科学技術計算における最適化に伴う分割の正しさを検査するユニットテストフレームワーク HPCUnit

穂積 俊平[1], 佐藤 芳樹[2], 千葉 滋[3]

[1 3] 東京大学大学院情報理工学系研究科創造情報学専攻
[2] 東京大学情報基盤センター
[1] hozumi@csg.ci.i.u-tokyo.jp

概 要 典型的な科学技術計算のプログラムは, 反復処理の計算空間を分割したり, その順序を変更して実行性能を高めている. だが一方で, 計算空間の分割に伴い, 部分空間の境界を判定するコードが増加し, 計算の抜けや重複といったバグが問題になってくる. そこで本研究では, 実行ログを利用し, 計算空間の分割の正しさを検査できるユニットテストフレームワークを開発した. ログの取得方法は, 専用言語を用いて, 制御フローやクラスを絞って柔軟に指定できる. また, ログを集合として扱うための API を提供しており, ログの検証や比較を集合演算で記述できるようにした. さらに, MPI に特化した集約テストと分散テストを備えており, 実行環境や問題の規模に応じてテスト方法を選択できる. 本フレームワークの有用性を評価するために, グラムシュミットの正規直交化法のプログラムに対して計算の抜けのテストを行った際の, テストコードの行数やオーバーヘッドを計測した.

1 はじめに

物理シミュレーションなどの科学技術計算では, 実行性能が重要視されるため, プログラムには計算順序の変更によるキャッシュヒット率の向上の最適化が施され, 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. 典型的な科学技術計算は, 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空間である計算空間は, 最適化や分散実行により分割される場合がある. 計算空間の分割は, 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計算空間の分割によって発生するバグを検出するための Java 向けユニットテストフレームワーク HPCUnit を提案する. HPCUnit は指定したカーネル計算の実行ログを取ることで実際に計算した空間を取得し, 正しい空間と突合・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. 計算空間の取得や計算空間の検証・突合は, HPCUnit が提供する専用言語や 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. 計算空間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計算空間の大きさや形状を任意に変更できる.

以下では, 2 章で計算空間の分割を引き起こす最適化の例と, それに伴うバグについて述べ, 3 章で 計算空間の分割の正しさを検査するためのユニットテストフレームワーク 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.

図 1. Java で記述した GS 法プログラム

```
1  public class NormalGS {
2    public void calc(double[] vectors, int numVec, int lengthVec) {
3      for (int i = 0; i < numVec; i++) {
4        for (int j = 0; j < i; j++) {
5          double ip = 0;
6          for (int k = 0; k < lengthVec; k++) { // 内積を計算
7            ip += vectors[i * lengthVec + k] * vectors[j * lengthVec + k];
8          }
9          for (int k = 0; k < lengthVec; k++) { // 正射影を減算
10           kernel(vectors, lengthVec, i, j, k);
11         }
12       }
13       normalize(vectors, numVec, lengthVec, i);
14     }
15   }
16
17   /* カーネル計算（中心となる計算）*/
18   private void kernel(int i, int j, int k, double[] vectors, int lengthVec){
19     vectors[i * lengthVec + k] -= ip * vectors[j * lengthVec + k];
20   }
21
22   /* 正規化*/
23   private void normalize(double[] vectors, int numVec, int lengthVec, int target) {..}
24 }
```

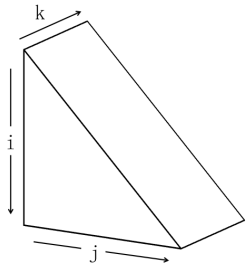


図 2. 最適化前の GS 法の計算空間

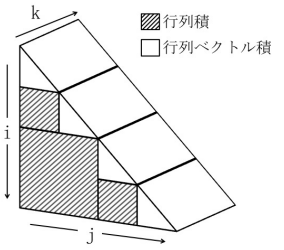


図 3. 最適化後の GS 法の計算空間

2 計算空間の分割に伴うバグ

2.1 計算空間の分割

典型的な科学技術計算プログラムでは実行性能を向上させるために, 計算対象とする空間を分割することが多い. 計算空間の分割は, 計算順序の変更による最適化や, 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.

計算空間の分割を伴う計算順序の変更は, 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, 周辺メモリ空間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物理シミュレーションなどの科学技術計算プログラムは, 配列化した行列やベクトルなどで表現される計算空間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

図 4. 最適化を施した GS 法

```
1  public class OptimizedGSWithKernel {
2    final static int DIVIDED_AREA_SIZE = 4; //分割後の領域のサイズ
3
4    public void calc(double[] vectors, int numVec, int lengthVec) {
5      calcInner(vectors, numVec, lengthVec, 0, numVec - 1, 0, numVec - 1, numVec);
6    }
7
8    private void calcInner(double[] vectors, int numVec, int lengthVec,
9        int iStart, int iEnd, int jStart, int jEnd, int areaSize) {
10     int dividedIEnd, iLength, dividedJEnd, jLength;
11     for (int i = iStart; i <= iEnd; i += areaSize) {
12       dividedIEnd = min(i + areaSize - 1, iEnd);
13       iLength = dividedIEnd - i + 1;
14       for (int j = jStart; j <= jEnd; j += areaSize) {
15         dividedJEnd = min(j + areaSize - 1, jEnd);
16         dividedJEnd = min(dividedJEnd, dividedIEnd - 1);
17         jLength = dividedJEnd - j + 1;
18         if (jLength <= 0)
19           continue;
20         if (i >= dividedJEnd + 1)
21           calcWithDgemm(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd);
22         else if (iLength <= DIVIDED_AREA_SIZE)
23           calcWithDgemv(vectors, numVec, lengthVec, i, dividedIEnd, j);
24         else
25           calcInner(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd,
26               max(areaSize / 2, DIVIDED_AREA_SIZE));
27       }
28     }
29   }
30
31   /* 行列積を用いて計算*/
32   private void calcWithDgemm(double[] vectors, int numVec, int lengthVec,
33                              int iStart, int iEnd, int jStart, int jEnd) {
34     .. BLAS.dgemm(..); ..
35   }
36
37   /* 行列ベクトル積を用いて計算*/
38   private void calcWithDgemv(double[] vectors, int numVec, int lengthVec,
39                              int iStart, int iEnd, int jStart) {
40     .. BLAS.dgemv(..); ..
41   }
42
43   /* 正規化*/
44   private void normalize(double[] vectors, int numVec, int lengthVec, int target) {..}
45 }
```

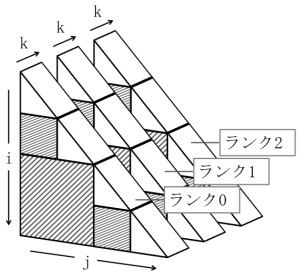
図 5. 行列積、行列ベクトル積の実装

```
1  public class BLAS {
2    public static void dgemm(char transa, char transb, int m, int n, int k,
3                             double alpha, double[] A, int offsetA, int ldA,
4                             double[] B, int offsetB, int ldB,
5                             double beta, double[] C, int offsetC, int ldC) {..
6      for (int i = 0; i < m; i++) {
7        for (int j = 0; j < n; j++) {
8          double sum = 0.0;
9          for (int..2 = 0; j2 < k; j2++) {
10           kernel(..);
11         } ..
12       }
13     }
14   }
15
16   public static void dgemv(char trans, int m, int n,
17                            double alpha, double[] A, int offsetA, int ldA,
18                            double[] x, int offsetx, int incx,
19                            double beta, double[] y, int offsety, int incy) {..
20     for (int i = 0; i < m; i++) {
21       double sum = 0;
22       for (int j = 0; j < n; j++) {
23         kernel(..);
24       } ..
25     }
26   }
27
28   private static void kernel (int i, int j, int k, ..) {..}
29 }
```

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, 行列積や行列ベクトル積に置き換え, 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. 図 1 のように, GS 法は, 行列 vectors を numVec 本の長さ lengthVec のベクトルの組とすると, 4 つの for ループを用いたプログラムとして表現できる. 10 行目のカーネル計算は, 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や行列ベクトル積に置き換えると, プログラムは図 4 のように修正される. また巡回する空間は, 図 3 のように複数の空間に分割されることになる.

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, 計算空間は更に分割される. 例えば, MPI を利用し, 3 ノードで並列に実行される GS 法プログラムの計算空間は図 6 のように分割された空間となる. 分割された計算空間は, 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空間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

図 6. MPI を利用した GS 法の計算空間



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 計算空間の分割に伴う計算の抜けや重複の発生

計算空間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．一般に計算空間の分割は, 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空間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, 意図しない計算空間が指定されるようなバグを引き起こしやすい. 意図しない計算空間の縮小は誤った計算結果を導出し, 逆に計算空間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．

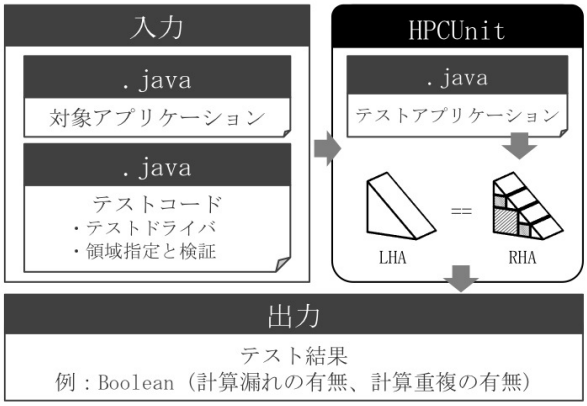
このような計算漏れや計算重複の有無を人為的な突合や機械的な比較で検査する事は難しい．多くの科学技術計算では, 浮動小数点型のデータ演算を多用するため, シミュレーションの計算過程で桁落ちや情報落ちが発生する．そのような計算誤差を考慮して, 最適化前後での計算結果を比較する必要がある．さらに, シミュレーションの初期状態のデータは乱数をベースに生成されることが多いため, 同一のシミュレーションプログラムを用いても同じ計算結果が得られるとは限らない．

3 HPCUnit

本研究では, 科学技術計算における最適化に伴う分割の正しさを検査するユニットテストフレームワーク HPCUnit を提案する．HPCUnit は, 指定したカーネル計算の実行時ログによる実行モデルを検査するような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をテストできる．HPCUnit における実行モデルは, プログラムが計算空間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, プログラムの巡回した計算空間の各要素を重複を許した順序付き集合として取得するための専用言語と, 計算空間の検証や突合を集合演算として記述するためのライブラリを提供する．

我々は, HPCUnit を JUnit を拡張した Java アプリケーション向けのユニットテストフレームワークとして開発し, 従来の単体テストプロセスと並行してテスト実行できるようにした．ユーザの記述するテストコードには, カーネル計算を指定する専用言語によるアノテーションと, 対象アプリケーションを起動するテストドライバ, 計算空間の検証コードが含まれる．テストコードによって起動されるアプリケーションには, ロード時に実行ログを取得するコードが織り込まれ, テスト実行時にログ取得と計算空間の検証が行われる．検証は, 基となる計算空間 (LHA: Left Hand Area) と,

図 7. HPCUnit の全体像



実行した計算空間 (RHA: Right Hand Area) の比較で行われる. (図 7) HPCUnit が提供する専用言語を利用すると, 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. また, RHA には計算空間の各座標だけでなく, MPI のランクなどの実行時情報を含められる.

図 8 に，HPCUnit を使った最適化後の GS 法を対象とした計算漏れと計算重複のテスト例を示す. HPCUnit のテストクラス (GSTest クラス) には, JUnit が提供する@RunWith アノテーションを付加し, その引数として “ HUTestRunner.class ” を与える. GSTest 内のメソッドには，GS 法をテスト実行させるテストドライバを指定する@HUBeforeClass アノテーションと，計算空間を検査するテストメソッドのための @HUTest アノテーションが，それぞれ collectingCalculationArea メソッド，kernelTest メソッドに付加される.

3.1 計算空間の取得

取得する計算空間は, HPCUnit が提供する集合表記と @HUSet アノテーションを利用して宣言的に記述する. 集合表記には内包的記法を採用し, 条件として AspectJ [6] のポイントカットライクなカーネル計算を指定できるように設計した. AspectJ は, Java を 拡張したアスペクト指向言語であり, そのポイントカット記述によってプログラム中の処理を柔軟に指定できる. ポイントカットライクなカーネル計算指定を条件とする内包的記法により, 簡潔に実行ログを選択できるようになる. ただし, HPCUnit は, カーネル計算がメソッドとして定義されている事を前提としており, 指定できる実行ログはメソッド呼び出し引数と, 表 1 に示す予約語を利用した一部の実行コンテキストに限られる. この集合表記を, @HUSet アノテーションの引数として文字列で記述し, テストメソッドの引数に付加すれば, 実際に取得する計算空間が HUSet オブジェクトとして引数で受け渡される. また, 集合表記の条件では, 位置 (クラス, メソッド) やコントロールフロー, レシーバの型を限定する事ができ, 取得するログを細かく指定する事ができる.

例えば, 図 8 のテストコードでは, テストメソッドである kernelTest の各仮引数に, @HUSet アノテーションが付加されており, その引数として各計算空間の取得方法が記述されている. kernelTest メソッドの実行時には, NormalGS クラス内, dgemv メソッドの制御フロー以下, dgemm メソッドの制御フロー以下, それぞれで呼び出された kernel メソッドの実引数値が, LHA, trianglesi, squares に束縛される.

観測位置として記述できるのは, メソッド呼び出しである. 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, 専用言語を使い, カーネル計算であるメソッド呼び出しを指定し, 指定された呼び出しの実引数とコンテキストを観測する. 観測対象とするメソッド呼び出しは, 位置 (クラス, メソッド) によって制限する within キーワード, コントロールフローに

図 8. 最適化後の GS 法の計算空間の正しさを検証するプログラム

```
1  @RunWith("HUTestRunner.class")
2  public class GSTest {
3
4   @HUBeforeClass
5   public static void collectingCalculationArea() {
6     new NormalGS().calc(..);
7     new OptimizedGS().calc(..);
8   }
9
10  @HUTest
11  public void kernelTest(
12   @HUSet("{(i,j,k)|call(void kernel(int i,int j,int k, ..)) && within(NormalGS)}")
13    HUSet<HUTuple3<Integer, Integer, Integer>> LHA,
14   @HUSet("{(i,j,k)|call(void kernel(int i,int j,int k, ..)) && cflow(call(dgemv(..))}")
15    HUSet<HUTuple3<Integer, Integer, Integer>> triangles,
16   @HUSet("{(i,j,k)|call(void kernel(int i,int j,int k, ..)) && cflow(call(dgemm(..))}")
17    HUSet<HUTuple3<Integer, Integer, Integer>> squares,
18  ) {
19    HUSet<HUTuple3<Integer,Integer,Integer>> nullArea = HUSet.getNull();
20    assertThat(LHA , is(triangles.union(squares))); // 計算漏れ
21    assertThat(nullArea, is(triangles.intersection(squares))); // 計算重複
22  }
23 }
```

表 1. メソッド呼び出しのコンテキストを取得する予約語

予約語	意味
count	メソッド呼び出しを観測した回数
time	メソッド呼び出しを観測した時間
threadID	メソッド呼び出しが行われたスレッドの ID
mpiRank	メソッド呼び出しが行われたノード番号
mpiSize	メソッド呼び出しが行われた際のノード数

よって制限する cflow キーワード, メソッド呼び出しのレシーバの型によって制限する receiver キーワードを使い限定できる. 例えば, 図 8 の kernelTest メソッドの LHA 仮引数では, within キーワードを利用して NormalGS クラスにおける kernel メソッド呼び出しのみに限定している.

計算空間の形状を定義する際には, 観測位置として指定されたメソッド呼び出しの仮引数, もしくはメソッド呼び出しのコンテキストを取得する予約語を利用できる. 例えば, 図 8 の LHA 仮引数においては, kernel メソッドの仮引数である i, j, k を利用している. コンテキストを取得する予約語は, 表 1 の通りである. 科学技術計算において頻繁に利用される MPI をサポートした予約語も定義している. 例えば, MPI のランク情報を含めるには以下のように記述すればよい.

```
@HUSet(‘‘{(mpiRank, i, j, k) | call(void kernel(int i, int j, int k, ..))}’’)
```

3.2 取得した計算空間の検査や突合

計算空間の検証・突合は, 表 2 に示した HUSet 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 OptimizedGS クラスの中で, 行列積計算を利用する計算空間 (squares) と, 行列ベクトル積を利用する空間 (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, 計算漏れのテストは 20 行目のように, union メソッドを利用し,

表 2. HUSet API (集合演算, 集合等価性)

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

表 3. HUSet API (計算空間生成)

HUSet⟨T⟩ API	意味
static ⟨U⟩ HPCList⟨U⟩ getNull ()	空集合生成を生成
static HUSet⟨HUTuple1⟨Integer⟩⟩ getLine (int L1)	長さ L1 の線分を生成
static HUSet⟨HUTuple2⟨Integer, Integer⟩⟩ getSquare (int L1)	一辺の長さが L1 の正方形を生成
static HUSet⟨HUTuple2⟨Integer, Integer⟩⟩ getRight-Triangle (int L1, int L2)	隣辺の長さが L1, L2 の直角三角形を生成
static HUSet⟨HUTuple3⟨Integer, Integer, Integer⟩⟩ getCube (int L1)	一辺の長さが L1 の立方体を生成
static HUSet⟨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

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, 表 3 のように, LHA として任意の計算空間を与えられるよう, 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, 図 8 のように, 最適化を施していない GS 法の計算空間を 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, HUSet 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計算空間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計算空間を復元する必要が出てくる. map メソッドを利用すれば, 各空間のオフセット値を計算し, その空間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, 計算空間は図 6 のように, 変数 k の方向に 3 分割され, 分割された空間の奥行きは 30 となる. 分割された空間から全体の空間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 法から取得した計算空間を表す集合を 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の検査を記述するために, HUSet は fold メソッドを提供する. fold メソッドを用いると, 計算空間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計算空間 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.

図 9. map メソッド：相対アドレス化された計算空間の絶対アドレス化

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

図 10. fold メソッド：減少列のテスト

```
// 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
gs.fold(HUFolder.decreasingOrder, origin);
```

3.3 効率的なテスト実行のためのサポート

JUnit の拡張として実装した HPCUnit のテストコードには, 計算抜けや重複の検査だけでなく, 通常の機能テストも余分なオーバーヘッド無しで共存させる事ができる. HPCUnit は, 専用のクラスローダを用いて, JUnit とは別にテスト対象アプリケーションをロードし, プログラム変換によってログ取得コードを埋め込む. ロード時のプログラム変換は, @HU で始まるアノテーションの付加されたテストメソッドの呼び出しの際に行われる. したがって, 通常の JUnit での機能テスト時には, アプリケーションのロードは JUnit 標準のクラスローダへ委譲され, ログ取得コードの埋め込まれないプログラムをテストできるようになる.

HPCUnit では, テスト実行を効率化するために, MPI アプリケーションに特化した分散/集約テスト実行や, テストの仮実行をサポートしている. 多くの場合, 大規模な計算空間の比較や突合は分散ノード上で別々に実行する方がスケーラビリティの観点で優れている. 一方, 疎行列処理のように, 各計算ノードに分割される計算空間が非均等であったり, 動的に決定される場合, データが小規模であればランク 0 の計算ノードに集約して比較や突合を行うテストコードの方がシンプルに記述できる. そのため, HPCUnit は, テストメソッドに付加する @HUTest アノテーションの替わりに, 分散テスト実行のための @HUDistributedTest アノテーションと, 集約テスト実行のための @HUGatheredTest アノテーションをサポートしている. また, @HUSet アノテーションには, テストを仮実行させる @HUSkip アノテーションを付加する事もできる. @HUSkip アノテーションによる仮実行時のテストメソッド呼び出し中では, カーネル計算が抑制され, 計算空間の取得とその検査のみが実行されるようになる.

4 実験

HPCUnit の有用性を確かめるために, GS 法に対して HPCUnit を用いていくつかのシナリオを基にテストを行った際のオーバーヘッド, メモリ使用量, コード行数を計測した. シナリオは, GS 法の出力行列への書き込みを観測し, 計算の網羅性を確かめるテスト (a), カーネル計算において計算漏れが発生していないことを確かめるテスト (b) , GS 法の入力行列の読み込みを観測し, 局所性を評価するテスト (c) の 3 つである. 実験は以下の環境で行った.

- FUJITSU Supercomputer PRIMEHPC FX10 1 ノード

表 4. HPCUnit による実行ログ取得のオーバーヘッド

	実行時間（秒）	メモリ使用量 (GB)	取得した計算空間の大きさ
元の GS 法プログラム	23.8 ± 0.16	4.7	なし
(a) 行列書き込み観測	32.1 ± 0.29	5.1	L * M
(b) カーネル計算観測	35.7 ± 0.23	8.8	L * (L - 1) / 2 * M
(c) 行列読み込み観測	53.3 ± 0.71	15.5	L * (L - 1) * 2 * M

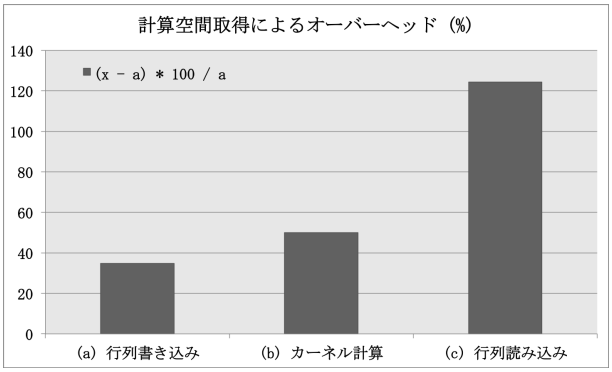


図 11. 計算空間取得によるオーバーヘッド

- 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 -Xmx28000m -Xms28000m -Xmn24000m -XX:SurvivorRatio=10

4.1 HPCUnit によるオーバーヘッドとメモリ使用量

実行ログ取得コードのオーバーヘッドとメモリ使用量の増加を測定した. 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メモリ使用量, 取得した計算空間の大きさを示す. また, (a), (b), (c) の実行時間に含まれるオーバーヘッドの, 元の GS 法プログラムの実行時間に対する比率を図 11 に示す. GS 法の入力行列 (ベクトル数 L × ベクトルの長さ M) は, L を 128 , M を 4096 とした.

図 11 を見ると, (c) では, 元プログラムの 124% という大きなオーバーヘッドがかかっている. この原因は, 取得する計算空間が大きい事が考えられる. GS 法のオーダーは, 入力となる行列のベクトル数が L, ベクトルの長さが M であるとき, $O(L \times L \times M)$ であり, (c) で取得した計算空間は $L \times (L-1) \times 2 \times M$ 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くなってしまう. 一方, (a) では, オーバーヘッドは 34.9% に抑えられている. (a) 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, (a) で取得した計算空間の大きさは $L \times M$ である. GS 法全体のオーダー $O(L \times L \times M)$ に比べ, (a) 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. 一方, メモリ使用量では, 取得する計算空間が大きい (c) で, 多くのメモリを使用している事が分かる.

表 5. 専用言語で記述されたコードと, そこから生成された AspectJ のコード行数

	専用言語 (行)	AspectJ(行)
(a) 行列書き込み観測	4	52
(b) カーネル計算観測	4	52
(c) 行列読み込み観測	4	52

次に, HPCUnit でのテストコード開発の生産性を評価するために, プログラム変換前後でのコード行数を測定した. HPCUnit では, 専用言語を AspectJ のソースコードにコード変換することで, 計算空間取得コードの埋め込みを行っている. そこで, 専用言語で記述したコードと, そこから生成された AspectJ のコードの行数と測定した. 測定結果を表 5 に示す. なお生成された AspectJ のソースコードは, Eclipse の標準フォーマットでフォーマットした上で行数をカウントした. 結果を見ると, 専用言語のコード行数が少なくなっていることがわかる. 生成された AspectJ のソースコードには, Aspect の宣言文や, 計算空間を保存するためのフィールド, コンテキスト取得のためのメソッドなどボイラープレートコードが含まれている. そのため, 専用言語で記述した計算空間取得コードに比べると, AspectJ を利用した場合でさえ, テスト開発者の実装するコード行数が大幅に増大する事が分かる.

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.

我々の先行研究 [12] では, 計算空間を取得するために Java API のみを提供しており, また JUnit との統合も実装されていなかった. 本研究では, 専用言語による計算空間の指定を可能とするため

の, 言語設計及び処理系の開発を行い, さらに JUnit の拡張テストランナー及び拡張クラスローダを実装した.

6 まとめ

科学技術計算における最適化に伴う分割の正しさを検査する ユニットテストフレームワーク HPCUnit を提案した. HPCUnit は, プログラムの計算空間を取得し, 科学技術計算で起こりやすい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 が提供する専用言語や API を利用することで, 計算空間の取得や検証・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計算空間は, 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.

今後はメモリ消費量を抑制する方法を考えたい. 複数のタプルをまとめたレンジ概念を導入することで対応できると考えている. レンジを導入することで, 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.

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 (2011).

[2] 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, *SIGPLAN Not.*, Vol. 40, No. 10, pp. 345–364 (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 (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 (2000).

[12] 穂積俊平, 佐藤芳樹 and 千葉滋: 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, SWoPP (2013).