

3.1 JAXA LBM コードの性能測定および改善

宇宙航空研究開発機構 石田 崇
富士通株式会社 三吉 郁夫

3.1.1 はじめに

航空宇宙分野，特に航空分野の数値流体力学（Computational Fluid Dynamics：CFD）においては，巡航状態の航空機翼面上で発生する衝撃波を精度良く捉えるために，Navier-Stokes 方程式をベースとした計算アルゴリズムが発展してきた．非構造格子法，近似リーマン解法，陰的時間積分，乱流モデルといった計算アルゴリズムの進展に加えて，大型計算機の性能向上によって，航空機の巡航状態における空力性能を精度良くかつ実用的な解析時間で予測できるようになってきたため，CFD は風洞試験と並んで航空機設計開発には欠かせないツールとしての地位を確立している．近年では CFD を航空機のオフデザイン（バフエットなどの非定常状態）における空力性能予測に適用するニーズが増えてきている．しかしながら非定常状態に対する Navier-Stokes 方程式をベースとする計算アプローチでは，計算精度・計算コスト・物理モデル依存の観点でまだ課題が多く，効率よく解析することが難しい．このような状況を打破するアプローチとして，近年格子ボルツマン法（Lattice Boltzmann Method：LBM）が注目されている．LBM では支配方程式がボルツマン輸送方程式であり，以下に示す特徴を持つ．

- ・支配方程式に非線形項が無い，
- ・支配方程式は非圧縮 Navier-Stokes 方程式に漸近するが，音の伝搬を直接計算できる，
- ・衝突（Collision）と並進（Stream）の 2 ステップで時間発展する，
- ・時間積分は CFL 数が 1 に固定された陽解法である，
- ・Navier-Stokes 方程式に比べて計算コストが小さい，
- ・格子点毎に依存関係が無いので容易に並列化が出来る．

JAXA では航空機の離着陸形態における空力騒音問題や低速バフエットの解析に向けた大規模非定常流体解析コードとして LBM コードを開発・整備しており，本稿ではメニーコア WG 内で行った FX100 を用いたコードの性能評価および改善について報告する．

3.1.2 JAXA LBM コードの概要

JAXA LBM コードでは、直交格子法の一つである Building-Cube Method (BCM) をフレームワークとして採用し、LBM と組み合わせて大規模非定常解析コードとして整備している。以降では、LBM および BCM の概要について説明する。

3.1.2.1 LBM のアルゴリズム概要

LBM における支配方程式は、

$$\frac{D}{Dt} f(\mathbf{x}, \mathbf{e}, t) = \left[\frac{\partial}{\partial t} + \mathbf{e} \cdot \nabla \right] f = \Omega \quad (1)$$

であり、格子上で離散化すると以下の離散式が得られる。

$$f_i(\mathbf{x} + \mathbf{e}_i \times dt, t + dt) = f_i(\mathbf{x}, t) + dt \times \Omega_i, \quad i = 1, \dots, b \quad (2)$$

ここで f は状態分布関数、 \mathbf{x} は位置ベクトル、 \mathbf{e} は粒子の速度、 Ω は衝突項、 b は粒子の自由度の数であり、 Ω には以下に示す BGK モデルを用いた Single Relaxation Time (SRT) を用いるのが一般的である。

$$\Omega_i^{BGK} = -\frac{1}{\tau} (f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)) \quad (3)$$

ここで f^{eq} は平衡状態分布関数、 τ は緩和係数である。時間積分は衝突 (Collision) と並進 (Stream) の 2 ステップからなり、衝突は (4)、並進は (5) のようにそれぞれ記述できる。

$$f_i^*(\mathbf{x}, t) = f_i(\mathbf{x}, t) + \Omega_i, \quad i = 1, \dots, b \quad (4)$$

$$f_i(\mathbf{x} + \mathbf{e}_i \times dt, t + dt) = f_i^*(\mathbf{x}, t), \quad i = 1, \dots, b \quad (5)$$

SRT の特徴は分散誤差が少ない点であるが、航空機の空力性能予測で対象とするような高 Re 数 (低粘性) の流れでは計算が非常に不安定ですぐに破綻してしまう。SRT では全ての状態分布関数を同一の緩和係数で緩和するが、この過程に計算を不安定化させる高次のモードをダンプする仕組みが無いためである。この問題を解決するために様々な衝突項モデルが世界的に研究されており、JAXA LBM コードでは Dr. Mertin Geier によって提案された Cascaded LBM を衝突項に実装している。Cascaded LBM は状態分布関数を (6) に示す central moment に変換し、central moment space で異なる緩和係数で緩和 (特に計算を不安定化させる高次のモーメントを dissipative な緩和係数で緩和) させることにより高 Re 数流れでも安定に計算することが出来る。

$$\begin{aligned} \rho \tilde{\mathbf{M}}_{p,q,r} &= \sum_i (e_{ix} - u_x)^p \cdot (e_{iy} - u_y)^q \cdot (e_{iz} - u_z)^r \cdot f_i \\ &= \mathbf{C} \cdot \mathbf{f} \end{aligned} \quad (6)$$

状態分布関数から central moment への変換行列 \mathbf{C} は、 $\mathbf{C} = \mathbf{C}(\mathbf{u})$ とローカルな速度の関数になっているため、格子点毎に計算しなくてはならない。また、central moment space で緩和した後は状態分布関数に逆変換しなければならず、これらの行列演算にかかる計算負荷は大きい。参考に2次元9速度(D2Q9)モデルにおける変換行列 \mathbf{C} およびその逆行列である \mathbf{C}^{-1} を(7)および(8)にそれぞれ示す。

3.1.2.2 Building-Cube Method (BCM)

Building-Cube Method では、計算領域を立方体領域 (Cube) に分割し、各 Cube 内に等間隔直交格子 (Cell) を生成して解析するフレームワークである。流体解析のプログラム部分は single block 用に作成し、Cube ループで全体の計算を回す間に適宜 Cube 間の情報交換を実施する。

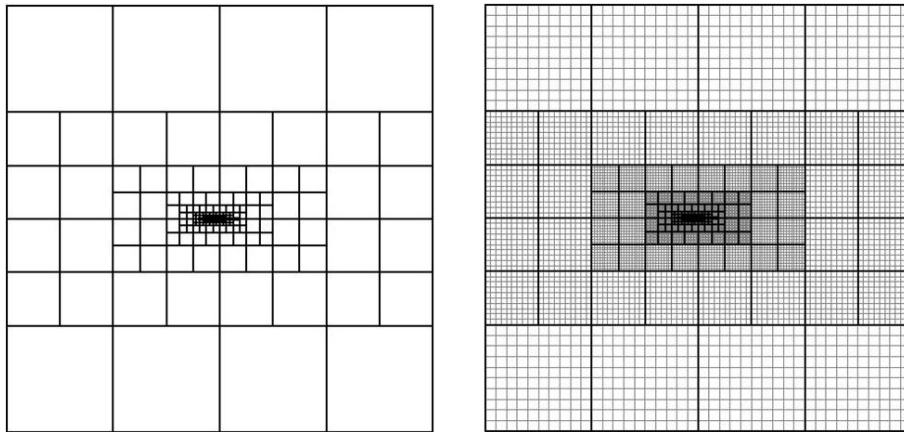


図 6 Cube (左図) と Cell (右図)

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-u & 1-u & -u & -1-u & -u & 1-u & -1-u & -1-u & 1-u \\
-v & -v & 1-v & -v & -1-v & 1-v & 1-v & -1-v & -1-v \\
u^2-v^2 & (1-u)^2-v^2 & u^2-(1-v)^2 & (-1-u)^2-v^2 & u^2-(-1-v)^2 & (1-u)^2-(1-v)^2 & (-1-u)^2-(1-v)^2 & (-1-u)^2-(-1-v)^2 & (1-u)^2-(-1-v)^2 \\
u^2+v^2 & (1-u)^2+v^2 & u^2+(1-v)^2 & (-1-u)^2+v^2 & u^2+(-1-v)^2 & (1-u)^2+(1-v)^2 & (-1-u)^2+(1-v)^2 & (-1-u)^2+(-1-v)^2 & (1-u)^2+(-1-v)^2 \\
uv & (-1+u)v & -u(1-v) & (1+u)v & -u(-1-v) & (1-u)(1-v) & (-1-u)(1-v) & (-1-u)(-1-v) & (1-u)(-1-v) \\
-u v^2 & (1-u)v^2 & -u(1-v)^2 & (-1-u)v^2 & -u(-1-v)^2 & (1-u)(1-v)^2 & (-1-u)(1-v)^2 & (-1-u)(-1-v)^2 & (1-u)(-1-v)^2 \\
-u^2 v & -(1-u)^2 v & u^2(1-v) & -(-1-u)^2 v & u^2(-1-v) & (1-u)^2(1-v) & (-1-u)^2(1-v) & (-1-u)^2(-1-v) & (1-u)^2(-1-v) \\
u^2 v^2 & (1-u)^2 v^2 & u^2(1-v)^2 & (-1-u)^2 v^2 & u^2(-1-v)^2 & (1-u)^2(1-v)^2 & (-1-u)^2(1-v)^2 & (-1-u)^2(-1-v)^2 & (1-u)^2(-1-v)^2
\end{pmatrix} \quad (7)$$

$$\begin{pmatrix}
(-1+u^2)(-1+v^2) & 2u(-1+v^2) & 2(-1+u^2)v & \frac{1}{2}(-u^2+v^2) & \frac{1}{2}(-2+u^2+v^2) & 4uv & 2u & 2v & 1 \\
-\frac{1}{2}u(1+u)(-1+v^2) & -\frac{1}{2}(1+2u)(-1+v^2) & -u(1+u)v & \frac{1}{4}(1+u+u^2-v^2) & \frac{1}{4}(1-u(1+u)-v^2) & -(1+2u)v & -\frac{1}{2}-u & -v & -\frac{1}{2} \\
-\frac{1}{2}(-1+u^2)v(1+v) & -uv(1+v) & -\frac{1}{2}(-1+u^2)(1+2v) & \frac{1}{4}(-1+u^2-v(1+v)) & \frac{1}{4}(1-u^2-v(1+v)) & -u(1+2v) & -u & -\frac{1}{2}-v & -\frac{1}{2} \\
-\frac{1}{2}(-1+u)u(-1+v^2) & -\frac{1}{2}(-1+2u)(-1+v^2) & -(-1+u)uv & \frac{1}{4}(1+(-1+u)u-v^2) & \frac{1}{4}(1+u-u^2-v^2) & v-2uv & \frac{1}{2}-u & -v & -\frac{1}{2} \\
-\frac{1}{2}(-1+u^2)(-1+v)v & -u(-1+v)v & -\frac{1}{2}(-1+u^2)(-1+2v) & \frac{1}{4}(-1+u^2+v-v^2) & \frac{1}{4}(1-u^2+v-v^2) & u-2uv & -u & \frac{1}{2}-v & -\frac{1}{2} \\
\frac{1}{4}u(1+u)v(1+v) & \frac{1}{4}(1+2u)v(1+v) & \frac{1}{4}u(1+u)(1+2v) & \frac{1}{8}(-u(1+u)+v+v^2) & \frac{1}{8}(u+u^2+v+v^2) & \frac{1}{4}(1+2u)(1+2v) & \frac{1}{4}(1+2u) & \frac{1}{4}(1+2v) & \frac{1}{4} \\
\frac{1}{4}(-1+u)uv(1+v) & \frac{1}{4}(-1+2u)v(1+v) & \frac{1}{4}(-1+u)u(1+2v) & \frac{1}{8}(u-u^2+v+v^2) & \frac{1}{8}((-1+u)u+v+v^2) & \frac{1}{4}(-1+2u)(1+2v) & \frac{1}{4}(-1+2u) & \frac{1}{4}(1+2v) & \frac{1}{4} \\
\frac{1}{4}(-1+u)u(-1+v)v & \frac{1}{4}(-1+2u)(-1+v)v & \frac{1}{4}(-1+u)u(-1+2v) & -\frac{1}{8}(u-v)(-1+u+v) & \frac{1}{8}((-1+u)u+(-1+v)v) & \frac{1}{4}(-1+2u)(-1+2v) & \frac{1}{4}(-1+2u) & \frac{1}{4}(-1+2v) & \frac{1}{4} \\
\frac{1}{4}u(1+u)(-1+v)v & \frac{1}{4}(1+2u)(-1+v)v & \frac{1}{4}u(1+u)(-1+2v) & -\frac{1}{8}(1+u-v)(u+v) & \frac{1}{8}(u+u^2+(-1+v)v) & \frac{1}{4}(1+2u)(-1+2v) & \frac{1}{4}(1+2u) & \frac{1}{4}(-1+2v) & \frac{1}{4}
\end{pmatrix} \quad (8)$$

3.1.3 コード改善

本章では, FX100 (JSS2 MA システム) を用いたコード改善による性能評価について報告する. 性能評価には C++ で記述された 2 次元 9 速度 (D2Q9) モデルのプログラムを用い, Lid Driven Cavity を計算対象として格子点数 128×128 の single block の直交格子を用いた. 時間積分は 1000 ステップ分実施して性能を評価した. なお, 計算は 1core で実施し, コンパイルには以下オプションを使用した.

コンパイルオプション : `-Kfast -Nlst=t -std=c++11`

以降で出てくるサンプルコード内の変数を以下に定義する.

Q : 自由度の数 (2 次元 : $Q=9$, 3 次元 : $Q=27$)
ftmp : 衝突後の状態分布関数
rho : 密度
CM : central moment $\tilde{\mathbf{M}}$
omg : モード毎の緩和係数
M : 変換行列 \mathbf{C}
Minv : 逆変換行列 \mathbf{C}^{-1}

3.1.3.1 Collision の SIMD 化

central moment を central moment space でモード毎に緩和させた後に，元の状態分布関数に逆変換する操作において，衝突後の分布関数を格納する配列：ftmp の定義にダブルポインタを用いていたため，コンパイル最適化が十分に効かなかった．そこで SIMD 化を促進するため，以下の変更を実施した．その結果，表 に示す通り 1.74 倍程度高速化することが出来た．

変更前

```
for (int n = 0; n < Q; ++n) {
    ftmp[I][n] = 0;
    for (int m = 0; m < Q; ++m) {
        ftmp[I][n] += rho[I] * (Minv[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
    }
}
```

変更後

```
T ftmp_[Q];
for (int n = 0; n < Q; ++n) {
    ftmp_[n] = 0;
}
for (int m = 0; m < Q; ++m) {
    for (int n = 0; n < Q; ++n) {
        ftmp_[n] += rho[I] * (Minv[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
    }
}
for (int n = 0; n < Q; ++n) {
    ftmp[I][n] = ftmp_[n];
}
```

表 1

	変更前	変更後
Elapsed(s)	29.0696	16.7274
MFLOPS	588.2525	895.9001
MFLOPS/PEAK (%)	1.6655	2.5452

3.1.3.2 central moment $\tilde{\mathbf{M}} = \mathbf{C} \cdot \mathbf{f}$ を求める行列演算の書き下し

central moment を求める行列は速度の関数になっており，格子点毎に実施する変換行列の定義およびそれらの演算はコストが大きい．そこで行列演算を書き下して性能が変化するか調査した．その結果，表 に示す通り更に 1.13 倍程度高速化することが出来，2次元9速度モデルでは行列演算の書き下しにある程度効果があることが分かった．

変更前

```

//! Transform matrix from f to Central Moment
T M[Q][Q] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1,
    -u, 1 - u, -u, -1 - u, -u, 1 - u, -1 - u, -1 - u, 1 - u,
    -v, -v, 1 - v, -v, -1 - v, 1 - v, 1 - v, -1 - v, -1 - v,
    u*v, -v * (1 - u), -u * (1 - v), -v * (-1 - u), -u * (-1 - v), (1 - u)*(1 - v), (-1 - u)*(-1 - v), (-1 - u)*(-1 - v), (1 - u)*(-1 - v),
    u2 - v2, ua - v2, u2 - va, ub - v2, u2 - vb, ua - va, ub - va, ub - vb, ua - vb,
    u2 + v2, ua + v2, u2 + va, ub + v2, u2 + vb, ua + va, ub + va, ub + vb, ua + vb,
    -u * v2, v2*(1 - u), -u * va, v2*(-1 - u), -u * vb, (1 - u)*va, (-1 - u)*va, (-1 - u)*vb, (1 - u)*vb,
    -u2 * v, -v * ua, u2*(1 - v), -v * ub, u2*(-1 - v), ua*(1 - v), ub*(1 - v), ub*(-1 - v), ua*(-1 - v),
    u2*v2, v2*ua, u2*va, v2*ub, u2*vb, ua*va, ub*va, ub*vb, ua*vb
};
//! Calculation of Central Moment
for (int l = 0; l < Q; ++l) {
    CM[l] = 0;
    for (int n = 0; n < Q; ++n) {
        CM[l] += M[l][n] * f[l][n];
    }
    CM[l] = CM[l] / ro;
}

```

変更後

```

CM[0] = f_ [0] + f_ [1] + f_ [2] + f_ [3] + f_ [4] + f_ [5] + f_ [6] + f_ [7] + f_ [8];
CM[1] = (-u)*f_ [0] + (1 - u)*f_ [1] + (-u)*f_ [2] + (-1 - u)*f_ [3] + (-u) * f_ [4] +
(1 - u) * f_ [5] + (-1 - u) * f_ [6] + (-1 - u) * f_ [7] + (1 - u) * f_ [8];
CM[2] = (-v)*f_ [0] + (-v)*f_ [1] + (1 - v)*f_ [2] + (-v)*f_ [3] + (-1 - v) * f_ [4] +
(1 - v) * f_ [5] + (1 - v) * f_ [6] + (-1 - v) * f_ [7] + (-1 - v) * f_ [8];
CM[3] = (u * v)*f_ [0] + (-v * (1 - u))*f_ [1] + (-u * (1 - v))*f_ [2] + (-v * (-1 - u))*f_ [3] + (-u * (-1 - v))*f_ [4] +
((1 - u)*(1 - v))*f_ [5] + ((-1 - u)*(1 - v))*f_ [6] + ((-1 - u)*(-1 - v))*f_ [7] + ((1 - u)*(-1 - v))*f_ [8];
CM[4] = (u2 - v2)*f_ [0] + (ua - v2)*f_ [1] + (u2 - va)*f_ [2] + (ub - v2)*f_ [3] + (u2 - vb)*f_ [4] +
(ua - va)*f_ [5] + (ub - va)*f_ [6] + (ub - vb)*f_ [7] + (ua - vb)*f_ [8];
CM[5] = (u2 + v2)*f_ [0] + (ua + v2)*f_ [1] + (u2 + va)*f_ [2] + (ub + v2)*f_ [3] + (u2 + vb)*f_ [4] +
(ua + va)*f_ [5] + (ub + va)*f_ [6] + (ub + vb)*f_ [7] + (ua + vb)*f_ [8];
CM[6] = (-u * v2)*f_ [0] + (v2*(1 - u))*f_ [1] + (-u * va)*f_ [2] + (v2*(-1 - u))*f_ [3] + (-u * vb)*f_ [4] +
((1 - u)*va)*f_ [5] + ((-1 - u)*va)*f_ [6] + ((-1 - u)*vb)*f_ [7] + ((1 - u)*vb)*f_ [8];
CM[7] = (-u2 * v)*f_ [0] + (-v * ua)*f_ [1] + (u2*(1 - v))*f_ [2] + (-v * ub)*f_ [3] + (u2*(-1 - v))*f_ [4] +
(ua*(1 - v))*f_ [5] + (ub*(1 - v))*f_ [6] + (ub*(-1 - v))*f_ [7] + (ua*(-1 - v))*f_ [8];
CM[8] = (u2 * v2)*f_ [0] + (v2*ua)*f_ [1] + (u2*va)*f_ [2] + (v2*ub)*f_ [3] + (u2*vb)*f_ [4] +
(ua*va)*f_ [5] + (ub*va)*f_ [6] + (ub*vb)*f_ [7] + (ua*vb)*f_ [8];

```

表 2

	変更前	変更後
Elapsed(s)	16.7274	14.7741
MFLOPS	895.9001	849.2416
MFLOPS/PEAK (%)	2.5452	2.4126

3.1.3.3 状態分布関数 $\mathbf{f} = \mathbf{C}^{-1} \cdot \tilde{\mathbf{M}}$ を求める行列演算の書き下し

2.6.3.2 で行列演算の書き下しによる効果が得られたため, central moment から状態分布関数を求める逆変換の行列演算も書き下して性能が変化するか調査した. その結果, 表に示す通り 1.64 倍程度高速化することが出来, 2次元9速度モデルにおける逆行列演算に関する書き下しには大きな効果があることが分かった.

変更前

```
T ftmp_[Q];
for (int n = 0; n < Q; ++n) {
    ftmp_[n] = 0;
}
for (int m = 0; m < Q; ++m) {
    for (int n = 0; n < Q; ++n) {
        ftmp_[n] += rho[l] * (MiC2[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
    }
}
for (int n = 0; n < Q; ++n) {
    ftmp[l][n] = ftmp_[n];
}
```

変更後

```
T CMpost[Q];
for (int n = 0; n < Q; ++n) {
    CMpost[n] = ro * (CM[n] * (1 - omg[n]) + CMeq[n] * omg[n]);
}
ftmp[l][0] = (u2 - 1) * (v2 - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 4 * u * v * CMpost[3] + 0.5 * (v2 - u2) * CMpost[4] + 0.5 * (u2 + v2 - 2) * CMpost[5] + 2 * u * CMpost[6] + 2 * v * CMpost[7] + 1 * CMpost[8];
ftmp[l][1] = -0.5 * u * (u + 1) * (v2 - 1) * CMpost[0] + 0.5 * CMpost[1] + 0 * CMpost[2] + -(2 * u + 1) * v * CMpost[3] + 0.25 * (u2 + u - v2 + 1) * CMpost[4] + 0.25 * (-u2 - u - v2 + 1) * CMpost[5] + -(u + 0.5) * CMpost[6] + -v * CMpost[7] + -0.5 * CMpost[8];
ftmp[l][2] = -0.5 * v * (v + 1) * (u2 - 1) * CMpost[0] + 0 * CMpost[1] + 0.5 * CMpost[2] + -u * (2 * v + 1) * CMpost[3] + 0.25 * (u2 - v2 - v - 1) * CMpost[4] + 0.25 * (-u2 - v2 - v + 1) * CMpost[5] + -u * CMpost[6] + -(v + 0.5) * CMpost[7] + -0.5 * CMpost[8];
ftmp[l][3] = -0.5 * u * (u - 1) * (v2 - 1) * CMpost[0] + -0.5 * CMpost[1] + 0 * CMpost[2] + v * (1 - 2 * u) * CMpost[3] + 0.25 * (u2 - u - v2 + 1) * CMpost[4] + 0.25 * (-u2 + u - v2 + 1) * CMpost[5] + (0.5 - u) * CMpost[6] + -v * CMpost[7] + -0.5 * CMpost[8];
ftmp[l][4] = -0.5 * v * (v - 1) * (u2 - 1) * CMpost[0] + 0 * CMpost[1] + -0.5 * CMpost[2] + u * (1 - 2 * v) * CMpost[3] + 0.25 * (u2 - v2 + v - 1) * CMpost[4] + 0.25 * (-u2 - v2 + v + 1) * CMpost[5] + -u * CMpost[6] + (0.5 - v) * CMpost[7] + -0.5 * CMpost[8];
ftmp[l][5] = 0.25 * u * v * (u + 1) * (v + 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u + 1) * (2 * v + 1) * CMpost[3] + 0.125 * (-u2 - u + v2 + v) * CMpost[4] + 0.125 * (u2 + u + v2 + v) * CMpost[5] + 0.25 * (2 * u + 1) * CMpost[6] + 0.25 * (2 * v + 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[l][6] = 0.25 * u * v * (u - 1) * (v + 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u - 1) * (2 * v + 1) * CMpost[3] + 0.125 * (-u2 + u + v2 + v) * CMpost[4] + 0.125 * (u2 - u + v2 + v) * CMpost[5] + 0.25 * (2 * u - 1) * CMpost[6] + 0.25 * (2 * v + 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[l][7] = 0.25 * u * v * (u - 1) * (v - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u - 1) * (2 * v - 1) * CMpost[3] + 0.125 * (-u2 + u + (v - 1) * v) * CMpost[4] + 0.125 * (u2 - u + (v - 1) * v) * CMpost[5] + 0.25 * (2 * u - 1) * CMpost[6] + 0.25 * (2 * v - 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[l][8] = 0.25 * u * v * (u + 1) * (v - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u + 1) * (2 * v - 1) * CMpost[3] + 0.125 * (-u2 - u + (v - 1) * v) * CMpost[4] + 0.125 * (u2 + u + (v - 1) * v) * CMpost[5] + 0.25 * (2 * u + 1) * CMpost[6] + 0.25 * (2 * v - 1) * CMpost[7] + 0.25 * CMpost[8];
```

表 3

	変更前	変更後
Elapsed(s)	14.7741	9.0030
MFLOPS	849.2416	1457.6530
MFLOPS/PEAK (%)	2.4126	4.1411

3.1.3.4 Streamの方針変更

Stream では【自分⇒周囲】の順番で状態分布関数をコピーする操作をしていたが、キャッシュの効率を考慮して【周囲⇒自分】の順番に修正して性能が変化するか調査した。その結果、表 に示す通り更に 1.3 倍程度高速化することが出来た。

変更前

```
for (int I = 0; I < size; ++I) {
    ijk[0] = I % NDIM[0];
    ijk[1] = I / NDIM[0];

    for (int l = 0; l < Q; ++l) {
        indx[0] = ijk[0] - c[l][0];
        indx[1] = ijk[1] - c[l][1];
        indx[0] = (indx[0] >= 0) ? (indx[0] = indx[0]) : (indx[0] = 0);
        indx[1] = (indx[1] >= 0) ? (indx[1] = indx[1]) : (indx[1] = 0);
        indx[0] = (indx[0] < NDIM[0]) ? (indx[0] = indx[0]) : (indx[0] = NDIM[0] - 1);
        indx[1] = (indx[1] < NDIM[1]) ? (indx[1] = indx[1]) : (indx[1] = NDIM[1] - 1);
        J = indx[0] + NDIM[0] * indx[1];
        flag = !mask[I].isInner() * !mask[J].isInner();
        f[J][l] = ftmp[I][l] * (flag) + f[J][l] * (1 - flag);
    }
}
```

変更後

```
for (int I = 0; I < size; ++I) {
    ijk[0] = I % NDIM[0];
    ijk[1] = I / NDIM[0];
    for (int l = 0; l < Q; ++l) {
        indx[0] = ijk[0] - c[l][0];
        indx[1] = ijk[1] - c[l][1];
        indx[0] = (indx[0] >= 0) ? (indx[0] = indx[0]) : (indx[0] = 0);
        indx[1] = (indx[1] >= 0) ? (indx[1] = indx[1]) : (indx[1] = 0);
        indx[0] = (indx[0] < NDIM[0]) ? (indx[0] = indx[0]) : (indx[0] = NDIM[0] - 1);
        indx[1] = (indx[1] < NDIM[1]) ? (indx[1] = indx[1]) : (indx[1] = NDIM[1] - 1);
        J = indx[0] + NDIM[0] * indx[1];
        f_[l] = ftmp[J][l];
    }
    f[I][0] = f_[0];
    f[I][1] = f_[1];
    f[I][2] = f_[2];
    f[I][3] = f_[3];
    f[I][4] = f_[4];
    f[I][5] = f_[5];
    f[I][6] = f_[6];
    f[I][7] = f_[7];
    f[I][8] = f_[8];
}
```

表 4

	変更前	変更後
Elapsed(s)	9.0030	6.9164
MFLOPS	1457.6530	1897.4156
MFLOPS/PEAK (%)	4.1411	5.3904

3.1.3.5 コード改善後の性能評価

3.1.3.2~3.1.3.4 の改善によって、検証ケース（格子点数 128×128 ）では最終的におよそ 4.2 倍高速化することが出来た。そこで格子点数を変化させて SRT とコード改善した Cascaded LBM の解析時間の比較を行い、その結果を図 7 に示す。コード改善によりコストの高い Cascaded LBM が SRT と比較して、元々 5.5 倍であった解析コストが高々 1.25 倍程度にまで削減出来ていることが分かった。また、解析コスト増分は格子点数によらずほぼ一定であった。コード改善無しの 3 次元コードの場合、Cascaded LBM の解析コストは SRT の約 13 倍程度となっており（図 8）、今後は 2 次元におけるコード改善の結果を反映して 3 次元における性能調査を実施する予定である。

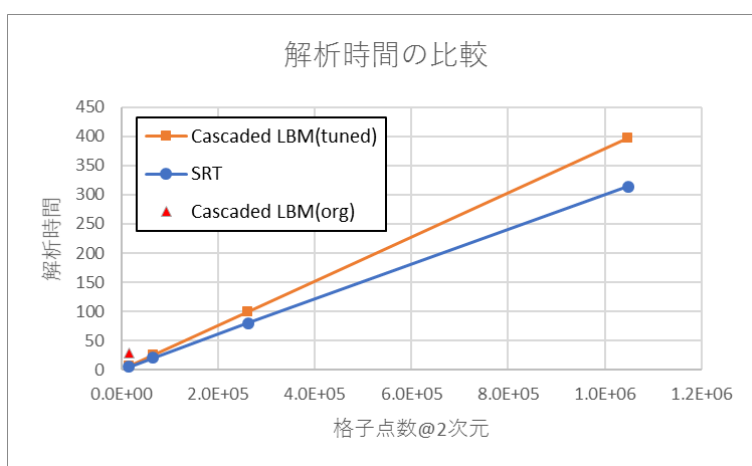


図 7 SRT と Cascaded LBM の解析時間の比較（2 次元）

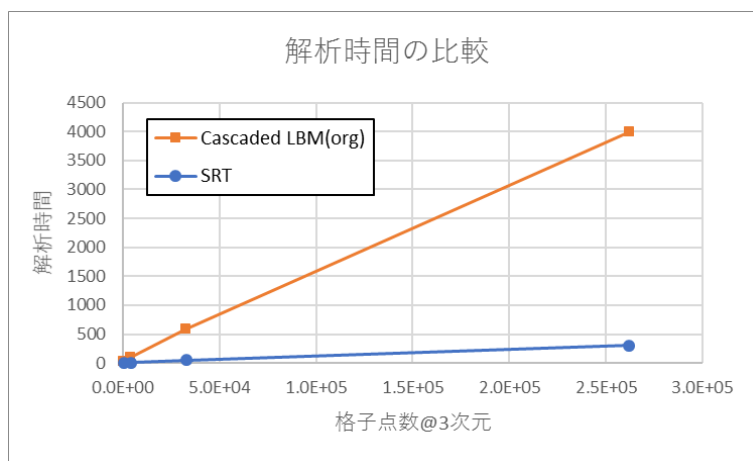


図 8 SRT と Cascaded LBM の解析時間の比較（3 次元）

3.1.4 スレッド並列性能

FX100 1node を用いて行ったスレッド並列性能調査の結果について以下に報告する。性能評価にはC++で記述された3次元27速度(D3Q27)モデルのプログラムを用い、衝突項にはSRTを採用した。計算格子には $64block \times 32^3cell$ のマルチブロック直交格子を用いて、スレッド数を 1^{32} に変化させて実行時間の変化を調べ、その結果を図9に示す。またその際のプロファイラの結果を図10に示すが、並列計算時のホットスポットはブロック間で情報通信を実施するinterfaceQという関数であった。LBMはNavier-Stokes方程式に比べて計算コストが小さいが使用変数が多く、コード改善によって演算に比べて通信がより際立つ結果となっている。今後は通信に関係する時間積分の箇所を分離する等、通信コストをうまく隠蔽できる実装に変更して性能調査を実施する予定である。

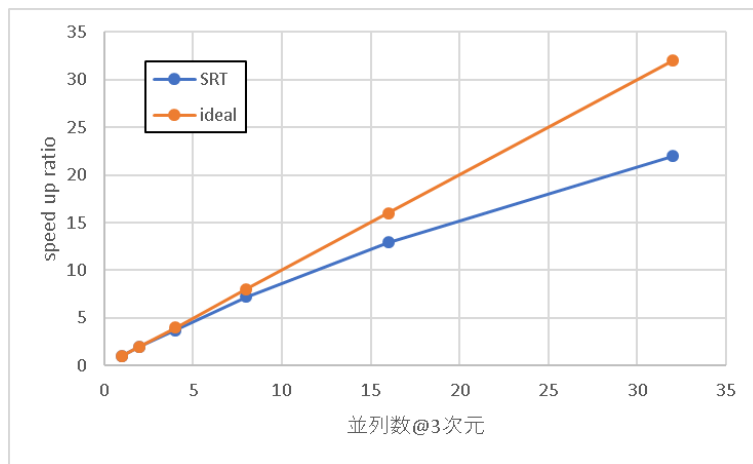


図9 FX100 1node を用いたスレッド並列性能

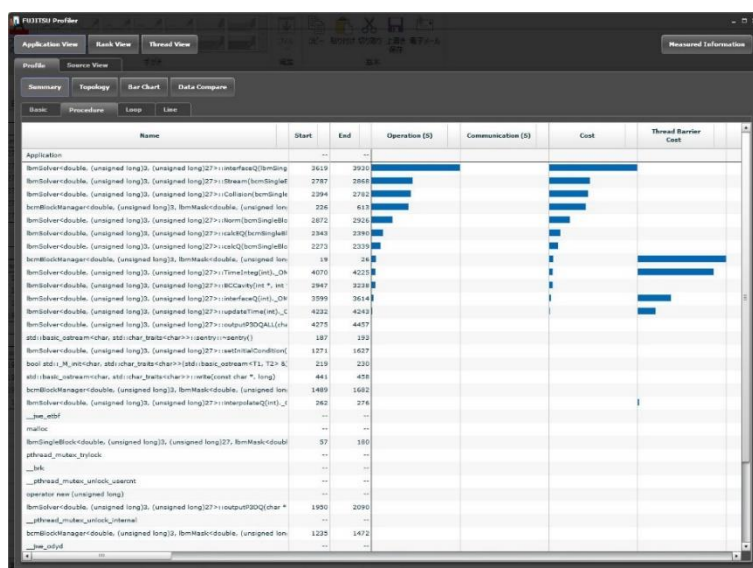


図10 プロファイラによるホットスポットの抽出

3.1.5 まとめ

メニーコア WG での JAXA LBM コードの性能調査の結果、以下の知見が得られた。

- central moment に関わる行列演算は、2次元の場合はSIMD化を促進するより書き下した方が性能向上が期待できる。
- SRTと比較して、コード改善したCascaded LBMの解析コストは2次元で約1.25倍程度にまで削減することが出来た。3次元の場合は 27×27 の行列を書き下すことになり、演算量や演算の複雑さが増すため、調査する必要がある。
- LBMはNavier-Stokes方程式ベースのアプローチより格子点当りの演算量は少ないが、格子点当りの変数が2次元では9変数(NSでは4変数+ α)、3次元では27変数(NSでは5変数+ α)であり、ノード内・ノード間の情報通信量が相対的に増えるため、ブロック間の情報通信を隠ぺいする等の処置が必要である。
- 現状では変数配列をダブルポインタで管理しているが、アーキテクチャやコンパイラの観点でどのようなデータの持たせ方が良いか (SoA or AoS)、検討する必要がある。

3.2 MUTSU コード高速化の検討

核融合科学研究所 三浦英昭

3.2.1 はじめに

MUTSU コードは、核融合・トーラスプラズマの不安定性研究や、これに関わる基礎研究のために使用する汎用流体コードである。MUTSU コードでは、基礎研究用の矩形形状（周期境界、非周期境界条件を含む；直角座標系を使用）や、核融合研究向けのトーラス構造を近似的に表現するための歪んだ矩形形状（この場合には非直交曲線座標系で数式を表現する）、さらには直角座標や非直交曲線座標領域を組み合わせた座標において、拡張 MHD 方程式や Navier-Stokes 方程式を解くことを想定している。対象となる方程式によって使用するモジュールを変える（コード名称も、方程式に応じて MUTSU/cXMHD3D, MUTSH/cNS3D などと名前を変える）が、空間微分や時間発展パートは共通となっている。

MUTSU コードは、入力パラメータで二次精度中心差分、三次精度風上差分(KK スキーム)、四次精度中心差分、六次精度コンパクト差分、八次精度コンパクト、十次精度コンパクト差分を切り替えるように設計されている。数値フィルターも、陽的なフィルターから十次精度コンパクトフィルターまでをサポートしている。また、ほぼ同じ構造のモジュールを使って、3方向に周期的な場合のフーリエ・擬スペクトル法によるシミュレーションが可能になっている。（こちらは、コード名末尾に”-T3”をつける。）また、矩形格子形状で複雑物体周りの流れを扱うために、volume penalization 用のモジュールも用意されている。

このコードの性能は、方程式の選択や微分・フィルターに関わるスイッチの選択で性能が大きく変わるため、ここではコンパクト差分法のパートと、MUTSU-T3 で使う 3次元 FFT のパートについて報告する。

3.2.2 MUTSU, MUTSU-T3 コードの開発・コンパイル

このコードの開発は、主に核融合科学研究所の”プラズマシミュレータ”（富士通株式会社製 FX100）で行っている。また、一部の開発は、東京大学の Oakforest-PACS（富士通株式会社製）で行っている。FX100 における主要なコンパイルオプションは、以下の通りである。

```
FC      = mpifrtpx
```

```
FFLAGS = -Cpp -Cfpp -Kfast -Kreduction -Kopenmp -Kocl -Kparallel -Koptmsg=2 -Qt
```

また、Oakforest-PACS での主要なコンパイルオプションは以下のとおりである。

```
FC      = mpiifort
```

```
FFLAGS = -fpp -O2 -axMIC-AVX512 -qopt-report -qopenmp  
-mcmmodel=medium -convert big_endian
```

3.2.3 コンパクト差分パートの高速化

コンパクト差分では、以下のような3重もしくは5重対角連立方程式を解く必要がある。

$$\begin{aligned}\beta f'_{i+2,j,k} + \alpha f'_{i+1,j,k} + f'_{i,j,k} + \alpha f'_{i-1,j,k} + \beta f'_{i-2,j,k} \\ &= c(f_{i+3,j,k} - f_{i-3,j,k}) + b(f_{i+2,j,k} - f_{i-2,j,k}) + a(f_{i+1,j,k} - f_{i-1,j,k}) \\ \beta f'_{i,j+2,k} + \alpha f'_{i,j+1,k} + f'_{i,j,k} + \alpha f'_{i,j-1,k} + \beta f'_{i,j-2,k} \\ &= c(f_{i,j+3,k} - f_{i,j-3,k}) + b(f_{i,j+2,k} - f_{i,j-2,k}) + a(f_{i,j+1,k} - f_{i,j-1,k}) \\ \beta f'_{i,j,k+2} + \alpha f'_{i,j,k+1} + f'_{i,j,k} + \alpha f'_{i,j,k-1} + \beta f'_{i,j,k-2} \\ &= c(f_{i,j,k+3} - f_{i,j,k-3}) + b(f_{i,j,k+2} - f_{i,j,k-2}) + a(f_{i,j,k+1} - f_{i,j,k-1})\end{aligned}$$

MUTSU コードにおいてこれらの微分評価を行うサブルーチンは、上記の方程式3セットの右辺の計算と、連立方程式の求解パートで構成される。右辺項の計算は、中心差分法などと同じ構造なので、比較的容易に、そこそこ高速な性能 (FX100 でピーク性能比 6-8%前後) を達成可能である。連立方程式の解法としては、ここでは LU 分解法を採用した。(メニーコア WG の会合では、問題規模から考えて、行列反転型の直接解法の方が速いとの指摘があった。)

八次精度のコンパクト差分法を使用した場合 (上の式で係数 $\beta=0$) について、LU 分解パートのループ融合などの最適化を行った。具体的には、

- ベクトル3成分の勾配 (速度勾配テンソル) など、複数のコンパクト差分を一括して計算することによって流量を増加
- LU 分解で使用する作業配列の次元を増やし、ループ融合
たとえば、入力データ b1, b2, b3 について

```
do k = 1, n2
do I = 1, n1
  dm(1, k, i) = b1(i, k, ijk)
  dm(2, k, i) = b2(i, k, ijk)
  dm(3, k, i) = b3(i, k, ijk)
end do
end do
```

などの処理について、事前に入力データ b1, b2, b3 を一つの配列 b にまとめ、

```
do k = 1, n2
do i = 1, n1
do j = 1, 3
  dm(j, k, i) = b(i, k, ijk, j)
end do
end do
end do
```

とする

- 融合配列の次元の入れ替え (最内側 \leftrightarrow 再外側) を行う

などの工夫を施した。この結果、ピーク性能比で 4-9%を実現した。なお、配列の第 2 成分(x, y, z の直角座標では y 成分)に関する微分操作については、ストライドアクセスが発生することによる速度の低下のため、3 方向についての微分の中では最も低速な演算(ピーク性比 4%)となった。

3.2.4 3次元 FFT の通信時間隠蔽による周期境界用コード MUTSU-T3 コードの高速化

3 方向に対して周期境界条件が課される流体シミュレーションでは、中心差分法やコンパクト差分法に代えて、擬スペクトル法を使うことが多い。以下は、非圧縮性 Hall MHD シミュレーション版(MUTSU /iHallMHD3D-T3)の場合についての報告である。

擬スペクトル計算の場合、3次元 FFT がコードの性能を決定づける。3次元 FFT では通信・データの転置が計算時間のかなりの部分を占めるため、スーパーコンピュータがその性能を十全に発揮するのは難しい。ここでは、3次元高速フーリエ変換ライブラリ“FFTE”(<http://www.ffte.jp>)をベースに、そのパーツを分解し、複数のフーリエ変換を束ねることで計算による通信時間の隠ぺいを図った。なお、この隠蔽によるシミュレーションコードの高速化は JHPCN 平成 30 年度共同研究課題にもなっており、共同研究者・FFTE の開発者である高橋大介博士から FFTE Ver. 6.0 および 6.2 α の提供を受けて行った。

この研究における通信時間の隠ぺいは、流体コードが往々にして保存形式で記述されることを利用している。たとえば速度勾配テンソルの演算は速度場 3 成分それぞれの勾配を求めることになり、3つのフーリエ変換を 3セット行うと考えることができる。1回のフーリエ変換は、一方向へのフーリエ変換、転置・通信を 3回繰り返すことで構成されるため、一つの変数のフーリエ変換と他の変数の通信・転置を重ねることで、通信時間をある程度隠蔽できると考えられる。

3.2.5 3次元 FFT における計算時間の隠ぺい結果

計算時間の隠ぺいを行う場合と行わない場合の MUTSU /iHallMHD-T3 における 1 ステップあたりの計算時間を、FX100 および Oakforest-PACS で測定した。この隠蔽の際の関数の呼び出しは、もう一つの著名な 3次元 FFT パッケージである P3DFFT (<https://www.p3dfft.net>) において、p3dfft_ftran_r2c_many, p3dfft_ftran_c2r_many といった名前の末尾に”_MANY”がつく関数と対応がつくため、P3DFFT と FFTE の比較も行った。

この計測結果は下表(表 1, 表 2)のとおりである。表 1 は FX100、表 2 は Oakforest-PACS で計測した結果である。表中で「(通常版)」は、3次元 FFT を個別にコールする場合を示している。P3DFFT の項目で「(_MANY)」は、p3dfft_ftran_r2c_many, p3dfft_ftran_c2r_many を用いて複数の変数に対する一括フーリエ変換を行う場合を示す。FFTE の「(隠蔽版)」項は、FFTE のコードを使用して複数のフーリエ変換(ここでは 3 変数)を行いつつ、通信について隠蔽を行った場合を示している。FX100 上での計測においては、アシスタントコアを使用していない。この点において、FX100 に特化した最適化によってさらに改善をもたらす余地は残っている。

この表からわかる通り、多くの場合において P3DFFT よりも FFTE の通常版の方が短時間で演算を終えていることがわかる。特に表 2 では、P3DFFT の _MANY 版の方が通常版よりも

7-30%高速であること、FFTE では通常版よりも隠蔽版が 11%程度高速であるが、最大規模の計算 ($N^3=4096^3$) では通常版の方が速い。最後の点についてはさらに調査が必要であり、また、P3D は比較的スレッド並列化を苦手とする (フラット MPI に近いほど、性能が顕著に高い) ためにスレッド並列数についての精査が必要であるなど、いくつかの課題は残る。しかし、概ね所期の目標を達成することができたと考えられる。他方、FX100 に比べて、Oakforest-PACS での性能はやや低いものとなっている。この点から、Oakforest-PACS についての最適化をさらに進める必要がある。

表 1. 核融合科学研究所 F100”プラズマシミュレータ”における 3 次元 FFT の通信時間の隠ぺい(MUTSU / iHallMHD3D-T3 の 1 ステップに要する時間 ; 単位[秒])

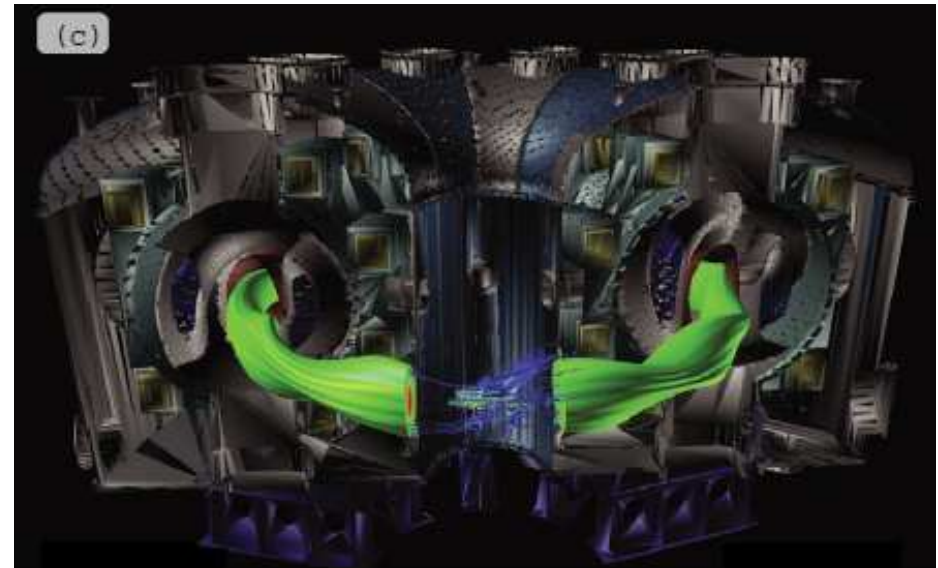
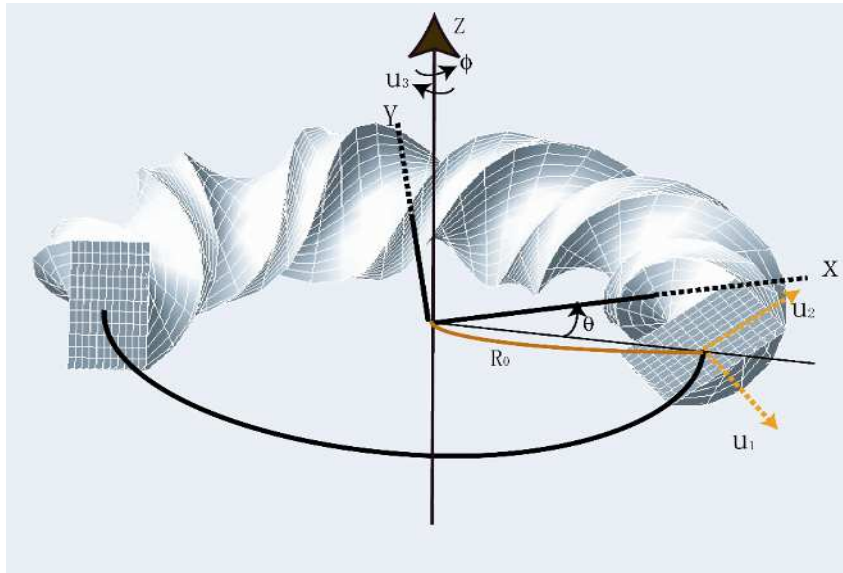
N^3	Nodes (Processes)	P3DFFT (_MANY)	FFTE(通常 版)	FFTE(隠蔽あ り)
2048 ³	512 (1024)	37.939	—	24.990
1024 ³	128 (256)	14.784	14.094	12.584
512 ³	32 (64)	5.000	4.846	3.742
256 ³	8 (16)	2.276	3.096	1.751

表 2. Oakforest-PACS における 3 次元 FFT の通信時間の隠ぺい(MUTSU / iHallMHD3D-T3 の 1 ステップに要する時間 ; 単位[秒])

格子点数	Nodes (Processes)	P3DFFT (通常版)	P3DFFT (_MANY)	FFTE (通常版)	FFTE (隠蔽版)
$N^3=4096^3$	1024(16384)	327.221	306.214	231.546	247.666
$N^3=2048^3$	512(16384)	42.553	38.879	30.668	27.524
$N^3=1024^3$	128(4096)	18.573	17.844	13.408	12.867
$N^3=512^3$	32(1024)	8.662	8.228	5.398	4.906
$N^3=256^3$	8(256)	4.321	2.700	2.700	2.550

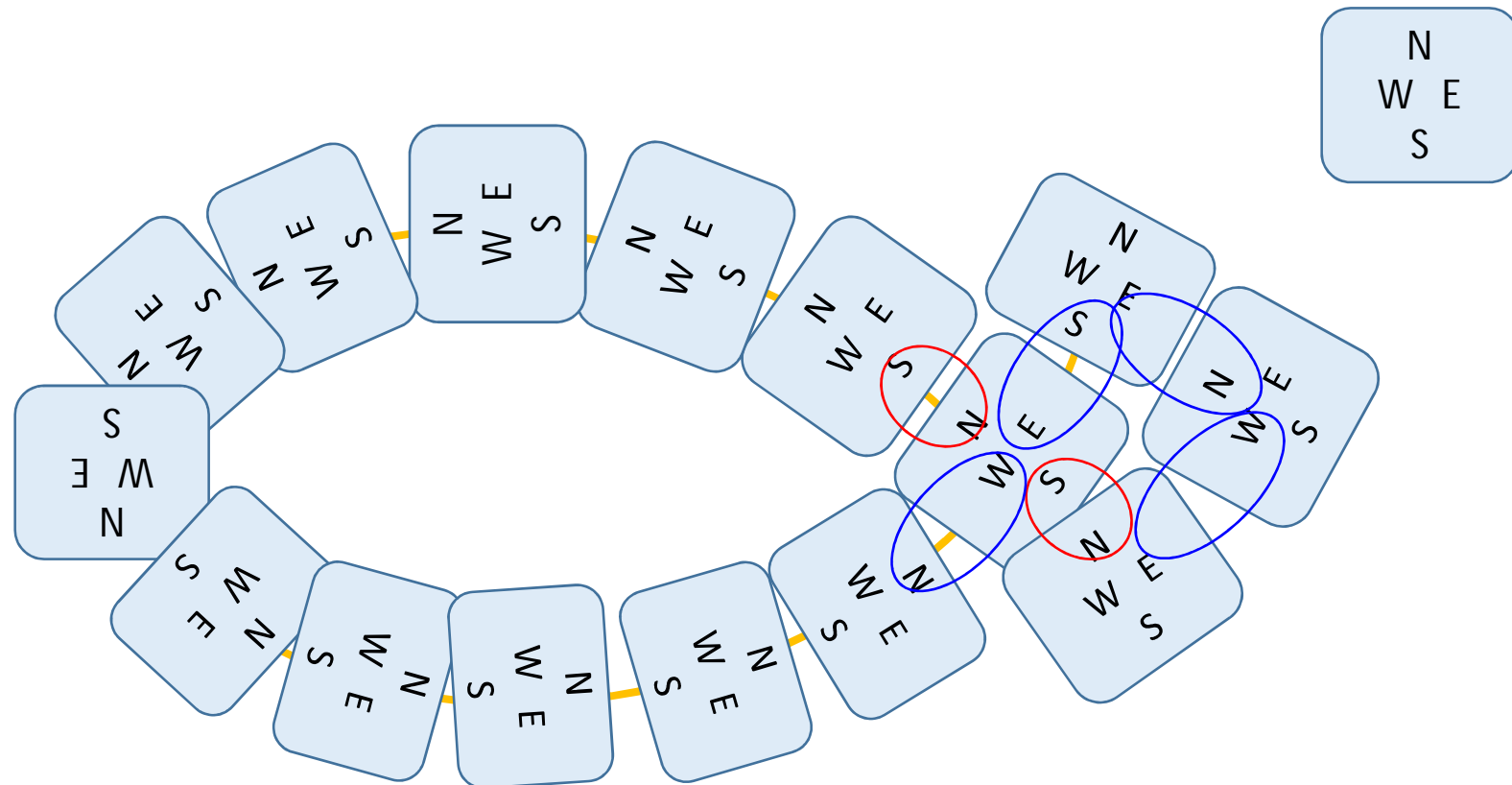
コード開発の目的

- 流体現象に関わる様々なシミュレーションを、矩形格子と高次差分で実行する
- 直角座標、
極座標等直交座標、
非直交曲線座標
- 周期境界・非周期境界



コード開発の経緯と現在の方針

- AMR コードとしてスタート
- AMR 部分はpending
- AMR構造を活かした柔軟な構造による多目的化 (たとえば C-grid, O-gridの変形など)を優先



コンパクト差分の実装

ベクトル3成分それぞれのgradientの計算
(多成分化による流量の増加)



```
subroutine compact_der3uniXYZ(a1, a2, a3, b1x, b1y, b1z,
& b3x, b3y, b3z, dx, dy, dz, iaflag1, iaflag2, iafla
g3)
  call compact_der_XYZ(a1, b1x, b1y, b1z, dx, dy, dz, iaflag1, iaflag2, iafla
g3)
  call compact_der_XYZ(a2, b2x, b2y, b2z, dx, dy, dz, iaflag1, iaflag2, iafla
g3)
  call compact_der_XYZ(a3, b3x, b3y, b3z, dx, dy, dz, iafla
g3)
  ifg = 1
  call Banbks3D_3vars (NWx,NWx,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
& work_Current%indx1,work_Current%ar1,work_Current%ar1L,
& b1x,b2x,b3x, ifg)
  ifg = 2
  call Banbks3D_3vars (NWy,NWy,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
& work_Current%indx2,work_Current%ar2,work_Current%ar2L,
& b1y,b2y,b3y, ifg)
  ifg = 3
  call Banbks3D_3vars (NWz,NWz,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
& work_Current%indx3,work_Current%ar3,work_Current%ar3L,
& b1z,b2z,b3z, ifg)
  return
end subroutine compact_der3uniXYZ
```

ベクトル3成分それぞれのコンパクト差分右辺計算
(中心差分と似た構造なので、この部分の効率は低くない)

3成分のコンパクト公式左辺のx,y,z一括LU分解

X-方向(Case1)

Y-方向(Case2)

Z-方向(Case3)

```

!option mp(p(0)), langlvl(save(0))
subroutine Banbks_2D_3vars (n,np,n1,n2,n3,m1,m2,mp,al,mpl,indx,ifg,ijk,a,b1,b
2,b3)
  real(dd), dimension(n1,n2,n3) :: b1, b2, b3

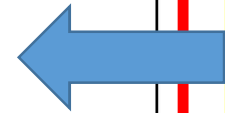
  select case (ifg)

  case (1)
!$omp parallel do default(none) &
!$omp&private(ijk,kk,ii,in2,i,k,l,dm1,dm1a_R,dm1b_R,dm1c_R,aval,ainv) &
!$omp&shared(n1,n2,n3,b1,b2,b3,a,al,indx,mm,m1)
    do ijk=1,n3

    do ii=1,n1,n1/NN
      do kk=1,n2,n2/NN
        do i=ii,min(ii+n1/NN-1,n1)
          do k=kk,min(kk+n2/NN-1,n2)
            dm1(1,k,i) = b1(i,k,ijk)
            dm1(2,k,i) = b2(i,k,ijk)
            dm1(3,k,i) = b3(i,k,ijk)
          enddo
        enddo
      enddo
    enddo
  enddo

```

Case1:
 $b(i,j,k)$
 ↑この成分への入出力
 (外側kで並列化可能)



```

  case (2)
!$omp parallel do default(none) &
!$omp& private(dm2a_R,dm2b_R,dm2c_R,l,k,ijk,i,aval,ainv,in1) &
!$omp& shared(n1,n2,n3,b1,b2,b3,m1,mm,a,al,indx)
    do ijk=1,n3

      l = m1
      do k = 1, n2
        i = indx(k)

        if(i.ne.k) then
          do in1 = 1, n1
            dm2a_R(in1) = b1(in1,k,ijk)
            b1(in1,k,ijk) = b1(in1,i,ijk)
            b1(in1,i,ijk) = dm2a_R(in1)

            dm2b_R(in1) = b2(in1,k,ijk)
            b2(in1,k,ijk) = b2(in1,i,ijk)
            b2(in1,i,ijk) = dm2b_R(in1)

            dm2c_R(in1) = b3(in1,k,ijk)
            b3(in1,k,ijk) = b3(in1,i,ijk)
            b3(in1,i,ijk) = dm2c_R(in1)
          end do
        end if
      end do
    end do

```

Case2:
 $b(i,j,k)$
 ↑この成分への入出力
 (ストライドアクセスの発生)



```

  case (3)
!$omp parallel do default(none) &
!$omp& private(dm3a_R,dm3b_R,dm3c_R,l,k,i,ijk,aval,ainv,in1) &
!$omp& shared(n1,n2,n3,b1,b2,b3,m1,mm,a,al,indx)
    do ijk=1,n2

      l = m1
      do k = 1, n3
        i = indx(k)

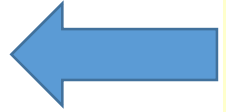
        if(i.ne.k) then
          do in1 = 1, n1
            dm3a_R(in1) = b1(in1,ijk,k)
            b1(in1,ijk,k) = b1(in1,ijk,i)
            b1(in1,ijk,i) = dm3a_R(in1)

            dm3b_R(in1) = b2(in1,ijk,k)
            b2(in1,ijk,k) = b2(in1,ijk,i)
            b2(in1,ijk,i) = dm3b_R(in1)

            dm3c_R(in1) = b3(in1,ijk,k)
            b3(in1,ijk,k) = b3(in1,ijk,i)
            b3(in1,ijk,i) = dm3c_R(in1)
          end do
        end if
      end do
    end do

```

Case3:
 $b(i,j,k)$
 ↑この成分への入出力
 (内側jで並列化可能)



コンパクト差分パート最適化

- 最適化前の高コスト部の状態

高コストルーチン	SIMD化	SWP化	自動並列化	経過時間 (コード全体200ステップ 898秒)
Banbks_2d_3vars	○			74.91
Banbks_2d_u2	○			61.91
Compact_der_xyz	○	○	○	12.80
Compact_der_z1	○	○		3.54

- OpenMP化が速い場合とFX100自動並列化の方が速い場合が混在 (ループ長に関する分岐判断のため)
全ループをOpenMP化すると低速化するので、個別に切り分け
- キャッシュミス等の分析と配列融合で最適化

配列融合について (1-1)

• banbks_2d_3vars の case 1 (修正前)
real(dd), dimension(n1,n2,n3) :: b1, b2, b3

```
do k = 1, n2
  do i = 1, n1
    dm(1,k,i) = b1(i,k,ijk)
    dm(2,k,i) = b2(i,k,ijk)
    dm(3,k,i) = b3(i,k,ijk)
  enddo
enddo
i = m1
do k = 1, n1
  i = indx(k)

  if(i.ne.k) then
    do in2 = 1, n2
      adm1 = dm(1,in2,k)
      dm(1,in2,i) = adm1
      adm2 = dm(2,in2,k)
      dm(2,in2,i) = adm2
      adm3 = dm(3,in2,k)
      dm(3,in2,i) = adm3
    end do
  end if
enddo
```

(中略)

```
do k = 1, n2
  do i = 1, n1
    b1(i,k,ijk) = dm(1,k,i)
    b2(i,k,ijk) = dm(2,k,i)
    b3(i,k,ijk) = dm(3,k,i)
  enddo
enddo
```

LU分解 banbks_*_3vars ... 3変数一括変換

配列融合前

Real(dd),dimension(n1,n2,n3)::b1,b2,b3

配列融合後 (内側)

Real(dd),dimension(3,n1,n2,n3)::b

配列融合後 (外側)

Real(dd),dimension(n1,n2,n3,3)::b

配列融合について (1-2)

最内で融合

• banbks_2d_3vars の case 1 (修正前)
real(dd), dimension(n1,n2,n3) :: b1, b2, b3

```
do k = 1, n2
  do i = 1, n1
    dm(1,k,i) = b1(i,k,ijk)
    dm(2,k,i) = b2(i,k,ijk)
    dm(3,k,i) = b3(i,k,ijk)
  enddo
enddo
l = m1
do k = 1, n1
  i = indx(k)

  if(i.ne.k) then
    do in2 = 1, n2
      adm1 = dm(1,in2,k)
      dm(1,in2,i) = dm(1,in2,i)
      adm2 = dm(2,in2,k)
      dm(2,in2,i) = dm(2,in2,i)
      adm3 = dm(3,in2,k)
      dm(3,in2,i) = dm(3,in2,i)
    end do
  end if
```

(中略)

```
do k = 1, n2
  do i = 1, n1
    b1(i,k,ijk) = dm(1,k,i)
    b2(i,k,ijk) = dm(2,k,i)
    b3(i,k,ijk) = dm(3,k,i)
  enddo
enddo
```



• banbks_2d_3vars の case 1 (修正後-最内で配列融合)
real(dd), dimension(3,n1,n2,n3) :: b

```
do k = 1, n2
  do i = 1, n1
    do j=1,3
      dm(j,k,i) = b(j,i,k,ijk)
    enddo
  enddo
  l = m1
do k = 1, n1
  i = indx(k)

  if(i.ne.k) then
    do in2 = 1, n2
      adm1 = dm(1,in2,k)
      dm(1,in2,i) = dm(1,in2,i)
      adm2 = dm(2,in2,k)
      dm(2,in2,i) = dm(2,in2,i)
      adm3 = dm(3,in2,k)
      dm(3,in2,i) = dm(3,in2,i)
    end do
  end if
  if(l.lt.n1) l = l + 1
  do i = k+1, l
    aval = a1(k,i-k)
    do in2 = 1, n2
      dm(1,in2,i) = dm(1,in2,i) - aval * dm(1,in2,k)
      dm(2,in2,i) = dm(2,in2,i) - aval * dm(2,in2,k)
      dm(3,in2,i) = dm(3,in2,i) - aval * dm(3,in2,k)
    end do
  end do
end do
```

(中略)

```
do k = 1, n2
  do i = 1, n1
    do j=1,3
      b(j,i,k,ijk) = dm(j,k,i)
    enddo
  enddo
enddo
```

配列融合について (1-3)

最外で融合

• banbks_2d_3vars の case 1 (修正前)

```
real(dd), dimension(n1,n2,n3) :: b1, b2, b3
do k = 1, n2
  do i = 1, n1
    dm(1,k,i) = b1(i,k,ijk)
    dm(2,k,i) = b2(i,k,ijk)
    dm(3,k,i) = b3(i,k,ijk)
  enddo
enddo
i = m1
do k = 1, n1
  i = indx(k)
  if(i.ne.k) then
    do in2 = 1, n2
      adm1 = dm(1,in2,k)
      dm(1,in2,i) = dm(1,in2,k)
      dm(1,in2,i) = adm1
      adm2 = dm(2,in2,k)
      dm(2,in2,i) = dm(2,in2,k)
      dm(2,in2,i) = adm2
      adm3 = dm(3,in2,k)
      dm(3,in2,i) = dm(3,in2,k)
      dm(3,in2,i) = adm3
    end do
  end if
```

(中略)

```
do k = 1, n2
  do i = 1, n1
    b1(i,k,ijk) = dm(1,k,i)
    b2(i,k,ijk) = dm(2,k,i)
    b3(i,k,ijk) = dm(3,k,i)
  enddo
enddo
```



• banbks_2d_3vars の case 1 (修正後-最外で配列融合)

```
real(dd), dimension(n1,n2,n3,3) :: b
do k = 1, n2
  do i = 1, n1
    do j = 1, n1
      dm(j,k,i) = b(i,k,ijk,j)
    enddo
  enddo
  i = m1
do k = 1, n1
  i = indx(k)
  if(i.ne.k) then
    do in2 = 1, n2
      adm1 = dm(1,in2,k)
      dm(1,in2,i) = dm(1,in2,k)
      dm(1,in2,i) = adm1
      adm2 = dm(2,in2,k)
      dm(2,in2,i) = dm(2,in2,k)
      dm(2,in2,i) = adm2
      adm3 = dm(3,in2,k)
      dm(3,in2,i) = dm(3,in2,k)
      dm(3,in2,i) = adm3
    end do
  end if
```

(中略)

```
do j=1,3
  do k = 1, n2
    do i = 1, n1
      b(i,k,ijk,j) = dm(j,k,i)
    enddo
  enddo
enddo
```


配列融合について (1-4)

結果比較(1ノード4プロセス500回コール時間)

Banbks_2D_3varsの実行時間

	Case 1	Case 2	Case 3	計(秒)	改善率
修正前	3.24	1.44	1.90	6.59	-
最内	3.17 (4.09%)	1.53 (9.09%)	1.74 (6.52%)	6.43	2.30%
最外	3.63	1.45	1.90	6.99	-6.12%

Banbks_2D_u2 (Banbks_2D_3vars類似・変数長偶数限定) 実行時間

	Case 1	Case 2	Case 3	計(秒)	改善率
修正前	0.76	0.39	0.66	1.81	-
最内	0.94	0.72	0.97	2.63	-45.01%
最外	0.75	0.39	0.66	1.80	0.59%

Case 1-3の実行回数にも依存するが、最内がやや有利

非保存的な方程式などでは、高速化の保証がない

Case 1 がコンパクト差分最適化 (ピーク性能比向上) のための問題点

3次元FFT ‘FFTE ’を用いた 計算時間の隠蔽

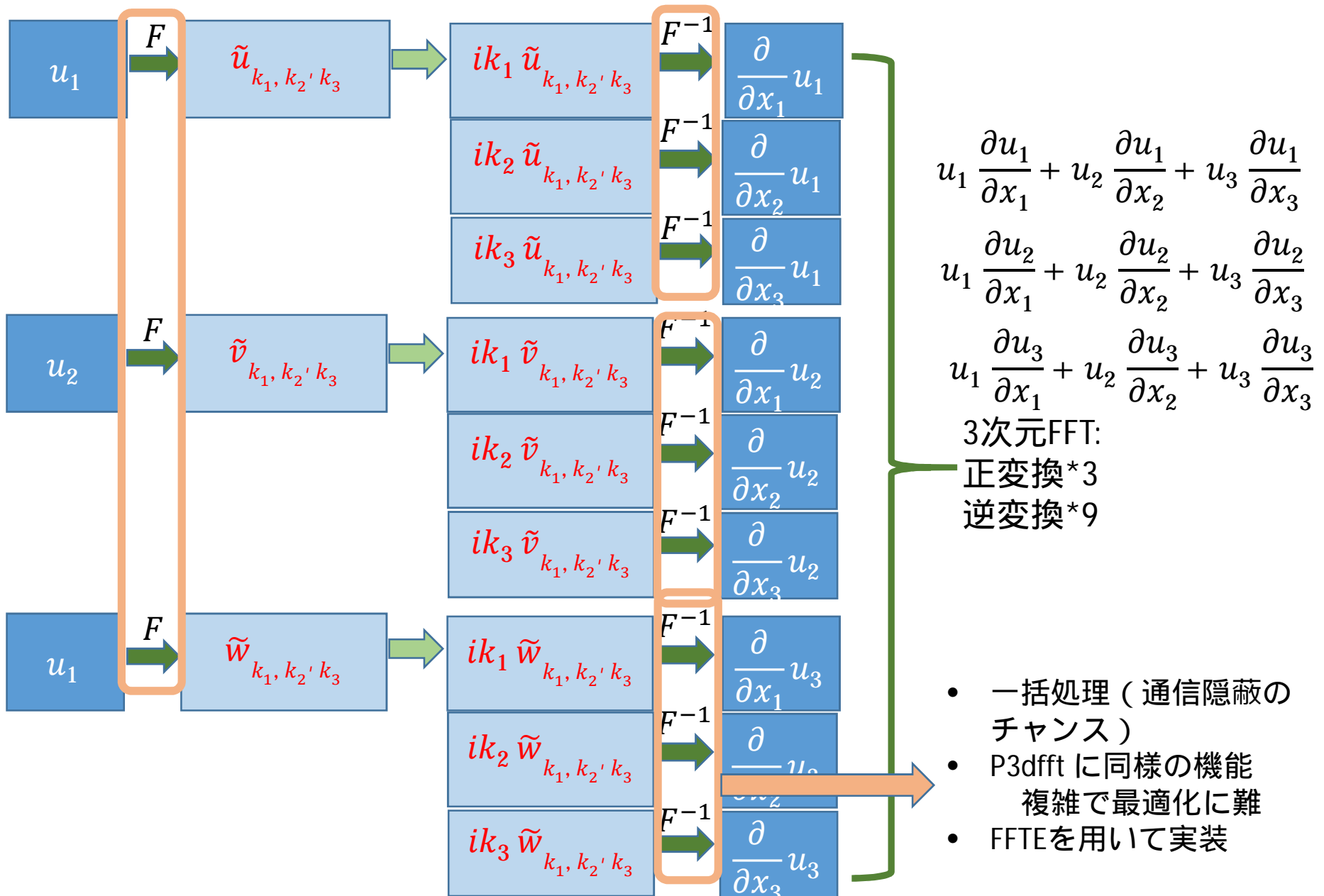
- 平成30年度JHPCN 共同研究課題
「電磁流体力学乱流の高精度・高並列LESシミュレーションコード開発研究」
共同研究者：高橋大介先生（筑波大）他
- 3次元FFT ... FFTE (<http://www.ffte.jp>) ver 6.0
- 基本方針
3次元FFT を分解
多重FFT=>転置 多重FFT=>転置 多重FFT
通信と計算・転置のオーバーラップ
* p3dfft の _MANY 機能と同様

FX100性能測定詳細（ループ交換等最適化前後比較）

	Elapsed(secs)			1次キャッシュミス率(%)		
	変更前	変更後	改善率	変更前	変更後	改善率
FFTINVXL1	42.56	34.00	20.1%	29.97%	26.94%	10.1%
FFTINVYL1	31.14	26.74	14.1%	30.32%	26.93%	11.2%
FFTINVZL1	36.86	29.26	20.6%	30.20%	26.96%	10.7%

nproc	変更	timesteploop	正変換	逆変換	a) 正変換 (1回)	b) 逆変換 (1回)	比(b/a)
256	前	782.4	247.8	412.5	1.2388	1.4029	113.2%
	後	757.2	240.0	394.7	1.2002	1.3423	111.8%
512	前	491.1	161.8	289.8	0.8088	0.9859	121.9%
	後	480.8	158.3	281.9	0.7916	0.9589	121.1%
1024	前	249.6	88.2	145.8	0.4409	0.4958	112.5%
	後	244.7	86.5	142.6	0.4323	0.4851	112.2%
2048	前	105.9	35.3	62.7	0.1766	0.2132	120.7%
	後	105.4	34.6	62.9	0.1732	0.2138	123.5%
4096	前	76.5	26.4	46.4	0.1319	0.1579	119.8%
	後	76.3	26.2	46.3	0.1309	0.1576	120.4%

擬スペクトル計算における“_MANY”



隠蔽効果の検証 (FX100)

測定データ 格子数 : 2048x2048x2048

ステップ数 : 10

スレッド数 : 16

Time step loop							
	P3dfft 2.7.5		ffte(Xペンシル)		ffte(Zペンシル)		
nproc	通常版	many版	通常版	隠蔽版	通常版	隠蔽版	
256	2283.016	2138.858	1577.800	1139.746	1092.112	821.853	
512	1036.223	986.424	1160.650	973.610	762.255	549.588	
1024	482.646	428.720	511.279	349.656	342.041	246.963	
2048	198.744	199.216	201.124	135.184	132.365	93.802	
4096	88.054	89.913	140.027	106.548	86.406	61.901	

- 川島康弘氏 (富士通(株)) による計測
- 注 : 現在 FFTE 6.0 6.2α移行、隠蔽版は若干速度低下(最適化中)

隠蔽効果の検証(Oakforest-PACS)

- ライブラリ

fftw ... fftw 3.3.6-pl2

p3dfft ... p3dfft3.7.7

ffte ... fftw ver 6.0

- コンパイル

mpiifort -fpp -O2

-axMIC-AVX512 -qopt-report -qopenmp

-mcmmodel=medium -convert big_endian

隠蔽効果の検証(Oakforest-PACS)

		Time for 1-time-step (secs)			
		P3dffft		FFTE (z-pencil)	
	Node,procs	通常版	MANY版	通常版	MANY版
$N^3=4096^3$	1024, 16384	327.221	306.214	231.546	247.666
$N^3=2048^3$	512, 16384	42.553	38.879	30.668	27.524
$N^3=1024^3$	128, 4096	18.573	17.844	13.408	12.867
$N^3=512^3$	32,1024	8.662	8.228	5.398	4.906
$N^3=256^3$	8,256	4.321	2.700	2.700	2.550

FFTE $N^3=4096^3$ で通常版とMANY版の計算時間の逆転については、要調査。

3.3 生体分子粗視化シミュレータ CafeMol の FX100 での性能測定とチューニング

理化学研究所 情報システム部

検崎博生

富士通株式会社 TC ソリューション事業本部

渡邊健太

3.3.1 はじめに

CafeMol はタンパク質や核酸のような生体分子の粗視化シミュレータである。CafeMol で用いる粗視化モデルでは、全原子モデルに比べて粒子数が 100 分の 1 程度まで減り、計算量が大幅に減る。このことにより長時間の時間発展の計算が可能になり、生体分子の大規模な構造変化を取り扱うことができるのであるが、同時に並列性能を出しにくいことにもつながっている。具体的には、時間発展の 1 step が 1 ms 以下になることも多く、ノード間通信が問題になってくるので複数ノードでの並列計算を行うことは難しく、1 ノードでの性能向上を図ることが大事になってきている。

本 WG では、CafeMol の性能測定とチューニングについて検崎による 2 回の発表と富士通渡邊による 1 回の発表があったので、ここではそれらをまとめることとする。最初に検崎による 1 回目の発表では、FX100 と skylake の比較と計算量が一番多い部分について配列の構造を Structure of Array (SoA) から Array of Structure (AoS) への変更を行うチューニングを行った。次に、渡邊の発表では FX100 が Skylake に比べて性能が出ていない部分について性能測定とチューニングが行われた。最後に、検崎による 2 回目の発表では、渡邊により示されたタイマーのオーバーヘッドの確認と追加のチューニングを行った。

3.3.2 プログラム概要

分子動力学シミュレーションでは、分子の動きを時間積分することにより系を時間発展させるわけであるが、分子間に働く力の計算が最も計算時間が掛かる部分となっている。CafeMol では力の計算にネイバリングリスト方式を用いて計算量を減らしており、ネイバリングリストを分割する形で MPI と OpenMP によるハイブリッド並列化を行っている。

粗視化された粒子間に働く相互作用はさまざまなものがあり、相互作用毎に異なるチューニングが必要になってくる。ここではタンパク質/DNA 複合体であるヌクレオソームの 10,000 step の時間発展の性能測定を行う。ヌクレオソームは粗視化粒子数は 2,000 程度の系で、計算量が一番多いのが静電相互作用の計算で粒子数の 2 乗程度の計算量であり、それ以外に粒子数の数倍程度の計算量であるさまざまな相互作用がある。

3.3.3 測定環境

検崎による性能測定には、理化学研究所の HOKUSAI システムを使用した。

- FX100 : SPARC64TMXIfx(1.975H)、富士通コンパイラ、-Kfast
 - Intel Xeon : Intel Xeon Gold 6148(3.1GHz)、インテルコンパイラ
- 富士通渡邊による性能測定には、以下のシステムを用いた。
- FX100 : SPARC64TMXIfx(1.5H)、富士通コンパイラ
 - Intel Xeon : Intel Xeon Gold 6148(3.1GHz)、インテルコンパイラ

3.3.4 検崎による 1 回目の発表のまとめ

以前の性能測定で、FX100 で 1 ノード 32 コアを用いた時は、MPI による 2 プロセス並列と OpenMP による 16 スレッド並列の組合せが一番速くなることを確認していた。Skylake の 1 ノード 40 コアでは、4 プロセスと 10 スレッド並列の組合せが一番速くなった。FX100 と Skylake を比較すると、FX100 は一番計算量の大きい静電相互作用は SIMD 化により Skylake よりも速くなった。一方、他の相互作用は SIMD 化が困難で Skylake に比べて数倍程度遅くなるものが多く、全体の計算時間も FX100 の方が遅くなった。

次に、skylake において静電相互作用のさらなるチューニングを行った。具体的には静電相互作用につかうネイバリングリストの配列の構造を Array of Structure (AoS) から Structure of Array (SoA) に変更した。結果として、静電相互作用の計算時間が 6.1 s から 4.6 s に高速化された。条件によってどちらの構造を使った方がいいかは変わるようだが、配列の構造を変えることがチューニングの 1 手法として有効であることを確認できた。

3.3.5 富士通渡邊による発表のまとめ

上記の検崎による測定結果で、多くの相互作用で FX100 の性能が Skylake に比べて性能が悪かったので、富士通渡邊により性能測定とチューニングが行われた。状況を簡単にするために、ボンド長、ボンド角、Go ポテンシャルの 3 つの相互作用について調べることにした。

まず、MPI_wtime により時間計測を行っていたが、オーバーヘッドが大きいことが示された。オーバーヘッドが少ないタイマーとして、FX100 では gettod を、Skylake では clockx を使うこととした。

チューニングとしてはボンド長について行われ、手動によるループアンスイッチングと” !ocl simd_redundant_vl () ” を配列式の直前に挿入することが行われ、10%程度の性能向上が得られた。また、ループ最後のリダクション演算部にインダイレクトアクセスがあり、データの重なりがないようなレイアウトにすると大きな最適化の効果を得られることが推測された。

3.3.6 検崎による 2 回目の発表のまとめ

富士通渡邊による指摘を受け時間計測の方法についての確認とボンド長とボンド角の相互作用計算のチューニングを行った。

タイマーについては、MPI_wtime と gettod に加えて clock_time も試してみたところ、MPI_wtime のオーバーヘッドが大きいことが再確認され、gettod と clock_time の差は小さかったので、移植性を考え clock_time を採用することとした。

ボンド長のチューニングについては、OpenMP のスレッド並列のチャンクサイズを 1 にすることによって行った。これは、ボンド長のネイバリングリストでは、リダクション部分がスレッド並列毎に異なる領域を持ち、ネイバリングリストの作成方法から 4 つ以上離れていればリダクション部分が重ならないようになっていることを利用している。このことから、チャンクサイズを 1 にしてスレッド並列数が 3 以上ならば、リダクション部分に

重なりがないことが保証されるのである。よって、” !ocl norecurrence() ” を付けることにより、ボンド長の計算が SIMD 化できた。ただし、ボンド長ではチャンクサイズを 1 にすることによるオーバーヘッドが大きく、SIMD 化による性能向上は小さかったので、結果として高速化にはならなかった。

また、ボンド角についても同様のチューニングが行った。こちらは SIMD 化による性能向上の効果が比較的大きく、オーバーヘッドを込みで 30% 程度高速化した。ただし、Skylake に比べると性能はまだ不十分ではあり、さらなる高速化が求められる。

3.3.7 まとめ

CafeMol を FX100 と Skylake 上で性能測定を行い、いくつかのチューニングを試みた。最も計算量の多い静電相互作用については、skylake について AoS から SoA の変更により性能が向上した。それ以外の計算量が少ない相互作用について FX100 についてチューニングを行ったところ若干の性能向上に成功したが、今回のチューニングにより差が縮まったとはいえ Skylake に比べて十分な性能を出すことが出来なかった。この理由は CPU やコンパイラによる out of order などの命令の充実に差があるためと考えられ、ポスト京では一部改善が見込めると考えられる。

また、MPI_Wtime については無視できないオーバーヘッドがあり、時間計測を行う際は system_clock やシステム依存なルーチンの利用などを行う必要がある場合がある。この指摘に対し、FX100 では経過時間を測定する際、NTP による時間自動調整結果が補正されないように clock_gettime システムコールを用いているためであると富士通より報告があった。なお、ポスト京で MPI_Wtime のオーバーヘッドは FX100 で用意されている gettod 相当の処理に変更され、MPI_Wtime の性能は改善される見込みであることも併せて富士通から報告されている。

2017/10/20

SS研メニーコア時代のアプリ性能WG

第4回会合

生体分子粗視化シミュレータCafeMol Skylakeでのチューニング

検崎博生

理研 情報システム部

概要

- 粗視化モデルとCafeMolについて
 - 生体分子シミュレーションと粗視化
 - CGタンパク質/DNAモデル
 - 計算方法と並列化
- Skylakeでのベンチマーク結果
 - 1ノードでのMPI並列数とスレッド並列数
- Skylakeでのチューニング結果
 - 静電相互作用のSIMD化
 - データ構造変換

HOKUSAI System

- GreatWave Massively Parallel Computer (GWMPC)
 - 1,080 nodes, 32 cores/node, 34,560 cores
 - 1.092 PFLOPS (1.975 Hz x 16 FP x 32 cores x 1,080 CPUs)
 - CPU: SPARCTMXIfx (1.975GHz, 32 cores, 1 CPU/node)
 - Memory: 32GB/node
 - Memory BW: 480 GB/s/node
 - Interconnect: Tofu2 6D-Mesh/Torus (12.5GB/s, bidirectional)
- Application Computing Server (ACS)
 - Large memory server: 2 nodes, 1.5 TB/node
 - GPU server: 30 nodes, 4 GPU/node, Tesla K20X
- BigWaterfall Massively Parallel Computer (BWMPC)
 - 840 nodes, 40 cores/node, 33,600 cores
 - 2.58 PFLOPS (2.4 GHz x 32 FP X 20 cores x 1680 CPUs)
 - CPU: Intel Xeon Gold 6148(2.4 GHz, 20 cores, 2 CPUs/node)
 - Memory: DDR4-2666 96GB/node
 - Memory BW: 255 GB/s/node
 - Interconnect: InfiniBand EDR (12.6 GB/s, bidirectional)

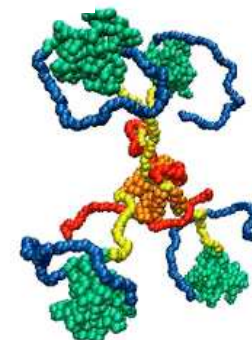
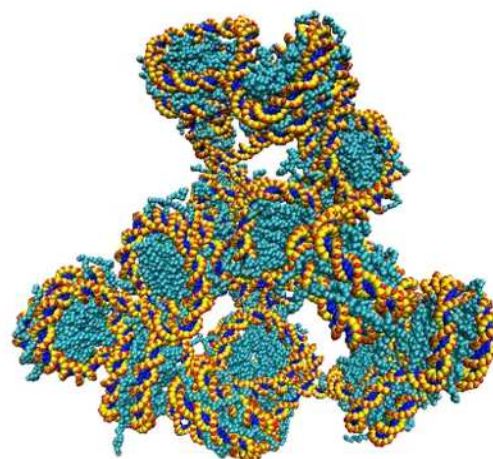
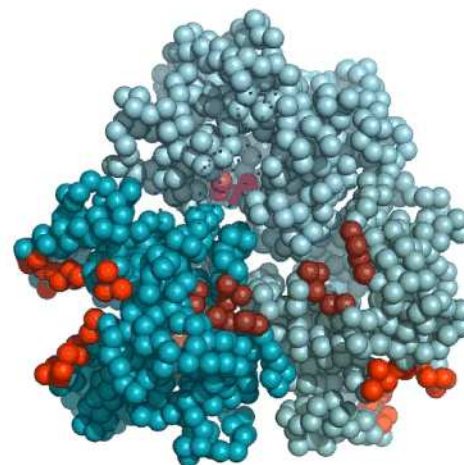
CafeMol: Coarse-grained biomolecular simulation software

H.Kenzaki, et al, J. Chem. Theor. Chem. (2011)



- CafeMol

- Coarse-grained (CG) Protein/Nucleic acid/lipid models
- CafeMol 3.0 source code and documentation are released at <http://www.cafemol.org>
- Takada Lab (Kyoto Univ)



CG protein model: Go-like model

C. Clementi, H. Nymeyer, and J.N. Onuchic, *J. Mol. Biol.* (2000)

Based on the energy landscape theory

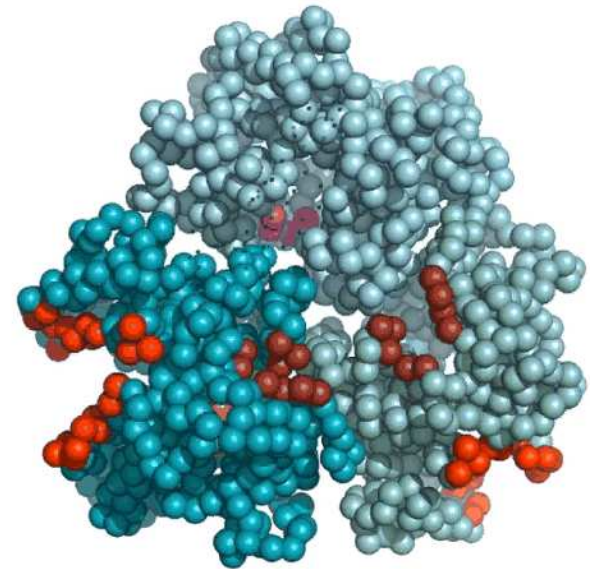
$$V_{protein} = V_{local} + V_{go} + V_{ex}$$

θ : bond angle
 ϕ : dihedral angle
(0 means native state)

$$V_{local} = K_b \sum_i (r_{i,i+1} - r_{0i,i+1})^2 + K_\theta \sum_i (\theta_i - \theta_{0i})^2 \\ + K_\phi^1 \sum_i (1 - \cos(\phi_i - \phi_{0i})) + K_\phi^3 \sum_i (1 - \cos 3(\phi_i - \phi_{0i}))$$

$$V_{go} = \epsilon_{go} \sum_{i,j}^{native} \left[5 \left(\frac{r_{0ij}}{r_{ij}} \right)^{12} - 6 \left(\frac{r_{0ij}}{r_{ij}} \right)^{10} \right]$$

$$V_{ex} = \epsilon_{ex} \sum_{i,j}^{nonnative} \left(\frac{\sigma}{r_{ij}} \right)^{12}$$



X. Yao, H. Kenzaki, S. Murakami,
and S. Takada, *Nature Comm.* (2010)

CG DNA model: 3SPN.1 force field (local, base pair and excluded volume interactions)

E.J. Sambriski, D.C. Schwartz, and J.J. de Pablo, Knotts, Biophys. J. (2009)

$$V_{dna} = V_{local} + V_{stack} + V_{bp} + V_{ex} + V_{qq} + V_{solv}$$

$$V_{local} = K_{b1} \sum_i (r_{i,i+1} - r_{0i,i+1})^2 + K_{b2} \sum_i (r_{i,i+1} - r_{0i,i+1})^4 + K_{\theta} \sum_i (\theta_i - \theta_{0i})^2 + K_{\phi} \sum_i (1 - \cos(\phi_i - \phi_{0i}))$$

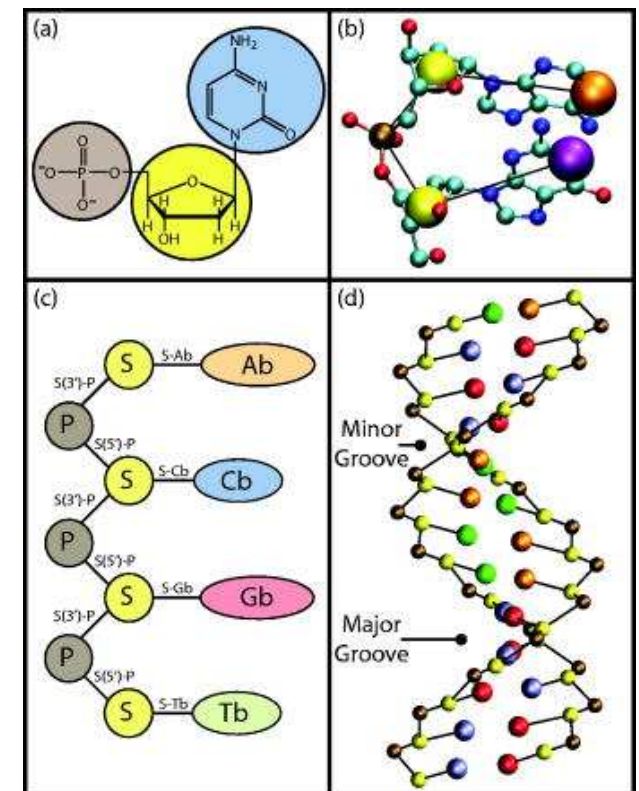
θ : bond angle
 ϕ : dihedral angle
(0 means B-type DNA)

$$V_{stack} = 4\epsilon_1 \sum_{i,j} \left[\left(\frac{\sigma_{0ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{0ij}}{r_{ij}} \right)^6 \right]$$

$$V_{bp} = \sum_{i,j} 4\epsilon_{bpi} \left[5 \left(\frac{r_{0ij}}{r_{ij}} \right)^{12} - 6 \left(\frac{r_{0ij}}{r_{ij}} \right)^{10} \right]$$

$$V_{ex} = 4\epsilon_1 \sum_{i,j} \left[\left(\frac{\sigma_0}{r_{ij}} \right)^{12} - \left(\frac{\sigma_0}{r_{ij}} \right)^6 \right] + \epsilon_1 \text{ (if } r_{ij} < d_{cut}),$$

$$= 0 \text{ (if } r_{ij} > d_{cut})$$



CG DNA model: 3SPN.1 force field (electrostatic and solvation interactions)

$$V_{qq} = \sum_{i,j}^N \left(\frac{q_i q_j}{4\pi\epsilon_0\epsilon(T,C)r_{ij}} \right) e^{-r_{ij}/\kappa_D}$$

Debye-Huckel theory

$$\epsilon(T,C) = \epsilon(T)a(C) \quad \leftarrow \quad \epsilon = 78$$

$$\epsilon(T) = 249.4 - 0.788T / K + 7.20 \times 10^{-4} (T / k)^2$$

$$a(C) = 1.000 - 0.2551C / M$$

$$+ 5.151 \times 10^{-2} (C / M)^2 - 6.889 \times 10^{-3} (C / M)^3$$

$$V_{solv} = \sum_{i < j}^{N_{solv}} \epsilon_s \left[1 - e^{-a(r_{ij} - r_s)} \right]^2 - \epsilon_s$$

$$\alpha^{-1} = 5.333 \text{ \AA}$$

$$r_s = 13.38 \text{ \AA}$$

$$\epsilon_0 = 0.504982\epsilon$$

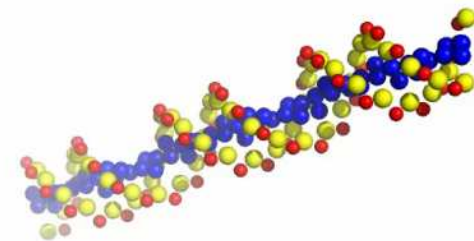
$$\epsilon_s = \epsilon_N A_I$$

$$e_N = e_0 (1 - [1.40418 - 0.268231 N_{nt}]^{-1})$$

$$A_I = 0.474876 (1 + \{0.148378 + 10.9553 [Na^+]\}^{-1})$$

Debye length

$$\kappa_D = \left(\frac{\epsilon_0 \epsilon RT}{2 N_A^2 e_q^2 I} \right)$$



計算規模と並列化

- 計算規模

- 数十から数万粒子数程度
- ヌクレオソーム1個の系: 1,854粒子

- 並列化

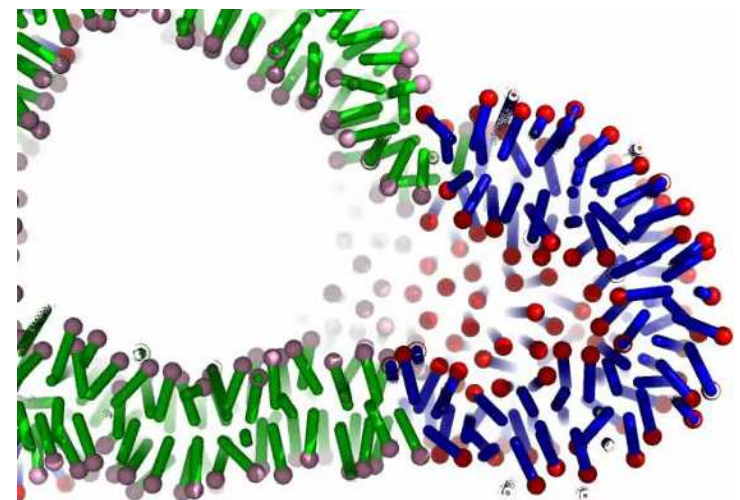
- 系の時間発展をMPIとOpenMPでハイブリッド並列化
 - 数十並列程度の並列化
 - 力の計算はネイバリングリスト方式
- レプリカ交換によるMPI並列
 - 通信量が少ないので大規模並列化
 - 平衡量を求める計算に使える

ネイバリングリストの例 (2体の相互作用、 $i < j$ だけをリストアップ)

i	1	2	3	4	5	6	7	8
j	2 4 7 9	5 8	4 5 9	7 8	6 7 9	8 9	8 9	9

力場の計算はネイバリングリスト方式

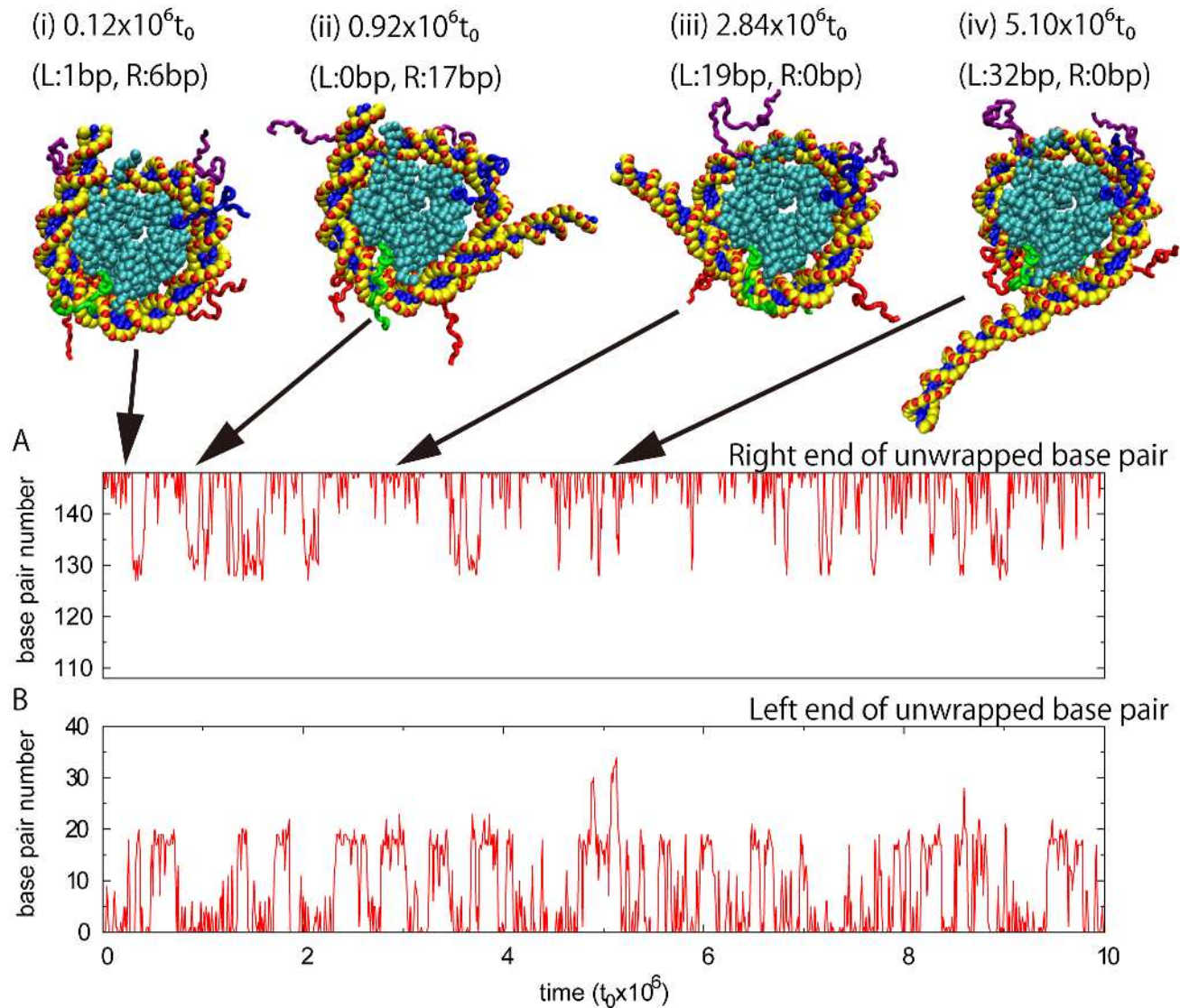
- 100stepに1回程度計算
- ネイバリングリストはノード毎に分割
 - ローカルな相互作用以外は基本的に2体の相互作用の形になるので、あらかじめ各プロセスで計算する1つめの粒子iのリストを作っておく
 - 各プロセスは割り当てられた粒子iのリストについて力場を計算する。
 - 最も大きな配列(粒子数 $\times 100 \sim 10,000$)
- 相互作用の到達範囲毎にネイバリングリストを作成
 - ローカルな相互作用
 - ボンド長、ボンド角、二面角
 - 計算を始める前にネイバリングリストを作成しておく
 - 近距離相互作用(<20Å)
 - 郷ポテンシャル、排除体積、塩基対
 - 中距離相互作用(20-50Å)
 - 疎水相互作用
 - 多体のため複雑なネイバリングリストに
 - 遠距離相互作用(>50Å)
 - 静電相互作用
 - イオン強度により相互作用距離が変化



Simulation of nucleosome

H.Kenzaki and S.Takada, PLoS. Comp. Biol. (2015)

Ion strength=300mM

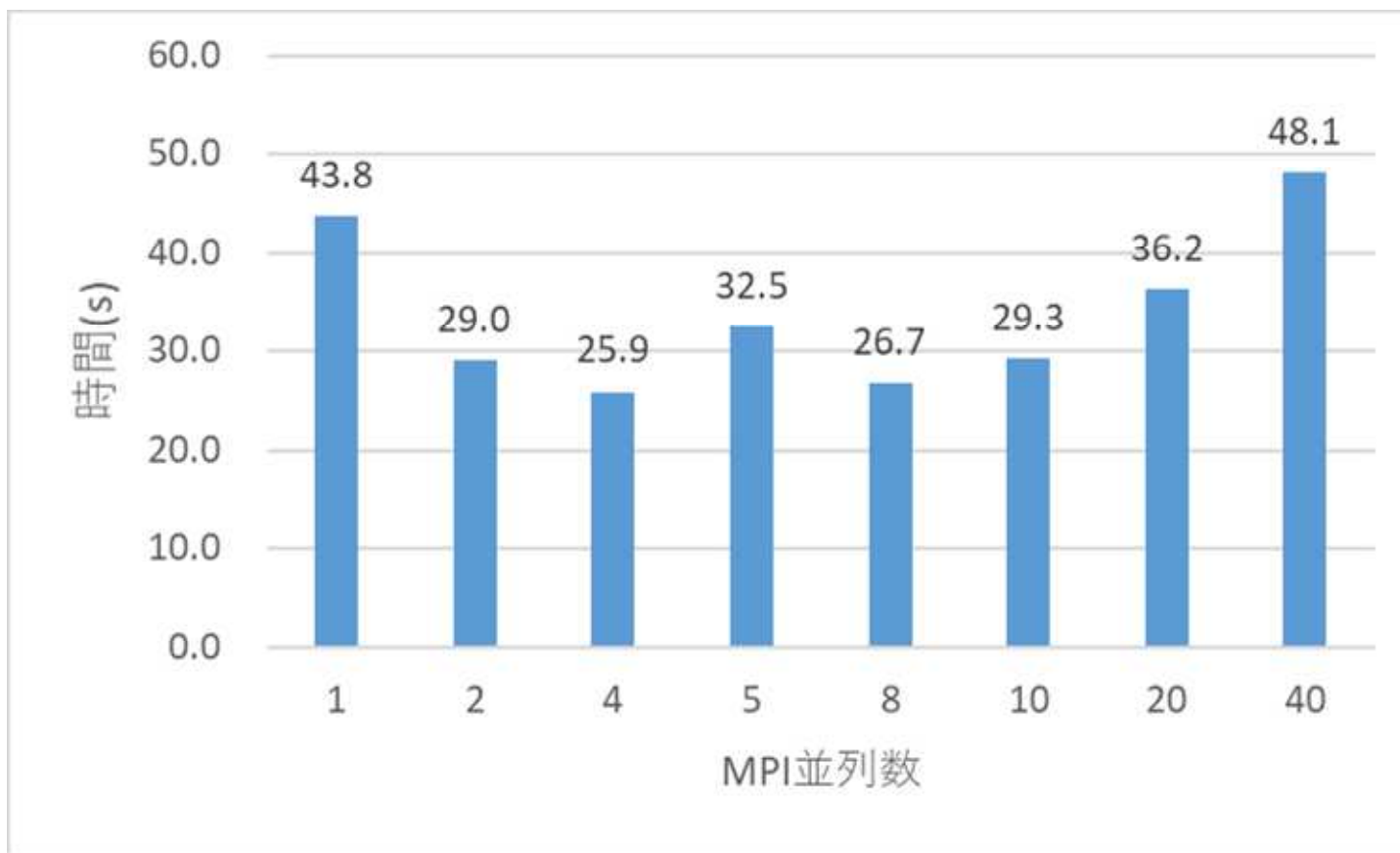


Skylakeでの性能測定 nucleosome (10^5 step, 100mM)

1 node (40 cores)利用で、MPI並列数xスレッド並列数=40を固定して、MPI並列数を変えて性能測定

コンパイラは、インテルコンパイラ17.0.4

最適化オプションは-fopenmp -O3 -xCORE-AVX512



1nodeでは4MPI並列x10OpenMP並列のときに一番高い性能

FX100との比較 nucleosome (10⁵step, 100mM)

	FX100(32 cores) -Kopenmp -Kfast -Kparallel 2MPIx16OpenMP	Skylake(40 cores) -fopenmp -O3 -xCORE-AVX512 4MPIx10OpenMP
force	36.7	18.1
_force(comm)	6.2	5.5
_force(local)	8.8	2.0
_force(go)	1.4	0.2
_force(pnl)	11.8	2.2
_force(ele)	5.8	7.9
random	7.0	4.0
neighbor	12.7	1.9
update	3.3	2.5
output	0.5	1.0
ope	59.9	21.9
comm	6.8	5.9
main_loop	65.7	27.8

Skylakeの方が2倍以上高速化
静電相互作用以外の相互作用とネイバリングリストで4-6倍

静電相互作用の計算方法

1. ネイバリングリストiele2mp(2, lele)を各プロセスで計算

i < jとなるペアだけを格納

2. simu_force.F90

```
!$omp parallel private(tn)
```

```
call simu_force_ele(force_mp_l(1, 1, tn))
```

スレッド毎にforce_mpの異なる領域を用意

3. simu_force_ele.F90

```
!$omp do private(...)
```

```
do iele1 = 1, lele
```

```
imp1 = iele2mp(1, iele1)
```

```
imp2 = iele2mp(2, iele2)
```

```
v21(1:3) = xyz_mp(1:3, imp2) - xyz_mp(1:3, imp1)
```

```
dist2 = v21(1)**2 + v21(2)**2 + v21(3)**2
```

```
if(dist2 > cutoff2) cycle
```

```
dist1 = sqrt(dist2)
```

```
rdist1 = 1.0/dist1
```

```
dvdw_dr = coef(iele)*rdist1*rdist1*(rdist1+rcdist)*exp(-dist1*rcdist)
```

```
force_mp(1:3, imp1) = force_mp(1:3, imp1) - dvdw_dr*v21(1:3)
```

```
force_mp(1:3, imp2) = force_mp(1:3, imp2) + dvdw_dr*v21(1:3)
```

```
end do
```

```
!$omp end do nowait
```

SIMD化困難

!\$omp atomicでは遅くなる

静電相互作用のSIMD化

1. ネイバリングリストiele2charge_k(ncharge, ncharge_mpi)を各プロセスで計算。
電荷をもっている粒子のリストicharge2mp(ncharge)と座標xyz_ele(3, ncharge)を作っておく。
タンパク質は4/20が、DNAは1/3の粒子が電荷をもっているため。

2. simu_force.F90

```
!$omp parallel private(tn)
```

```
call simu_force_ele2(force_mp_l(1, 1, tn))
```

3. simu_force_ele2.F90

```
!$omp do private(...)
```

```
do icharge1 = 1, ncharge
```

```
  imp1 = icharge2mp(icharge)
```

```
  for(1:3) = 0.0
```

```
  do iele = 1, lele_k(icharge)
```

```
    jcharge = iele2charge_k(iele, icharge)
```

```
    v21(1:3) = xyz_ele(1:3, icharge) - xyz_ele(1:3, jcharge)
```

```
    ...
```

```
    for(1:3) = for(1:3) + dvdw_dr*v21(1:3)
```

```
  end do
```

```
  force_mp(1:3, imp1) = force_mp(1:3, imp1) + for(1:3)
```

```
end do
```

```
!$omp end do nowait
```

電荷を持った粒子だけの座標

SIMD化OK

ただし計算量は

本来必要な量の2倍

計算時間は7.9s 6.1s

静電相互作用のデータ構造変換

- Array of Structure(AoS)からStructure of Array(SoA)にデータ構造を変換。
 - xyz_ele(1:3, ncharge)をxyz_ele(ncharge, 1:3)に変換。

	Skylake(40 cores)	Skylake(40 cores) SIMD	Skylake(40 cores) SIMD+DATA
force	18.1	16.1	14.8
_force(comm)	5.5	5.1	5.2
_force(local)	2.0	2.1	2.0
_force(go)	0.2	0.2	0.2
_force(pnl)	2.2	2.2	2.2
_force(ele)	7.9	6.1	4.6
random	4.0	4.0	3.9
neighbor	1.9	1.8	1.9
update	2.5	2.5	2.5
output	1.0	1.1	1.0
ope	21.9	20.6	19.0
comm	5.9	5.6	5.6
main_loop	27.8	26.2	24.6

計算時間は6.1s 4.6s

まとめ

- Skylakeでのベンチマーク結果
 - 1ノードでは、4MPI並列x10スレッドの時に一番高い性能。
 - FX100に対して2倍以上の高速化。
 - 静電相互作用は余り変わらない。
 - 静電相互作用以外やネイバリングリングリストの計算で4-6倍の差。
- Skylakeでの静電相互作用のチューニング
 - SIMD化により1.3倍高速化。
 - データ構造を変えることにより1.3倍高速化。

SS研 メニーコア時代のアプリ性能検討WG

[A.I. 3]

CafemolのFX100とIntel Skylakeの性能差について

2018年7月20日

富士通株式会社

TCソリューション事業本部

渡邊 健太

- 目的
- 評価環境
- 現状分析
- チューニング
- まとめ

[A.I. 3] FX100における cafemol の性能を分析する

- ・ Intel Skylakeとの性能差の原因を調査する
- ・ FX100で高速化の検討

剣崎さんから評価対象のサブルーチンを選定いただいています。

[メールの一部を抜粋]

- > ○調べていただきたいサブルーチン
- > simu_force_bond.F90(タイマーの名称: _force(bond))
- > simu_force_angle.F90(タイマーの名称: _force(angle))
- > simu_force_nlocal_go.F90(タイマーの名称: _force(go))
- >
- > bondとangleは鎖で繋がった局所的な2体、3体の相互作用で、メモリアクセスが連続的
- > になっています。
- > goはランダムアクセスを含むような2体の相互作用となっています。
- > どれもskylake1にくらべて6倍程度遅くなっています。

■ FX100 と Intel Skylake

		FX100	Intel Skylake
C P U	名称	SPARC64 Xlfx	Intel Xeon Gold 6148
	動作周波数	<u>1.50 GHz</u>	[Base] 2.40 GHz [MAX] Normal : 3.10 GHz AVX512 : 2.20 GHz
	コア数	32コア + アシスタントコア	20コア
	キャッシュメモリ	L1I\$, L1D\$: 64KB/コア L2\$: 24MB/CPU	L1I\$, L1D\$: 32KB/コア L2\$: 1MB L3\$: 27.5MB
	理論ピーク性能	768GFlops/CPU (倍精度)	1,408GFlops/CPU (AVX512,倍精度)
	メモリ帯域	(240+240)GB/s/CPU	128GB/s/CPU (DDR-2666, 6chnnels)
	ノ ード	CPU数, コア数	1CPU/ノード, (32コア + アシスタントコア)/ノード
理論ピーク性能		768GFlops/CPU (倍精度)	2,816GFlops/CPU (AVX512,倍精度)
メモリ帯域		(240+240)GB/s	256GB/s
コンパイラー		富士通コンパイラ	Intel Compiler v18.1

■ 各計算機の性能を確認

■ 測定条件

[コンパイルオプション]

FX100 : -Kopenmp,fast,parallel,optmsg=2 -Qa,m,p,t,x

SKL : -O3 -qopenmp -qopt-report=5

■ 測定結果

		FX100		Intel Skylake		性能値の比較 ② / ④
評価環境		① 渡邊	② 剣崎さん	③ 渡邊	④ 剣崎さん	
評価環境	測定者	渡邊	剣崎さん	渡邊	剣崎さん	② / ④
	計算機	FX100 (1.5GHz)	FX100 (1.975GHz)	Skylake (2.40GHz)	Skylake (2.40GHz)	
	並列数	2p x 16t	2p x 16t	4p x 10t	4p x 10t	
タイマーの結果	_force(bond)	0.6316	0.462	0.1142	0.0912	<u>5.1</u>
	_force(angle)	2.0688	1.5737	0.2970	0.2803	<u>5.6</u>
	_force(go)	1.8975	1.4399	0.2090	0.1935	<u>7.4</u>
		#1		#2		

#1 クロック周波数の差と同じ性能差

#2 BIOS設定SNC(Sub-NUMA Clustering)が違う?

③ : SNC disable

④ : SNC enable

■ コンパイラの最適化状況

コンパイラ最適化	_force(bond)		_force(angle)		_force(go)	
	FX100	SKL	FX100	SKL	FX100	SKL
SIMD	×	×	×	×	×	×
SWP	×	-	×	-	×	-
FULLUNROLLING*1	○	○	○	○	○	○
UNSWITCHING*2	○	-	○	-	×	-
PREFETCH	×	×	○	×	×	×

*1 : doループにではなく、ループ内部の配列式に対して適応

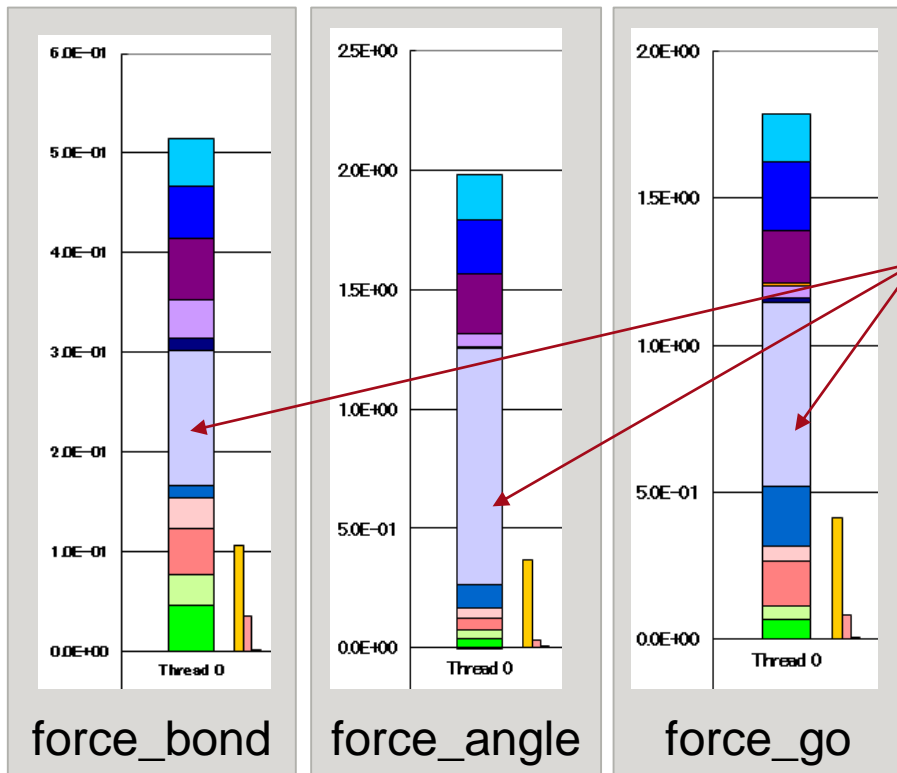
*2 : ループ内で不変な不等式をループ外に出す最適化

■ 実行効率の差

SIMD化されていないため、各サブルーチンの実行効率の差は下表となる

サブルーチン	① FX100(1.975GHz)	② SKL (3.1GHz) *No-SIMD	③ No-SIMD時のピーク性能比 FX100 / SKL	実行効率の差 ① / ② x ③
_force(bond)	0.462	0.0912	FX100 / SKL = 0.51	2.6
_force(angle)	1.5737	0.2803		2.9
_force(go)	1.4399	0.1935		3.8

■ PA情報(実行時間内訳)



	_force(bond)	_force(angle)	_force(go)
4命令コミット	9.14%	9.68%	9.27%
2/3命令コミット	10.24%	11.42%	13.08%
1命令コミット	12.09%	12.42%	9.98%
浮動小数点演算待ち	26.53%	50.01%	35.12%
整数演算待ち	2.41%	4.77%	11.27%
浮動小数点ロード L1アクセス待ち	5.85%	2.26%	2.96%
浮動小数点ロード L2アクセス待ち	9.21%	2.52%	8.57%
整数ロード L1アクセス待ち	5.97%	1.90%	2.65%
整数ロード L2アクセス待ち	8.88%	1.81%	3.58%

✓ 命令スケジュールの改善で高速化できる部分 (浮動小数点/整数演算待ち、浮動小数点/整数L1アクセス待ち)

_force(bond)	_force(angle)	_force(go)
40.76%	58.94%	52.00%

✓ 各サブルーチンの××命令コミットの割合は30%を超えており、命令数もボトルネックになっている。

■ PA情報(メモリ・キャッシュスループット情報)@FX100

■ メモリ・キャッシュスループット情報

サブルーチン	L1ビジー率	L2ビジー率	メモリビジー率	L2スループット	メモリスループット
_force(bond)	21%	7%	0%	32.08GB/s	0.00GB/s
_force(angle)	19%	2%	0%	8.13GB/s	0.00GB/s
_force(go)	23%	5%	0%	20.77GB/s	0.00GB/s

■ キャッシュミス情報

サブルーチン	L1Dミス率	L1Dミスdm率	L2ミス率	L2ミスdm率	μDTLBミス率	mDTLBミス率
_force(bond)	2.63%	91.05%	0.00%	70.27%	1.44%	0.00%
_force(angle)	0.51%	81.49%	0.00%	83.91%	0.33%	0.00%
_force(go)	1.15%	87.43%	0.00%	85.24%	0.34%	0.00%

- ✓ メモリビジー率・メモリスループットから、メモリアクセスはほとんど無い。
- ✓ L1キャッシュミス率は、理論値(3.15%)以下である。

■ 基本プロファイル情報の採取

[実行時のコマンド]

```
fipp -C -d fipp_dir -l hwm,cpu -P nouserfunc
```

[プロファイル情報のテキスト出力]

Procedures profile

```
*****  
Application - procedures  
*****
```

Cost	% Operation (S)	Barrier	% Start	End		
19792	100.0000	1873.0138	1695	8.5641	--	-- Application
2551	12.8890	241.4149	1600	62.7205	406	515 simu_tintegral.get_random_number._OMP_1_
2311	11.6764	218.7002	0	0.0000	--	-- !!! lock wait
2029	10.2516	192.0169	0	0.0000	5	143 simu_force_pnl_
2015	10.1809	190.6891	0	0.0000	--	-- __pthread_mutex_unlock_usercnt
1990	10.0546	188.3222	0	0.0000	45	87 simu_neighbor_list._OMP_1_
1276	6.4470	120.7537	0	0.0000	4	88 simu_force_ele2_
1023	5.1688	96.8112	0	0.0000	7	424 simu_force_pnl2_
806	4.0724	76.2771	0	0.0000	510	549 mt_stream.mt_genrand_int32_
538	2.7183	50.9133	0	0.0000	1032	1080 gf2xe.mult_i32_
496	2.5061	46.9388	0	0.0000	84	124 simu_neighbor_list_ele2._PRL_3_
443	2.2383	41.9240	0	0.0000	--	-- __g_dsin
416	2.1019	39.3690	0	0.0000	708	729 mt_stream.mt_genrand_double1_
413	2.0867	39.0839	0	0.0000	14	113 simu_force_nlocal_go_
378	1.9099	35.7725	0	0.0000	288	376 simu_force_fdih.calc_force_fdih_
231	1.1671	21.8582	0	0.0000	6	329 mloop_flexible_local_
186	0.9398	17.6023	0	0.0000	166	236 simu_force_fdih.calc_phi_
152	0.7680	14.3840	0	0.0000	--	-- __GI__gettimeofday_internal
146	0.7377	13.8166	0	0.0000	4	191 simu_force_dih_
139	0.7023	13.1556	0	0.0000	--	-- __g_dscn2
131	0.6619	12.3968	0	0.0000	13	770 simu_neighbor_assign_

コンパイルリスト (_force(bond))

```
55      !$omp do private(imp1,imp2,v21,dist,ddist,ddist2,for, &
56      !$omp&      force,efull,iunit,junit,isys,istat)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< UNSWITCHING
      <<< Loop-information End >>>
57  1 p 2s      do ibd = ksta, kend
58  1
59  1 p 2v      imp1 = ibd2mp(1, ibd)
60  1 p 2v      imp2 = ibd2mp(2, ibd)
61  1
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< FULL UNROLLING
      <<< Loop-information End >>>
62  1 p 6v      v21(1:3) = xyz_mp_rep(1:3, imp2,irep) - xyz_mp_rep(1:3, imp1,irep)
63  1
64  1 p 2v      dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
65  1 p 2v      ddist = dist - bd_nat(ibd)
66  1 p 2v      ddist2 = ddist**2
67  1
68  1          ! calc force
69  1 p 2v      for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * &
70  1          (-2.0e0_PREC * ddist / dist)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< FULL UNROLLING
      <<< Loop-information End >>>
71  1 p 6m      force(1:3) = for * v21(1:3)
72  1
73  2 p 2v      if(inmgo%i_multi_mgo == 0) then
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< FULL UNROLLING
      <<< Loop-information End >>>
74  2 p 6      force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
75  2 p 6      force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
```

→ メインループの最適化情報
SIMD/SWPIはなし

→ インダイレクトアクセス

→ If文 ループ中で不変

```
76  2
77  2 p 2v      else
78  2          ! calc energy
79  2 p 2v      efull = (coef_bd(1, ibd) + coef_bd(2, ibd) * ddist2) * ddist2
80  2 p 2v      iunit = imp2unit(imp1)
81  2 p 2v      junit = imp2unit(imp2)
82  2 p 2m      ene_unit(iunit, junit) = ene_unit(iunit, junit) + efull
83  2
84  2 p 2v      isys = ibd2sysmbr_mgo(1, ibd)
85  3 p 2s      if(isys == 0) then
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< FULL UNROLLING
      <<< Loop-information End >>>
86  3 p 6s      force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
87  3 p 6s      force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
88  3 p 2s      else
89  3 p 2s      istat = ibd2sysmbr_mgo(2, ibd)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< FULL UNROLLING
      <<< Loop-information End >>>
90  3 p 6s      force_mp_mgo(1:3, imp1, istat, isys) = force_mp_mgo(1:3,
imp1, istat, isys) - force(1:3)
91  3 p 6s      force_mp_mgo(1:3, imp2, istat, isys) = force_mp_mgo(1:3,
imp2, istat, isys) + force(1:3)
92  3 p 2      end if
93  2 p 2v      end if
94  1 p 2v      end do
95          !$omp end do nowait
```

→ If文

- ✓ サブルーチン内のメインループはIntel SkylakeでもSIMD化されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応

コンパイルリスト (_force(angle))

```

52      !$omp do private(imp1,imp2,imp3,v21,v32,c11,c22,c21, &
53      !$omp&      co_theta,dba,t3,for,force_21,force_32, &
54      !$omp&      efull,iunit,junit,isys,istat)
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< UNSWITCHING
    <<< PREFETCH      : 2
    <<<   ba_nat: 2
    <<< Loop-information End >>>
55  1 p s      do iba=ksta,kend
56  1
57  1 p m      if (coef_ba(1, iba) < ZERO_JUDGE .and. coef_ba(2, iba) <
ZERO_JUDGE) cycle
58  1
59  1 p s      imp1 = iba2mp(1, iba)
60  1 p s      imp2 = iba2mp(2, iba)
61  1 p s      imp3 = iba2mp(3, iba)
62  1
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< FULL UNROLLING
    <<< Loop-information End >>>
63  1 p 3s     v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
64  1 p 3s     v32(1:3) = xyz_mp_rep(1:3, imp3, irep) - xyz_mp_rep(1:3, imp2,
irep)
65  1
66  1 p s      c11 = v21(1) * v21(1) + v21(2) * v21(2) + v21(3) * v21(3)
67  1 p s      c22 = v32(1) * v32(1) + v32(2) * v32(2) + v32(3) * v32(3)
68  1 p s      c21 = v32(1) * v21(1) + v32(2) * v21(2) + v32(3) * v21(3)
69  1
70  1 p s      co_theta = - c21 / sqrt(c11 * c22)
71  1
72  2 p s      if(co_theta > 1.0e0_PREC) then
73  2 p m      co_theta = 1.0e0_PREC
74  2 p s      else if(co_theta < -1.0e0_PREC) then
75  2 p s      co_theta = -1.0e0_PREC
76  2 p s      end if
77  1
78  1 p s      dba = acos(co_theta) - ba_nat(iba)
79  1
80  1 p s      t3 = c11 * c22 - c21**2
81  2 p s      if(t3 <= 1.0e0_PREC) then
82  2 p s      t3 = 1.0e0_PREC
83  2 p s      end if
84  1

```

→ メインループの最適化情報
SIMD/SWPはなし

→ cycle文

→ インダイレクトアクセス

```

85  1          ! calc force
86  1 p s      for = 2.0e0_PREC * coef_ba(1, iba) * dba / sqrt(t3) - &
87  1          coef_ba(2, iba) / sqrt(c11 * c22)
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< FULL UNROLLING
    <<< Loop-information End >>>
88  1 p 3s     force_21(1:3) = for * (v21(1:3) * (c21 / c11) - v32(1:3))
89  1 p 3s     force_32(1:3) = for * (v32(1:3) * (c21 / c22) - v21(1:3))
90  1
91  2 p s      if(inmgo%i_multi_mgo == 0) then
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< FULL UNROLLING
    <<< Loop-information End >>>
92  2 p 3      force_mp(1:3, imp1) = force_mp(1:3, imp1) - force_21(1:3)
93  2 p 3      force_mp(1:3, imp2) = force_mp(1:3, imp2) + force_21(1:3) -
force_32(1:3)
94  2 p 3      force_mp(1:3, imp3) = force_mp(1:3, imp3) + force_32(1:3)
95  2
96  2 p s      else
          :
114 2 p s      end if
115 1
116 1 p v      end do
117          !$omp end do nowait

```

→ If文 ループ中で不変

- ✓ サブルーチン内のメインループは Intel SkylakeでもSIMD化されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応
- ✓ cycle文あり

ソースコード (force_go)

```
67      !$omp do private(imp1,imp2,rcut_off2,v21,dist2,roverdist2,roverdist4, &
68      !$omp&      roverdist8,roverdist12,roverdist14,dgo_dr,for,imirror)
69  1 p      do icon=ksta,kend
70  1
71  1 p      imp1 = icon2mp(1, icon)
72  1 p      imp2 = icon2mp(2, icon)
73  2 p      if (iclass_mp(imp1) == CCLASS%RNA .AND. iclass_mp(imp2) ==
CLASS%RNA) then
74  2 p          rcut_off2 = rcut_off2_rna
75  2 p      else
76  2 p          rcut_off2 = rcut_off2_pro
77  2 p      endif
78  1
79  2 p      if(inperi%i_periodic == 0) then
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< FULL UNROLLING
<<< Loop-information End >>>
80  2 p 3      v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
81  2 p      else
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< FULL UNROLLING
<<< Loop-information End >>>
82  2 p 3      v21(1:3) = pxyz_mp_rep(1:3, imp2, irep) - pxyz_mp_rep(1:3, imp1,
irep)
83  2 p      call util_pbneighbor(v21, imirror)
84  2 p      end if
85  1
86  1      ! v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
87  1
88  1 p      dist2 = v21(1)*v21(1) + v21(2)*v21(2) + v21(3)*v21(3)
89  1
90  1 p      roverdist2 = go_nat2(icon) / dist2
```

→ インダイレクトアクセス

→ If文

→ If文

```
91  1 p      if(roverdist2 < rcut_off2) cycle
92  1
93  1 p      roverdist4 = roverdist2 * roverdist2
94  1 p      roverdist8 = roverdist4 * roverdist4
95  1 p      roverdist12 = roverdist4 * roverdist8
96  1 p      roverdist14 = roverdist12 * roverdist2
97  1
98  1 p      dgo_dr = 60.0e0_PREC * coef_go(icon) / go_nat2(icon) *
(roverdist14 - roverdist12)
99  1
100 2 p      if(dgo_dr > DE_MAX) then
101 2      ! write (*, *) "go", imp1, imp2, dgo_dr
102 2 p      dgo_dr = DE_MAX
103 2 p      end if
104 1      ! if(dgo_dr > 5.0e0_PREC) dgo_dr = 5.0e0_PREC
105 1
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< FULL UNROLLING
<<< Loop-information End >>>
106 1 p 3      for(1:3) = dgo_dr * v21(1:3)
107 1 p 3      force_mp(1:3, imp2) = force_mp(1:3, imp2) + for(1:3)
108 1 p 3      force_mp(1:3, imp1) = force_mp(1:3, imp1) - for(1:3)
109 1
110 1 p      end do
111      !$omp end do nowait
```

→ cycle文

→ If文

- ✓ サブルーチン内のメインループはIntel SkylakeでもSIMD化されていない
- ✓ UNSWITCHINGは適応されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応
- ✓ cycle文あり

- 3つのサブルーチンの1ノードのFX100とSKLの性能は、5~7倍SKLが良い
 - FX100 (1.5GHz)とFX100 (1.975GHz)の性能差から、CPUの演算性能(周波数)の差が実測性能に直結
 - PA情報からメモリアクセスはほとんど無い
 - FX100/SKLでSIMD化されていない。
 - 1ノードのFX100/SKLのピーク演算性能差が2倍程度異なるため、**FX100とSKLの実行効率の差は2.6~3.6倍**になる
- PA情報(実行時間内訳)から、
 - 命令スケジューリングの改善で高速化できる部分は、40~58%ある。
 - **SWP化の検討**
 - 命令数(XX命令コミット)の削減で高速化できる部分は、約30%ある。
 - **SIMD化の検討**

プログラム全体に対する性能改善 1

- 基本プロファイラから得られた “__lll_lock_wait” に対する対応
- 実行時環境変数を指定

```
export XOS_MMM_L_ARENA_LOCK_TYPE=0
```

- 「0」を指定することで、malloc要求を並列処理することができる。
- 「1」がデフォルト値、逐次処理される。

■ 測定結果 (プログラム全体の経過時間)

指定なし	指定あり	改善率
130.75 (s)	115.63	13.1%

メインループおよび調査対象のサブルーチンの性能への影響はない。

プログラム全体に対する性能改善 2

- オーバーヘッドが大きい時間計測ルーチンを置き換える
- タイマールーチンの置換え

オリジナル	変更後	
	FX100	SKL
MPI_wtime	gettod	clockx

■ タイマー結果

サブルーチン	FX100		SKL		MPI_wtimeの 性能差	タイマー置換え 後の性能差	
	FX100 (1.5GHz)		FX100 (1.975GHz)				
	MPI_wtime	gettod	gettod	MPI_wtime			clockx
force_bond	0.6316	0.4509	0.3424	0.1142	0.1005	5.1	3.4
force_angle	2.0688	1.8957	1.4398	0.2970	0.2809	5.6	5.1
force_go	1.8975	1.7119	1.3002	0.2090	0.1999	7.4	6.5

- ✓ 経過時間が短いサブルーチンほど、タイマーのオーバーヘッドの影響が大きくなる

■ チューニング方針

1. 手動でループアンスイッチングを適応
2. `!ocl simd_redundant_vl()`を配列式の直前に挿入
→ この指示行を指定しないとSWP化されない。コンパイラのメッセージは、
「スケジューリング結果を得られなかったため、ソフトウェアパイプラインを適用できません。」

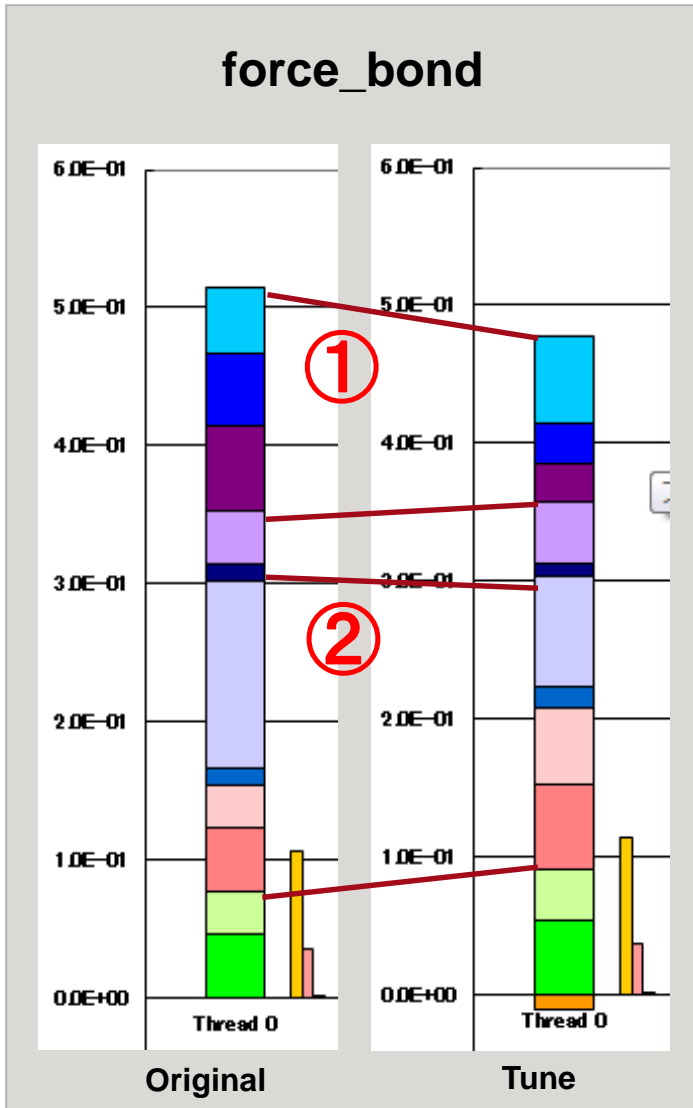
simd_redundant_vl指示行

ループ回転数がSIMD長で割り切れない場合でも、マスク付きSIMD命令を利用し、SIMD実行するループを生成する。

```
57 1      if(inmgo%i_multi_mgo == 0) then      ← 手動ループアンスイッチング
58 1
59 1      !$omp do private(imp1,imp2,v21,dist,ddist,ddist2,for, &
60 1      !$omp&      force,efull,iunit,junit,isys,istat)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SOFTWARE PIPELINING
      <<< Loop-information End >>>
61 2 p      do ibd = ksta, kend
62 2
63 2 p      imp1 = ibd2mp(1, ibd)
64 2 p      imp2 = ibd2mp(2, ibd)
65 2
66 2      ! write(*,*) "fj*** imp1=",imp1,"imp2=",imp2
67 2      !ocl simd_redundant_vl(3)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< FULL UNROLLING
      <<< Loop-information End >>>
68 2 p 1v      v21(1:3) = xyz_mp_rep(1:3, imp2,irep) - xyz_mp_rep(1:3,
imp1,irep)
69 2
70 2 p      dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
71 2 p      ddist = dist - bd_nat(ibd)
72 2 p      ddist2 = ddist**2
73 2
```

```
74 2      ! calc force
75 2 p      for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * &
76 2      (-2.0e0_PREC * ddist / dist)
77 2      !ocl simd_redundant_vl(3)
      <<< Loop-information Start >>>
      <<< [OPTIMIZATION]
      <<< SIMD(VL: 4)
      <<< FULL UNROLLING
      <<< Loop-information End >>>
78 2 p 1v      force(1:3) = for * v21(1:3)
79 2      !ocl simd_redundant_vl(3)
80 2 p 1v      force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
81 2      !ocl simd_redundant_vl(3)
82 2 p 1v      force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
83 2 p      enddo
84 1      !$omp end do nowait
85 1
86 1      else      ← 手動ループアンスイッチング
```

■ 実行時間内訳の比較



■ チューニング効果

(#) タイマー(gettod)の経過時間から

Original	Tune	向上率
0.4509秒	0.4120秒	9.4%

① 命令コミットのコストは少し改善

→ ループ内の配列式をSIMD化したので命令数が減少した。

② 命令スケジューリングで改善できる部分のSWPによる効果が小さい

- SWPに必要な回転数 :44
(コンパイラメッセージから)
- 実際のループ回転数 :約900

■ SWPの効果が小さい = 命令を重ねられていない原因

- ループ最後のインダイレクトアクセスのあるリダクション演算部に対して、コンパイラは依存関係を見切れず、逐次スケジューリングになっていると考えられる。

```
79  2      !ocl simd_redundant_vl(3)
80  2 p 1v      force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
81  2      !ocl simd_redundant_vl(3)
82  2 p 1v      force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
```

- しかし、データには重なりがあるため、!OCL NORECURRENCEが使えない

```
fj*** imp1= 1 imp2= 2
fj*** imp1= 1 imp2= 3
fj*** imp1= 3 imp2= 4
fj*** imp1= 4 imp2= 5
fj*** imp1= 4 imp2= 6
fj*** imp1= 6 imp2= 7
fj*** imp1= 7 imp2= 8
fj*** imp1= 7 imp2= 9
fj*** imp1= 9 imp2= 10
fj*** imp1= 10 imp2= 11
:
```

データに重なりが無いようなレイアウトにすることで
コンパイラ最適化(SWP)の効果を最大限に引き出せる
と考える

■ FX100とSkylakeの5~7.4倍の性能差の分析


- タイマールーチンの置換えにより、性能差は3.4~6.5倍
- さらに、ピーク演算性能の差を考慮すると、その差は、1.7~3.3倍
- FX100では、PA情報の採取から、命令数と演算/データアクセス待ちがボトルネックとなっていた。

■ 高速化の検討

- 手動ループアンスイッチングと!ocl simd_redundant_vlの指定により、SIMD化/SWP化を促進することで、約10%性能が改善した。
- インダイレクトアクセスのデータに対して、データの重なりがないようレイアウトを変更できれば、より最適化の効果を得ることができると考える。

■ 残り2ルーチンに対して

- `_force(bond)`との一番の違いはcycle文があること。
- cycle文直前でループ分割し、後半のループに対して、`_force(bond)`と同様のチューニングをすることが可能



FUJITSU

shaping tomorrow with you

2019/2/25

SS研メニーコア時代のアプリ性能WG

第9回会合

生体分子粗視化シミュレータCafeMol タイマーとローカルな力のSIMD化

検崎博生

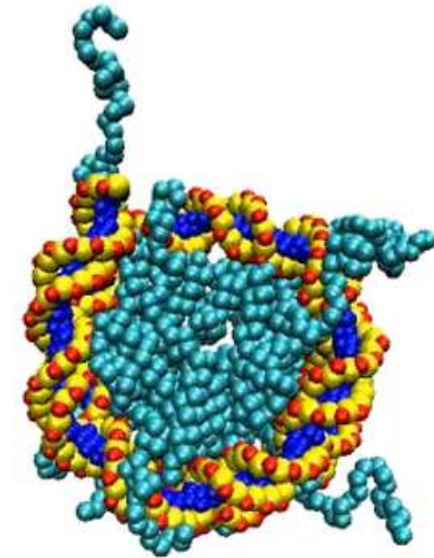
理研 情報システム部

概要

- 粗視化モデルとCafeMolについて
 - 生体分子シミュレーションと粗視化
 - CGタンパク質/DNAモデル
 - 計算方法と並列化
- タイマーによるオーバーヘッド
 - `mpi_wtime`, `system_clock`, `gettod`
- 1ノードでのMPI並列数とスレッド並列数
- ローカルな相互作用のSIMD化
 - ボンド長とボンド角相互作用
 - OpenMPのスレッド並列のチャンクサイズを変更

Model

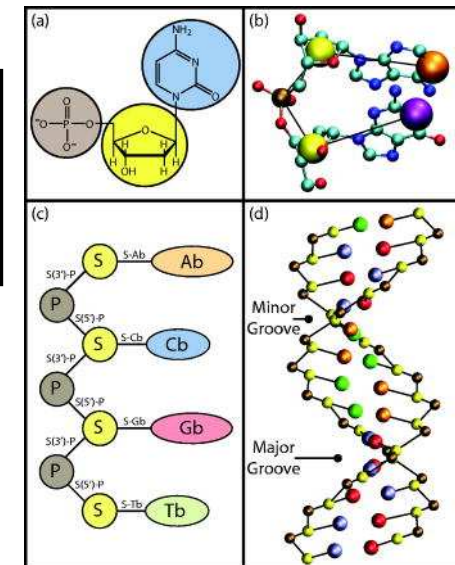
- CafeMol H.Kenzaki, et al, J. Chem. Theor. Chem. (2011)
- Protein
 - AICG2+ model W Li, et al PNAS (2012)
 - FLP model for disordered regions in X-ray structure Knotts et al, J. Chem. Phys. (2007)
- DNA
 - 3SPN.1 model



ローカルなポテンシャル

θ : bond angle
 ϕ : dihedral angle
 (0 means native state)

$$V_{local} = K_b \sum_i (r_{i,i+1} - r_{0i,i+1})^2 + K_\theta \sum_i (\theta_i - \theta_{0i})^2 + K_\phi^1 \sum_i (1 - \cos(\phi_i - \phi_{0i})) + K_\phi^3 \sum_i (1 - \cos 3(\phi_i - \phi_{0i}))$$



ボンド長、ボンド角、二面角などの相互作用からなるが、モデルによって詳細はさまざまである。

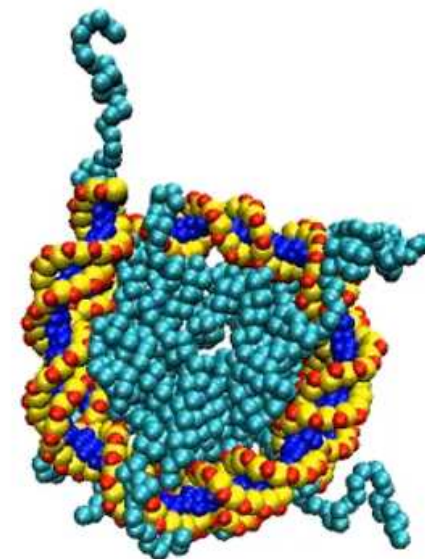
今回の計算対象と計算方法

- 計算対象

- ヌクレオソーム1個の系: 1,854粒子

- 計算方法

- ネイバリングリストを作っておく
 - ローカルな相互作用は最初に作っておく
 - それ以外は100stepに1回更新
- MPIとOpenMPでネイバリングリストを分割して並列計算
 - プロセスとスレッド毎に力を保存して、最後に通信を行い足し合わせる
- 速度や座標のアップデートは全プロセスで同じものを行う。



ネイバリングリストの例 (2体の相互作用)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	1				2		3			4		5			6		7		8
j	2	4	7	9	5	8	4	5	9	7	8	6	7	9	8	9	8	9	9

タイマーの比較 nucleosome (10⁵step, 100mM)

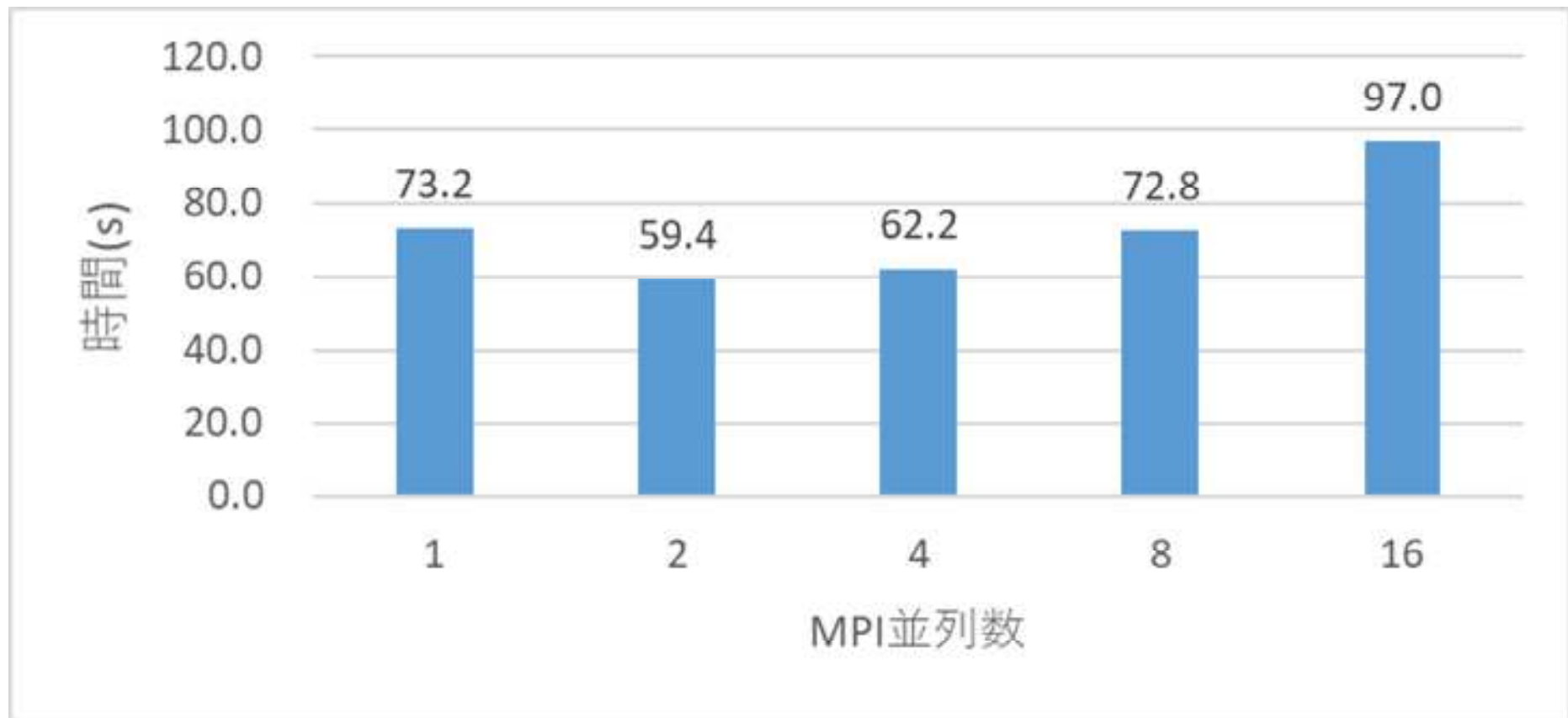
- FX100 1 node: SPARC64 Xlfx (1.975GHz, 32 cores, 1 CPU/node)
- コンパイラは、Fujitsu Fortran Compiler 2.0.0-07
- 最適化オプションは-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- 1ノード(32コア)で2MPI並列X16スレッド並列で実行

	FX100(32 cores) mpi_wtime	FX100(32 cores) system_clock	FX100(32 cores) gettod
force	38.33	34.92	34.94
_force(comm)	6.12	6.15	6.19
_force(local)	10.42	8.72	8.68
__force(bond)	0.48	0.42	0.38
__force(bangle)	1.54	1.39	1.41
_force(go)	1.43	1.31	1.30
_force(pnl)	11.70	11.55	11.51
_force(ele)	5.87	5.69	5.68
ope	60.52	53.03	52.75
comm	6.75	6.60	6.65
main_loop	67.27	59.62	59.40

mpi_wtimeはオーバーヘッドが大き過ぎる。
system_clockとgettodは同程度。

FX100での性能測定 nucleosome (10⁵step, 100mM)

- FX100 1 node: SPARC64 Xlfx (1.975GHz, 32 cores, 1 CPU/node)
- コンパイラは、Fujitsu Fortran Compiler 2.0.0-07
- 最適化オプションは-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- 1ノード(32コア)でMPI並列数X16スレッド並列数=32に固定して実行



1nodeでは2MPI並列x16OpenMP並列のときに一番高い性能

force(bond)の計算方法

1. ネイバリングリストiele2mp(2, lele)を各プロセスで計算

i < jとなるペアだけを格納

2. simu_force.F90

```
!$omp parallel private(tn)
```

```
call simu_force_bond(force_mp_l(1, 1, tn))
```

3. simu_force_bond.F90

```
!$omp do private(...)
```

```
do ibd = ksta, kend
```

```
  imp1 = ibd2mp(1, ibd)
```

```
  imp2 = ibd2mp(2, ibd)
```

```
  v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1, irep)
```

```
  dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
```

```
  ddist = dist - bd_nat(ibd)
```

```
  ddist2 = ddist**2
```

```
  for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * (-2.0e0_PREC * ddist / dist)
```

```
  force(1:3) = for * v21(1:3)
```

```
  force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
```

```
  force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
```

```
end do
```

```
!$omp end do nowait
```

スレッド毎にforce_mpの異なる領域を用意

```
ibd=1 imp1=1 imp2=2  
ibd=2 imp1=1 imp3=3  
ibd=3 imp1=3 imp2=4  
ibd=4 imp1=4 imp2=5  
ibd=5 imp1=4 imp2=6  
ibd=6 imp1=6 imp2=7
```

force_mpへの足しこみが原因でSIMD化できない

force(bond)のSIMD化

• 方針

- force_mpへの足し込みをループ内で被らないようする。
- force_mpがスレッド毎に別なので、各スレッド毎に被らないようする。
- ボンド長の場合、ネイバリングリストで3つ以上離れていれば被らない。
- よって、チャンクサイズを1にし、スレッド並列数が3以上であれば、SIMD化が可能はず。

• tune1

- 今回のモデルに関係ない部分を削除しておく。
- チェック用のifなど、if文は全て削除する。

• tune2

- !omp do schedule(static,1)としてチャンクサイズを1に設定する。
- この段階では、メモリアクセスが遅くなると予想できる。

• tune3

- !ocl norecurrence()を付けてSIMD化を行う。

• ボンド各force(bangle)についても同様にSIMD化を行うが、FLPモデルについては、中で関数を読んでいるのでSIMD化に失敗。

- 他の相互作用でもif文を外せないなどSIMD化できないことはよくある。

force(bond)とforce(bangle)のSIMD化

- tune1: 今回の計算に使わない部分を削除しソースを単純化
- tune2: OpenMPのスレッド並列の順序を変更
- tune3: !ocl norecurrence()を付けてSIMD化
- コンパイルオプションは-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- 1ノード(32コア)で2MPI並列X16スレッド並列で実行

	FX100(32 cores) tune1	FX100(32 cores) tune2	FX100(32 cores) tune3
force	34.90	35.44	35.00
_force(comm)	6.18	6.21	6.22
_force(local)	8.71	8.99	8.79
__force(bond)	0.37	0.62	0.51
__force(bangle)	1.42	1.44	1.06
_force(go)	1.27	1.29	1.29
_force(pnl)	11.52	11.65	11.47
_force(ele)	5.68	5.67	5.68
ope	52.80	53.24	52.80
comm	6.65	6.66	6.65
main_loop	59.44	59.90	59.44

bondはSIMD化の副作用の方が大きく遅くなった。
bangleはSIMD化により30%程度速くなった。

まとめ

- Timerの比較
 - MPI_WTIMEはオーバーヘッドが大きい。
 - system_clockとgettodは同程度。
- MPI並列とOpenMP並列の兼ね合い。
 - 1ノードでは、2MPI並列x16スレッドの時に一番高い性能。
- FX100でのローカルな相互作用のSIMD化
 - スレッド並列の割り当てることにより、SIMD化に成功。
 - 複雑な力場を使う場合はif文や関数呼び出しを含むので難しい。
 - 性能の向上は限定的。
 - スレッド並列の割り当てるオーバーヘッドとの兼ね合い。

3.4 ADVENTURE における多倍長精度演算の利用に向けた検討

名古屋大学情報基盤センター 荻野 正雄

3.4.1 はじめに

ADVENTURE とは、設計用大規模計算力学システム開発プロジェクト(通称 ADVENTURE プロジェクト、<https://adventure.sys.t.u-tokyo.ac.jp/>)において開発されているオープンソース CAE システムである。大規模メッシュを用いて自然物や人工物を丸ごと詳細にモデル化し、多様な並列分散計算機環境のもとで固体の変形や熱・流体の流れ等の力学解析から可視化、設計最適化までを行える特徴がある。無料公開されていることもあり、教育・研究から産業応用まで幅広く利用されているソフトウェアである。ここでは、有限要素法による 3 次元電磁場解析モジュール ADVENTURE_Magnetic を取り上げる。電磁場には静磁場、低周波電磁場、高周波電磁場など様々な種類があり、対象とする問題の特性に応じて Maxwell 方程式から偏微分方程式を導出する。それらに辺要素有限要素法を適用することで、様々な線形方程式が得られる。例えば、時間調和渦電流問題や高周波電磁場問題では複素対称線形方程式を解くことになる。しかし、係数行列が悪条件になりやすく、共役直交共役勾配(COFG)法などの反復解法で高い精度の収束解を得ることが困難な場合がある。そこで現在、IEEE 754-2008 が規定する四倍精度浮動小数点数や D.H. Bailey や K. Briggs などが提案した倍精度数 2 つを用いる疑似四倍精度数などの多倍長精度演算に着目し、反復計算における丸め誤差を低減させることで、電磁場解析を効率化することを目指している。今回は、富士通 FX100 を含むいくつかの環境において、任意精度・多倍長精度ライブラリの性能評価を行った。また、ライブラリ利用における知識や技術の蓄積と共有を行い、さらに富士通 TCS 環境の課題を明らかにする。

3.4.2 任意精度・多倍長精度演算ライブラリの概要

表 3.4.1 に今回用いた任意精度・多倍長精度演算ライブラリ、実数及び複素数の浮動小数点数型を示す。

表 2.9.1 主な任意精度・多倍長精度浮動小数点数

Library	Precision	Real data type	Complex data type
C data type	extended	long double	long double _Complex
Fortran data type	quadruple	real(kind=16)	complex(kind=16)
libquadmath	quadruple	__float128	__complex128
Intel's _Quad	quadruple	_Quad	complex<_Quad>
QD	pseudo-quad	dd_real	complex<dd_real>
Fujitsu's fast_dd	pseudo-quad	dd_real	complex<dd_real>

ARPREC	arbitrary	mp_real	mp_complex
exflib	arbitrary	exfloat	complex<exfloat>
MPFR/GMP	arbitrary	mpfr_t	-
MPC		-	mpc_t

四倍精度としては、Fortran データ型、GCC 拡張である C 言語の libquadmath ライブラリ (<https://gcc.gnu.org/onlinedocs/libquadmath/>)、Intel コンパイラ拡張である C 言語の _Quad を用いる。libquadmath ライブラリは GCC 4.6.0 以降から利用可能である。QD ライブラリ (<http://crd-legacy.lbl.gov/~dhbailey/mpdist/>) は倍精度浮動小数点数を 2 つ用いる double-double 形式の疑似四倍精度演算ライブラリである。富士通 fast_dd は、富士通 TCS に含まれる高速 4 倍精度基本演算ライブラリであり、QD と同じく double-double 形式の疑似四倍精度である。比較用として、C データ型の拡張倍精度、任意精度演算ライブラリである ARPREC (<http://crd-legacy.lbl.gov/~dhbailey/mpdist/>)、exflib (<http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/>)、並びに MPFR (<http://www.mpfr.org/>) とその複素数版である MPC (<http://www.multiprecision.org/>) を用いる。

ここで、Intel _Quad、QD、fast_dd、並びに exflib では複素数の基本演算機能が提供されていないため、C++ 複素数テンプレートクラスと組み合わせて用いた。

また、任意精度・多倍長精度演算ライブラリは利用マニュアルが十分でないことが多く、サンプルコードからプログラミング方法を学ぶ必要がある。今回作成したいくつかのプログラムをスライド 1~8 に示す。

3.4.3 測定環境

性能測定には、汎用 PC、名古屋大学の FX100、及び名古屋大学の CX400 を使用した。それぞれの環境における CPU、OS、コンパイラを表 2.9.2 に示す。また、それぞれの環境において性能測定を行うことができたライブラリを表 2.9.3 に示す。性能測定できなかった理由は次の通りである。

- libquadmath は、FX100(fcc) は富士通コンパイラのため、FX100(gcc) は gcc-6.3.0 だが当該ライブラリ未提供のため、CX400 は gcc-4.4.7 のため
- Intel's _Quad は、PC ではライセンス未導入のため、FX100 は SPARC64 プロセッサのため
- fast_dd は、FX100 のみ利用可能であるため
- exflib は x64 バイナリ版のみを利用したため

表 3.4.2 実験に用いた計算機とソフトウェア

Name	CPU	OS	Compiler	Site
PC	Intel Core i7-8650U (Kabylake-R)	Ubuntu 18.04 TLS	gcc-7.3. 0	-
FX100(fcc)	Fujitsu SPARC64 XIfx	XTC OS 2.1.1	fcc-2.0. 0	Nagoya
FX100(gcc)			gcc-6.3. 0	Nagoya
CX400	Intel Xeon E5-2697v3 (Haswell-EP)	RHEL	icc-14. 0.3	Nagoya

表 3.4.3 使用ライブラリと計算環境における利用可否

Library	Version	PC	FX100(fcc)	FX100(gcc)	CX400
libquadmath	(gcc-4.6.0 or later)	○	-	-	-
Intel's _Quad	icc-14.0.3	-	-	-	○
QD	2.3.20	○	○	○	○
fast_dd	2.3.2	-	○	○	-
ARPREC	2.2.19	○	○	○	○
exflib	x64-bin-20180620	○	-	-	○
GMP	4.0.1				
MPFR	6.1.2	○	○	○	○
MPC	1.1.0				

3.4.4 計算精度に関する数値実験

a) 問題設定

実数を係数にもつ2次方程式 $ax^2 + bx + c = 0$ を考える。この式は解の公式により、 $x_1 = (-b - \sqrt{b^2 - 4ac}) / (2a)$ と $x_2 = (-b + \sqrt{b^2 - 4ac}) / (2a)$ で得られる。このとき、桁落ちしやすい係数 $a = 1.01$ 、 $b = 2718281$ 、 $c = 0.01$ を用いて x_2 を計算したときの精度を評価する。この問題では加減算、乗算、除算、平方根の計算を行う。

真の解は $x_2 = -3.678795532912165323920499 \times 10^{-9}$ である。比較として、exflib ライブラリで100桁精度で計算した結果は $\tilde{x}_2 = -3.$

$6787955329121653239204992803347 \times 10^{-9}$ であった。

b) 実験結果

スライド9に実験結果を示す。任意精度演算ライブラリについては、4倍精度相当となるように計算精度を設定している。表の結果に書かれている数値は、太字・下線のところまで正しく計算できていることを表している。表より、普

通に倍精度数で計算したのでは小数点以下 1 桁までしか正しく計算できていないことが分かる。これに対し、疑似四倍精度の QD と fast_dd は 16~17 桁まで正しく計算できており、Fortran や C の libquadmath といった四倍精度の結果 (18~19 桁) に近い計算精度が得られている。これより、名大 FX100 環境などにおいて、多倍長精度計算ライブラリを正しく利用できていることが示された。

ここで、x86 系である PC の環境では long double 型は小数点以下 3 桁程度の精度で拡張倍精度として倍精度より高い精度で計算しているようであるがこの問題にとっては精度不足である。一方、FX100 の long double 型は四倍精度相当の結果が得られている。long double 型変数が占めるサイズを調べたところ 128 ビットであり、きちんと 128 ビット分の精度が出ていることになる。富士通の C 言語処理系でとりあえず四倍精度数を使いたい場合に有用である。

また、Intel's _Quad は倍精度と同じ程度の結果となった。これは、平方根を含め数学関数の四倍精度版が提供されていなかったためである。実験当時の Intel コンパイラの説明として _Quad は実験的なものと述べられていた。

なお、任意精度演算ライブラリの ARPREC と exflib は想定以上の精度が出ているが理由は不明である。

3.4.5 計算時間に関する数値実験

a) 問題設定

以下のロジスティック写像を考える。

$$x_{n+1} = ax_n(1 - x_n)$$

$a = 4$ とすると、 x_n は区間 $[0, 1]$ 全体で非周期に変動するカオスとなることが知られている。このとき、初期値 $x_0 = 0.7501$ とし、 n が 10^6 になるまで計算したときの計算時間を評価する。また、 x_n を実数としたとき、複素数としたときの 2 ケースを評価する。この問題では、乗算と減算のみを行う。

b) 実験結果

スライド 10 と 11 に実数と複素数のときの実験結果をそれぞれ示す。表において、QD(C++) と QD(C) はそれぞれプログラム記述言語が C++ と C であることを表している。QD ライブラリ自体は C++ で記述されており、C++ プログラムから利用する場合向けに四則演算等の演算子オーバーロードをしている一方で、C プログラム向けに関数コールでの利用も可能となっている。C++ であればプログラムは書きやすいが、計算時間の観点で最適なコードが生成されない可能性がある。そこで比較のために 2 つの実装を行った。

スライド 10 より、PC では QD(C++) が最速であった。これは C++ の最適化が十分に行えているのだと推測される。また、QD(C++) による疑似四倍精度演算は倍精度演算に比べて 3 倍遅い程度であった。また、疑似四倍精度演算は四倍精度演算よりも 3 倍程度速く、前の実験で計算精度が同程度であったことから、実用性が高いと言える。FX100(fcc) では、同じく QD(C++) が最速であったが、倍精

度演算よりも7倍程度の計算時間であり、C++コードのさらなる最適化が望まれる。CX400ではIntel's _Quadが最速であり、QD(C++)もほぼ同程度であった。

次に、ターゲットとする複素数演算の性能を、スライド11で見てみる。実数のときと異なり、全ての環境においてQD(C)が最速となった。よって、QD(C)の利用が良いように思えるが、プログラム例のスライド4を見てわかるように、QDライブラリをC言語から複素数で利用する場合は複素数の四則演算などの関数を自作する必要があり、さらにプログラミングコストが高くなり、可搬性も損なわれる。C++の演算子オーバーロードを利用すれば、疑似四倍精度数であっても、複素数であっても、 $z=x+y$ のように記述できる。しかし、実際には、実部と虚部それぞれの加算に展開され、加算は疑似四倍精度数の加算になり、その加算1回あたりではKnuthによる加算の無誤差変換に基づいて倍精度数の加減算が11回行われる。つまり、 $z=x+y$ という記述は22回の演算に展開される。 $z=x*y$ は100回以上、 $z=x/y$ は250回以上の演算にもなる。実際のプログラムではもっと長い計算式が書かれ、forループブロック内であればループボディが非常に長くなることを意味する。スライド14と15に、QD(C++)を用いた疑似四倍精度の密行列ベクトル積コードをFX100の富士通C/C++コンパイラでコンパイルしたときの最適化メッセージを示す。これより、SIMD化もソフトウェアパイプライン化も行われていないことが分かる。コンパイラによる最適化が望まれる。なお、インライン展開に関するメッセージが大量にでてくるため、メッセージを調査しづらい問題があった。

また、名大FX100において富士通C/C++コンパイラの32レジスタ制限版を用いた実験も行った。その結果をスライド12と13に示す。このケースでは通常環境と性能差はなかった。しかし、上述したように現状でもコンパイラにはもう少し最適化を頑張ってもらいたいため、32レジスタ制限環境の影響を受けないかどうかは判断できないと言える。

3.4.6 複素対称線形方程式の数値実験

複素対称線形方程式における多倍長精度計算の有効性を評価する数値実験をPC環境で行った。疎行列データベースUF Sparse Matrix Collectionで公開されている2つの小規模行列を用いて、IC前処理付きCOCG法で解いた実験結果をスライド16~18に示す。これより、疑似四倍精度数を用いると倍精度数の場合と比べて反復回数を削減できることが分かる。このケースでは総計算時間は倍精度の場合が高速であったが、反復回数の削減効果は問題依存のため、その効果が大きい場合であれば、倍精度数よりも疑似四倍精度数を使う方が高速になる可能性があると言える。

スライド19と20に、高周波電磁場の有限要素解析で得られた複素対称行列による数値実験結果を示す。このケースでは、わずかにであるが疑似四倍精度が高速であった。

3.4.7 富士通 fast_dd について

富士通 TCS に含まれる富士通 fast_dd は QD と同様に double-double 形式の疑似四倍精度演算機能を提供するライブラリであるが、スライド 10~13 に示すように計算時間の面では QD ライブラリよりも低い性能となっている。しかし、富士通 fast_dd は 2 要素ずつ同時に計算するマルチ関数やベクトル関数を提供しており、それらが利用できるケースでの性能改善が目的の 1 つにある。前述の計算は単純な漸化式であったことからマルチ関数等の利用は行っていない。そこで、第 9 回 WG において富士通側報告として行われたマルチ関数・ベクトル関数の性能評価例をスライド 21~23 に示す。これより、複数要素の同時計算が行える場合では fast_dd は QD よりも高い性能が期待できることが明らかになった。

3.4.8 HOWTO

今回の数値実験を通して得られた多倍長精度演算ライブラリの利用に関する知識について、HOWTO としてまとめ、知識共有を行う。得られた知識は以下の通りである。

- ① 任意精度・多倍長精度演算ライブラリには、計算順序の入れ替えを行うと計算精度が低下するものがある。よって、例えば富士通 C/C++ コンパイラでは“-Keval”オプションは指定すべきではない。“-Keval”は“-Kfast”に含まれているので“-Kfast”の利用も避けるべきである。しかし、環境によっては、コンパイラの標準オプションとして“-Kfast”が指定されている場合があるので注意が必要であり、“-Knoeval”を使うのが望ましい。
- ② 富士通 C/C++ 処理系で任意精度・多倍長精度演算ライブラリを利用するときは、コンパイラのインライン展開オプション“-x-”を指定した方が高速となる。
- ③ 富士通 C/C++ コンパイラで QD ライブラリをビルドするときは、コンパイラに“-Kfp_contract”オプションを指定すれば FMA 命令をきちんと使ってくれる。①、②とあわせて、“-O3 -Kfp_contract -Knoeval -x-”が基本となる。
- ④ 名大 FX100 の gcc-6.3.0 環境では FMA 命令を使うよう指定して QD ライブラリをビルドすると、計算結果が誤ったものになる。SPARC64 XIfx の正しい FMA 命令が出されないのだと思われる。また、libquadmath はビルドされていない。
- ⑤ 富士通 fast_dd は SSL II の一部として提供されているが、GCC など富士通 TCS 以外のコンパイラで SSL II ライブラリを使うときは“-SSL2”ではリンク

できない。例えば今回指定したのは、インクルードパス指定は“-I/opt/FJSVmxlang/include”、リンク指定は“-L/opt/FJSVmxlang/lib64 -lssl2mtfoursimd_prexi -lssl2mtfoursimd_com -lssl2mtfoursimd_postxi -lfj90i -lfj90fmt -lfj90f -lfj90rt -lfjrtcl -ltrt -L/usr/lib64 -lm -lelf”である。Intel MKL Link Line Advisor のようなリンク方法を調べる手助けがあるのが望ましい。

- ⑥ 富士通 C/C++処理系では long double 型は 128 ビットの 4 倍精度浮動小数点数である。
- ⑦ 任意精度・多倍長精度の浮動小数点数に定数を代入するときは、定数の精度に気を付ける必要がある。例えば QD ライブラリでは文字列定数を使えば、疑似四倍精度数に変換した上で代入してくれる。
- ⑧ 富士通 fast_dd は疑似四倍精度定数の指定方法が不明だが、富士通 C/C++処理系のときは 4 倍精度である long double 型定数で代用できる。
- ⑨ 任意精度・多倍長精度演算ライブラリを使うときは、出力関数が対応しているかも気にする必要がある。

3.4.9 まとめ

今回は将来的に ADVENTURE へ多倍長精度演算を組み込むことを目的に、任意精度・多倍長精度演算ライブラリを名大 FX100 環境に移植し、数値実験によって性能評価を行った。当初は、想定する計算結果が得られないことが多く、丸め回数が少ない FMA 命令を使えているか、コンパイラによる計算順序入れ替えは起こっていないか、高精度な定数はどう指定するか、高精度な数学関数はあるのか、あたりを確認していけば正しく移植できることが見えてきた。疑似四倍精度数かつ複素数となると 1 回の四則演算あたりの計算量が多く、それを含む DO ループ / for ループの最適化は難しいと思われるが、ポスト京のコンパイラが効率良いコードを出してくれることに期待する。また、多倍長精度計算の専門家との協調作業が必要と思われるが、そのためには大量に出力されるコンパイラの最適化メッセージの改善も望まれる。

複素数版プログラム例 (FORTRAN)

```
program quadformula_complex_real16
  implicit none
  complex(kind=16) :: a, b, c, x2
  a = 1.01Q0
  b = 2718281.0Q0
  c = 0.01Q0
  x2 = (-b+sqrt(b*b-4.0*a*c))/(2.0*a)
  print *, real(x2)
end program quadformula_complex_real16
```

4倍精度複素数の宣言

4倍精度定数の代入

複素数版プログラム例 (libquadmath, C)

```
#include <stdio.h>
#include <stdlib.h>
#include <quadmath.h>

int main(void)
{
    __complex128 a, b, c, x2;
    char str[1024];

    a = 1.01Q;
    b = 2718281.0Q;
    c = 0.01Q;
    x2 = (-b+csqrtq(b*b-4.0*a*c))/(2.0*a);
    quadmath_snprintf(str, 1024, "%31.30Qe , %31.30Qe",
    __real__ x2, __imag__ x2);
    printf("x2= %s¥n", str);
}
```

4倍精度複素数の宣言

4倍精度定数の代入

4倍精度数学関数の呼び出し

4倍精度専用の出力関数

複素数版プログラム例 (QD, C++)

```
#include <iostream>
#include <iomanip>
#include <complex>
#include <stdlib.h>
#include <qd/dd_real.h>
using namespace std;

int main(void)
{
    complex<dd_real> x2;
    unsigned int oldcw;
    fpu_fix_start(&oldcw);

    complex<dd_real> a("1.01", "0.0");
    complex<dd_real> b("2718281.0", "0.0");
    complex<dd_real> c("0.01", "0.0");

    x2 = (-b+sqrt(b*b-
complex<dd_real>(4.0,0.0)*a*c))/(complex<dd_real>(2.0,0.0)*a);
    cout << "x2= " << scientific << setprecision(31) << x2 <<
"¥n";
}
```

疑似4倍精度変数の宣言

80ビット拡張倍精度を使うIntel x86向け設定

4倍精度定数の代入

複素数版プログラム例 (QD, C)

```
...
#include <qd/dd_real.h>
#include <qd/c_dd.h>
#define TO_DOUBLE_PTR(a, ptr) ptr[0] = a.x[0]; ptr[1] = a.x[1];

/* (a_r + b_r) + (a_i + b_i) I */
void c_zdd_add(const double *a_r, const double *a_i, const double *b_r,
const double *b_i, double *c_r, double *c_i) {
    dd_real cr, ci;
    cr = dd_real(a_r) + dd_real(b_r);
    ci = dd_real(a_i) + dd_real(b_i);
    TO_DOUBLE_PTR(cr, c_r);
    TO_DOUBLE_PTR(ci, c_i);
}
...
int main(void){
    double a_r[2], a_i[2], b_r[2], b_i[2], c_r[2], c_i[2], x2_r[2], x2_i[2];
    ...
    c_dd_copy_d(1.01, a_r);
    ...
    c_zdd_mul(b_r, b_i, b_r, b_i, t1_r, t1_i); /* b*b */
    c_zdd_mul(a_r, a_i, c_r, c_i, t2_r, t2_i); /* a*c */
    ...
    c_dd_write(x2_r);
}
```

マクロ関数の定義

倍々精度複素数の
四則演算関数は自作

変数1個につき、実部と虚部それぞれ大きさ2個の倍精度配列

四則演算は関数呼び出し

複素数版プログラム例 (Fujitsu fastdd, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <complex>
#include <fast_dd.h>

using namespace std;
int main(void)
{
    complex<dd_real> a, b, c, x1, x2;
    long double lx;

    a = 1.01L;
    b = 2718281.0L;
    c = 0.01L;

    x2 = (-b+sqrt(b*b-4.0*a*c))/(2.0*a);
    lx = to_long_double(real(x2));
    cout << "x2= " << scientific << setprecision(31) << lx <<
    "¥n";
}
```

疑似4倍精度複素数の宣言

4倍精度定数の作り方が用意されていないので、
long double定数を代入

4倍精度の出力機能は用意されていないのでlong
doubleとかに変換してから出力する必要あり

複素数版プログラム例 (ARPREC, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <arprec/mp_complex.h>

using namespace std;
int nr_digits = 32; 精度の設定

int main(void)
{
    mp::mp_init(nr_digits);
    mp_complex x2; 任意精度変数の宣言
    mp_complex a("1.01E+0", "0E+0");
    mp_complex b("2718281.0E+0", "0E+0"); 任意精度定数の代入
    mp_complex c("0.01E+0", "0E+0");

    x2 = (-b+sqrt(b*b-4*a*c))/(2*a);
    cout << "x2= " << scientific << setprecision(31) << x2.real << "
" << x2.imag << "¥n";

    mp::mp_finalize();
}
```

複素数版プログラム例 (exflib, C++)

```
#define PRECISION 32
```

精度の設定

```
#include <iostream>
```

```
#include <iomanip>
```

```
#include <complex>
```

```
#include <stdlib.h>
```

```
#include <exfloat.h>
```

```
using namespace std;
```

```
int main(void)
```

```
{
```

```
    complex<exfloat> a, b, c, x2;
```

任意精度複素数の宣言

```
    a = "1.01";
```

```
    b = "2718281.0";
```

```
    c = "0.01";
```

任意精度定数の代入

```
    x2 = (-b+sqrt(b*b-  
complex<exfloat>(4,0)*a*c))/(complex<exfloat>(2,0)*a);
```

```
    cout << "x2= " << scientific << setprecision(31) << x2 <<  
"¥n";
```

```
}
```

複素数版プログラム例 (MPC, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
#include <mpc.h>

#define PRECISION 106
using namespace std;
int main(void)
{
    mpc_t a, b, c, _four, _two, x2, t1, t2, t3;

    mpc_init2(a,PRECISION);
    mpc_init2(b,PRECISION);
    mpc_init2(c,PRECISION);
    mpc_init2(x2,PRECISION);
    mpc_init2(t1,PRECISION);
    mpc_init2(t2,PRECISION);
    mpc_init2(t3,PRECISION);
    mpc_init2(_four,PRECISION);
    mpc_init2(_two,PRECISION);
```

精度の設定

任意精度変数の宣言

変数の初期化や
任意精度定数の
代入

```
mpc_set_str(a, "(1.01 0.0)", 10, MPC_RNDNN);
mpc_set_str(b, "(2718281.0 0.0)", 10, MPC_RNDNN);
mpc_set_str(c, "(0.01 0.0)", 10, MPC_RNDNN);
mpc_set_str(_four, "(4.0 0.0)", 10, MPC_RNDNN);
mpc_set_str(_two, "(2.0 0.0)", 10, MPC_RNDNN);
```

```
mpc_mul(t1, b, b, MPC_RNDNN);
mpc_mul(t2, a, c, MPC_RNDNN);
mpc_mul(t3, t2, _four, MPC_RNDNN);
mpc_sub(t2, t1, t3, MPC_RNDNN);
mpc_sqrt(t1, t2, MPC_RNDNN);
mpc_sub(t2, t1, b, MPC_RNDNN);
mpc_mul(t3, a, _two, MPC_RNDNN);
mpc_div(x2, t2, t3, MPC_RNDNN);
```

乗算

```
cout << "x2= ";
mpc_out_str(stdout, 10, 0, x2, MPC_RNDNN);
cout << "¥n";
```

```
mpc_clear(a);
mpc_clear(b);
mpc_clear(c);
mpc_clear(x2);
mpc_clear(t1);
mpc_clear(t2);
mpc_clear(t3);
mpc_clear(_four);
mpc_clear(_two);
```

オブジェクトの解放

```
}
```

計算精度に関する数値実験結果

- 最適化オプション“-O3”, MPFRの丸めモードは最近接偶数丸め
- 任意精度ライブラリは4倍精度相当で計算

	言語	精度	結果
double	C	倍	- <u>3.6</u> 88406236100904979616609403840e-09
long double	C	拡張倍	- <u>3.678</u> 838531936214554367889299446e-09
long double@FX	C	128ビット	- <u>3.6787955329121653239</u> 60311671534e-09
real(kind=16)	F90	4倍	- <u>3.6787955329121653239</u> 60311671534e-09
libquadmath	C	4倍	- <u>3.678795532912165323</u> 7603627688244e-09
_Quad	C++	4倍	- <u>3.6</u> 884062361009050123322488651663e-09
QD	C++	疑似4倍	- <u>3.67879553291216532</u> 2960567157986e-09
fast_dd	C++	疑似4倍	- <u>3.6787955329121653</u> 94583236142940e-09
ARPREC	C++	32桁	- <u>3.6787955329121653239204992803347</u> e-09
exflib	C++	32桁	- <u>3.678795532912165323920499</u> 457919e-09
MPFR	C	106ビット	- <u>3.6787955329121653</u> 309585232663659e-09

計算時間[s]に関する実験結果 (実数)

	言語	PC (gcc)	FX100 (fcc)	FX100 (gcc)	CX400 (icc)
double	C	0.0105	0.0116	0.00546	0.0060
long double	C	0.0110	0.7882	0.781	0.0095
real(kind=16)	F	0.1050	0.0000	0.781	0.0746
libquadmath	C	0.0921	N/A	N/A	N/A
_Quad	C++	N/A	N/A	N/A	0.0714
QD (C++)	C++	0.0304	0.0732	0.0333	0.0798
QD (C)	C	0.0409	0.2884	0.0221	0.1364
fast_dd	C++	N/A	0.1441	0.144	N/A
ARPREC	C++	0.8968	6.6298	7.493	0.6934
exflib	C++	0.0856	N/A	N/A	0.1060
MPFR	C	0.3329	1.3545	0.597	0.3939

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果 (複素数)

	言語	PC / gcc	FX100 / fcc	FX100 / gcc	CX400 / icc
double	C	0.0227	0.0234	0.0228	0.0087
long double	C	0.0457	1.4669	1.288	0.0155
complex(kind=16)	F	0.3250	0.0000	1.453	0.2133
libquadmath	C	0.2141	N/A	N/A	N/A
_Quad	C++	N/A	N/A	N/A	0.2004
QD (C++)	C++	0.1349	0.3232	0.342	0.1552
QD (C)	C	<u>0.1334</u>	<u>0.2862</u>	<u>0.241</u>	<u>0.1472</u>
fast_dd	C++	N/A	0.7464	0.803	N/A
ARPREC	C++	1.5565	13.6597	5.515	1.3135
exflib	C++	0.2591	N/A	N/A	0.2864
MPC	C	0.1928	0.5868	0.422	0.2342

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果2 (実数)

	言語	PC / gcc	FX100 / fcc	FX100 / fcc(32reg)	FX100 / gcc
double	C	0.00679	0.00548	0.00546	0.00546
long double	C	0.00645	0.788	0.785	0.781
real(kind=16)	F90	0.0730	0.313	0.324	0.781
libquadmath	C	0.0636	N/A	N/A	N/A
QD (C++)	C++	0.0285	0.0333	0.0322	0.0692
QD (C)	C	0.0323	0.0221	0.217	0.0851
fast_dd	C++	N/A	0.144	0.143	0.155
ARPREC	C++	0.408	7.493	7.247	2.90
exflib	C++	0.0727	N/A	N/A	N/A
MPFR	C	0.166	0.597	0.574	0.469

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果2 (複素数)

	言語	PC / gcc	FX100 / fcc	FX100 / fcc(32reg)	FX100 / gcc
double	C	0.00542	0.0205	0.0205	0.0228
long double	C	0.0248	1.455	1.452	1.288
complex(kind=16)	F90	0.186	0.787	0.584	1.453
libquadmath	C	0.138	N/A	N/A	N/A
QD (C++)	C++	0.0579	0.173	0.128	0.342
QD (C)	C	0.0569	0.112	0.113	0.241
fast_dd	C++	N/A	0.743	0.733	0.803
ARPREC	C++	0.704	13.549	13.801	5.515
exflib	C++	0.205	N/A	N/A	N/A
MPC	C	0.132	0.528	0.487	0.422

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

FX100における疑似四倍精度実行列ベクトル積演算(QD(C++))の最適化メッセージ例

```
38         for (i = 0; i < n; i++) {
39     i         q[i] = 0.0;
40         #pragma loop noalias
41     2         for (j = 0; j < n; j++) {
42     i     2         q[i] += A[i*n+j] * p[j];
43     2         }
44     }
```

- ・SIMD化されず
- ・SWPLされず
- ・ループ展開2倍

...

jwd6101s-i "logisticmap_real_qd_quad.cpp", line 41: SIMD conversion is not applied because a statement that prevents SIMD conversion exists.

jwd8670o-i "logisticmap_real_qd_quad.cpp", line 41: This loop is not software pipelined because the loop contains a branch instruction which is not for loop iteration.

jwd8202o-i "logisticmap_real_qd_quad.cpp", line 41: Loop unrolling expanding 2 times is applied to this loop.

jwd8101o-i "logisticmap_real_qd_quad.cpp", line 42: Inline expansion is applied to the user defined function ' ZmlRK7dd_realS1 '.

jwd8101o-i "logisticmap_real_qd_quad.cpp", line 42: Inline expansion is applied to the user defined function ' ZmlRK7dd_realS1 '.

インライン展開のメッセージが大量にでる

FX100における疑似四倍精度複素行列ベクトル積演算(QD(C++))の最適化メッセージ例

```
40         for (i = 0; i < n; i++) {
41     i             q[i] = c_zero;
42         #pragma loop noalias
43             for (j = 0; j < n; j++) {
44     i             q[i] += A[i*n+j] * p[j];
45             }
46         }
```

- ・SIMD化されず
- ・SWPLされず
- ・ループ展開なし

...

jwd6101s-i "logisticmap_complex_qd_quad.cpp", line 43: SIMD conversion is not applied because a statement that prevents SIMD conversion exists.

jwd8670o-i "logisticmap_complex_qd_quad.cpp", line 43: This loop is not software pipelined because the loop contains a branch instruction which is not for loop iteration.

jwd8101o-i "/opt/FJSSVmxlang/bin/./include/c++/std/stl/_string_io.c", line 44: Inline expansion is applied to the user defined function '_ZNKSt8ios_base5flagsEv'.

jwd8101o-i "logisticmap_complex_qd_quad.cpp", line 44: Inline expansion is applied to the user defined function '_ZStmI7dd_realESt7complexIT_ERKS3_S5_'.

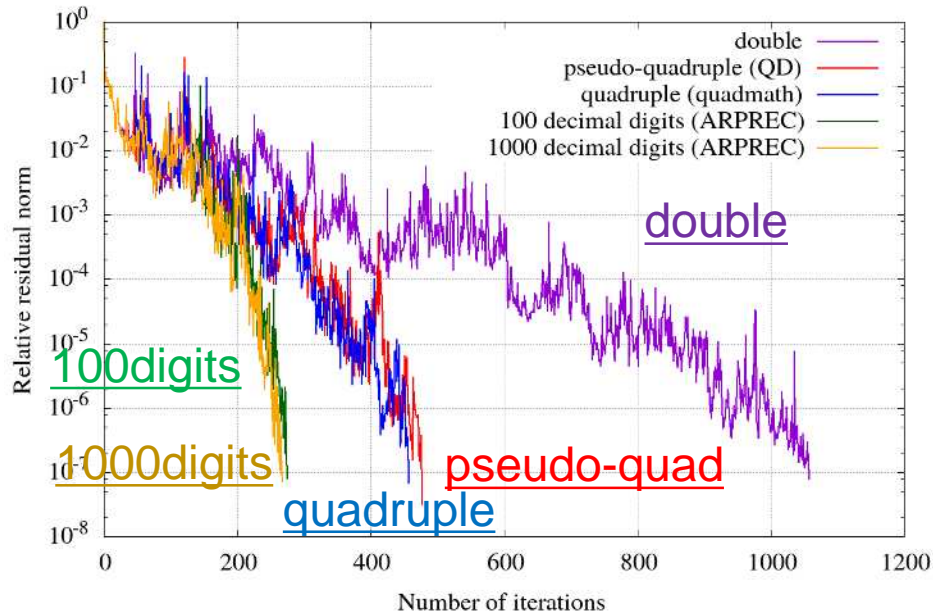
jwd8101o-i "logisticmap_complex_qd_quad.cpp", line 44: Inline expansion is applied to the user defined function '_ZNSt7complexI7dd_realEpLERKS1_'.

Numerical experiments on the convergence of the iterative method

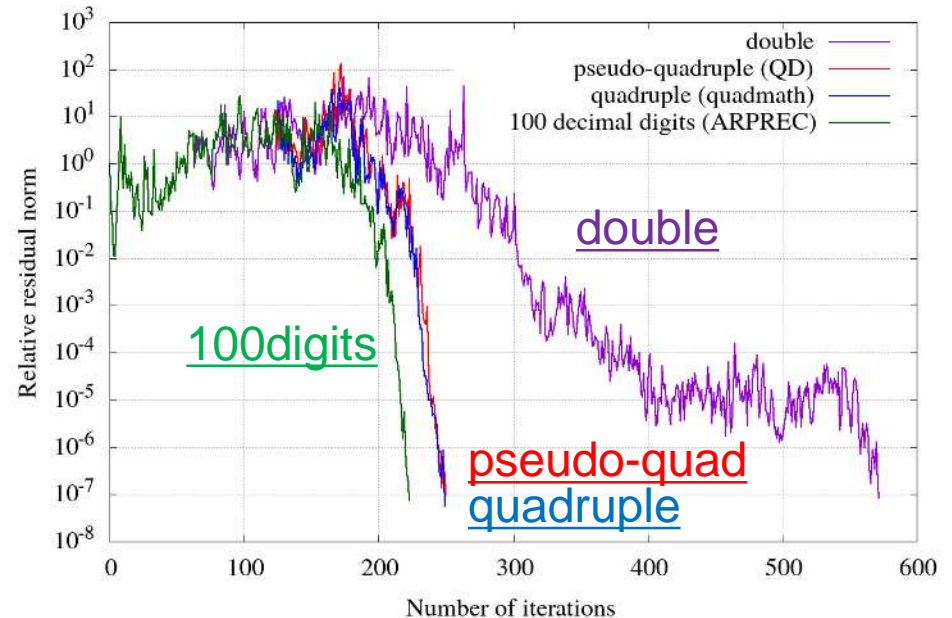
- Solving complex symmetric linear systems
$$A\mathbf{x} = \mathbf{f}$$
 - coefficient matrices (from the SuiteSparse Matrix Collection)
 - **dwg961b**: $n = 961$, num. of nonzero = 19,591
 - **qc2534**: $n = 2,534$, num. of nonzero = 463,360
an electromagnetic field problem and a complex symmetric matrix
 - $\mathbf{f} = A \times (1 + i, 1 + i, \dots, 1 + i)^T$
- Iterative methods
 - **COCG (Conjugate Orthogonal Conjugate Gradient) method** w/ incomplete Cholesky preconditioning
- Multiple-precision libraries
 - **libquadmath** (GCC 5.4.0), **QD** 2.3.17, **ARPREC** 2.2.18
- Computers
 - Intel Core i7-6500U / Ubuntu 16.04 TLS / gcc-5.4.0

Comparison of number of iterations with different precision arithmetic

Convergence criterion: 10^{-7}



Convergence history for dwg961b



Convergence history for qc2534

- “100digits” means 100 decimal digits of precision calculated by ARPREC as reference.
- In solving complex symmetric linear systems,
 - multiple-precision got faster convergence compared with double
 - (pseudo-)quadruple is enough precision

Comparison of calculation time with different precision arithmetic

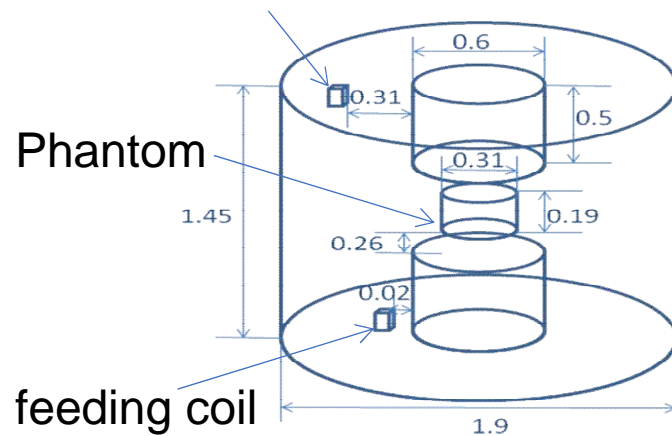
Convergence criterion: 10^{-7}

Matrix	Library	Precision name	Decimal digits	# of Iter.	Time [s]
dwg961b	non-use	double	15	1,058	0.480
	<u>QD</u>	pseudo-quadruple	32	479	<u>1.216</u>
	quadmath	quadruple	34	457	4.132
	ARPREC	pseudo-quadruple	32	383	32.503
qc2534	non-use	double	15	572	9.437
	<u>QD</u>	pseudo-quadruple	32	248	<u>20.841</u>
	quadmath	quadruple	34	249	73.272
	ARPREC	pseudo-quadruple	32	240	748.447

- QD is faster than GCC's libquadmath and ARPREC in total.
- QD takes **at most 3 times longer** than double precision in small-scale cases.

TEAM Workshop Problem 29

- Benchmark model for hyperthermia simulation
 - Problem size: 134,573
 - Frequency: 300 (MHz)
 - Relative permittivity of Phantom: 80.0
 - Electrical conductivity of Phantom: 0.52 (S/m)
- Linear solver
 - Iterative method: COCG w/ Symmetric SOR ($\omega = 1.05$) preconditioning
 - Convergence tolerance: relative residual norm $< 10^{-7}$
- Computers
 - Intel Core i7-6500U / Ubuntu 16.04 TLS / gcc-5.4.0

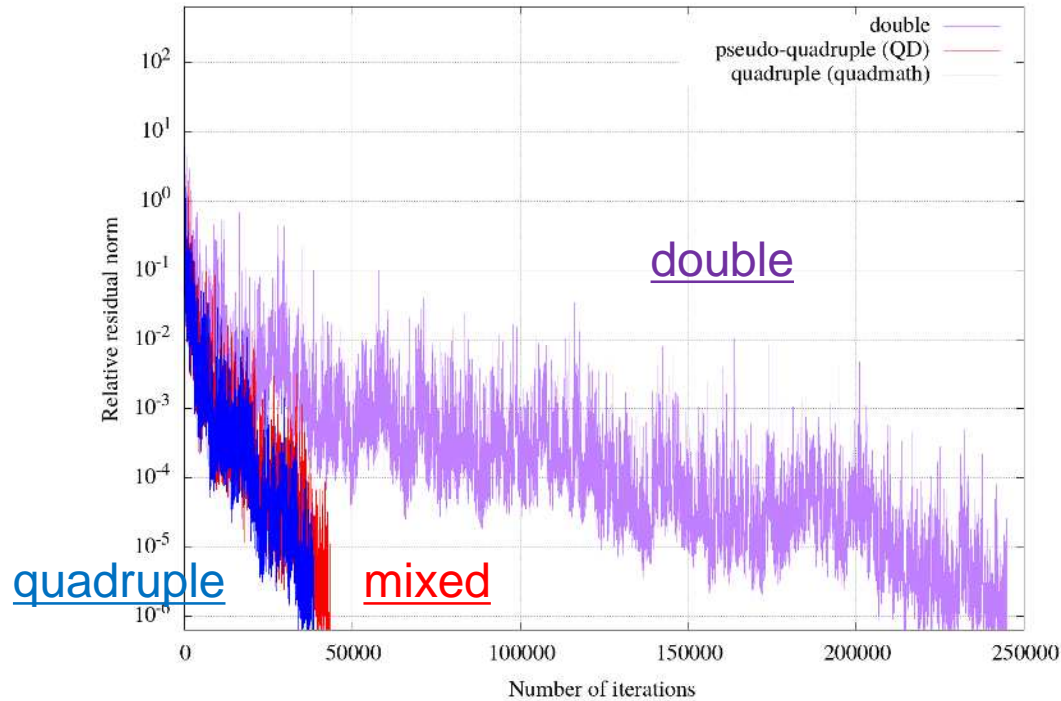


Model



Finite element mesh

Computational performance of 300 (MHz)



Precision	# of iter.	Time [hr.]
double	24,5105	5.78
pseudo-quadruple (QD)	43,598	<u>5.69</u>
quadruple (libquadmath)	38,643	14.09

■ IEEE 754-2008のbinary128形式

- ソフトでエミュレーションするため低速

■ double-double形式

- 倍精度を2つ並べた形式

- 倍精度演算器を使って計算できるためIEEE形式よりも高速

- 加減乗算の演算方法は広く知られている

- double-double形式の実装例

- フリーソフト：QDなど
- 富士通：高速4倍精度基本演算ライブラリ(以降fast_dd)
- 他ベンダー
 - IBMでは4倍精度としてdouble-double形式をサポートするシステム有り
 - NEC：ASLQUAD
- この他に、ユーザ自身が実装する場合もあり

■ QDの実装

- C++版はヘッダファイル内に演算が書かれており、インライン展開される
 - 翻訳時に最適化が適用されることで高速化を実現
- Fortran版は、wrapperのみでCの関数を呼び出している
- 注意点
 - C++版は、インライン展開されないと高速化されず、また、演算順序を変更する最適化が適用されると結果が正しくなくなる
 - Fortranは1要素ずつ関数を呼び出すため高速化されない

■ fast_ddの実装

- ユーザの指定する最適化オプションによって性能や結果に影響が出ることを避けるため、インライン展開方式は採用しない
 - 演算は基本的に1要素ずつ関数を呼び出すことで実現
- 高速化のために、マルチ関数(2要素ずつ計算)・ベクトル関数(入出力が1次元配列)を用意
- マルチ関数・ベクトル関数を使用して頂くことを期待
 - 7/20ご報告の評価で使われたロジスティック写像は漸化式でベクトル関数が使えないためQDのC++版(インライン)が高速

- 加算の性能
FX100 1.5Ghz
N=1024 (L1に乗るサイズ)

```

use fast_dd
INTEGER :: N
TYPE(dd_real) :: X(N), Y(N), Z(N)
DO I=1, N
    Z(I)=X(I)+Y(I)
END DO
    
```

マルチ関数、ベクトル関数は関数呼出し化した版を使用

		性能 mflops(注1)	備考
QD	C++	68.2	インライン、SWPLのみ
	Fortran	21.9	1要素ずつ関数呼出し
fast_dd 単体関数	C++	13.1	1要素ずつ関数呼出し
	Fortran	22.2	1要素ずつ関数呼出し
fast_dd マルチ関数	C++	43.0	2要素ずつ関数呼出し
	Fortran	114.0	2要素ずつ関数呼出し
fast_ddベクトル関数	C++	805.0	SIMD+SWPL
	Fortran	874.0	SIMD+SWPL

注1) double-double形式の加算1回を1flopとしたときの性能