: } \text{Approximate solution } z \\
1: \tilde{r} := r/||D^{-1}r|| \\
2: \text{Solve } D^{-1}M\tilde{y}_1 = D^{-1}\tilde{r} \text{ in FP32; load/store in FP16.} \\
3: \text{for } i = 1, 2, \ldots \text{ until convergence do} \\
4: \quad \text{Compute } \tilde{u}_i := D^{-1}\tilde{r} - D^{-1}A\tilde{y}_i \\
\quad \text{in FP32; load/store in FP16.} \\
5: \quad \text{Solve } D^{-1}M\tilde{v}_i = D^{-1}\tilde{u}_i/||D^{-1}\tilde{u}_i|| \\
\quad \text{in FP32; load/store in FP16.} \\
6: \quad \tilde{y}_{i+1} := \tilde{y}_i + ||D^{-1}\tilde{u}_i||\tilde{v}_i \\
7: \text{end for} \\
8: z := ||D^{-1}r||\tilde{y}_{i+1}
\]

[Algorithm 4: IR-based preconditioner]
Single node benchmark tests using P-CG solvers

Fugaku (240x150x1,024 ~ 37M DOFs)

<table>
<thead>
<tr>
<th>precondition</th>
<th>n iteration</th>
<th>f [flop/grid]</th>
<th>b [byte/grid]</th>
<th>t_R [s]</th>
<th>t [s]</th>
<th>t_R/t efficiency</th>
<th>t[s]/n</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BJ(z-div,FP64)</td>
<td>1818</td>
<td>39xFP64</td>
<td>224</td>
<td>18.7</td>
<td>40.7</td>
<td>46%</td>
<td>0.0224</td>
<td>1.00</td>
</tr>
<tr>
<td>BJ(zx-div,FP64)</td>
<td>2866</td>
<td>36xFP64</td>
<td>236</td>
<td>30.9</td>
<td>37.2</td>
<td>83%</td>
<td>0.0130</td>
<td>1.09</td>
</tr>
<tr>
<td>IR(zx-div,FP64)</td>
<td>1484</td>
<td>62xFP64</td>
<td>408</td>
<td>27.7</td>
<td>31.4</td>
<td>88%</td>
<td>0.0211</td>
<td>1.30</td>
</tr>
<tr>
<td>IR(zx-div,FP16)</td>
<td>3583</td>
<td>35xFP64+27xFP16</td>
<td>234</td>
<td>38.6</td>
<td>47.0</td>
<td>82%</td>
<td>0.0131</td>
<td>0.87</td>
</tr>
<tr>
<td>IR(zx-div,FP16/32)</td>
<td>1484</td>
<td>26xFP64+36xFP32</td>
<td>234</td>
<td>16.4</td>
<td>18.2</td>
<td>90%</td>
<td>0.0123</td>
<td>2.23</td>
</tr>
</tbody>
</table>

※Single IR iteration showed the best performance for all IR preconditioners

- Improved computing efficiency by BJ(zx-div) is almost cancelled by convergence degradation
- IR(FP64) dramatically improves convergence, but performance gain is limited by memory access
- IR(FP16) accelerates t/n, but leads to significant convergence degradation
- IR(FP16/32) accelerates t/n with keeping the same convergence property as IR(FP64)

→IR(FP16/32) is 2.2x faster than the original BJ(FP64)
To clarify required precision, different data formats are implemented using bit operations in C.

- Keeps almost the same convergence up to FP12/32, and breaks down at FP10/32.

→ Required precision is 6 bits of significand, and FP16/32 has extra 4 bits before convergence degradation.

- bfloat16 may be the best choice on GPUs and Intel Copper Lake.
  - Satisfy the above requirement on significand.
  - Keep the same exponent as FP32, and works without normalization.
- All solvers show good strong scaling up to 8,000 CPUs.
- IR preconditioner shows better convergence, and is 2x faster than BJ preconditioner.
- 5.7x speedup between P-CG/IR on Fugaku and P-CG/BJ on OFP greatly exceeds memory BW ratio ~2x.

  Computation ~ 2.96x,  All_Reduce ~ 39.0x,  Halo comm. ~ 2.03x

  → Large impacts of hybrid FP16/32 implementation and TofuD interconnect.
All solvers show good strong scaling up to 8,000 CPUs with ~10x speedups from P-CG solvers.

- Number of iterations $n$ is reduced to ~1/1000, leading to negligible All_Reduce.
- IR preconditioner is 1.4x faster than BJ preconditioner.
- 3.4x speedup between MGCG/IR on Fugaku and MGCG/BJ on OFP.

Computation $\sim 2.54x$, All_Reduce $\sim 29.2x$, Halo comm. $\sim 6.23x$

→Less performance gain between BJ in FP32 and IR in FP16/32, and small impact from All_Reduce.
New mixed-precision iterative refinement (IR) preconditioner was designed on Fugaku
- Fine BJ blocks improved wide SIMD optimization
- Convergence degradation due to fine BJ blocks was avoided by IR method
- IR preconditioning was accelerated by FP16-based mixed-precision computing
  - Normalization of linear systems and residual vectors
  - Hybrid FP16/32 implementation

Robustness of IR preconditioner was examined by scanning precision or bit length of significand
- Required precision was FP12/32 and FP16/32 has extra 4 bits before convergence degradation

P-CG and MGCG solvers with IR preconditioner showed good strong scaling up to 8,000 CPUs on Fugaku
- P-CG and MGCG solvers with IR showed 5.7x and 3.4x speedups from conventional solvers on OFP
  - Large impacts from hybrid FP16/32 implementation and TofuD interconnect