Vol.2019-HPC-170 No.42
2019/7/26

アスペクト指向言語を用いたHPC向けDSL作成プラットフォームの構築

石村 脩[a)]　吉本 芳英[b)]

概要：ドメイン特化言語（DSL）は、科学計算用プログラムのコードのポータビリティの実現や、大きな労力なしに悪くないパフォーマンスを達成するために有望なアプローチの一つである。しかし、DSL プラットフォーム自身にポータビリティがあることはあまりなく、パフォーマンスの向上にも大きな労力がかかる。この問題を解決するため、科学計算用カーネルコードを、HPC システムの階層構造に合わせて再帰的に最適化し、適応する DSL を構築するプラットフォームを開発した。私たちの開発したプラットフォームでは、最適化と適応をするシステムはそれぞれアスペクト指向プログラミングにおけるアスペクトとして、HPC システムの各レイヤー (MPI を用いたノードレベル、OpenMP を用いたスレッドレベル、OpenACC を用いたアクセラレータレベル等) に対応する形でモジュール化されている。これにより、システムデザイナーはターゲットとなる HPC システム向けの DSL プラットフォームを、DSL 開発者の提供するモジュールの組み合わせから容易に作成することが可能となる。本発表では、ステンシル計算向けの DSL をデモとして構築し、性能評価を行った。

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, ハイパフォーマンスコンピューティング

Aspect-oriented programming based DSL constructing platform for HPC

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

Abstract: Domain Specific Language is a promising approach to achieve programmability and portability of scientific program codes and obtain reasonable performance without pain. However, DLS platforms themselves often lacks in portability. To solve this issue, we developed a platform to construct DSLs that optimize and adapt the scientific computation kernel codes recursively concerning the hierarchal structure of HPC systems. On our platform, the optimization and adaptation system is modularized as aspects, which is a type of object in Aspect-oriented Programming, corresponding to each layer of the HPC system (like node-level parallelism by MPI, thread-level parallelism by OpenMP, accelerator-level parallelism by OpenACC, and so on). Therefore, system designers can easily construct the DSL platforms for their HPC system by combining the modules provided by DSL developers using our platform. In this paper, we demonstrate and evaluate our platform by constructing a simple DSL for stencil computation.

Keywords: domain specific language, aspect-oriented programming, high performance computing

1. はじめに

ドメイン特化言語 (DSL) は HPC システムの構造が複雑化してゆく中で、有望なアプローチの一つである。DSL の分野では村主らの formura [1,2] や村山らの Physis [3] など

†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

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。そこで、一つのドメイン特化言語のアプリケーション基盤を複数の計算機システムむけに展開することが可能となれば、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。

そこで以前の研究で、ステンシル計算向けに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アプリケーションの適応と最適化を行う開発基盤を提案した [4]。当開発基盤では、開発基盤の設計者・開発基盤の DSL 用モジュールの開発者・モジュールの選択とパラメータの設定を行う計算機システムの管理者・エンドユーザーという形で分離することを目指している。当研究では、ステンシル計算の領域分割を繰り返し行っても、各格子点を更新するロジックに変化がない点と計算機システムの階層構造に着目し、各階層構造 (MPI, OpenMP, Cache) ごとに、領域分割・袖通信・システムの初期化/終了処理を行う機能を持つアスペクトを用いたモジュールを作成し、これらを任意の組み合わせで利用できるようにした。さらに、テンポラルブロッキングなどの最適化アルゴリズムもまた同様に、適用しても各格子点を更新するロジックに変化がない点にも着目し、テンポラルブロッキングによる最適化機能も、モジュールに実装した。

本研究では、以前の研究では割り付けの単位が一格子当たりであったことによる、「SIMD 化や GPU へのオフロードが難しい」、「各レイヤーのデータ転送を行うアスペクトが複雑になりコード量が増加する」、「一回のメモリアクセスに対して 5 %程度のオーバーヘッドが存在する」という課題を、ブロック単位での領域管理を実装することで解決することを試みた。

なお、本開発基盤の実装は、以前と同じく、AOP の処理系の一つである AspectC++を用いて MPI 向けのモジュールを実装している。また、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、Reedbush 上で性能評価を行い、以前の研究の結果と比較した。

本稿の構成は下記のとおりである。まず、第二節でアスペクト指向プログラミングについて説明する。次に、第三節で本開発基盤の設計の詳細、第四節で性能評価について述べる。そして、第五節で関連研究の紹介を行い、最後に第六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、OOP では依存関係の形式によって取り除くことのできない横断的関心事を分離するプログラミングパラダイムである [5, 6]。

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。

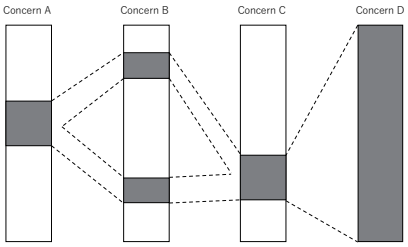

図 1: 横断的関心事の分散
Fig. 1 Distribution of crosscutting concern

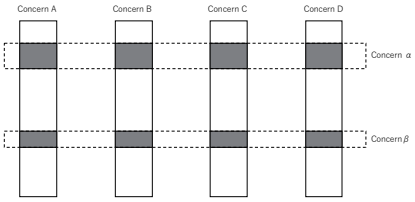

図 2: 横断的関心事の分離
Fig. 2 Partition of crosscutting concerns

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）と呼び、ポイントカット (pointcut) と呼ばれるパタンマッチによって特定される。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特定がされたジョイントポイントに対して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。

このような AOP の特性は、「オブジェクトとして分離される処理が、ロジックのどの部分で実行されるかを、その分離されたオブジェクト自身から管理する場合」や、「ロジックのパターンが同様の場合、常に同じ処理を挿入したい場合」などに有用である。

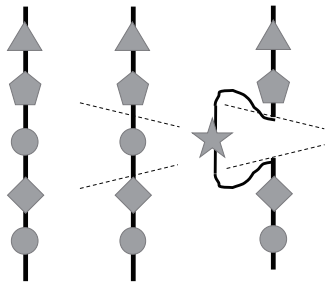

図 3: Aspect の織り込み手順
Fig. 3 Aspect weaving process

3. 開発基盤設計

2 で述べた通り、AOP の特性は、「オブジェクトとして分離される処理が、ロジックのどの部分で実行されるかを、その分離されたオブジェクト自身から管理する場合」や、「ロジックのパターンが同様の場合、常に同じ処理を挿入したい場合」などに有用である。HPC アプリケーションにおいては、前者がライブラリの初期化や、後者が通信にあたる。

図 4 は、当該開発基盤の構造である。当構造は以前の研究 [4] と変更はない。

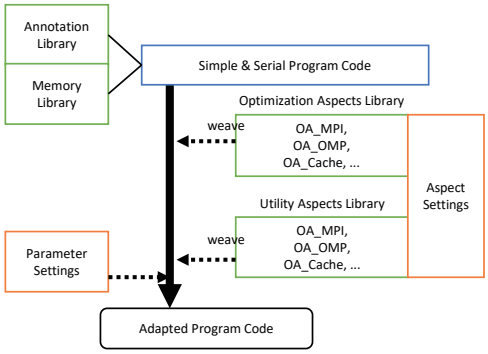

図 4: 開発基盤の構造
Fig. 4 Structure of the platform

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。

- アノテーションライブラリ (3.1 節)
- メモリライブラリ (3.2 節)
- アスペクトライブラリ (3.3 節)
- ユーティリティアスペクトライブラリ
- アプリケーションロジックコード (3.4 節)

3.1 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し、ジョイントポイントを作成する。ソースコード 2 では、アノテーションライブラリの `SUGOI_Target` クラスを継承し `My_Target` クラスとしてプログラムを作成している。`SUGOI_Target` クラスは仮想関数として `Initialize`、`Main`、`Finalize` を持つ。これらに対して、後ほど説明するアスペクトライブラリで織り込みが行われる。

3.2 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。この実装が、以前の研究での実装と大きく変わった部分である。以前の研究では、すべてのデータに対して、そのノードやスレッドが保持しているデータかどうかをチェックしていたため、メモリアクセスに 5%程度のオーバーヘッドが存在した [4]。また、カーネル関数を格子点ごとに呼び出す実装となっていたため、SIMD 化や GPU へのオフロードの実装が難しかった、そこで本研究では、上記の問題を解決するためデータ領域をブロック単位に分割して管理することとした。これにより、メモリアクセスの際にデータの所持確認を行うのは、ブロックのエッジ部分へのアクセスの際のみでよくなるため、メモリアクセスのオーバーヘッドを減らすことができる。さらに、カーネル関数をブロック単位で呼び出す実装とすることができるため、カーネル関数内で、ユーザーが SIMD 化や GPU へのオフロードを実装することができるようになる利点もある。

アノテーションライブラリと同様に、メモリライブラリ内の各関数に対して、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し、ジョイントポイントを作成する。メモリライブラリは内部でダブルバッファリングが採用されており `refresh` 関数を呼び出すことによってバッファの入れ替えが行われる。そのため、メモリライブラリへの代入操作は `refresh` 関数の呼び出し時にまとめて更新される。Copy Mode と Swap Mode の二つのモードが用意されており、Kernel でブロックのすべてのデータを更新している場合のみより高速な Swap Mode を利用することが可能である。

全データ領域をブロック単位に分割し、図 5 の示す木構造を用いて管理する。これは、負荷分散や、動的なメッシュ分割に対応するためである。内部構造の主要な構成要素は Server, iNode, Leaf の三種である。Server は次に述べる木構造を生成/管理する機能や、特定のデータを取得する場合に、そのデータのタグから、そのデータを含む Leaf ノードのポインタを検索する機能をもつ。Leaf は単一ブロックを示し、実際にデータを保持する。iNode は Leaf の集合

体を木構造として管理するための節ノードである。なお、Server でデータ領域を生成すると、木のルートの iNode のポインタが返される。(generate 関数。ソースコード 2 の行 9) Server は木構造のルートへの参照を持ち、木の iNode および Leaf はすべて Server への参照を持つ。Server は木の iNode 及び Leaf への参照のキャッシュを持つことで、ノード間の検索の高速化を行っている。

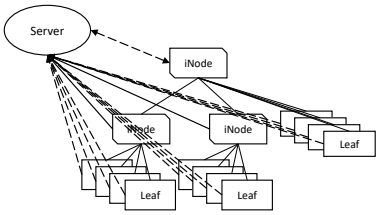

図 5: メモリライブラリ内の実装
Fig. 5 Structure of the memory library

各プログラムでは、必要とするデータ領域を生成する Server から、全データ領域のデータを表すメモリ領域を生成する。プログラム内では、「全データへのデータアクセス法」、もしくは「全データ領域から、内部の単一ブロックを取り出し、その単一ブロックへのデータ操作を行うアクセス法」の二つが提供される。パフォーマンスが必要なコードでは、ソースコード 2 の行 13-15 及び行 23-29 の様にパフォーマンスの良い後者のアクセス法を利用する。

ステンシル計算向け実装では、多次元空間を複数の固定サイズの多次元ブロックの集合体として管理する。ユーザーは単一ブロックに対しては、多次元配列と同様のメモリアクセスが可能である。ソースコード 2 の Kernel 関数内でも利用されているが、getLD 関数を用いた場合、もしくはオーバーロードされている配列アクセス演算子を利用した場合は、対象の単一ブロックの読み込み/書き込みバッファへの直接アクセスとなる。単一ブロックの外部のメモリブロックへのアクセスは、単一ブロックのインスタンスが持つ相対アドレスでのアクセスを行う関数 (getL 関数) を用いるか、全体ブロックの持つ絶対アドレスでアクセスを行う関数 (getG 関数) を用いることで可能である。

3.3 アスペクトライブラリ

アスペクトライブラリは、アスペクトをライブラリとして提供する。各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。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、アスペクト内に実装する。MPI 向けアスペクトの実装では、カーネル関数のメモリアクセスが常に一定である場合、カーネル関数が必要とするメモリブロックをカーネル関数の呼び出し前にプリフェッチする機能がある。これは、Dryrun 機能によって実装される。Dryrun では、カーネル関数を結果を破棄しつつ実行し、その際にアクセスしたメモリブロックを記録し、それに応じてノード間通信を行うことよって実現する。

DSL を実装する際のサンプルコードをソースコード 1 に示す。なお、このコードは説明の簡略化及びコード量の問題から、実際のコードとは省略と変更を加えた。このコードでは、SUGOI_Aspect アスペクトで、次の 4 つのポイントカットを定義している。

- sugoi_main(int argc, char **argv)
- pointcut sugoi_env_inode_refresh() = ... // 略
- pointcut sugoi_SUGOI_Grid_2D_createLeaf(/* 略*/) = ... //略

1、2 は汎用のポイントカット、3 はステンシル計算専用のポイントカットである。これらが一つのアスペクトに実装されている点については 6 で説明する。そして、これらのポイントカットに対して、SUGOI_Aspect_MPI でアドバイスを実装している。

なお、サンプルコード内の tjp->proceed 関数は、ジョイントポイントに存在したコードの実体を実行するために、AspectC++言語で定義されている関数である。

1 のポイントカットは、メイン関数にマッチするポイントカットである。MPI 向けアスペクトでは、メイン関数が実行時の前処理と後処理として、MPI の初期化と終了処理を行っている。2 のポイントカットでは、データ領域のダブルバッファリングの更新の呼び出しにマッチするポイントカットである。MPI 向けアスペクトでは、まずそれが初回の refresh 関数の呼び出しであった場合は、データ領域のバッファの結果を Dryrun であったとして破棄するとともに、その時のアクセスパターンから必要なノード間通信のリストを作成する（ソースコード 1、行 24 の dryrun 関数)。そして、以降はプリフェッチ機能が、リストに従いノード間通信を行う（ソースコード 1、行 26 の data_transfer 関数)。

3 のポイントカットは 2 次元ステンシル計算用に拡張のされたメモリライブラリの Leaf が作成されるかどうかのチェックを行う関数にマッチするポイントカットである。MPI 向けアスペクトでは、自身のノードが保持する Leaf であるかどうかをノード id、全体のノード数の数から判断し、チェックを行う関数の結果を上書きする（ソースコード 1、行 32 の balancing 関数)。

```
1 aspect SUGOI_Aspect
2 {
3     //変数宣言及び内部関数は省略
4 public:
```

```
    pointcut sugoi_main(int argc, char **argv) = ... //略
    pointcut sugoi_env_inode_refresh() = ... //略
    pointcut sugoi_SUGOI_Grid_2D_createLeaf( /*略*/ ) = ... //略
};
aspect SUGOI_Aspect_MPI : public SUGOI_Aspect {
    //変数宣言及び内部関数は省略
    advice sugoi_main(argc, argv) : around(int argc, char **argv){
        layer_level = Aspect::current_layer_num;
        Aspect::current_layer_num++;
        int mpi_mode;
        MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &mpi_mode);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
        tjp->proceed();
        MPI_Finalize();
    }
    //一部略
    advice sugoi_env_inode_refresh() : around() {
        if (!flag_dryrun) tjp->proceed();
        else dryrun();
        MPI_Barrier(MPI_COMM_WORLD);
        data_transfer();
        MPI_Waitall(req_send_count, req_send, stat_send);
        MPI_Waitall(req_recv_count, req_recv, stat_recv);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    advice sugoi_SUGOI_Grid_2D_createLeaf( uuid, i, i_max, j, j_max ) : around(/*略*/) {
        auto ni = balancing(uuid, i, i_max, j, j_max);
        tjp->proceed();
        if (ni != mypid) *(tjp->result()) = false;
    }
};
```

ソースコード 1: プラットフォームにおける DSL 実装コード例（ステンシル計算向け）

3.4 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。サンプルコードをソースコード 2 に示す。なお、このコードは説明の簡略化及びコード量の問題から、実際のコードとは省略と変更を加えた。

ソースコード 2、行 23-29(及び省略されたコード) が、拡散方程式に基づいて計算を行い、ブロックのデータを更新するコードとなっている。行 26 では、メモリ領域はすべて単一ブロック内であるため、配列演算子を用いたメモリアクセスを行っている。行 24 および行 28 ではエッジ部分の計算を行っており、ブロック外の値を必要であるため、getL 関数をもちいて、外部のブロックのデータを取得している。

```
class My_Target : SUGOI_Target
{
    using ENVT = SUGOI_Env_inode_Grid<double, std::tuple<int64_t, int64_t>>;
    using BLKT = SUGOI_Env_leaf_Grid<double, std::tuple<int64_t, int64_t>>;
    //変数宣言と内部関数は省略
public:
    void Initialize(int argc, char **argv) override {
        envs = init_envs();
        env = envs->generate(2048, 2048);
    }
    void Main() override {
        for (int c = 0; c < 10000; c++) {
            for (auto el : *env->get_blocks()) {
                auto p = static_cast<BLKT *>(el);
                Kernel(*p);
            }
            env->refresh();
        }
    }
    void Finalize() override {    }
    void Kernel(BLKT &blok) {
        //エッジ部分一部省略
        for (uint64_t j = 1; j < BLKT::SIZE_Y - 1; j++) {
            blok[j][0] = (blok[j][0] + blok[j - 1][0] + blok.getL(-1, j) + blok[j + 1][0] + blok[j][1]) / 5; //エッジ部分
            for (uint64_t i = 1; i < BLKT::SIZE_X - 1; i++) {
                blok[j][i] = (blok[j][i] + blok[j - 1][i] + blok[j][i - 1] + blok[j + 1][i] + blok[j][i + 1]) / 5; //メインのカーネル
            }
            blok[j][BLKT::SIZE_X - 1] = (blok[j][BLKT::SIZE_X - 1] + blok[j - 1][BLKT::SIZE_X - 1] + blok[j][BLKT::SIZE_X - 2] + blok[j + 1][BLKT::SIZE_X - 1] + blok.getL(BLKT::SIZE_X, j)) / 5; //エッジ部分
        }
    }
};
```

ソースコード 2: プラットフォームにおけるアプリケーションコード（ステンシル計算向け）

4. 性能評価

4.1 評価 1

簡単なステンシル計算用の MPI アスペクトを作成し、それを利用して二次元格子上五点拡散方程式のプログラムを作成し、手書きコードとのコード量を比較した。なお、今回の実装では、カーネル関数の SIMD 化や GPU へのオフロードは行っていない。

表 1 に結果を示す。なお、表内の A,B,C はそれぞれプログラムコードを次に分類して集計を行った。

- A: プログラムコード行数
- B: アスペクトコード行数 (ステンシル計算向け)
- C: アスペクトコード行数 (全体)

A は最終的にエンドユーザーが作成するコードに相当、B は DSL の作成者が作成するコードに相当する。

プログラムコード行数は、すべて手書きのコードと比較し半分以下の分量となっている。また、ステンシル計算向けアスペクトコード行数を加えたとしてもすべて手書きのコードと比較し、約半分となっている。なお、このステンシル計算向けアスペクトコードの内容は、初期化時にノード間でのバランシングを行うコードのみである。バランシングは、各ブロックを順にできる限り均一の個数ずつすべてのノードが所持するように割り付けており、通信量の最適化は行っていない。

表 1: コード量比較

Table 1 Code amount comparison

名前	手書きコード	プラットフォームを用いたコード
A	211 行	87 行
B	-	26 行
C	-	400 行

4.2 評価 2

次に、評価 1 で作成したプラットフォームが出力するコードと、以前の研究で作成した作成したプラットフォームが出力するコードの性能を東京大学情報基盤センターの計算機システムである Reedbush 上で計測し、比較した。実行環境の詳細は次の通り

- 最大ノード数: 420
- CPU: Xeon E5-2695v4 * 2
- Cores: 18 * 2
- Frequency: 2.1GHz
- コンパイラ: intel2018.1.163
- MPI ライブラリ: intel_mpi2018.1.163

AspectC++コンパイラは ac++2.2 を利用した。なお、ac++2.2 は Reedbush 上で動作しなかったため、他の環境で織り込みを行い、出力された C++のコードを Reedbush 上でコンパイルしている。

五点拡散方程式プログラムのパラメータは下記である。

- ステップ数: 1000
- 全領域のデータサイズ: 1024 * 1024 (Double Precision Floating Point)
- ブロックのデータサイズ: 64 * 64 (Double Precision Floating Point)
- ノード数: 1,2,4,8,16,32,64,128
- 一ノード当たりの MPI Process 数: 1

図 6 は当評価の結果を示す。

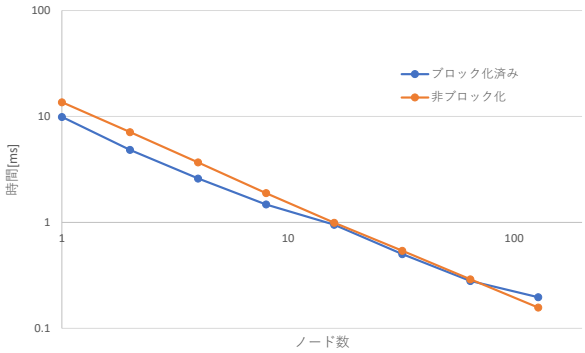

図 6: ブロック化および非ブロック化のパフォーマンス比較

Fig. 6 Performance comparison between blocked and non-blocked

上記の結果から、ノード数が少ない場合は、ブロック化したものが性能が良いものの、ノード数が多くなるにつれ、非ブロック化のものの性能が上回る結果となった。

ノード数が少ないケースにおいては、アドレス計算のオーバーヘッドの減少および、プラットフォーム初期化のロジックの修正によるパフォーマンスの向上が要因と考えられる。ノード数が多いケースにおいてブロック化のパフォーマンスが下回ったのは、ブロック化のバージョンでは、データの転送がブロック単位で行われており、エッジ部分のみを送信する非ブロック化のものと比較し著しく多いことが原因と考えられる。

5. 関連研究

AOP を HPC アプリケーション開発に利用した先行研究は次のものが存在する。

- MPI の通信を Aspect として分離した John S. Dean らの研究 [7]
- ループの並列化を Aspect を用いて行った B. Harbulot ら [8] および João L. Sobral らの研究 [9]

上記のいずれも AspectJ をベースとして用いている。

6. 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のブロック化実装を提案した。ブロック化実装のパフォーマンスが、ノード数が少ないケースにおいては非ブロック化を上回ることを確認した。しかし、ノード数が多いケースにおいては、パフォーマンスが劣化した。この問題を解決するため、

- ブロックのノードへの割り付けの最適化
- ブロックサイズの変更
- 通信のオーバーラップによる隠蔽
- テンポラルブロッキングなど、通信回数を減らす最適化アルゴリズムの実装

などを検討してゆく必要がある。

現時点のプラットフォームの拡張として、今後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の実装
- 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
- 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。

また、現在の AspectC++コンパイラは Template への対応が Experimental であるため、「テンプレートインスタンス内の関数呼び出しにはマッチングしない」、「テンプレート関数およびテンプレートクラスの関数への execution でのマッチングがされない」といった問題がある。これにより、一部 Aspect が決め打ちされたパラメータ以外では動作しないという問題がある。この問題の解決のため、マクロもしくはテンプレートのプリプロセッサによる一部展開をコンパイルのフローに追加する等を検討している。

謝辞 本研究では、東京大学情報基盤センターの Reedbush 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。

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] 石村 脩，吉本芳英：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，技術報告 35，東京大学, 東京大学 (2018).

[5] 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).

[6] 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).

[7] Strazdins, P. and Strazdins, P.: A High Performance, Portable Distributed BLAS Implementation, *IN SIXTH PARALLEL COMPUTING WORKSHOP*, pp. P2-K-1–P2-K-10 (1996).

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

[9] Sobral, J., Cunha, C. and Monteiro, M.: Aspect oriented pluggable support for parallel computing, *High Performance Computing for Computational Science-VECPAR 2006*, Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 93–106 (2007).