RC-010

スーパーコンピュータ「京」における C++アプリケーションの評価

Evaluation of Compiler Optimization of C++ application on the K computer

千葉 修一[†1] Shuichi Chiba
ファム バン フック[†2] Pham Van Phuc
南 一生[†3] Kazuo Minami
青木 正樹[†1] Masaki Aoki

1. はじめに

古くから HPC 分野における研究者やアプリケーション開発者は，FORTRAN や C を中心とした言語を利用してプログラミングを行うことが一般的であった．しかし，オープンシステムを利用する分野では，規模が大きくなり複雑化するコードの生産性を確保するため新しい言語の開発が必要であった．その一つが，1979 年にベル研究所のコンピュータ科学者 Bjarne Stroustrup によって考案された，C を拡張したオブジェクト指向言語 C++である．

C++は，C の構造化プログラミングの考え方を最大限に生かし，より効果的にプログラムを組織化するためのカプセル定義，多重継承，仮想関数，テンプレート等の多種多様な機能を備え，プログラムの生産性や柔軟性を高めている．近年では，コミュニティを通じてオープンソースソフトウェアの開発が盛んに行われており，C++の特徴を活かし並行開発が行われている．

このようなオープンソースソフトウェアのアプリケーションおよびライブラリは，数値解析シミュレーションを中心とした HPC 分野でもプログラム開発の生産性を高めるために広く利用されるようになってきた．スーパーコンピュータ「京」(以降，「京」)[1]における利用課題選定，利用促進，利用支援業務の実施機関である一般財団法人高度情報科学技術研究機構の調べ[2]では，平成 24 年度から平成 27 年度までの「京」の産業利用課題で利用されているアプリケーションの上位３つの内２つが C++で開発されたアプリケーションであることが分かっている．中でも数値流体解析の CAE 分野で利用が拡大しているオープンソースのライブラリの一つである OpenFOAM[3] (Open source Field Operation And Manipulation)は，C++技術を最大限に活用し幅広い課題に対応することができている．

OpenFOAM は乱流，燃焼，および混相流など様々な物理モデルが用意されており，対象に合わせてソルバ，クラスライブラリを選択することができる．これらは，オブジェクト指向における Strategy パターンで実現されており，各種のアルゴリズムを動的に選択することが可能となっている．また，命令レベルの高速化を実現するために，高度な C++技術を利用している．その一つが Expression Template 技術の利用である．OpenFOAM を構成する基底クラスでこの技術を利用することで，関数呼び出し，および一時的なメモリ利用を削減し，演算コードの局所性を向上させる利点を得ている．

しかしながら，HPC 分野ではプログラミング技術による高速化だけではシステムの性能を十分に引き出すことはできず，コンパイラによる最適化も必要となる．演算器への命令供給をスムーズにするための命令スケジューリング，レジスタの効率的に利用するためのレジスタ割り付けなどがある．特に「京」では，HPC-ACE (High Performance Computing - Arithmetic Computational Extensions) と呼ばれる SPARC-V9 アーキテクチャに対する科学技術計算向けの拡張命令セットの追加，レジスタ数の拡張，SIMD (Single Instruction Multiple Data) 処理の搭載など様々な拡張が行われており，これらはコンパイラの最適化によって高い性能効果が得られる．

「京」では，Fortran コードを中心に多くの性能評価が行われているが，OpenFOAM のような高度な C++技術を利用するコード，および C++コンパイラに対する評価が非常に少ない．本論文では，ベンチマークコード，および OpenFOAM を利用した実アプリケーションのプロセッサ単体性能の評価を通して「京」における C++コードの実行性能を分析し評価する．

2. 「京」の CPU 概要

プロセッサ単体性能について評価する上で，「京」の CPU である富士通社製の SPARC64 ™ VIIIfx[4]の概要について述べる．1 つの CPU には 8 つのプロセッサコア，コア間共有の 2 次キャッシュメモリ (6MB, 12 way, Write back 方式)，メモリ制御ユニットを保持している．各コアは，1 次データキャッシュメモリ(32KB, 2way, Write back 方式)，4 つの積和演算器を保持している．レジスタは，SPARC-V9 の倍精度浮動小数点レジスタ 32 本に比べ，256 本と大幅に拡張されている．Table 1に CPU の詳細情報を示す．

Table 1 SPARC64™ VIIIfx Specification

	Specification
Peak performance	128GFLOPS (16GFLOPS×8cores)
Core	8
Floating-point Execution units (Core spec)	FMA : 2 x 2 (SIMD) DIVIDE : 2
	Floating-point register (64bit) : 256 General purpose register (64bit) : 188
Cache	L1I$: 32KiB (2way) L1D$: 32KiB (2way) L2$: Shared 6MiB (12way)
Memory throughput	64GiB (0.5B/F)
Clock	2.0GHz
Chip size	22.7mm x 22.6mm
Number of transistors	760M
Power	58W (30℃, Water Cooling)

†1 富士通株式会社
FUJITSU LIMITED
†2 清水建設株式会社
SHIMIZU Corporation
†3 国立研究開発法人理化学研究所
RIKEN

3. ベンチマークコードによる評価

3.1 ベンチマークコードの概要

ベンチマークコードとして，姫野ベンチマーク[5]を利用する．姫野ベンチマークは，非圧縮流体解析における性能評価を行うことができ，HPC 分野の性能評価では広く利用されているベンチマークである．実装は，ヤコビ法を利用し Poisson 方程式の解を得るものである．

本評価用に Fortran で記載されたオリジナルコードを基に C++コードで実装した新たなベンチマークコード(himenoBMT@C++)を作成した．この時，C++技術を使い分け 3 種類のコードを生成し，それぞれの性能を比較評価することで C++技術と実行性能の関係を評価する．

Figure 1の array type は，Fortran コードと同様に解析データを静的な配列宣言で記述したものである．コンパイラが最適化を適用する時，静的にデータサイズが決定することで，より高度な最適化を適用することができ，Fortran コードと同等の性能が期待できる．

```
static float  p[MI][MJ][MK];
static float  a[4][MI][MJ][MK],
              b[3][MI][MJ][MK],
              c[3][MI][MJ][MK];

jacobi(arguments) {
  ...
  for(condition) {
    s0 = a[0][i][j][k] * p[i+1][j  ][k  ]
       + a[1][i][j][k] * p[i  ][j+1][k  ]
    ...
```

Figure 1 himenoBMT@C++ array type

Figure 2の pointer type は，解析データを動的に確保し，ポインターを利用してアドレス計算を行いアクセスするものである．ポインターを利用したコードは，コンパイラによるメモリアクセスの依存関係が解析できない場合，最適化を適用することが難しく性能低下につながる．

```
struct mat {
  float* addr;
  int rows, cols, deps; // dimensions
} *p, *a, *b, *c;
#define VA(mat,n,r,c,d) ¥
  mat->addr[ ¥
    (n) * mat->rows * mat->cols * mat->deps + ¥
    (r) * mat->cols * mat->deps + ¥
    (c) * mat->deps + ¥
    (d) ]

jacobi(arguments) {
  ...
  for(condition) {
    s0 = VA(a,0,i,j,k) * VA(p,0,i+1,j,  k)
       + VA(a,1,i,j,k) * VA(p,0,i,  j+1,k)
    ...
```

Figure 2 himenoBMT@C++ pointer type

Figure 3の template type は，C++で利用頻度の高い Boost[6]を利用したものである．Boost はテンプレートを利用して実現されており，生産性を向上させる上で非常に有効なライブラリである．ただし，テンプレートや operator キーワードが利用されるため，暗黙的に関数呼び出しが多く登場するコードとなる．そのため，演算コードの局所性が低くなり命令スケジューリングの最適化を適用することが難しい．

```
new_Mat(Matrix *& Mat, int nums,
        int rows, int cols, int deps)
{
  Mat = new boost::multi_array<float,4>(
    boost::extents[nums][rows][cols][deps] );
  return (Mat != NULL) ? 1:0;
}
main(){
  new_Mat(a,4, imax, jmax, kmax);
  ...
  gosa= jacobi(nn,*a,*b,*c,*p,*bnd,*wrk1,*wrk2);
}
jacobi(arguments) {
  ...
  for(condition) {
    s0 = a[0][i][j][k] * p[i+1][j  ][k  ]
       + a[1][i][j][k] * p[i  ][j+1][k  ]
    ...
```

Figure 3 himenoBMT@C++ template type

3.2 ベンチマークコードの実行結果および分析

それぞれ，Figure 1の array type は Fortran コード，Figure 2の pointer type は C コード，Figure 3の template code は C++コードで利用されるプログラミングモデルを想定した実装といえる．「京」において，これらのコードの 1 コア実行性能(MFLOPS)をオリジナルの Fortran コードの実行性能と比較したデータをFigure 4に示す．以降，本論文で利用するコンパイラは，2016 年 1 月現在「京」で公開されている富士通コンパイラ FCCpx を利用し，翻訳時オプションは，すべて-Kfast とする．Figure 4のデータでは，Fortran の実効性能を 100%とし，その比率を表している．表の上位に行くほど性能が高いことになる．

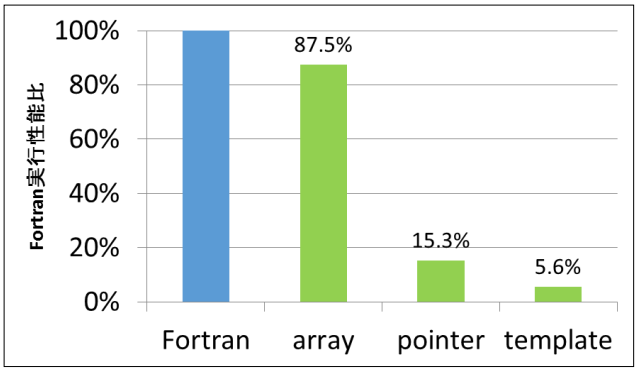

Figure 4 himenoBMT@C++ 実行性能の比較

array type は，Fortran とほぼ同等の性能となっている．性能差は言語規格の違いにより最適化の適用量が変化したと考えられる．Fortran は科学計算に特化した言語であるため，ループ内の制御変数（ループの回転に利用する変数）の更新が許可されていない．そのため，コンパイラは，メモリアクセス先などで制御変数が更新されることがないため，最適化を広く適用することができる．これに対し，C++規格では，ループ内で制御変数の更新が許されている．制御変数の更新が不明確な場合は，制御変数を参照する命令に対する一部の最適化が抑止されている．ただし言語規格としては，柔軟性が高いといえる．

pointer type は，1 つの配列要素へアクセスするごとに，ポインター変数(たとえば a)を経由したメモリアクセスが 6 回行われる．コンパイラの最適化によりこれらのオーバーヘッドは最低限に抑えられるが，命令スケジューリングにおいてメモリアクセス命令の依存関係が判断できず，命令の並列性が低下していることが考えられる．そこで Figure 5のように宣言時に restrict キーワードを付加し，各ポインター変数がアクセスするメモリに依存がないことを指示する．

```
struct mat {
  float* addr;
  int rows, cols, deps; // dimensions
};
struct mat * restrict p;
struct mat * restrict a;
struct mat * restrict b;
struct mat * restrict c;
```

Figure 5 restrict キーワードの指示

これにより Figure 6のように array type とほぼ同等の性能となり，ポインター変数に対する最適化の影響で性能差が生じていることが確認できた．pointer type では，必要に応じて restrict キーワードを付与することが有効なチューニングの一つであると言える．

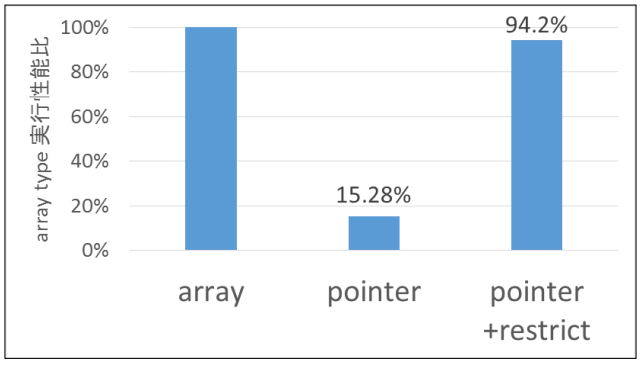

Figure 6 restrict キーワードの効果

template type は，pointer type のようなポインターアクセスに加え，テンプレートの実装となる暗黙的な関数呼び出しが含まれる．オブジェクトの生成単位で動作するコンストラクタ，デストラクタもこれに含まれ，作業単位で多くの関数呼び出しが行われている．そこで，アセンブリコード上に静的に登場する関数呼び出し箇所を計上したものをFigure 7に示す．ここでは，コンパイラによる最適化の有無による変化も記載している．

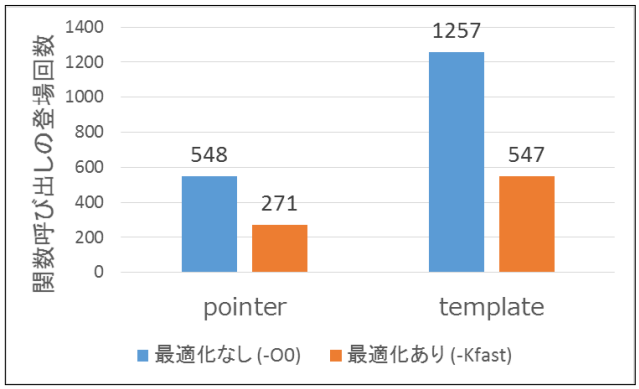

Figure 7 関数呼び出しの登場回数の比較

このように template type では，pointer type と比較しても 2 倍の関数呼び出しが残存している．これらのほとんどは暗黙的な関数呼び出しであり，手修正による性能チューニングが難しい．そのため，コンパイラの最適化への期待は高い．実際に利用されるアプリケーションの場合は，高い生産性を確保するため，データごとにクラス化されるため，明示的な関数呼び出しも増加する．

前述したように，生産性の高い template type は，C++のコードとして一般的であり，HPC 分野でも増加している．template type のコードを評価することは非常に重要である．そこで，template type に近い実アプリケーションを利用し C++コードをさらに検証する．

4. 実アプリケーションによる評価

実アプリケーションは，「京」での実績も多い OpenFOAM を利用したシミュレーションコードを用いる．前述した template type に近いコードであり，多くの C++技術が使われている．

4.1 アプリケーションの概要

今回は，「京」における大規模並列の評価報告[7][8][9]があった Pham Van Phuc らのコードを利用して評価を行う．解析モデルは，風洞測定部を再現するもので，解析領域は長さ 16m ×幅 3m×高さ 2m を想定し，1 プロセスあたりの計算格子数は 262,144 (=64×64×64)となっている．解析ソルバは，OpenFOAM 2.2.2 で提供される pisoFOAM を利用している．pisoFOAM は，PISO 法(Pressure-Implicit with Split Operators)のアルゴリズムを搭載したソルバとなっており，単層流非圧縮性解析を行う．実装は，初期化，運動方程式，および圧力方程式の 3 つの処理区間に分けられる．

Figure 8，Figure 9，Figure 10に示すように各区間のコードは，処理を抽象化したファイル，またはクラスを呼び出すことで実現されている．特に各方程式のコードでは，一般的な数値解析で記述される行列ベクトル積など計算コードが登場しない．これらは，C++の機能である

operator キーワードを利用し演算子をオーバーロードすることで実現されている．オーバーロードされた演算子は暗黙的な関数呼び出しとなるため，他の開発言語で実装したコードと比較して暗黙的な関数呼び出しが多くなる理由の一つである．

```
#include "setRootCase.H"

#include "createTime.H"
#include "createMesh.H"
#include "createFields.H"
#include "initContinuityErrs.H"
```

Figure 8 初期化のコード

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
);

UEqn.relax();

if (momentumPredictor)
{
    solve(UEqn == -fvc::grad(p));
}
```

Figure 9 運動方程式のコード

```
volScalarField rAU(1.0/UEqn.A());

U = rAU*UEqn.H();
phi = (fvc::interpolate(U) & mesh.Sf())
    + fvc::ddtPhiCorr(rAU, U, phi);

adjustPhi(phi, U, p);

// Non-orthogonal pressure corrector loop
for (condition) {
    // Pressure corrector

    fvScalarMatrix pEqn
    (
        fvm::laplacian(rAU, p) == fvc::div(phi)
    );

    pEqn.setReference(pRefCell, pRefValue);
    ...
```

Figure 10 圧力方程式のコード(一部)

4.2 アプリケーションの実行結果および分析

「京」で利用可能な詳細プロファイラ fapp を利用し取得した 1 コアにおける実行時間の処理内訳をFigure 11に示す．

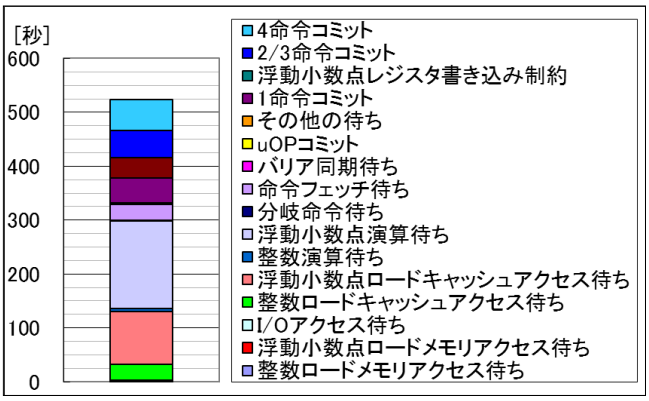

Figure 11 詳細プロファイラによる実行時間の内訳

詳細プロファイラの情報から実行時間が増加する要因として命令フェッチ待ち，浮動小数点演算待ち，浮動小数点ロードキャッシュアクセス待ち，および整数ロードキャッシュアクセス待ちが挙げられる．これらの待ちが発生する要因として２つの原因が考えられる．一つは，命令が集中することで演算器の処理が追いつかず，処理待ちが発生しているケース．Figure 11では，浮動小数点演算待ちの割合が多く，浮動小数点演算が集中していることが想定される．またもう一つは，命令のスケジューリングが最適に行われておらず，スーパースカラ処理における命令の実行サイクル(レイテンシ)が隠蔽できていないケース．これらの要因は，コンパイラによるソフトウェア的なスケジューリング(ソフトウェアパイプライニング)やハードウェアによるスーパースカラ処理に依存するため，直接的に原因を検証することができない．そのため，PA イベントカウンタや翻訳時オプションを変化させた性能データなど多面的な情報から分析を行う．

まず，コード内で利用されている命令の種類を把握するため，詳細プロファイラ情報から命令数の割合を比率化したグラフをFigure 12に示す．浮動小数点のメモリアクセス，および整数系の命令が全体の 80%を占めており，浮動小数点の演算は 20%程度しか存在しないことが分かる．この内容から，浮動小数点演算命令が集中したことが要因である可能性は低いと判断する．

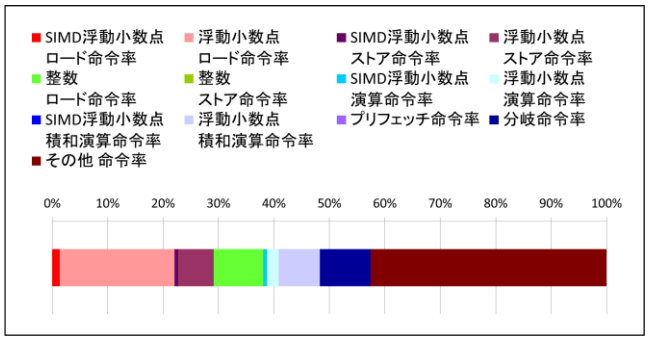

Figure 12 実行命令の比率

次に命令スケジューリングの効果を検証する．命令スケジューリングの最大の効果は，命令の並列性を解析し命令の追い越しを発生させることで，スーパースカラ処理を効率的に適用することである．そのため，命令の並列性の解析を阻害する要因があった場合は，最適なスケジューリングを行うことができない．C++コードでは，前述したように関数呼び出しが多用される．この関数呼び出しは，関数の呼び出し元，呼び出し先コードに登場するメモリアクセス命令の依存が不明確になる．そのため，関数呼び出しが性能を低下させる要因となる．そこで，コンパイラはインライン展開を適用し演算コードを集約することで命令スケジューリングを促進する．ただし，すべてのコードを一か所へインライン展開した場合は，命令キャッシュミスやレジスタ不足が発生する可能性もある．そのため，コンパイラは性能のホットスポットを静的に解析し，インライン展開を効果的に行うことで，命令スケジューリングを促進している．

そこでインライン展開に着目し検証を行う．まず本アプリケーションにおいて，関数がどのくらい深く呼び出されているか解析した結果をFigure 13に示す．ただし，再起呼び出しによる関数呼び出しはカウントしていないため，呼び出し回数は完全なデータではない．このデータから判断できることは，関数呼び出しが非常に深いことである．

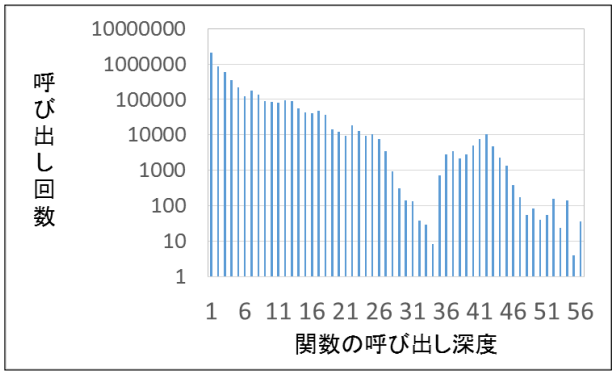

Figure 13 関数呼び出し深さ

最大 56 となる深さをもつ関数呼び出しに対して，コンパイラによるインライン展開がどのように効果しているかを検証する．コンパイラの翻訳時オプションを利用しインライン展開の有無による実行時間を測定し，比較したグラフをFigure 14に示す．

初期化，運動方程式では大きな性能効果が得られていることが分かる．これらの区間は，シンプルなループ構造で構成されている．そのため，インライン展開を行わない場合は，浮動小数点命令（浮動小数点演算，および浮動小数点メモリアクセス命令）が集約されず十分な命令スケジューリングができていないことが考えられる．

また，大きな効果がなかった圧力方程式は，ループ内の処理が複雑である．そのためインライン展開を適用しない場合でも，ある程度の浮動小数点命令が集約されており，命令スケジューリングの効果があったと考えられる．

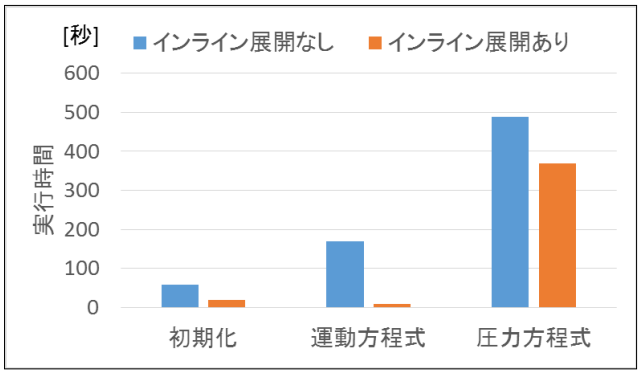

Figure 14 インライン展開による実行時間の比較

各区間の浮動小数点命令数はTable 2のようになっている．圧力方程式の区間は非常に多くの浮動小数点命令が処理されていることが分かる．命令の局所性は判断できないが，豊富な命令数の中で命令スケジューリングが行われることで，インライン展開されない場合でも，命令の並列性が確保できたと考える．

Table 2 各区間の浮動小数点命令数

初期化	運動方程式	圧力方程式
495, 663, 402	8, 388, 370, 175	334, 056, 721, 240

またインライン展開の効果は，展開されたコードによる命令スケジューリングの促進だけではない．関数呼び出しによるオーバーヘッドの軽減である．関数呼び出しは，呼び出し元の状態保存や呼び出し先へのデータ渡しを行う必要があり，これらの作業はオーバーヘッドとして遅延につながる．SPARC64™ VIIIfx アーキテクチャでは，関数呼び出しによるオーバーヘッドの軽減策としてレジスタウィンドウが搭載されている．

レジスタウィンドウは，レジスタファイルをウィンドウという単位で区切り関数呼び出し単位でウィンドウを切り替える．この時，Figure 15のように呼び出し元と呼び出し先でレジスタを共有することでレジスタ内容の退避やデータの受け渡しによるオーバーヘッドを軽減する．

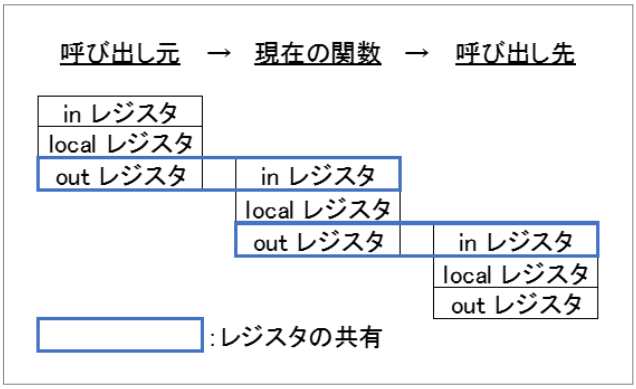

Figure 15 レジスタウィンドウ

しかしながらレジスタウィンドウの切り替え自体にもオーバーヘッドは存在する．たとえば，レジスタウィンドウの移動は，CALL 命令の呼び出し時に起こるものではなく，SAVE 命令，および RESTORE 命令の発行契機に CWP (Current Window Pointer)が加減算されることで行われる．そのため，命令の実行サイクル分のオーバーヘッドが必要となる．また，レジスタウィンドウの切り替え時は，内部的な排他処理が行われるため命令発行が抑止される．インライン展開はこれらのオーバーヘッドを削減していると言える．そこで，インライン展開とレジスタウィンドウの切り替えによるオーバーヘッドの関係性を検証する．レジスタウィンドウの切り替えを可視化するために，SPARC64™ VIIIfx Extensions で公開されている PA イベントカウンタを利用する．イベント情報として，実行区間の CPU サイクルを計上する cycle_counts，およびレジスタウィンドウの切り替えにより命令発行ができなかったサイクル数を計上する regwin_intlk を計測する．

regwin_intlk の情報は，詳細プロファイラ fapp では計測することができない．そのため，PCR(Performance Control Register)，および PIC(Performance Instrumentation Counter Register)を直接利用して計測する評価ツールを作成し検証を行った．運動方程式，および圧力方程式における計測データをFigure 16に示す．

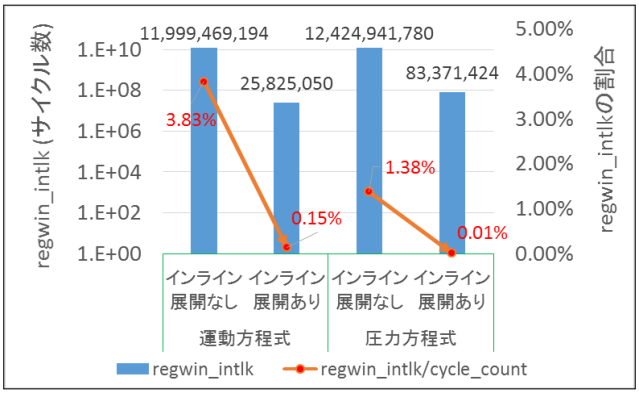

Figure 16 レジスタウィンドウ関連の性能情報

インライン展開前の計測における regwin_intlk のサイクル数は，運動方程式，圧力方程式ともに同じような値となっている．ただし，全体の実行サイクル数との比率を取ると，運動方程式は 3.83%，圧縮方程式が 1.38%となっている．インライン展開による regwin_intlk の減少率を見た時，運動方程式の改善率が高いことが分かる．実際には，レジスタウィンドウの切り替えごとに SAVE 命令，および RESTORE 命令の実行サイクルも加算されるため，本データ以上に性能効果があると言える．この結果から，Figure 14において運動方程式で大幅な改善が見られたことについては，レジスタウィンドウの切り替えによるオーバーヘッドの削減が少なからず影響していると言える．

これらの結果から C++アプリケーションの評価において，レジスタウィンドウの切り替えによるオーバーヘッドの計測は重要であり，PA イベントカウンタ regwin_intlk の情報は有効な指標であると言える．

4.3 分析結果に基づいた性能改善

分析の結果からインライン展開による命令スケジューリングの促進に着目し，「京」での性能チューニングを試みる．コンパイラでは，インライン展開の適用度合を指示するための翻訳時オプションが用意されている．これらの翻訳時オプションを利用し実行性能を計測する．

コンパイラは，翻訳時オプション-Kfast を指定することで，インライン展開を制御するオプション-x-が自動的に指定される．-x-はコンパイラの判断によりホットスポットに対してインライン展開を適用するオプションである．今回は，コンパイラの判断ではなく，強制的にインライン展開を促進する指示を与えることで，実行性能の改善効果を検証する．

強制的なインライン展開は，Figure 17のようにコンパイラの翻訳時オプションに-x*NUM* を指定することで適用できる．コンパイラは，関数に含まれる実行文の数が *NUM* に指定された値以下の関数を対象にインライン展開を適用する．

```
% FCCpx -xNUM foo.cpp

  -xNUM: 実行文の数が NUM で指定された値以下の
         関数をインライン展開の対象とする
  -x-  : コンパイラが自動判断
```

Figure 17 強制的なインライン展開の翻訳時オプション

NUM に対して 10，50，100 を指定した場合の実行時間と-x-オプションを指定した場合の実行時間の比較をFigure 18に示す．NUM の数値が大きいほど実行文の数が多い関数もインライン展開の対象になるため，より多くの関数が展開されることになる．また，展開する関数が増えることで翻訳時間も増加するため大きな値を指定する場合は注意が必要であり，100 より大きな値の測定は見送った．今回の結果からは，展開数を増加した場合でも，-x-と比べて有効な実行性能は得られなかった．

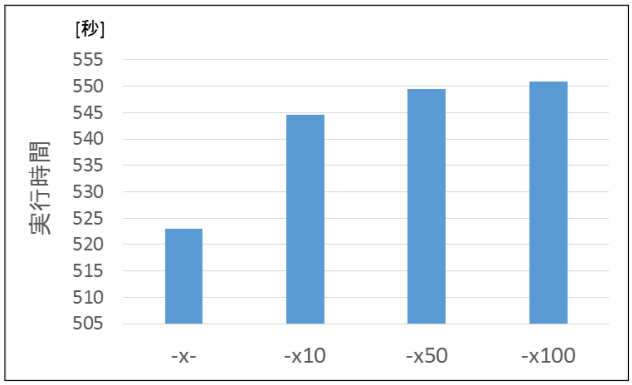

Figure 18 強制的なインライン指示による実行時間の変化

実行時間だけではインライン展開の適用効果が判断できないため，レジスタウィンドウの切り替えによるオーバーヘッドを検証するため，PA イベントカウンタ regwin_intlk を計測する．今回は，インライン展開による

性能が支配的であった運動方程式を計測した結果をFigure 19に示す．-x10 から-x100 まではインライン展開が促進されたことで，regwin_intlk の値が減少していることが分かる．しかし，-x-は実行文の数に関係なくホットスポットに対してインライン展開を適用するため，regwin_intlk の値が一番小さくなっている．また，運動方程式区間の実行サイクル数 cycle_counts との比率を見た時も-x-による計測値が一番低くなっており，レジスタウィンドウの切り替えによるオーバーヘッドが一番低いことが分かる．

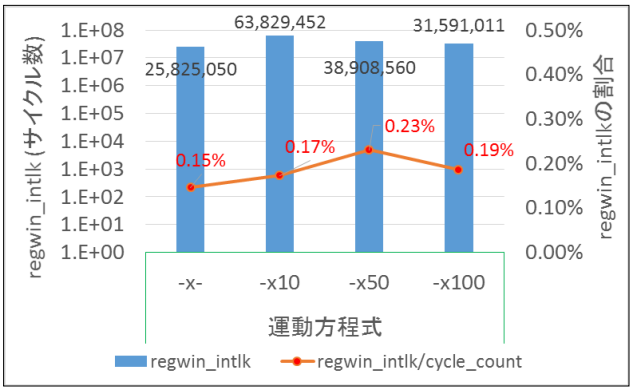

Figure 19 運動方程式における regwin_intlk 情報

本アプリケーションについては，強制的にインライン展開した場合，性能が悪化することが確認された．この結果から，実行文の数がインライン展開の有効性と単純には直結していないと言える．また，性能チューニングにおいても regwin_intlk を利用した評価は有効である．

5. まとめ

姫野ベンチマークと OpenFOAM の 2 種類のベンチマークコードを評価することで C++コードの特性に応じた性能，および特徴を評価することができた．また，C++で主流となる template type に関しては，実アプリケーションでの評価も行った．実アプリケーションにおいてインライン展開の効果を検証することは重要であること．またその評価指標の一つとして PA イベントカウンタ regwin_intlk によるレジスタウィンドウの切り替えのオーバーヘッドの計測が有効であることが導き出せた．C++アプリケーションの普及に合わせて，PA イベントカウンタ regwin_intlk の必要性を発信し，標準ツールで取得できるように働きかけたい．

今後は，他の実アプリケーションに対する評価も行い C++アプリケーション特有の評価情報を拡充したいと考えている．

謝辞

本報告の結果は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものである．ここに記して謝意を表する．

参考文献

[1] 理化学研究所：「京」のシステム概要, http://www.aics.riken.jp/jp/k/system.html
[2] 新宮 哲："「京」を中核とする HPCI 産業利用のご案内", 第 3 回 OpenFOAM ワークショップ発表資料, 2015
[3] OpenFOAM：http://www.openfoam.com/
[4] Fujitsu Limited：SPARC64™ VIIIfx http://www.fujitsu.com/downloads/JP/archive/imgjp/jhpc/sparc64viiifx-extensionsj.pdf
[5] 理化学研究所情報基盤センター：姫野ベンチマーク, http://accc.riken.jp/supercom/himenobmt/
[6] Boost C++ Libraries http://www.boost.org/
[7] ファム バン フック, 井上 義昭, 浅見 暁, 内山 学, 千葉修一："「京」コンピュータでの C++型流体コードにおける MPI の評価"第 151 回ハイパフォーマンスコンピューティング研究会, pp1-9(2015)
[8] PHAM VAN PHUC, 内山 学：" OpenFOAM の超大規模解析「10^5 並列規模の計算テクニック」", 第 3 回 OpenFOAM ワークショップ発表資料, 2015
[9] 千葉 修一："「京」コンピュータにおける富士通の取り組みと OpenFOAM の評価", 第 3 回 OpenFOAM ワークショップ発表資料, 2015