HPCアプリケーション向け 計算網羅性・計算順序をテストするツール

穂積 俊平[1,a)]　佐藤 芳樹[1]　千葉 滋[1]

概要：多くの High Performance Computing (HPC) アプリケーションでは, メモリアクセス最適化で大幅な性能向上が期待できる. 例えば, 行列積計算を部分行列へ分割するブロック化は, 連続した配列アクセスを局所化しキャッシュヒット率を向上させられる. 一方, ブロック化の効果を最大化するために施す計算順序の変更や, それに伴う計算カーネルの分割によって, 実装コードは散在し可読性や保守性の低下を招く場合がある. さらに, ブロックの境界条件を判定するコードの増加によって, 一部の行列要素の計算漏れや重複計算のような新たなバグの発生も問題となってくる. そのような最適化にともなうバグは, 最適化の前後のプログラムの計算結果を突合すれば発見できるが, 手動による突合は現実的ではない. また単純な機械的な突合では, 浮動小数点の切り上げや乱数を使った初期パラメータ生成への対応が難しい. 本研究では, 実行ログを利用した HPC アプリケーションの計算順序及び計算網羅性のテストツールを提案する. このテストツールは, 指定した計算カーネルでの配列アクセスの等の実行ログを記録し, ログの検証やログ同士を突合するための機能を提供する. アスペクト指向のポイントカットに基づいた記述により, 散在した計算カーネルの位置や取得するログ内容を柔軟に記述できる. また, 実行ログを集合として扱うための API を提供し, 配列のアクセスシーケンスの比較や, 同じ配列インデックスでの計算結果の突合などを単純な集合演算として記述できるようにした. 提供している API は, MPI など分散環境にも対応できるようになっている.

キーワード：HPC, 単体テスト, 実行時検査

Hozumi Shumpei[1,a)]　Sato Yoshiki[1]　Chiba Shigeru[1]

1. はじめに

HPC では実行性能が重要視されるため, プログラムにはメモリアクセス局所化などの最適化が施され, クラスタコンピュータなどの大規模並列分散環境で実行される. 最適化や分散実行は, プログラムの実行性能を高める一方で, プログラムにバグが混入する要因となり得る. HPC アプリケーションの多くは, カーネル計算と呼ばれる核となる計算を繰り返し実行するプログラムである. カーネル計算が巡回する領域であるシミュレーション領域は, 最適化や分散実行により分割される場合がある. シミュレーション領域の分割は, プログラム中の境界判定コードを増加させ保守性や可読性の低下を招く. 結果として, 人為ミスを誘発し, 計算漏れや計算重複といったバグをプログラムに混入させる. これらのバグは単純には計算結果の突き合わせにより検証できるが, 浮動小数点演算の誤差などの問題があり, 簡単ではない.

本稿では, 計算漏れや計算重複といった HPC 特有のバグを検出するためのテストツール HPCUnit を提案する. HPCUnit は対象プログラムの実行ログを取ることで実際に計算した領域を取得し, 正しい領域と突合・検証することで, プログラムの計算網羅性や計算順序をテストする. シミュレーション領域の取得やシミュレーション領域の検証・突合は, HPCUnit が提供する API を利用することで, 対象プログラムから分離した形で簡潔に記述できる. シミュレーション領域の取得はコントロールフローやクラスによって細かく指定でき, テスト内容に応じてシミュレーション領域の大きさや形状を任意に変更できる.

以下では, 2 章で HPC アプリケーションのテストの難しさを指摘し, 3 章で HPC アプリケーションのテストツール HPCUnit について説明する. 4 章では, HPCUnit のオーバーヘッドを計測した実験結果を示し, 5 章で関連研究を紹

[1] 東京大学 情報理工学系研究科創造情報学専攻
Dept. of Creative Informatics, The University of Tokyo
[a)] hozumi@csg.ci.i.u-tokyo.ac.jp

```
1  for(i = 0; i < N; i++) {
2   for(j = 0; j < i; j++) {
3    ip = 0
4    for(k =0 ; k < M; k++)
5     ip += Q[j][k] * A[i][k]
6    for(k = 0; k < M; k++)
7     Q[i][k] -= Q[j][k] * ip
8   }
9   Q[i] /= |Q[i]|
10 }
```

図 1 最適化前の GS 法の擬似コード

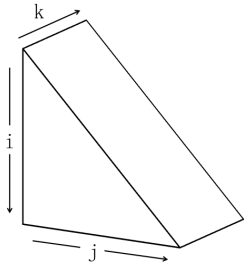

図 2 最適化前の GS 法のシミュレーション領域

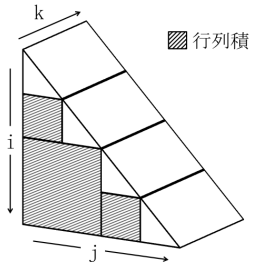

図 3 最適化後の GS 法のシミュレーション領域

介する. そして最後に, 6 章でまとめと今後の課題を述べる.

2. HPC アプリケーションのテストの難しさ

2.1 性能向上のためのシミュレーション領域の分割

HPC アプリケーションでは実行性能を向上させるために, 計算対象のシミュレーション領域を分割して計算する事が多い. シミュレーション領域の分割は, メモリアクセス局所化による最適化や, GPGPU やスーパーコンピュータを利用するためのデータ分散に伴って必要となる.

シミュレーション領域の分割を伴うメモリアクセスの局所化は, キャッシュヒット率を向上させる最適化手法としてよく知られている. そのような最適化には, 同一データアクセスの局所化（時間的局所性）や, 周辺メモリ領域へのデータアクセス局所化（空間的局所性）がある. 例えば, 行列積計算における行列のブロック化は, 行列を部分行列（ブロック）に分割し, 部分行列に行列要素の参照を集中させ, 短い時間に同じ要素を複数回参照させる. スーパーコンピュータで実行される多くの物理シミュレーションプログラムは, 配列化した行列やベクトルなどで表現されるシミュレーション領域の要素に対して同一の計算（カーネル計算）を繰り返すため, データ依存の無い計算の順序変更が容易でありプログラムの局所性を向上させやすい. 2011 年のゴードン・ベル賞を受賞した物理シミュレーションソフトウェア RSDFT [1], [2] においても, 中核を担うモジュールであるグラムシュミットの正規直交化法 (GS 法) の一部を, 行列積に置き換え, 局所性を高めている [3]. GS 法とは, ある線型独立なベクトルの組が与えられたとき, そこから正規直交系を作り出すアルゴリズムのことである. GS 法は, 行列 A を与える N 本の長さ M のベクトルの組, 行列 Q を GS 法によって得られる正規直交系とすると, 図 1 のように 4 つの for ループを用いたプログラムとして表現できる. このプログラムは, 7 行目のカーネル計算が, 図 1 中の 3 つの変数 i, j, k が作り出す図 2 のような三角柱全体を巡回して反復的に計算するプログラムと捉えられる. GS 法に行列のブロック化を施し, 計算の一部を行列積に置き換えると, 巡回する領域は, 図 3 のように複数の領域に分割されることになる.

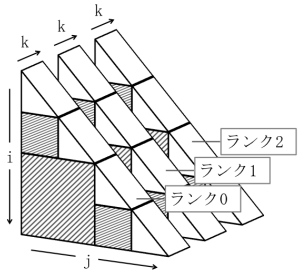

図 4 MPI を利用した GS 法のシミュレーション領域

一方, スーパーコンピュータ上でのシミュレーションのために, MPI [4] を利用した並列分散プログラムでは, 計算の一部を各計算ノードに割り当てるため, シミュレーション領域は更に分割される. 例えば, MPI を利用し, 3 ノードで並列に実行される GS 法プログラムのシミュレーション領域は図 4 のように分割された領域となる. 分割されたシミュレーション領域は, 別々の分散メモリ上に分離される. そのため, 単一ノードでは絶対アドレスでアクセスしていたデータでも, 各ノード毎に割り当てられたデータ範囲に応じた相対アドレスによるデータアクセスが必要となる. 相対アドレスによるデータアクセスへの変更は, GPGPU を利用したプログラムでも必須である. GPGPU の大量の計算コアを効率よく利用するためには, キャッシュ効率を意識して, ストリーミングマルチプロセッサの共有メモリにデータを配置する必要がある. 例えば, NVIDIA CUDA [5] では, 複数の CUDA コアを持つ SIMD 演算用のストリーミングマルチプロセッサ (SMX) 毎に共有メモリを持つため, MPI を利用したプログラムと同様のデータ分割や相対

アドレスによるデータアクセスが必要となってくる. また, CUDA では, メモリバスサイズに合わせて CUDA コアが処理するスレッドを複数まとめ, メモリアクセスを一括処理させる事で更なる性能向上が期待できる（コアレッシング）. そのため, メモリバスサイズに合わせたデータアクセスを行うよう, ループ処理を適切に再構成するような変更も必要となる.

2.2 領域分割による計算漏れ・計算重複

シミュレーション領域の分割されたプログラムは, 計算漏れや計算重複のようなバグを引き起こしやすい. 一般にシミュレーション領域の分割は, カーネル計算の分割を伴う修正である. 分割されたカーネル計算のプログラムには, 対象要素が分割された領域に含まれるか否かを逐一判定するコードが含まれる. そのような判定コードの増加によって, 分岐やループ判定条件が複雑になり可読性や保守性が著しく低下する. 複雑化したコードは人為的な実装ミスを誘発し, 意図しないシミュレーション領域が指定されるようなバグを引き起こしやすい. 意図しないシミュレーション領域の縮小は誤った計算結果を導出し, 逆にシミュレーション領域の拡大による不要な計算は実行時間を増加させ, 最悪の場合, プログラムが無限ループで停止しない事もある.

このような計算漏れや計算重複の有無を人為的な突合や機械的な比較で検査する事は難しい. 多くの物理シミュレーションでは, 浮動小数点型のデータ演算を多用するため, シミュレーションの計算過程で桁落ちや情報落ちが発生する. そのような計算誤差を考慮して, 最適化前後での計算結果を比較する必要がある. さらに, シミュレーションの初期状態のデータは乱数をベースに生成されることが多いため, 同一のシミュレーションプログラムを用いても同じ計算結果が得られるとは限らない.

3. HPCUnit

本研究では, HPC アプリケーションの計算網羅性及び計算順序のテストツール HPCUnit を提案する. HPCUnit は, サンプリングした実行時ログによる実行モデルを元にした単体テストを可能にし, プログラムの計算漏れや計算重複, カーネル計算の依存関係が正しく保たれているか等をテストできる. HPCUnit における実行モデルは, プログラムがシミュレーション領域のどの部分を計算したかを表現し, 実行時ログから作られる. そのため, プログラムが巡回したシミュレーション領域の各要素を順序付き集合として取得するための API と, シミュレーション領域の検証・突合を集合演算として記述するための API を提供する.

我々は, HPCUnit を Java アプリケーション向けに開発し, アスペクト指向に基づきユーザが柔軟にテスト対象の実行ログを抽出できるようにした. これにより, ユーザは

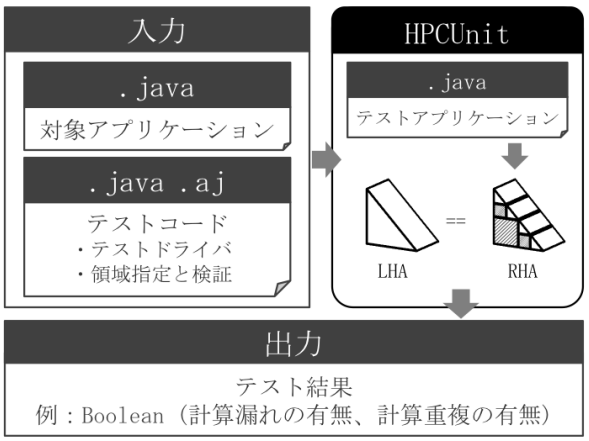

図 5 HPCUnit の全体像

HPCUnit が提供する AspectJ [6] の API を用いて, テストコードをテスト対象プログラムとは分離して記述できる. テストコードには, 実行ログとしてのシミュレーション領域の指定（ポイントカット記述）, 及びシミュレーション領域を検証もしくは突合する Java コードが含まれる. テストコードから呼び出される対象アプリケーションには, コンパイル時にテストコードが織り込まれ, テスト実行時にログ取得と領域検証が行われる（図 5）. 領域の検証は, 正しい領域 (LHA: Left Hand Area) と, 実際に取得したシミュレーション領域 (RHA: Right Hand Area) の比較で行う. HPCUnit API を利用すると, コントロールフローやクラスを細かく指定し, RHA を目的に合わせた任意の大きさ, 形状で取得できる. 取得する RHA は任意の次数のタプルの順序付き集合としても取得でき, 各座標での計算結果に加え, MPI のランクなどの実行時情報を含めることもできる

3.1 シミュレーション領域を抽出する API

シミュレーション領域の取得は, AspectJ と HPCUnit が提供する API を利用して記述する. AspectJ は, Java を拡張したアスペクト指向言語であり, シミュレーション領域の取得をテスト対象プログラムと分離して記述できる. ポイントカット記述によって指定されたカーネル計算に対して, HPCUnit が提供する集合 API を利用し変数を選択的に取得する. 現段階では, HPCUnit ではカーネル計算がメソッドとして定義されていることを前提としており, カーネル計算メソッド呼び出し時の実引数を観測する. 例えば, メモリアクセスの局所化を施した GS 法が図 6 のように定義されている場合, OptimizedGS クラスと Matmul クラス, それぞれの kernel メソッド呼び出し時の実引数値を観測し, シミュレーション領域を得る. ポイントカット記述による観測位置の指定では, コントロールフローやクラスを限定することができ, 観測対象のコードを細かく指定することができる. GS 法を例にすると, 行列積が担当し

```
1  class OptimizedGS {
2   static void calc () {
3    ... kernel(i, j, k, ..);
4    ... Matmul.calc(..);
5    ...
6   }
7   static void kernel(int i, int j, int k, ..) {}
8  }
9
10 class Matmul {
11  static void calc () {
12   ... kernel(i, j, k, ..);
13   ...
14  }
15  static void kernel(int i, int j, int k, ..) {}
16 }
```

図 6 メモリアクセス局所化を施した GS 法

```
1  public aspect MonitoringKernel {
2   /* 領域指定*/
3   pointcut atMatmulKernel (int i, int j, int k, ..) :
4    args(i, j, k) &&
5    within(Matmul) &&
6    call(void kernel(int, int, int, ..));
7
8   before(int i, int j, int k):atMatmulKernel (i, j, k) {
9    GSTest.squares.add(new HUTuple3<>(i, j, k));
10  }
11 }
```

図 7 GS 法のシミューレション領域の指定

ている領域のみを取得したい場合, 図 7 の atMatmulKernel のように, within キーワードを利用して Matmul クラスに限定することができる.

観測内容の指定は, AspectJ のアドバイスと HPCUnit の集合 API を利用して記述する. 集合 API は, 実行状態を表す HUTuple とその順序付き集合である HUList で表現される. GS 法を例にすると, kernel メソッド呼び出し時の i, j, k をタプルとする集合を取得するプログラムは, 図 7 の before:atMatmulKernel アドバイスのように記述できる. タプルに含める情報は, 配列の添字だけでなく, MPI のランクや, 集合のサイズを参照することで計算に利用された順序も含めることができる.

3.2 シミュレーション領域を検証・突合する API

シミュレーション領域の検証・突合は, 表 1 に示した HUList が提供する集合演算と, その等価性の判定によって記述する. GS 法の場合, 図 8 のように, 最適化された

表 1 HUList API (集合演算, 集合等価性)

HUList⟨T⟩ API	意味
boolean equals(Object o)	集合の等価性
HUList⟨T⟩ union(HUList⟨T⟩ list)	和集合
HUList⟨T⟩ intersection(HUList⟨T⟩ list)	積集合
HUList⟨T⟩ difference(HUList⟨T⟩ list)	差集合
⟨U⟩ HUList⟨U⟩ map(HUMapper⟨T, U⟩ mapper)	写像
⟨U⟩ U fold(HUFolder⟨T, U⟩ folder, U origin)	折りたたみ

表 2 HUList API (シミュレーション領域生成)

HUList⟨T⟩ API	意味
static ⟨U⟩ HPCList⟨U⟩ getNull ()	空集合生成を生成
static HUList⟨HUTuple1⟨Integer⟩⟩ getLine (int L1)	長さ L1 の線分を生成
static HUList⟨HUTuple2⟨Integer, Integer⟩⟩ getSquare (int L1)	一辺の長さが L1 の正方形を生成
static HUList⟨HUTuple2⟨Integer, Integer⟩⟩ getRightTriangle (int L1, int L2)	隣辺の長さが L1, L2 の直角三角形を生成
static HUList⟨HUTuple3⟨Integer, Integer, Integer⟩⟩ getCube (int L1)	一辺の長さが L1 の立方体を生成
static HUList⟨HUTuple3⟨Integer, Integer, Integer⟩⟩ getTriangularPrism (int L1, int L2, int L3)	隣辺の長さが L1, L2 の直角三角形を底面とする高さ L3 の三角柱を生成

```
1  class GSTest {
2   static
3    HUList<HUTuple3<Integer,Integer,Integer>>
4     squares = new HUList<>(),
5     triangles= new HUList<>(),
6     nullList = HUList.getNull();
7     gs = HUList.getTriangularPrism(30, 29, 30),
8
9   public static void main(String[] args){
10   /* テストドライバ*/
11   ... OptimizedGS.calc(..);
12
13   /* 検証内容*/
14   // 計算漏れ
15   gs.equals(triangles.union(squares));
16   // 計算重複
17   nullList.equals(triangles.intersection(squares));
18  }
19 }
```

図 8 GS 法の計算漏れ・計算重複のテスト例

Optimized クラスの中で, 行列積計算を利用するシミュレーション領域 (squares) と, それ以外の領域 (triangles) の集合比較で計算漏れ・計算重複をテストできる. 例えば, 計算

```
1 gsMPI = gsRaw.map(
2  new HUMapper<HUTuple4<..>,HUTuple3<..>>(){
3   HUTuple3<..> calc(HUTuple4<..> t) {
4    return new HUTuple3<>(
5     t.el1, t.el2, t.el3 + t.el0 * 30);
6   }
7 });
```

図 9 map メソッド：相対アドレス化されたシミュレーション領域の絶対アドレス化

漏れのテストは 15 行目のように, union メソッドを利用し, triangles と squares の和集合を RHA とし, 正しい LHA を与え等価性を判定する事で達成できる. HPCUnit API は, 図 2 のように, LHA として任意のシミュレーション領域を与えられるよう, getTriangularPrism メソッドのような基本的な集合生成メソッドを提供する. さらに, 最適化を施していない GS 法のシミュレーション領域を LHA として与え, 網羅性や順序のテストだけでなく, 最適化前後での計算結果や巡回順序の突合も可能である.

また, HUList は集合全体に対する演算（写像）を記述するための汎用的な map メソッドを提供する. 例えば, 先述のように, MPI を利用した分散プログラムが生成するシミュレーション領域はノード毎に分割されるため, 単一ノード実行と比較可能なシミュレーション領域を復元する必要が出てくる. map メソッドを利用すれば, 各領域のオフセット値を計算し, その領域全体に足し合わせる操作を直感的に記述できるようになる. 例えば, GS 法がベクトルの長さ 90 で 3 ノードで並列に実行される場合を考える. この場合, シミュレーション領域は図 4 のように, 変数 k の方向に 3 分割され, 分割された領域の奥行きは 30 となる. 分割された領域から全体の領域を復元するためには, MPI.rank * 30 を k が生成した要素に足し込む必要がある. GS 法から取得したシミュレーション領域を表す集合を gs とし, 第一要素を MPI のランク, 以下 i, j, k それぞれの変数値とすると, この操作は, 図 9 のように map メソッドを利用して記述できる.

一方, 集合に対する実行順序を考慮したテストを記述するために, HUList は fold メソッドを提供する. fold メソッドを用いると, シミュレーション領域の巡回順序に従った繰り返し処理（畳み込み）が記述でき, カーネル計算の依存関係や, データ参照の局所性を評価に利用できる. 例えば, GS 法は変数 i に関してカーネル計算を昇順に行う必要がある. GS 法の依存関係テストは図 10 のように記述できる. fold メソッドを使い GS 法プログラムから取得したシミュレーション領域 gs を走査し, gs に含まれるタプルが変数 i に関して昇順に並んでいるかテストする. また, fold を使い gs を走査し, gs に含まれるタプルのインターバルの総和を求めれば, プログラムの局所性を評価することもできる.

```
1 // 前後のタプルを参照し, タプルの一番目の要素に関して非減少列となっているか確認する
2 gs.fold(HUFolder.decreasingOrder, origin);
```

図 10 fold メソッド：減少列のテスト

4. 実験

HPCUnit によるシミュレーション領域取得のオーバーヘッドを測定するために, マイクロベンチマークを行い, さらにいくつかのシナリオを基に GS 法へ適用した際のオーバーヘッドを測定した. 実験は以下の環境で行った.

- FUJITSU Supercomputer PRIMEHPC FX10 1 ノード
- Linux ベースの専用 OS (カーネル 2.6.25.8)
- SPARC64TM IXfx 1.848 GHz
- Memory 32 GB
- OpenJDK Runtime Environment (IcedTea6 1.11.5) (linux-gnu build 1.6.0_24-b24) OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)
- 実行オプション デフォルト

4.1 マイクロベンチマーク

マイクロベンチマークの対象プログラムは, 配列の要素を添え字の 2 倍にするプログラムである. このプログラムを元に 5 つのプログラムを作成し, 実行時間を測定した. 最初の 3 つのプログラムは, 元の対象プログラム (a), AspectJ を使い, 対象プログラムに HUList と HUTuple を使ったシミュレーション領域取得コードを織り込んだプログラム (b1), 手作業で, 対象プログラムに, HUList と HUTuple を使ったシミュレーション領域取得コードを織り込んだプログラム (b2) である. プログラム (b1) と (b2) では, 取得した領域をオブジェクトの集合で表現するが, これを配列で表現するものも用意した. 我々は, これを AspectJ を使って織り込んだプログラム (c1), 手作業で織り込んだプログラム (c2) についても測定した.

マイクロベンチマークの実験結果として, 表 3 にそれぞれのプログラムを 7 回実行し最大と最小を除いた 5 回の平均実行時間と標準偏差を示す. 表 3 から, (b) の実行時間 が (a) の実行時間に比べ, 大幅に上昇していることがわかる. 実行オーバーヘッドの原因は HUTuple のオブジェクト生成コストに起因すると考えられる. 一方, 配列を使い, シミュレーション領域を取得した (c) は, (b) と比較するとオーバーヘッドを大幅に減らせている. また, (c1) と (c2) を比べると, AspectJ による織り込みを行った方がオーバーヘッドが大きいこともわかる. 一般に，AspectJ を利用して織り込みを行ったコードには，実行ログを収集するアスペクト実装を参照する間接的なメソッド呼び出しが含まれる. そのため，マイクロベンチマークのように非

表 3 マイクロベンチマーク：シミュレーション領域収集

	実行時間（ミリ秒）	標準偏差
(a) 織り込みなし	42	0.5
(b1) HU AspectJ	12800	215
(b2) HU 手織り込み	12690	135
(c1) 配列 AspectJ	143	0.4
(c2) 配列 手織り込み	82	0.4

常に計算量が小さいプログラムの実行では，間接参照によるオーバーヘッドが顕著に観測されたと考えられる．マイクロベンチマークの結果を活かし, 配列化による最適化や AspectJ によるオーバーヘッドを減らす方法を HPCUnit に取り入れていきたい.

4.2 GS 法への HPCUnit の適用

シミュレーション領域が変更された時のオーバーヘッドを測定した．実験では, 元の GS 法プログラム (a), 計算漏れ・計算重複をテストするために, GS 法のカーネル計算を観測するプログラム (b), プログラムの局所性を評価するために, GS 法の対象となる行列要素の参照を観測するプログラム (c), GS 法が行列全てを網羅している事を保証するために, GS 法の結果を保存する行列への書き込みを観測するプログラム (d) を用いた.

実験結果として, 表 4 にそれぞれのプログラムを 7 回実行し最大と最小を除いた 5 回の平均実行時間と標準偏差を示す．また, (b), (c), (d) の実行時間に含まれるオーバーヘッドの, (a) の実行時間に対する比率を図 11 に示す. 図 11 を見ると, (b) では, 元プログラムの 188%, (c) では, 元プログラムの 468%という大きなオーバーヘッドがかかっている．大きなオーバーヘッドがかかっている原因は, シミュレーション領域取得にかかるオーダーが大きい事が考えられる．GS 法のオーダーは, 入力となる行列サイズが N x N であるとき, $O(N^3)$ であり, (b), (c) の領域取得にかかるオーダーも $O(N^3)$ である．そのため, 計算量の観点から, オーバーヘッドの占める割合が大きい．また, シミュレーション領域取得の際に, AspectJ の提供する cflow を利用したこともオーバーヘッドが大きい原因と考えられる．cflow は実行時のコントロールフローを利用し, アドバイスの on / off を制御する機構であり, 実行時にアドバイスを有効にするか判断するコードが挿入されるため, オーバーヘッドが大きい．一方, (d) では, オーバーヘッドは 14%に抑えられている．(d) 行列書き込み は, 最終的な実行結果の書き込みのみを観測しており, (d) のオーバーヘッドの オーダーは $O(N^2)$ である．GS 法全体のオーダーは, $O(N^3)$ であるため, (d) のオーバーヘッドが占める割合は低くなっている．多くのプログラムでは, GS 法と同様に最終的な計算結果の代入回数が読込回数よりも小さいオーダーであるため, HPCUnit によるシミュレーション領域取得のオーバーヘッドは許容可能であると考えられる.

表 4 GS 法のシミュレーション領域とログ取得時の実行時間

	実行時間（ミリ秒）	標準偏差
a) 元の GS 法プログラム	625	20
b) カーネル計算観測	1800	98
c) 行列参照観測	3556	15
d) 行列書き込み観測	715	35

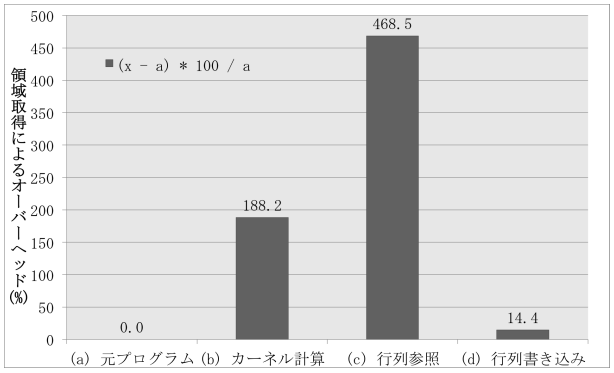

図 11 シミュレーション領域取得によるオーバーヘッド

5. 関連研究

従来から, プログラムの仕様及び動作の検証は, 様々なアプローチで取り組まれてきた．本章では, 一般的な単体テスト手法に加え, 動作時の情報を利用した実行時検査及び形式的に記述した性質を検査するモデル検査, について HPCUnit と比較する.

多くの単体テストツールとは異なり, HPCUnit は実行ログで表された動作モデルを検査対象とする．単体テストとは, 関数やメソッド等の各モジュール単位に, 実装が動作仕様に合致するかを検査する作業であり, プログラムの早期のバグ出しに有効なステップである．Java 向けの代表的な単体テストツール JUnit [7] では, 複数の単体テストを自動化し, 入力値や状態に応じた出力に対する様々なテストを容易に実行できる．しかし, 多くの単体テストツールでは, モジュールの内部状態や出力に対するテストを目的とするため, HPCUnit のように, 実行時動作を対象とした計算網羅性や計算漏れのようなテストは直接サポートされない.

HPCUnit は実行時検査の一種と捉えることができる．実行時検査とは, プログラムの実行時情報を取得し, プログラムの振舞を検査する手法である．プログラムの動作を対象としたトレースベースの検査ツールである．tracematch [8] や MOP [9] では, HPCUnit と同様にプログラム中の指定した部分の順序関係や状態を検査する事ができる．しかし, これらのトレースベースの検査ツールでは, 指定したモジュールの部分的な制御フローの検査を目的とするため, 動作ログ全体を対象とするような検査には適していない.

形式的モデル検査ツールである SPIN [10] や, Java のプログラム検査ツール Java Path Finder [11] では, プログラムの動作全体を対象とした計算網羅性や計算漏れを検査で

きる可能性がある．これらの検査ツールを用いれば, 形式的モデルや実プログラムで実装された全ての動作パスを実行し, 計算順序や計算網羅性が満たされるパスが存在するかを検証できる．一方, HPCUnit では, 特定の入力値や状態における動作テストを目的としている．さらに, 多くのモデル検査ツールは莫大な状態数を現実的な時間で検査する事が難しいため, HPC アプリケーションのような多くの入力値や状態を持つプログラムへは容易に適用できない．

6. まとめ

HPC アプリケーションの計算網羅性・計算順序をテストするツール HPCUnit を提案した．HPCUnit は, プログラムのシミュレーション領域を取得し, HPC 特有の計算漏れや計算重複といったバグが含まれていないかテストできるようにする．ユーザは, HPCUnit が提供する API を利用することで, シミュレーション領域の取得やシミュレーション領域の検証・突合を対象プログラムから分離した形で簡潔に記述できる．プログラムから取得するシミュレーション領域は, コントロールフローやクラスの制限を駆使することで, 大きさや形状を任意に決めることができ, テスト内容に合わせ実行モデルを柔軟に変更できる．

今後は, HPCUnit への Domain Specific Language (DSL) の導入, メモリ消費量の抑制を検討したい．DSL を導入する事で, 実装とユーザインターフェイスを切り離す事ができ, 実装の変更に柔軟に対応できる．また, シミュレーション領域の取得やシミュレーション領域の検証・突合をより簡潔に記述可能となる．複数のタプルをまとめたレンジ概念の導入もしたい．複数のタプルを 1 つにまとめることで, インスタンスの生成を減少させ, メモリ消費量の削減が期待できる．

謝辞 本研究を進めるにあたり, 助言をして頂いた東京大学の岩田潤一氏に感謝申し上げる．本研究の開発及び実験には, 東京大学情報基盤センターの富士通 PRIMEHPC FX10 System (Oakleaf-FX) を利用した．

参考文献

[1] Hasegawa, Y., Iwata, J.-I., Tsuji, M., Takahashi, D., Oshiyama, A., Minami, K., Boku, T., Shoji, F., Uno, A., Kurokawa, M., Inoue, H., Miyoshi, I. and Yokokawa, M.: First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 1:1–1:11 (online), DOI: 10.1145/2063384.2063386 (2011).

[2] ichi Iwata, J., Takahashi, D., Oshiyama, A., Boku, T., Shiraishi, K., Okada, S. and Yabana, K.: A massively-parallel electronic-structure calculations based on real-space density functional theory, *J. Comput. Physics*, pp. 2339–2363 (2010).

[3] 横澤拓弥, 高橋大介, 朴泰祐, 佐藤三久 et al.: 行列積を用いた古典 Gram-Schmidt 直交化法の並列化, **情報処理学会論文誌**. **コンピューティングシステム**, Vol. 1, No. 1, pp. 61–72 (2008).

[4] Snir, M., Otto, S. W., Walker, D. W., Dongarra, J. and Huss-Lederman, S.: *MPI: the complete reference*, MIT press (1995).

[5] Nvidia, C.: Programming guide (2008).

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An overview of AspectJ, *ECOOP 2001—Object-Oriented Programming*, Springer, pp. 327–354 (2001).

[7] Hunt, A., Thomas, D. and Programmers, P.: *Pragmatic unit testing in Java with JUnit*, Pragmatic Bookshelf (2004).

[8] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: Adding trace matching with free variables to AspectJ, *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, New York, NY, USA, ACM, pp. 345–364 (online), DOI: 10.1145/1094811.1094839 (2005).

[9] Chen, F. and Roşu, G.: Mop: an efficient and generic runtime verification framework, *ACM SIGPLAN Notices*, Vol. 42, No. 10, ACM, pp. 569–588 (2007).

[10] Holzmann, G. J.: The Model Checker SPIN, *IEEE Trans. Softw. Eng.*, Vol. 23, No. 5, pp. 279–295 (online), DOI: 10.1109/32.588521 (1997).

[11] Havelund, K. and Pressburger, T.: Model checking JAVA programs using JAVA PathFinder, *International Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 366–381 (online), DOI: 10.1007/s100090050043 (2000).

[12] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: Adding trace matching with free variables to AspectJ, *SIGPLAN Not.*, Vol. 40, No. 10, pp. 345–364 (online), DOI: 10.1145/1103845.1094839 (2005).