Vol.2018-HPC-165 No.35
2018/8/1

ハードウェア階層構造を持つHPCシステム向けステンシル計算コードの自動最適化プラットフォームの提案と評価

石村 脩[,a)]　吉本 芳英[,b)]

概要：ドメイン特化言語（DSL）はユーザプログラムから大規模計算機の複雑性を隠蔽し、その移植性を向上する有望な手法だが、これら自身の移植性が低いのが課題である。そこで、計算機システムの階層構造に対応して再帰的な最適化を行う機構を備えたステンシル計算向け DSL のプラットフォームを開発した。この提案では、各システム向けの最適化機構をアスペクト指向プログラミングによって各階層要素向けの最適化機構を組み合わせて構築することで移植性を高める。

キーワード：ドメイン特化言語, アスペクト指向プログラミング

Osamu Ishimura[,a)]　Yoshihide Yoshimoto[,b)]

1. はじめに

近年、HPC システムの構造は複雑化の一途をたどっている。メニーコアプロセッサや GPU が導入され、不揮発性メモリや FPGA、DSP などの導入も検討されている。このように複雑な計算機システム向けに、最適化されたアプリケーションを作成するのは難しい。また、特定のシステムに対する過度な最適化は、アプリケーションの最適化を損なう。これは、今後システム構成の多様化がよりすすむことで、アプリケーション開発における大きな問題となることは想像に難くない。

上記の問題を解決する手法として、ドメイン特化言語 (DSL) は有望なアプローチの一つである。前者は村主らの formura [1,2] や村山らの Physis [3] など多くの研究が行われている。しかし、これらの研究では、ドメイン特化言語のアプリケーション基盤は、ある特定の計算機システムや、特定の計算機システム構成の組み合わせにのみに対して作成されていた。

しかし、これらの開発基盤をより汎用に作成することが可能となれば、開発リソースの集中を行うことが可能となり、HPC アプリケーション開発の高速化を行うことが可能となる。理想としては、計算機システムの管理者が、アプリケーション開発基盤の 1 ユーザーとして容易に特定システム向けに最適化された開発基盤を提供できることが望ましい。

そこで本研究では、計算機システムの構成からボトムアップ的に構築可能な、対象システム向けにボトムアップ的に構築可能なアプリケーションの適応と最適化を行う開発基盤の提案、およびステンシル計算向けのプロトタイプの開発と評価を行った。

ステンシル計算では、並列計算機向けの適応は主にタイリングによって行われる。タイリングの行われるデータ領域はグリッドの集合体であり、タイリングによって各グリッドの計算処理は変更されない。つまり、ステンシル計算のタイリングを用いた適応は対象システムの各演算器へのグリッドの割り付け問題と考えることが可能である。同様に、ステンシル計算の最適化アルゴリズムの一つで、キャッシュ効率および通信効率の向上に利用されるテンポラルブロッキングも上記の特徴をもつ。本開発基盤は、上記の構造に着目し、計算機システムが持つ階層構造毎に対応するモジュールとして”階層の制御コード”・”データ領域の割り付け”・”データ転送”を実装し、これらの組み合わせで計算機システム向けの適応・最適化基盤を出力する。

本開発基盤では各機能をアスペクト指向プログラミング (AOP) で記述する。アプリケーションコードに置ける各階層向けの適応・最適化を行う部分コードを横断的関心事と捉え、それぞれを AOP におけるアスペクトとして実

†1 現在，東京大学
Presently with The University of Tokyo
a) oishimura@is.s.u-tokyo.ac.jp
b) yosimoto@is.s.u-tokyo.ac.jp

装する。各アスペクトは、対象とする階層の制御・ブロッキング・メモリ管理のコードを追加する機能を持つ。アプリケーションの適応・最適化の際には、アプリケーションのロジックのみが書かれたコードに対して、対象システムに対応するアスペクトを選択し、階層の順序に合わせて適用 (AOP における織り込み) する。各アスペクトによって、コードは変更されるが、あくまで自己相似性を持つ変換のみが行われるため、他のアスペクトを同様に適応可能なフォーマットの出力が行われる。

本開発基盤の実装は AOP の処理系の一つである AspectC++を用いて実装した。対象の階層構造として、分散メモリ並列 (MPI)・共有メモリ並列 (OpenMP)、CPU キャッシュ向けの適応および最適化を行うモジュールを作成した。簡単なベンチマーク用ステンシル計算アプリケーションを作成し、Oakforest-PACS および Reedbush 上で性能評価を行い、手書きのコードと比較して、大きなパフォーマンスの劣化がない事を確認した。

本稿の構成は下記のとおりである。まず、第二節〜第四節でアスペクト志向プログラミング・ステンシル計算・テンポラルブロッキングについて説明する。次に、第五節及び第六節で本開発基盤の設計の詳細を説明する。そして、第七節で関連研究の紹介を行い、最後に第八節で本稿のまとめと今後の展望について述べる。

2. Aspect Oriented Programming

アスペクト指向プログラミング (AOP) は、G. Kiczales らによって提案された、オブジェクト指向プログラミング (OOP) を拡張したプログラミングパラダイムである [4, 5], オブジェクト指向はプログラム処理のモジュール化を行うのに有用なアプローチであるが、依存関係の形式によっては取り除くことができないものもある。このようなものを AOP にでは横断的関心事と呼ぶ。

図 1 は、横断的関心事 (Cross-Cutting Concern) のイメージ図である。この図では、縦の線が各オブジェクトの処理、色で塗られた部分が分散した他のオブジェクトに依存関係を持つコード片である。オブジェクト A,B,C のぞれぞれの中に、オブジェクト D の処理が埋め込まれている。AOP ではこれらの処理を図 2 のように、オブジェクト A,B,C から分離し、一つのオブジェクトとしてまとめる。このオブジェクトのことを AOP ではアスペクト (Aspect) と呼ぶ。

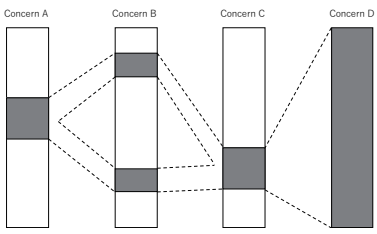

図 1: Distribution of Crosscutting Concern

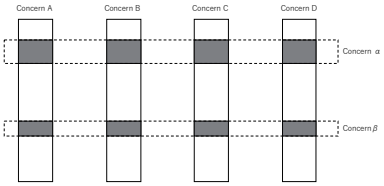

図 2: Partition of Crosscutting Concerns by Aspects

アスペクトはアドバイス (Advice) と内部型宣言 (Inter-type declarations) の二つで構成される。アドバイスは挿入や上書きされる処理、内部型宣言は変数や関数をオブジェクトに追加する処理である。

AOP を実現するモデルにはいくつかあるが、もっとも一般的なものは AspectJ や AspectC++で用いられるジョイントポイントモデル (JPM) である。JPM では元のオブジェクトの処理の中で、処理が挿入される箇所のことをジョイントポイント (Joint-Point) と呼ぶ。ジョイントポイントは、ポイントカットと呼ばれるアスペクトに含まれる条件宣言によって作成される。

また、コンパイル時やプログラム実行時に、ポイントカットによって変更箇所の特定がされアドバイスの処理が適応されることを織り込み (Weave) と呼ぶ。

図 3 は織り込みの処理例である。それぞれの縦線が、プログラムの処理を示す。図では、○の処理の前後に処理内容のパターンマッチでジョイントポイントが生成され、○の処理が☆の処理に置換されている。

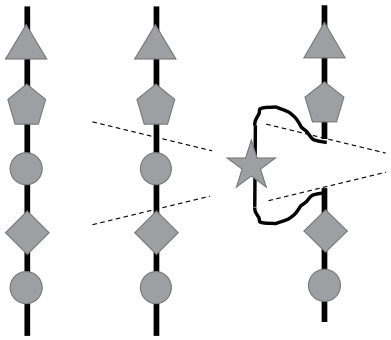
図 3: Aspect Weaving Process

2.1 AspectC++

AspectC++は Olaf Spinczyk らによって提案された C++の AOP 拡張である [6, 7]。AspectC++コンパイラが、AspectC++のコードをテンプレートを利用した C++のコードに変化する。AspectC++のコードは C++のコードとアスペクトコードの二つで構成される。まず、C++のコードは通常の C++のコード。そして、アスペクトコードは C++のプログラムコードのパターンマッチを行うポイントカットと、C++のコードで書かれたアドバイスで構成される。AspectC++のポイントカットは Name-Pointcut

と Code-Pointcut の二種類で構成される。Name-Pointcuts では型やクラス、変数、関数、名前空間など静的な要素に対するパターンマッチ表現を行う。Code-Pointcut では関数呼び出し・関数実行・変数書き込み・変数読み込みなどプログラムのコントロールフローに対する指定を行う。さらに、AspectC++は C++11 に対応しており、C++11 の属性表現をポイントカットで用いることが可能である。AspectC++のアドバイスは、単なる処理だけではなく、さらに追加の属性を持つことが可能である。追加の属性では”before”, ”after”, ”around”と、そのジョイントポイントに対して、いつアドバイスを実行するかを指定することが可能である。一つのジョイントポイントに対して、複数のアスペクトが適応された場合は、別途指定された優先順序にしたがって順番にすべてのアスペクトが織り込みされる。

3. ステンシル計算

ステンシル計算は計算科学において登場するもっと主要な計算パターンの一つであり、地震シミュレーションや画像処理、電磁界シミュレーションなどにおいて用いられる。

ステンシル計算のデータ構造は d 次元の配列である。配列内の各要素は各ステップごとに周囲の要素の値を用いて更新される。もし、境界要素を考慮しない場合、データは次の式でアップデートされる。

$$V(t+1,\vec{p}) = f(V(t,\vec{p}+\vec{d_1}),...,V(t,\vec{p}+\vec{d_n}))$$

$V(t+1,\vec{p})$ は t ステップ ($t \in \boldsymbol{N}$) 時の $\vec{p}$ ($\vec{p} \in \boldsymbol{Z}^d$) の値を示す。関数 $f()$ は周囲の n 点の値を受け取る関数である。$\{ \vec{d_1},...,\vec{d_n} \in \boldsymbol{Z}^d, n \in \boldsymbol{N} \}$ は対象の要素と周辺の要素との相対座標であり、静的な値である。

2 次元空間上の 5 点拡散方程式の単純なプログラムコードは下記のソースコード 1 のようになる。

ソースコード 1: 二次元格子空間上の五点拡散方程式

```
for(t=0; t<T_MAX; t++ ){
    for(i=0; i<X; i++){
        for(j=0; j<Y; j++){
            V[i,j] = beta*V[i,j] + alpha*(V[i−1,j]+V[i
                +1,j]+V[i,j−1]+V[i,j+1]);
        }
    }
}
```

このようなコードでは、一般的にメモリのバンド幅が実行時のボトルネックとなる。一ステップごとに n 点の周辺格子の要素が必要な場合、性能上限 L は次の式で表される。なお、T_c は一格子のアップデートに必要な計算コスト、BF は実行環境のバンド幅/Flops 値 (B/F 値) である。

$$L = \frac{T_c}{(n+1)*(\text{word-size})} * BF$$

この問題への対処として、一般的に空間タイリングが用いられる。格子空間をタイリングしデータサイズを小さくすることで、キャッシュの効果を高めるものである。

図 4 は空間タイリングの概念図である。例として $n*n*n$ の格子空間を $3*3*3$ 個のタイルに分割している。分割された各タイルは袖領域のサイズ h を含め、$(n/3+2h)*(n/3+2h)*(n/3+2h)$ となる。分割されたタイルのデータサイズがキャッシュに収まるサイズまで小さくすることで下記の式で示される L_{new} まで性能上限を上げることが可能となる。

$$L_{new} = \frac{T_c}{2(\text{word-size})} * BF$$

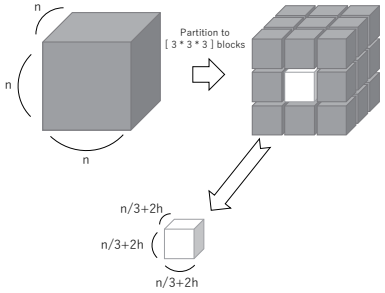

図 4: Spatial Tiling

4. テンポラルブロッキング

空間タイリングよりもさらに B/F 値の影響を軽減するための最適化手法として、テンポラルブロッキング [8,9] が存在する。テンポラルブロッキングでは空間だけではなく、時系列においてもブロッキングを行う。一回の通信ごとに、複数のステップ数データのアップデートを行うことで、データ転送時間を減少させる。

図 5 は、黒い点の要素を計算するために必要なデータの依存関係である。図の灰色の要素が依存関係を持つこととなる。このことは、T ステップ時の灰色のグリッドのデータを持つことで、他のタイルと通信することなく、$T+k$ ステップ時の黒点のデータを得ることをできることを示す。

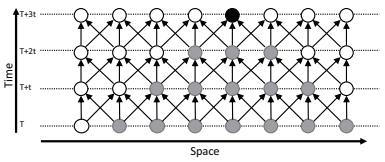

図 5: Data Dependency of Stencil Computation

図 6 は 1 次元空間上のテンポラルブロッキングを行う場合の例である。$t+s$ ステップ時の担当領域のデータを計算するために、図の台形部分の値のアップデートを行う。このとき、図の黒色の部分は、他のタイルでも同様に計算

を行う必要があるため、計算処理としては重複することとなる。しかし、ネットワークなどを対象とした場合の B/F 値は非常に小さいため、重複計算にかかる処理時間を考慮したとしても高速化を望むことが可能である。

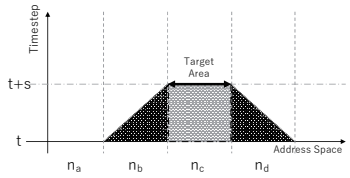

図 6: Temporal Blocking (Duplicated Area)

5. 開発基盤設計

本開発基盤は次のことを目標として作成する。対象のアプリケーションは一次元格子および二次元格子状の拡散方程式とする。

- 各計算機システムの階層向けの最適化モジュールは任意に組み合わせることが可能
- エンドユーザーが作成するシリアルなプログラムコードを計算機システム向けに適応・最適化可能
- エンドユーザーが、当該開発基盤上で作成したプログラムコードを他の計算機システム上に移植する際に、コードを一切変更しなくてよい

図 7 は、当該開発基盤の構造である。

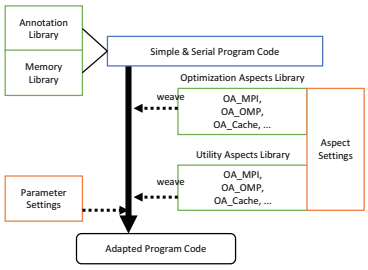

図 7: 開発基盤の構造

開発基盤が提供するものが図の緑色の部分、開発基盤の利用者で計算機システムの提供者が作成するものが図の橙色の部分、エンドユーザーが作成するのは図の青色の部分である。

開発基盤は下記のものを提供する

- アノテーションライブラリ (5.1 章)
- メモリライブラリ (5.2 章)
- 適応・最適化アスペクトライブラリ (5.3 章)
- ユーティリティアスペクトライブラリ (5.4 章)

開発基盤の利用者兼計算機システムの管理者が作成するのは次の二つである。

- 適応・最適化アスペクトの優先順位設定ファイル (5.5 章)
- 各適応・最適化アスペクトおよびメモリライブラリ用設定ファイル (5.6 章)

最後に、アプリケーションの開発を行うエンドユーザが作成するのは次である。

- アプリケーションロジックコード (5.7 章)

それぞれの詳細を説明する。

5.1 アノテーションライブラリ

アノテーションライブラリは、ユーザーがプログラムを作成する際に利用する仮想クラスを提供する。ユーザーがアプリケーションを作成する際には、仮想クラスを継承したクラスを作成し、そのクラスの提供する複数の関数の内部処理を実装することとなる。開発基盤の実行時には、アノテーションライブラリのクラスの型に対して適応・最適化アスペクトライブラリで定義されているポイントカットがマッチングを行いジョイントポイントを作成する。

5.2 メモリライブラリ

メモリライブラリは対象プログラムのグローバルデータを保存するためのストレージを提供する。

メモリライブラリ内のデータは、各関数内で宣言される変数とは異なり、常にアクセス可能なことが保障される。

メモリライブラリは複数のアドレッシングのインターフェースを提供する。主な二つは論理大域アドレス (LGA) 及び論理局所アドレス (LNA) である。LGA インターフェースを用いた場合、コードは明示的な支持なしに、他のノードの保有するデータにアクセス可能である。一方、LNA インターフェースを用いた場合は他のノードの保有するデータにはアクセスできないが、オーバーヘッドなくデータにアクセスすることが可能である。

開発基盤の実行時には、メモリライブラリもアノテーションライブラリと同様に、適応・最適化アスペクトライブラリによって処理の変更が行われる。メモリライブラリのインスタンスは、適応・最適化アスペクトライブラリによって作成される最も粒度の細かいコンテクスト毎に作成される。

また、メモリライブラリ内部はデータキャッシュを行うための機構を持ち、利用される適応・最適化アスペクトによっては、プリフェッチを行い、メモリライブラリ内にキャッシュする。

5.3 適応・最適化アスペクトライブラリ

適応・最適化アスペクトライブラリは、適応・最適化アスペクトをライブラリとして提供する。各適応・最適化アスペクトはシステム階層と対象アプリケーションの組み合わせごとに存在する。適応・最適化アスペクトは対象とする階層の制御とブロッキングを行うコードをアノテーショ

ンライブラリに従って、メモリ管理のコードをメモリライブラリに従って追加する。アプリケーションに対して最適化アルゴリズムを追加する場合は、適応・最適化アスペクト内に実装する。

適応・最適化アスペクトは下記の三種を作成した。下記の三種はすべて内部にステンシル計算用の最適化アルゴリズムとして時空間台形の形に分割するテンポラルブロッキングを実装している。そのため、複数のアスペクトを適用した場合、再帰的に時空間台形が生成される。

5.3.1 MPI Optimization Aspect

MPI Optimization Aspect(OA_MPI) は分散メモリ並列向けの最適化を行うアスペクトである。ソフトウェアスタックとしては MPI を利用する。OA_MPI はデータ領域の分割のほかに、MPI の初期化・終了処理、および非同期通信を用いたデータ通信を行う。さらに、拡散方程式向けの OA_MPI の機能として、テンポラルブロッキングを実装している。当拡張では、データ通信を最適化するため、グリッド値の更新式の他のグリッドへの依存関係の情報から、事前に袖領域のデータを得るために通信するノードを決定し、通信を一括して行ったうえでキャッシュする。

5.3.2 OpenMP Optimization Aspect

OpenMP Optimization Aspect(OA_OMP) は共有メモリ並列向けの最適化を行うアスペクトである。ソフトウェアスタックとしては OpenMP を利用する。OA_OMP はデータ領域の分割のほかに、スレッドの立ち上げ、共有メモリ領域の確保、および共有メモリを通した擬似的なスレッド間通信を行う。さらに、拡散方程式向けの機能として、OA_MPI と同様にテンポラルブロッキングを実装している。

5.3.3 Cache Optimization Aspect

Cache Optimization Aspect(OA_Cache) はキャッシュブロッキングによる最適化を行うアスペクトである。さらに、拡散方程式向けの拡張として、OA_MPI および、OA_OMP と同様にテンポラルブロッキングを実装している。しかし、OA_Cache では、対象システムのキャッシュからデータが溢れないよう、分割された各時空間台形はシリアルに実行され、データが集約される。

5.4 ユーティリティアスペクトライブラリ

ユーティリティとして、実行時間計測を行うライブラリおよびデバッグ用ライブラリをアスペクトとして提供している。これらのライブラリでは、各階層ごとの出力の取捨選択や、適切な計測関数の選択などを行う。

5.5 適応・最適化アスペクトの優先順位設定ファイル

当ファイルは、対象計算機システム向けに利用する適応・最適化アスペクトの選択と適応順序の設定である。開発基盤の利用者兼システムの管理者は、対象システムの階層構造に応じて、当ファイルを設定する。

表 1: 実行環境

名前	Oakforest-PACS	Reedbush
Type Name	regular-cache[*1]	reedbush-u
最大ノード数	8192	420
CPU	Xeon Phi 7250	Xeon E5-2695v4 * 2
Cores	68	18 * 2
Frequency	1.4GHz	2.1GHz

5.6 各適応・最適化アスペクトおよびメモリライブラリ用設定ファイル

当ファイルは、対象計算機システム向けに利用する各適応・最適化アスペクトやメモリライブラリが利用するパラメータの設定である。例を挙げると、OA_MPI ではテンポラルブロッキングの深さや、キャッシュブロッキングのサイズを当ファイルで設定することが可能である。開発基盤の利用者兼システムの管理者は、対象システムの階層構造に応じて、当ファイルを設定する。

5.7 アプリケーションロジックコード

このファイルは、アプリケーション開発を行うエンドユーザーが作成するファイルで、アプリケーションのロジックをシリアルなプログラムとして作成し、記載する。プログラムの作成には、開発基盤の提供するアノテーションライブラリとメモリライブラリを利用する。

6. 性能評価

当性能評価の事前実験として、利用する機能の範囲では AspectC++プラットフォーム自体にはオーバーヘッドがないことを確認した。

6.1 計算機システム

本性能評価では、評価 1 を Oakforest-PACS, 評価 2 を Reedbush 上で行った。Oakforest-PACS と Reedbush は双方とも東京大学情報基盤センターの計算機システムである。Oakforest-PACS は Intel XeonPhi のクラスタマシン、Reedbush は Intel Xeon+Nvidia Tesla のクラスタマシンである。なお、本評価では、Reedbush を Xeon のみのクラスタマシンとして利用した。表 1 に実行環境の詳細を示す。

なお、AspectC++コンパイラは ac++2.2 及び ag++0.9 を利用した。(ag++は ac++で gnu 拡張を利用可能とするためのラッパーである。)

[*1] Oakforest-PACS の regular-cache ノードでは MCDRAM はキャッシュとして利用される。また、コアは Quadrant mode で動作する。

6.2 評価 1

まず、簡単なステンシル計算のプログラムを作成し、手書きコードとプラットフォームが出力するコードのパフォーマンスを Oakforest-PACS 上で計測した。なお、双方ともに、ブロッキング以外の最適化は行っていない。

- Program: 二次元格子上五点拡散方程式
- Depth of Temporal Blocking: 10
- The numbers of Total Steps: 1000
- Global Size of Target Grid per Node: 512 * 512 - 4096 * 4096 (Double Precision Floating Point)
- The number of MPI processes : 64, 256, 576, 1024, 1600, 2304, 3136, 4096
- Optimization Aspect: MPI

グラフ 8 は当評価の結果を示す。グラフの***H の表記は手書きコードであること、***P はプラットフォームの出力コードであることを示す。それぞれの***にあたる部分の数値は、一ノード当たりのグリッドの割り当て数を示す。このグラフは弱スケーリングの結果を示す。このグラフの結果から、十分にデータ領域が大きければパフォーマンスの大きな劣化はないことがわかる。なお、一ノード当たりのグリッドの割り当て数が少ない際のプラットフォームの出力コードのパフォーマンスの劣化は、主にプラットフォームの初期化によるものと考えられる。また、一ノード当たりのグリッドの割り当て数が大きくなっても残存するパフォーマンスの劣化はメモリアドレスへの Aspect の挿入によって生じていると思われる。メモリアクセスがどのメモリ領域（ノードの保持するメモリ領域、キャッシュする他のノードのメモリ領域など）にアクセスを行うか判定することでメモリアクセスごとにオーバーヘッドが生じてしまっている。

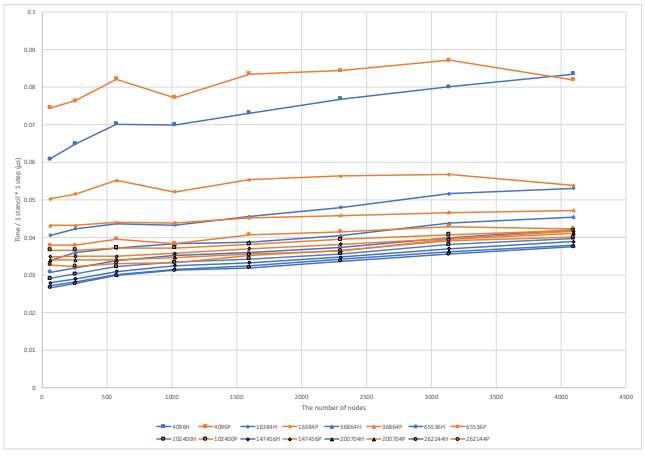

図 8: The Performance Comparison between Hand-optimized and Generated Code for MPI on Oakforest-PACS

6.3 評価 2

次に、アスペクトの組み合わせに応じた最適化性能を Reedbush 上で計測した。

- Program: 二次元格子上五点拡散方程式
- Depth of Temporal Blocking: 10
- The numbers of Total Steps: 1000
- Global Size of Target Grid: 1024 * 1024 (Double Precision Floating Point)
- The number of MPI processes and OpenMP threads per Node: (MPI: 2, OpenMP: 16), (MPI: 32)
- Block Size for cache layer: 30 * 30
- Optimization Aspect: MPI, MPI+OpenMP, MPI+Cache, MPI+OpenMP+Cache

グラフ 9 は当評価の結果を示す。グラフはプログラムの合計実行時間を示す。この結果より、MPI 及び MPI+Cache の組み合わせに関しては、各ノード当たりのデータサイズが十分に大きければある程度のパフォーマンスの向上が見られるという想定の結果となった。これは Cache ブロッキングによる性能向上とテンポラルブロッキングの重複計算の増加による性能劣化の複合によりこのような結果となったと推察される。しかし OpenMP 用のモジュールを利用した結果は上記の二つと比較し大きくパフォーマンスの劣化が見られた。そのため、OpenMP 用のモジュールの設計もしくは実装に問題があると思われる。

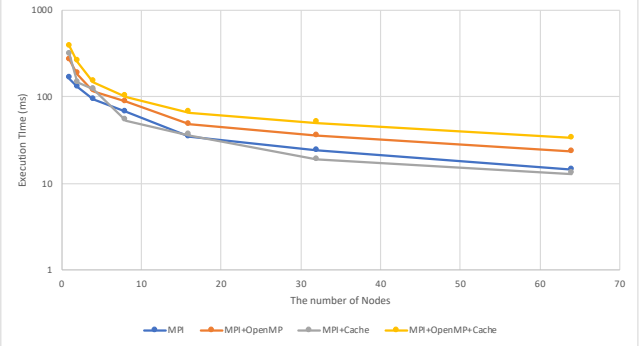

図 9: The Performance Comparison of the Combination of Optimization Aspects on Reedbush-U

7. 関連研究

DSL の処理基盤とハードウェア向けの適応と最適化の間に階層を設け、汎化した開発基盤としては Bernstein Gilbert Louis らの Ebb があげられる [10]。Ebb は流体シミュレーション向けの DSL プラットフォームで、Terra と呼ばれる、メタプログラミングの一種として、Lua コードに埋め込まれるように設計された低レベルのプログラミング言語 [11,12] を用いて CPU のみか CPU+GPU の計算機システム向けに実装されている。Ebb は三階層で構築される。第一階層ではユーザーがユーザーがアプリケーション

ロジックを記載する。第二階層ではメッシュ構造を定義し、第三階層では第二階層で定義されたメッシュ構造をハードウェアに適応する。このことにより、メッシュ構造によらず、アプリケーションを開発することを可能としている。

AOP を HPC アプリケーション開発に利用した先行研究としては、MPI の通信を Aspect として分離した John S. Dean らの研究 [13] やループの並列化を Aspect を用いて行った B. Harbulot ら [14] や João L. Sobral ら Sobral2007 の研究が存在する。

8. 終わりに

本研究では、AOP を用いて、ハードウェア階層に対応したモジュールから作成可能な、アプリケーション適応および最適化プラットフォームを提案した。また、ステンシル計算向けにテンポラルブロッキングによる最適化と計算機システムへの適応を行う開発基盤を開発した。当プラットフォームの評価は Oakforest-PACS 及び Reedbush-U 上で行い、パフォーマンスの劣化が非常に少ないことも確認した。

現時点のプラットフォームの問題点および変更すべき点および解決策で、現在取り組んでいるものは次である。

- CUDA や AVX など、SIMD/SIMT モデルを用いるシステム階層への適応。この問題に関しては、各カーネルコードの最小単位をグリッドから、グリッドの集合体へ変更することで、エンドユーザーのプログラム開発の難易度とのトレードオフとはなるが、ある程度解決可能であると考える。また、この変更は、メモリアクセスのオーバーヘッドを削減するという点においても有用である。
- テンポラルブロッキングを繰り返し適応することで、計算の重複が大きくなり、通信量の削減での利点に見合わなくなる点への対応。この問題は、台形以外の時空間分割のテンポラルブロッキングに対応し、状況に応じて選択可能とすることで解決を目指す。
- Adaptive Mesh や粒子法の負荷分散など、ステンシル計算以外のデータ領域の割り付け問題へのプラットフォームの汎化
- OpenMP 用モジュールの修正

また、開発基盤を今後実用的なものとしてゆくために解決すべき問題と検討している解決方法は次のとおりである。

- 各モジュールが利用するパラメータ用の自動チューニング機能の追加
- 各モジュールからの対象アルゴリズムの分離。現在のプラットフォームの構成では各階層のモジュールごとに対象アルゴリズムへの拡張を持つ必要がある。そのため、（階層の種類）×（対象アルゴリズムの種類）のアスペクトの実装を持つ必要がある。AspectC++上では Aspect on Aspect を行うことはできないため、アルゴリズム、もしくは階層を横断的関心事として分離することはできない。これらを可能とするため、マクロや関数適用を利用することを検討している。
- Layer-by-Layer でないシステム階層への適用。現在の開発基盤では、Layer-by-Layer のシステム階層にしか対応することができない。これは、Advice 内で複数回の元処理の呼び出しができないことに由来する。コンテクスト指向プログラミングの手法などを応用することによってこの問題を解決することを目指す。

謝辞　本研究では、東京大学情報基盤センターの Reedbush 及び Oakforest-PACS を利用した。Reedbush の利用は東京大学情報理工学系研究科の”計算科学アライアンス”による。

研究活動全般においてアドバイスをいいただい平木敬教授、本研究に関し多くのアドバイスをいただいた松本正晴先生、吉本研究室の皆様へ心から感謝の気持ちと御礼を申し上げ、謝辞にかえさせていただきます。

参考文献

[1] Muranushi, T., Nishizawa, S., Tomita, H., Nitadori, K., Iwasawa, M., Maruyama, Y., Yashiro, H., Nakamura, Y., Hotta, H., Makino, J., Hosono, N. and Inoue, H.: Automatic Generation of Efficient Codes from Mathematical Descriptions of Stencil Computation, *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, Vol. 1, No. 212, pp. 17–22 (online), DOI: 10.1145/2975991.2975994 (2016).

[2] Muranushi, T., Hotta, H., Makino, J., Nishizawa, S., Tomita, H., Nitadori, K., Iwasawa, M., Hosono, N., Maruyama, Y., Inoue, H., Yashiro, H. and Nakamura, Y.: Simulations of Below-Ground Dynamics of Fungi: 1.184 Pflops Attained by Automated Generation and Autotuning of Temporal Blocking Codes, *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, pp. 23–33 (online), DOI: 10.1109/SC.2016.2 (2016).

[3] Naoya, M., Tatuo, N., Kento, S. and Satoshi, M.: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12 (online), DOI: 10.1145/2063384.2063398 (2011).

[4] Chiba, S.: アスペクト指向入門 -Java ・ オブジェクト指向から AspectJ プログラミングへ，技術評論社 (2005).

[5] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming, *The Spring Framework Reference Documentation*, Springer, Berlin, Heidelberg, pp. 220–242 (online), DOI: 10.1007/BFb0053381 (1997).

[6] : The Home of AspectC++, `https://www.aspectc.org/`. (Accessed on 01/25/2018).

[7] Spinczyk, O., Gal, A. and Schröder-Preikschat, W.: {AspectC++}: an aspect-oriented extension to the {C++} programming language, *Proceedings of the Fortieth International Conference on Tools Pacific*, Australian Computer Society, pp. 53–60 (online), available from ⟨https://dl.acm.org/citation.cfm?id=564100

http://www.mendeley.com/catalog/aspectc-aspectoriented-extension-c-programming-language/⟩ (2002).

[8] Wellein, G., Wittmann, M. and Hager, G.: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory, *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Vol. 00, pp. 1–7 (online), DOI: 10.1109/IPDPSW.2010.5470813 (2010).

[9] 知輝”河村，直也丸山，聡”松岡：”並列ステンシル計算における通信の自動最適化に向けた性能モデルの評価”，技術報告”32”，”東京工業大学, 東京工業大学／理化学研究所／科学技術振興機構 CREST, 東京工業大学／科学技術振興機構 CREST ／国立情報学研究所” (”2012”).

[10] Bernstein, G. L., Shah, C., Lemire, C., DeVito, Z., Fisher, M., Levis, P. and Hanrahan, P.: Ebb: A DSL for Physical Simulation on CPUs and GPUs, *ACM Transactions on Graphics*, Vol. 35, No. 2, pp. 1–12 (online), DOI: 10.1145/2892632 (2015).

[11] : Terra, `http://terralang.org/`. (Accessed on 01/27/2018).

[12] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P. and Vitek, J.: Terra, *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, Vol. 48, No. 6, New York, New York, USA, ACM Press, p. 105 (online), DOI: 10.1145/2491956.2462166 (2013).

[13] Strazdins, P. and Strazdins, P.: A High Performance, Portable Distributed BLAS Implementation, *IN SIXTH PARALLEL COMPUTING WORKSHOP*, p. pages (online), available from ⟨http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3576⟩ (1996).

[14] Harbulot, B. and Gurd, J.: Separating concerns in scientific software using aspect-oriented programming, PhD Thesis (2006).