Vol.2021-HPC-180 No.15
2021/7/21

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におけるメモリ管理手法の提案と評価

石村 脩[†1,a)]　吉本 芳英[†1,b)]

概要：ドメイン特化言語 (DSL) は HPC アプリケーションのポータビリティやプログラム作成の簡易化を行う有望なアプローチの一つとして幅広く用いられれている。しかし DSL プラットフォーム自身はポータビリティを持たず、他のプラットフォームへの移行や最適化は開発者の大きな労力によって成し遂げられている。この問題を解決する手法の一つとして、我々はアスペクト指向プログラミング (AOP) を用いた DSL 作成プラットフォームを提案している。当プラットフォームでは HPC アプリケーションの主目的以外の HPC システム向けの適用、最適化を AOP に置ける横断的関心事ととらえ、アスペクトとして分離を行う。さらに、HPC システムの各レイヤーとそのランタイムの階層構造を、モジュール化されたアスペクトの集合に対応させ、コードの再利用性を高めている。本研究では、当プラットフォームを実装するにあたり問題となるアスペクトから効率よく制御可能なメモリ構造とその管理手法を考案し、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

Memory Management System on Aspect-oriented programming based DSL constructing platform for HPC

Osamu Ishimura[†1,a)]　Yoshihide Yoshimoto[†1,b)]

Abstract: Domain-Specific Language(DSL)is widely used for achieving the programmability and portability of HPC applications as one of the promising approaches. However, DLS platforms themselves often lack in portability. It demands much effort for maintainers to port and optimizes them to other platforms. To solve this issue, we developed Aspect-oriented programming based DSL constructing platform. On our platform, to increase re-usability, each layer of HPC systems and structure of runtime for them are modularized as aspects. To improve our platform's performance, an easily-controllable memory structure from the aspects is essential. In this study, we proposed a new memory structure and management method and evaluated it.

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

1. はじめに

一般的に、プログラムの最適化は、コード量（開発にかかる労力/工数/複雑性）・データ量・実行時間の観点から行われる。この中でもシステム向けの最適化では、コード量と実行時間のトレードオフとなる。また、コード量の観点では、システムやそれを構成するハードウェアの知識が豊富でなければ実施することもできないという問題がある。なお、外部ライブラリやコンパイラを利用する場合、エンドユーザーの視点においてはこの双方が両立しているが、ライブラリやコンパイラのロジックが複雑になっており、総体としてはこのトレードオフが成り立つ。

そこで重要となるのは、可能な限り最適化を行うコードを再利用可能とすることである。これにより、エンドユーザーの観点ではコード量の増加なく、実行時間の短縮を行うことが可能となる。

[†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

この観点で見たとき、DSL はコンパイラがコード量の増加を担うことでエンドユーザーのプログラム作成を容易とする手法である。しかし、コンパイラの開発には、少なくてもプログラミング言語に関する知識が必要であり、その対象が汎用的なものでなくなるほど開発と維持は困難となる。これは、このコンパイラ（もしくは DSL 基盤）では、コンパイラ自体の機能と対象のプログラムの構造を用いた最適化が一体となっており、双方に対する深い見識が必要となるためである。そのため、村主らの formura [1,2] や村山らの Physis [3] など多くの DSL 開発基盤は特定の計算機システム構成の組み合わせのみ対して作成されており、他のプラットフォームへの移植は盛んではない。一般的にプログラムコードの寿命はハードウェアの寿命よりも長いため、ハードウェア間の移植が容易でないことは問題である。

これらを分離、もしくは部分的に適用可能とした先行研究としては Legion [4] や ebb [5] などが存在する。しかし、これらは最適化機能のモジュール化と再利用という観点では開発されていない。そこで、我々はコンピューターシステム、特に HPC システムに存在すハードウェアの階層構造毎にモジュール化し再利用可能なレイヤーを持つ開発基盤を開発している [6,7]。当プラットフォームでは、アスペクト指向プログラミングを用いて、プログラムの構造を用いた各計算機レイヤー向けの最適化機構をアスペクトとしてモジュール化し、計算機の構成に合わせて組み合わせて利用可能としている。

以前の研究では、プロトタイプとしてステンシル計算向けの DSL を開発した。その中で、データ点ごとの割り付けではプログラムの構造はわかりやすいがオーバーヘッドが大きいという問題があった。そこで、データ点の集合（以下、ブロック）の割り付けに構造を変更することでこの問題を緩和することができた。[7] しかしここで二つの問題に直面した。一つ目は、エンドユーザーの作成するコード、開発基盤のコード、アスペクトのコードそれぞれすべてが、基本となるブロックの単位で作成する必要があり、他の対象のプログラムへの展開が容易でないという問題。二つ目は、すべての実行基盤上で同一のフォーマットを持つブロックが利用できなければならないという制限により、CPU、GPU、SIMD などのメモリレイアウトの異なるハードウェアを混在させて利用することが難しいという問題だ。

そこで当研究では上記の問題を解決するため、メモリ管理レイヤーを論理的なメモリ構造と物理的なメモリ構造に分離し、これらを管理するメモリ管理手法を考案し、簡単なベンチマーク用の 2 次元ステンシル計算アプリケーションを用いて、オーバーヘッドの性能評価を行った。

なお、対象のアプリケーションが単純な正規格子のみである場合、メモリ管理を行わず、直接メモリアクセスを行う方法もあるが、非正規格子や適合格子細分化法 (AMR) [8] への拡張を見据え、メモリ管理の改良を行っている。

本稿の構成は下記のとおりである。まず、2 章でアスペクト指向プログラミング、3 章で本開発基盤の概要について説明する。次に、4 章でデータの割り付け構造、5 章で性能評価について述べる。そして、6 で関連研究の紹介を行い、最後に 7 章で本稿のまとめと今後の展望について述べる。

2. Aspect Oriented Programming

生産性や再利用性を高めるためのプログラミングモデルとして、オブジェクト試行プログラミング (OOP) は頻繁に利用されている。しかし、OOP のモデルではあくまでメインとなる “ プログラムの目的（関心事）” に合わせてモジュール化が行われるデザインパターンとなっているため、モジュール化がうまくいかない関心事（横断的関心事）が存在する。一例を挙げると、プログラムのログの機能などが挙げられる。ログは、ログをなんらかの方法を持って出力するという目的においてはモジュール化を行うことができるが、プログラムのどの時点でログを出力するかという点においてはモジュール化を行うことができない。そのため、ログの出力タイミングを変更する場合には、元のコードを修正する必要がある。この問題を解決するプログラミングモデルが、G. Kiczales らによって提案されたアスペクト指向プログラミング（AOP）である [9,10]。AOP では横断的関心事をアスペクトというオブジェクトとは別のモジュールに分離する。

AOP を実現するモデルでもっとも用いられているモデルはジョイントポイントモデル (JPM) である。Java の AOP の実装系である AspectJ や、本研究で利用している C++ の AOP の実装系である AspectC++も JPM である。

JPM では PointCut と呼ばれるパターンマッチをソースコードに対して行い、パターンが一致した場所をジョイントポイントと呼ぶ。そして、そのジョイントポイントに対してアスペクトで定義された処理を織り込む。

AOP による横断的関心事の分離は、横断的関心事に関わる処理をモジュール化し再利用することを可能とするため、次の 2 つのケースで特に有用である。

- 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
- 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

3. プラットフォーム概要

本プラットフォームは多段の SPMD 型並列プログラミングモデルとなっており、対象の計算機を構成するハードウェア階層毎でアスペクトによるプログラムの並列化が行

われる。基本となるシリアルプログラムに対する入力の操作や、プログラム間のデータ移動、アプリケーション基盤の終了・初期化などをアスペクトによって制御することでエンドユーザーからは並列化や、ハードウェア階層の制御コードを隠すことを可能としている。

当プラットフォームではプラットフォーム自体は C++及び AspectC++で作成されている。そのため、プラットフォームの開発者と DSL の開発者はこれら 2 つを用いて開発を行う。エンドユーザーはプラットフォームの提供する対象とするシミュレーション向けの C++のアノテーションクラス (仮想クラス) を継承して C++でアプリケーションを作成する。

図 1 はプラットフォーム構成図である。

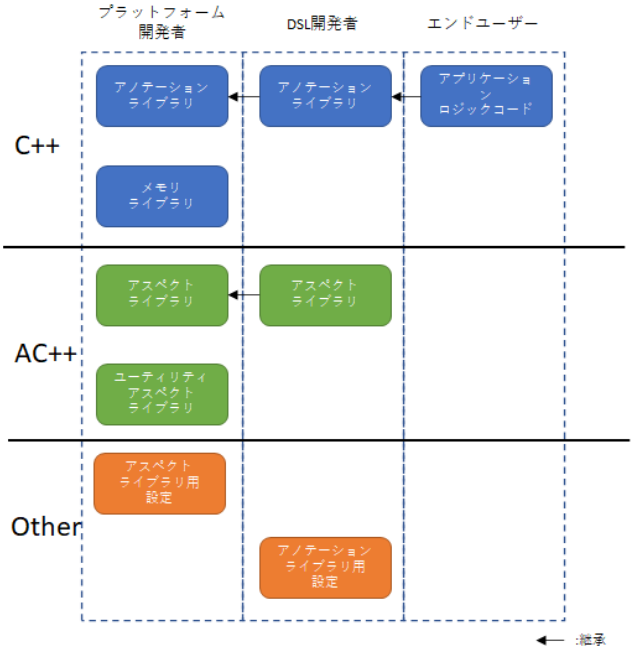

図 1: プラットフォーム構成概要
Fig. 1 Abstruct of platform structure

プラットフォームの主要コンポーネントは下記の 5 つである。

- 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

3.1 アノテーションライブラリ

エンドユーザーがアプリケーションを作成する際に継承するスーパークラスを提供する。プラットフォームの提供する汎用のスーパークラスのライブラリおよび、汎用のスーパークラスを継承して作成されたアプリケーションに特化されたスーパークラスで構成される。

コンパイル時には、スーパークラスに対してジョイントポイントが作成され、アスペクトの織り込みが行われる。それにより、エンドユーザーは利用するアスペクトモジュールを考慮せずにコードを作成可能となる。

3.2 メモリライブラリ

後述

3.3 アスペクトライブラリ

アスペクトライブラリは、コンパイル時に織り込まれる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。アスペクトは下記を行う。

- 対象とするハードウェア階層の制御と入力データの制御を行うコードをアノテーションライブラリのジョイントポイントに対して織り込む。
- メモリ管理のコードをメモリライブラリのジョイントポイントに対して織り込む。
- （任意）対象とするハードウェア階層およびアプリケーション向けの最適化を織り込む。

一例として、MPI の階層のアスペクトのメモリライブラリ向けの DryRun 機能によるデータ領域外へのメモリアクセスの調査とプリフェッチの機能などもアスペクトライブラリに含まれる。

3.4 コンパイル

プラットフォーム上でのプログラムのコンパイルは図 2 の様に行われる。

エンドユーザーが作成した C++のプログラムコードをライブラリとともに AspectC++コンパイラを用いて C++のコードへトランスコンパイルを行う。どのアスペクトモジュールを利用するかはアスペクトライブラリ用設定で定義する。エンドユーザーは出力されたプログラムコードを任意の環境でコンパイルして利用することができる。

なお注意点として、AspectC++の仕様により、ライブラリおよびエンドユーザーの作成するコードが C++11 で記載を行う必要がある。より新しいバージョンの C++を利用する場合は、マクロにより該当部分を除去したうえで、スタブを定義する。

3.5 プログラム例

実際の開発の例として、下記にプラットフォームが提供する汎用と MPI 向けのアスペクトのソースコード 1 および、エンドユーザーが作成するアプリケーションのコード 2 を載せる。なお、ソースコード 2 は 2 次元の熱拡散方程式の例で、境界条件として f(x)=0 の固定端を与えている。

メモリライブラリの提供する”環境”に対して環境ブロックの列挙を要請し、戻り値として返された各環境ブロック

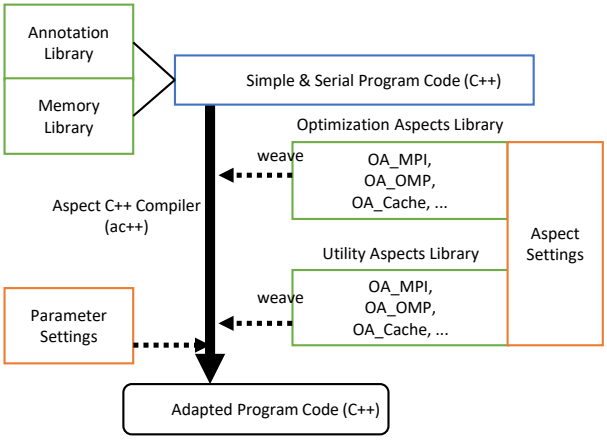

図 2: コンパイルフロー
Fig. 2 Compile Flow

に対してカーネル処理を行う。各ブロックに対するカーネル処理内部では、相対・絶対のいずれかのインデックスを用いて他のブロックを参照可能。参照はインデックスを用いて環境ツリーを探索し、そのインデックスが含まれるブロックとそのブロックに対する相対インデックス計算する。カーネル処理中のブロックでないブロックに対して書き込みを行うことも可能であるが、動作は保証しない。

```
aspect SUAspect
{
public:
pointcut su_main(int argc, char** argv) =
↪  execution("int main(...)") && args(argc,
↪  argv);
pointcut su_target_initialize() =
↪  call("void %::Initialize(...)" &&
↪  derived("sugoi::SU_Target"));
pointcut su_target_main() =
↪  call("void %::Main(...)" &&
↪  derived("sugoi::SU_Target"));
pointcut su_target_finalize() =
↪  call("void %::Finalize(...)" &&
↪  derived("sugoi::SU_Target"));

/*..略..*/
};

aspect SUAspect_MPI : public SUAspect
{
/*..略..*/
/*..MPI関連パラメータ..*/

protected:
advice su_main(argc, argv) : around(int argc,
↪  char** argv) {
  layer_level = SUAspect::current_layer_num;
  SUAspect::current_layer_num++;

  /*..MPI初期化..*/

  tjp->proceed();
  MPI_Finalize();
}

advice su_target_initialize() : around() {
  tjp->proceed();
  MPI_Barrier(MPI_COMM_WORLD);
}

advice su_target_main() : around() {
  flag_dryrun = true;
  updateBlockDistribution();
  tjp->proceed();
  MPI_Barrier(MPI_COMM_WORLD);
}

/*..略..*/
};
```

ソースコード 1: プラットフォーム用アスペクト例（汎用および MPI 向け）

```
#include "DEBUG.hpp"
#include <sugoi.hpp>
#include <sugoi_stencil.hpp>
using namespace std;

class My_Target2D :
↪  sugoi::stencil::SU_Target_Grid2D<double,
↪  4> {

  ENV env;
  int counter;
  const double alpah = 1.0 / 5;
  const double beta = 1.0 / 5;

public:
  void Initialize(int argc, char** argv)
  ↪  override {

    env.set_page_param(256);
    env.set_env_param();
    env.set_base_param(AddrT(128, 128));

    env.oa_gen = [](AddrT) { return 0.0f; };

    env.init_gen = [](AddrT addr) {
      auto& a = std::get<0>(addr), & b =
      ↪  std::get<1>(addr);
      return (double)sin(pi * (a + b) / 256);
    };

    env.generate();
  }
```

```
  void Main() override {
    for (int c = 0; c < 10000; ) {
      for (auto el : env.get_blocks()) {
        Kernel(*el);
      }
      if (!env.refresh()) c++;
    }
  }

  void Kernel(BLK& blk) {
    for (int64_t i = 0; i < BLK::SIZE_X; i++) {
      for (int64_t j = 0; j < BLK::SIZE_Y;
      ↪  j++) {
        blk[i][j] = alpah * blk[i][j] + beta *
        ↪  (blk[i - 1][j] + blk[i + 1][j] +
        ↪  blk[i - 1][j] + blk[i + 1][j]);
      }
    }
  }

  void Finalize() override {
    env.pretty_print();
  }
};
```

ソースコード 2: 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）

4. メモリ構造

概要で述べたように、本プラットフォームは SPMD 型並列プログラミングモデルとなっており、アスペクトを用いてプログラムに入力するデータを制御するすることで並列計算を実現している。

データの割り付けを行う際の単位として、プログラムデータの論理的な最小単位（ステンシル計算に置ける格子点や粒子法に置ける粒子）を用いた場合、データの過不足ない割り当てを行うことが可能であるが、メモリアクセスの時空間的局所性が失われる他、隣接のメモリアクセスでも探索を行う必要があるためオーバーヘッドが生じる問題点がある。初期のプラットフォームの実装では、上記の手法をとり、標準のアクセスと比較し約 5%の性能劣化が見られた。それに加え、エンドユーザーが作成するコードの複雑性とコード量も増大した [6]。

上記の問題を解決するため、データの割り付けを、プログラムデータの論理的な最小単位の集合（以下、ブロック）単位にまとめる、これらをツリー構造で管理する（図 3 参照）ことで上記の問題の解決を試みた。ことその結果、ブロック数が少ない場合は性能向上が見られたものの、ブロック数が多くなると従来のものより性能が劣化した。主な原因としてはブロックの探索に自体のオーバーヘッドの増加と最小単位通信の場合と比較してブロック単位の通信によって発生する余分な通信によりオーバーヘッドの増加があげられる。それに加え、ブロック単位のデータ管理を行った場合、CPU と GPU など、メモリレイアウトが異なるハードウェア上でプログラムを動作させることが難しくなるという弊害も発生した。

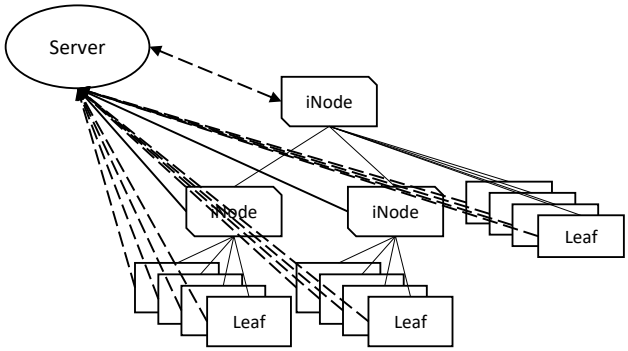

図 3: ブロックのツリー構造例 [7]
Fig. 3 Block Tree Structure

以上を踏まえ、プラットフォーム上で利用されるデータ構造と物理的なメモリ管理を分離する手法を検討する。

4.1 提案手法

物理メモリと仮想メモリを分離して管理する手法として、ページングが非常にスタンダートな手法であり、オペレーティングシステムのメモリ管理などで利用されている。

そこで、当プラットフォームのブロックをページングにおける仮想メモリに見立て、ページングを実装した。

メモリライブラリとして、次の 3 つを実装する。

- メモリプール (4.1.1 節)
- メモリページ (4.1.2 節)
- 環境ブロック (4.1.3 節)

なお、AMR の実装では動的なブロックの生成消滅が必要でとなるためガベージコレクション (GC) の実装が必須である。しかし現時点では、簡易化のため GC は実装しておらず、ページの再利用も行っていない。

4.1.1 メモリプール

基本の実装は一次元配列のメモリである。関数として、メモリの確保と必要量のメモリページの切り出しを持つ。アスペクトの畳み込みの際に、メモリの確保・切り出し、環境のアドレスからメモリアドレスへの変換の内部実装を置き換えることで、他のメモリのフォーマットにも対応させることができる。

4.1.2 メモリページ

メモリプール内のメモリへのポインタとページのデータを持つ。メモリページの表面上のサイズと実際に確保するメモリサイズに差を持たせることで、キャッシュの競合を防ぐことが可能である。

4.1.3 環境ブロック

プログラム基盤上でデータを扱う単位である。

アプリケーション特化のアノテーションライブラリで定義されたフォーマットのアドレスでアクセスを行う。アドレスはブロックの最初のデータを起点とする相対アドレスと、環境 (4.1.4 章を参照) 全体でユニークな絶対アドレスのいずれかを用いてアクセスすることが可能である。なお、添字演算子は相対アドレスでのアクセスのシンタックスシュガーとなっている。

プラットフォームのライブラリとして下記のような仮想クラスを提供し、アプリケーション特化のアノテーションライブラリではこれらを組み合わせて実体のブロックを作成する。

- 内部のデータ構造
  - データをもつ
  - 算術演算でデータを出力する
  - 他の環境ブロックへの参照を持つ
- アドレスのデータ構造
  - 一次元格子
    - アドレスでデータの保持/非保持を判定
    - 算術演算でデータの保持/非保持を判定
  - 二次元格子
    - アドレスでデータの保持/非保持を判定
    - 算術演算でデータの保持/非保持を判定
  - etc...

なお、データを持つ環境ブロックはリングバッファとしてデータを持っている。

4.1.4 環境

SPMD 並列計算モデルでのプログラムの入力となるデータの集合である。環境ブロックのツリー構造として構成される。例を図 4 にしめす。環境はアプリケーション特化のアノテーションライブラリで構築する。

ソースコード 2 の例の様に、環境は `Main` 関数内で修了条件を満たすまで更新を繰り返す。各ステップの更新は、環境のメンバ関数である `get_blocks` 関数を呼び、環境に属しているすべての更新が必要な環境ブロックを取得し、それらをアップデートすることで行う。なお、例では `Kernel` 関数を定義して各環境ブロックのアップデートを行っているが、必ずしもその必要はない。

その後、メンバ関数の `refresh` 関数を呼び、バッファの更新を行う。`refresh` 関数はバッファの更新に成功した際は false, 失敗した場合は true を返す。成功した場合は、次のステップに進み、失敗した場合は再度同じステップを実行する。これは、アスペクトによって織り込まれた処理によってカーネル関数の再実行が要請されることがあるためである。

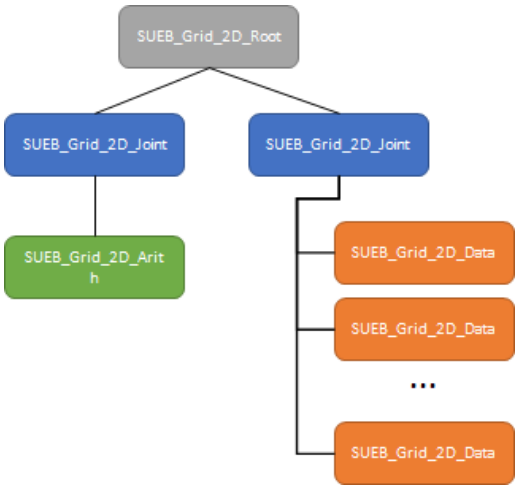

図 4: 環境のツリー構造例（二次元格子向け）
Fig. 4 Env Block Tree Structure

4.2 環境ブロック構築例

二次元格子向けに環境ブロックを定義した例をソースコード 3 に示す。

```
using AddrE = int64_t;
using AddrT = std::tuple<AddrE, AddrE>;
using BaseBLK = SU_EnvBlock_Grid_N<T, AddrT,
↪  AddrESeed>;

class SUEB_Grid_2D_Root final : public BaseBLK
{
public:
    SUEB_Grid_2D_Root()
        : SU_EnvBlock_Grid<T, AddrT>() {
        /*..略..*/
    }
};

class SUEB_Grid_2D_Joint final : public
↪  BaseBLK, public
↪  SU_EnvBlock_Grid_FilledRect<T, AddrT>
{
public:
    SUEB_Grid_2D_Joint(const AddrT ga_s, const
    ↪  AddrT ga_e)
        : SU_EnvBlock_Grid<T, AddrT>(ga_s,
        ↪  ga_e),
        ↪  SU_EnvBlock_Grid_FilledRect<T,
        ↪  AddrT>() {
        /*..略..*/
    }
};

class SUEB_Grid_2D_Data final : public
↪  BaseBLK, public
↪  SU_EnvBlock_Grid_FilledRect<T, AddrT>,
↪  public SU_EnvBlock_Data<T, AddrT>
{
```

```
public:
    SUEB_Grid_2D_Data(const AddrT ga_s, const
    ↪ AddrT ga_e, std::vector<std::vector<SU」
    ↪ _MemPage<T>*>> buffer, bool isNeedCopy
    ↪ = false, bool isVirtual = false)
        : SU_EnvBlock_Grid<T, AddrT>(ga_s,
        ↪ ga_e),
        ↪ SU_EnvBlock_Grid_FilledRect<T,
        ↪ AddrT>(), SU_EnvBlock_Data<T,
        ↪ AddrT>(buffer, isNeedCopy) {
        /*..略..*/
    }
};

class SUEB_Grid_2D_Arith final : public
↪ BaseBLK, public SU_EnvBlock_Arithmetic<T,
↪ AddrT>
{
public:
    SUEB_Grid_2D_Arith(std::function<T(AddrT)>
    ↪ func)
        : SU_EnvBlock_Grid<T, AddrT>(),
        ↪ SU_EnvBlock_Arithmetic<T,
        ↪ AddrT>(func) {
        /*..略..*/
    }
};

class SUEB_Grid_2D_Ref final : public BaseBLK,
↪ public SU_EnvBlock_Reference<T, AddrT>
{
public:
    SUEB_Grid_2D_Ref(std::function<AddrT(AddrT」
    ↪ )>
    ↪ func)
        : SU_EnvBlock_Grid<T, AddrT>(),
        ↪ SU_EnvBlock_Reference<T,
        ↪ AddrT>(func) {
        /*..略..*/
    }
};
```

ソースコード 3: メモリ構造の定義例（二次元格子向け）

上記例では、下記の 5 つの環境ブロックがそれぞれ定義されている。

- SUEB_Grid_2D_Root：環境のツリー構造のルートとなるブロック。
- SUEB_Grid_2D_Joint：環境のツリー構造を分割するためのブロック。後述。
- SUEB_Grid_2D_Data：実データをもつブロック。
- SUEB_Grid_2D_Arith：実データを持たず、アドレスに対して算術演算を行って結果を返すブロック。ディリクレ境界等で用いる。
- SUEB_Grid_2D_Ref：実データを持たず、他のブロックのデータを返すブロック。ノイマン境界や AMR 等で用いる。

従来の実装では、Root と Data のみがブロックとして定義されており、Arith と Ref は OuterArea として境界条件として別途定義されていた。しかし本実装ではこれらをすべて統一し、更に Joint ブロックを追加した。この実装の変更は 2 つの理由による。

1 つ目は、AMR の実装を見据え、ツリー構造が動的に変化した際に、ツリーの一部に対して外部からの参照を持つオブジェクトがいた場合、コードが複雑になり、また更新のコストもかかるためである。

2 つ目はブロック間をまたぐアクセスを行う際の、ブロックの検索の高速化のためである。各 Joint ブロックは直下のデータブロック群の間の相互参照の解決を担当しており、各データブロックは直上の Joint ブロックを使うことで、隣接ブロックの検索を行う。隣接ブロックが見つからなかった場合には、Joint ブロックは更に上の環境ブロックに検索を依頼することを繰り返して隣接ブロックの検索が行われる。k-d 木や Barnes-Hut ツリーアルゴリズムに現れる 4 分木のような、空間的局所性を反映するような木構造として、この環境のツリー構造を構成すると、隣接ブロックの検索は直上の Joint ブロックの呼び出しだけで終了する可能性が高くなる。また直上の Joint ブロックが管理するキャッシュテーブルのサイズは、木構造にすることでブロックの総数に関わらず一定サイズに小さく保つことが出来る。このようにして隣接ブロックの検索の高速化が可能になっている。このことにより、メモリアクセスの空間局所性が高いアプリケーションにおいては、図 4 に対して図 5 の様に Joint ブロックを挿入することにより速度向上が望める。

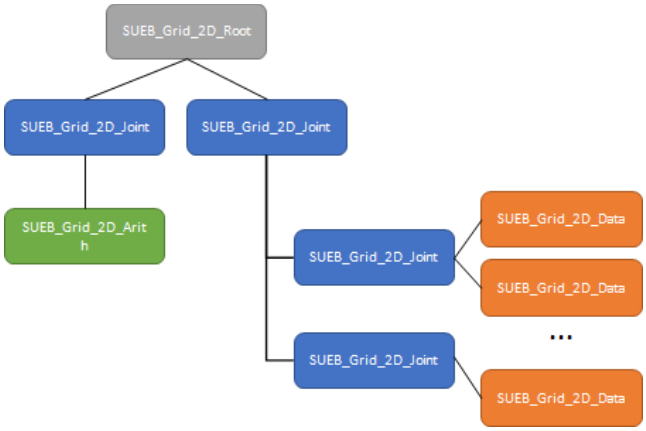

図 5: 環境のツリー構造例 2（二次元格子向け）

Fig. 5 Env Block Tree Structure

5. 性能評価

二次元格子の熱拡散方程式（倍精度）を実行し、理論値

に対するパフォーマンスを計測した。なお、メモリアクセスのパフォーマンスのみを計測するためアスペクトの織り込みは行っていない。また、SIMD 等のアクセラレータも利用していない。よって、シングルスレッドで 1 プロセスでの動作となる。

実行環境は次の通り。

- CPU: Xeon E5-2699v4
- Cores: 22 * 2
- Frequency: 2.2GHz
- Memory: PC4-19200 128GB
- コンパイラ: Visual C++ 2019 00435-60000-00000-AA968

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
- 要素のデータ形式: Double Precision Floating Point
- 全領域のデータサイズ: 128 * 128, 256 * 256, 512 * 512
- ブロックのデータサイズ: 4 * 4, 8 * 8, 16 * 16, 32 * 32, 64 * 64
- ページサイズ: 2KB, 4KB

結果は図 6 および図 7 の通り

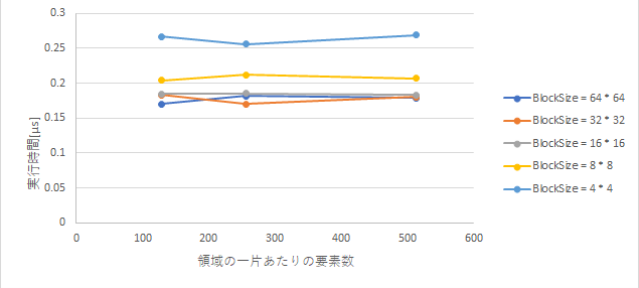

図 6: 一要素あたりの実行時間 (ページサイズ 2KB)

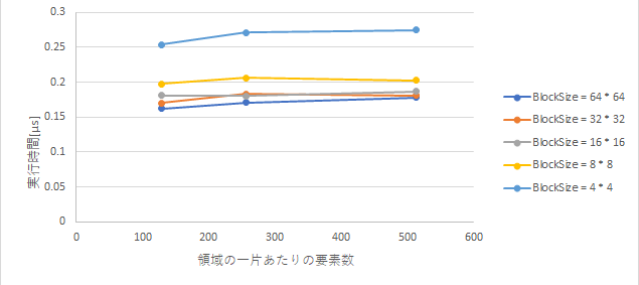

図 7: 一要素あたりの実行時間 (ページサイズ 4KB)

ページのサイズに関しては、4KB ページのものが概ね良いパフォーマンスであった。各ブロックのサイズが大きく、領域のサイズが小さいほど良い結果である点から、ブロック数が多い場合にパフォーマンスの劣化が発生している事が分かる。

6. 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 [11]
- ループの並列化を Aspect を用いて行った B. Harbulot ら [12] および João L. Sobral らの研究 [13]

上記のいずれも AspectJ をベースとして用いている。

7. 終わりに

パフォーマンスは SIMD を利用していない場合の理論性能の 0.3 シリアルなメモリアクセスでないことを考慮しても非常に遅いといえるだろう。パフォーマンスの劣化の要因は次の三つが推測される。

- ページングを OS の機能とは別途実装しているため、メモリアクセスの解決に必要な処理が 1 段階増加してしまっていること
- コードの複雑化の回避のため、ポリモーフィズムを多用していること。
- ブロックの検索に大きなオーバーヘッドがかかっている。

1 つ目は、環境ブロックと物理メモリの実装は分離されているため、ページングを OS のページングに沿う形の実装にすることでオーバーヘッドを減らすことが可能であると考えている。2 つ目は純粋な実装の問題であるため、不要なポリモーフィズムの利用の削減を行うことで改善することが可能であると考えている。3 つ目は、今回の性能測定ではアスペクトの織り込みを行わなかったため、Joint ブロックの挿入が行われていない状態での測定となったことが原因の一つと推測される。デフォルトでもある程度の木の分割を行うことで性能改善を行うことができると考えている。

メモリライブラリの拡張として、下記の実装を予定している

- ガベージコレクション・ページの再利用・コピーオンライトによるメモリ利用の効率化。そして AMR への対応。
- 複数のメモリプール対応を利用した、ファイルの分散読み込み対応や通信時のバッファへのコピーの削減

プラットフォームの既知の問題として、アスペクト言語の限界に起因する、関数/変数のブロックに対してのみしか制御の挿入を行うことができないという問題がある。これはプログラムコードの制御に対して操作を行うことが可能なディレクティブと比較し、大きく不利な点である。当問題を解決するため、我々はアスペクト言語の中間言語への拡張を検討している。

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] : Legion Overview – Legion Programming System, `http://legion.stanford.edu/overview/`. (Accessed on 01/27/2018).

[5] 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).

[6] 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).

[7] 脩　石村，芳英吉本：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，技術報告 42，東京大学, 東京大学 (2019).

[8] Berger, M. J. and Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations, *Journal of Computational Physics*, Vol. 53, No. 3, pp. 484–512 (online), DOI: https://doi.org/10.1016/0021-9991(84)90073-1 (1984).

[9] 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).

[10] 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).

[11] 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).

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

[13] 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).

付　　録

A.1 ソースコード

`https://github.com/hrontan/SUGOI`