東京大学
情報理工学系研究科 創造情報学専攻
修士論文

分割された計算空間の正しさを実行ログを用いて検査するユニットテストフレームワークの提案

A Unit Testing Framework for Verifying Divided Calculation Area by Using Runtime Logs

穂積 俊平
Shumpei Hozumi

指導教員　千葉 滋 教授

2014年1月

概要

本研究では、分割された計算空間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。多くの科学技術計算のプログラムは、計算を分割することで実行性能を高めている。だが一方で、計算の分割によって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となってくる。そこで本研究では、テスト対象空間で記録した実行ログから実行時情報を抽出し、その検証や比較を行うことで、計算の抜けや重複がないか検査するフレームワークを開発した。取得するログの範囲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 も提供しており、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。HPCUnit 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や重複のテストを行った場合の、テストコードの行数やオーバーヘッドを計測した。

Abstract

We propose HPCUnit, which is a unit test framework for verifying divided calculation area. In scientific computing, programs are optimized by dividing calculation area. However, dividing calculation area increases bounds-checking code and causes bugs such as calculation leak and calculation duplication. To address this problem, we propose a new unit test framework that detects bugs by using runtime logs. With this system, programmers can get runtime logs flexibly by selecting class and control flow. And programmers can verify runtime logs easily by using set operations. HPCUnit has special test methods for MPI applications such as distributed testing and gathered testing. Therefore, programmers can select a test method according to test environment and scale of a problem.

目次

第 1 章　序論 1
1.1　背景と問題点 1
1.2　提案 1
1.3　本論文の構成 2

第 2 章　背景と関連研究 3
2.1　科学技術計算における最適化とそれに伴う計算空間の分割 3
2.2　計算空間の分割に伴う計算漏れや計算重複 6
2.3　関連研究 7

第 3 章　HPCUnit 14
3.1　HPCUnit の全体像 14
3.2　計算空間の取得 15
3.3　計算空間の検証 21
3.4　効率的なテスト実行のためのサポート 25

第 4 章　実装 28
4.1　パーサコンビネータライブラリを活用した構文解析 28
4.2　AspectJ ソースコードの生成 28

第 5 章　実験 31
5.1　HPCUnit によるオーバーヘッドとメモリ使用量 31
5.2　専用言語の生産性 32
5.3　@HUSkip の効果 33

第 6 章　まとめと今後の課題 35
6.1　まとめ 35
6.2　今後の課題 35

発表文献と研究活動 37

参考文献 38

付録 A ソースコード 43

第 1 章

序論

1.1 背景と問題点

物理シミュレーションなどの科学技術計算では, 取り扱う問題が大きいため, プログラムの実行性能が重要視される. そのため, 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. また, 近年では GPU などのアクセラレータに一部の計算を委譲することもある. これらの最適化や並列分散実行, 計算のオフロードは, 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, 計算のオフロードによって分割される場合がある. 計算空間の分割は, カーネル計算の分割やプログラム中の境界判定コードの増加を引き起こし, プログラムの保守性や可読性の低下を招く. 結果として, 誤った計算空間の指定を誘発し, 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.

1.2 提案

本論文では, 計算空間の分割によって発生する計算漏れや計算重複といったバグを検出するためのユニットテストフレームワーク HPCUnit を提案する. HPCUnit は対象プログラムの実行ログとして実際に計算した空間を取得し, 正しい空間と突合することで, 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. 専用言語を用いることで, カーネル計算の指定を制御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. また, 集合演算 API を利用することで, 取得した計算空間をテストしやすい形に加工することができる. さらに, HPCUnit は, MPI [1] を利用したアプリケーションに特化した分散/集約テストを提供しており, 問題や実行環境に応じて, テスト方法を変更することができる. 我々は, HPCUnit の有用性を示すために, 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, 複数のテストケースを想定してテストを行い, それぞれのケースにおけるテストコード行数やオーバーヘッドを測定した.

1.3　本論文の構成

本論文の構成は以下の通りである. 第 2 章では, 典型的な科学技術計算における最適化とそれに伴うバグについて述べ, それらのバグを検出するのに有効なツールが存在しないことを指摘する. 第 3 章では, 計算空間の分割に伴い発生したバグを検出するためのユニットテストフレームワーク HPCUnit を提案する. 第 4 章では, HPCUnit の実装方法について説明する. その後, 第 5 章で HPCUnit の有用性を確認するために行った実験の結果と考察を示す. そして最後に, 第 6 章で まとめと今後の課題を述べる.

第 2 章

背景と関連研究

2.1 科学技術計算における最適化とそれに伴う計算空間の分割

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, 以下のような要素がプログラムに含まれることにより必要となる.

- 計算順序の変更による最適化
- MPI などを利用したデータ分散
- GPGPU [2] などの計算のオフロード

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 [3, 4] 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) に対し, 計算順序の変更による最適化を施している. 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, 配列 vectors を与えるベクトルの組, numVec をベクトルの本数, lengthVec をベクトルの長さとし Java [5] で記述すると, 図 2.1 の calc メソッドのように 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, 12 行目のカーネル計算が, 図 2.1 中の 3 つの変数 i, j, k が作り出す図 2.2 のような三角柱の形をした計算空間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 法のプログラムは, 計算順序の変更により計算の一部を行列積もしくは行列ベクトル積に置き換え, 局所性を向上させられることが知られており（[6]）, RSDFT の GS 法でもこの方

図 2.1. 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          // 内積を計算
7          for (int k = 0; k < lengthVec; k++) {
8            ip += vectors[i * lengthVec + k] * vectors[j * lengthVec + k];
9          }
10         // 正射影を減算
11         for (int k = 0; k < lengthVec; k++) {
12           vectors[i * lengthVec + k] -= ip * vectors[j * lengthVec + k];
13         }
14       }
15       normalize(vectors, numVec, lengthVec, i);
16     }
17   }
18
19   // 正規化
20   private void normalize(double[] vectors, int numVec, int lengthVec, int target)
         {
21     double size = 0;
22     for (int k = 0; k < lengthVec; k++) {
23       size += vectors[target * lengthVec + k] * vectors[target * lengthVec + k];
24     }
25     size = 1 / Math.sqrt(size);
26     for (int k = 0; k < lengthVec; k++) {
27       vectors[target * lengthVec + k] *= size;
28     }
29   }
30 }
```

法を採用している. 図 2.1 の calc メソッドは, 変数 j を利用した for ループを 2 つの二重 for ループに分割するように計算順序を変更することで, 図 2.4 の calc メソッドのように書き換えることができる. この 2 つの二重 for ループは行列ベクトル積に置き換えることができ, 図 2.5 のようにできる. 更に計算順序を変更すると, 一部を行列積に置き換えることができる. 行列積の方が行列ベクトル積を用いるよりも局所性を高められるため, RSDFT では再帰を用いて, なるべく多くの計算空間を行列積が担当するようにしている. 最終的には, GS 法のプログラムは図 2.6 のようになる. 行列積と行列ベクトル積を利用するようにプログラムを改変すると, 巡回する計算空間は, 図 2.3 のように複数の部分空間に分割されることになる.

さらに, 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 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, 計算空間はより細かく分割される. 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 2.7

図 2.2. 最適化前の GS 法の計算空間

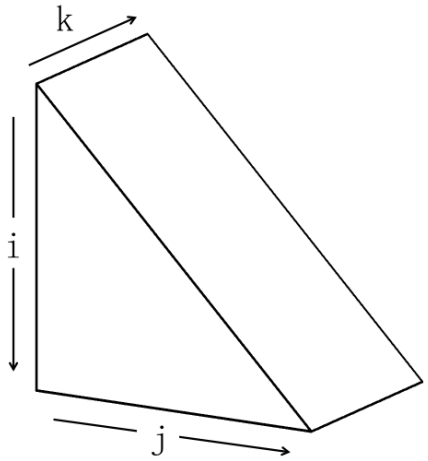
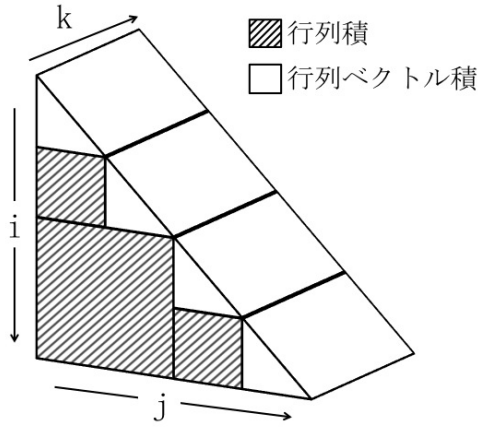


図 2.3. 行列積と行列ベクトル積を利用した GS 法の計算空間



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. GPU 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 [7] 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,

図 2.4. 計算順序を変更した GS 法プログラム

```
1  public class OrderChangedGS {
2    public void calc(double[] vectors, int numVec, int lengthVec) {
3      for (int i = 0; i < numVec; i++) {
4        double[] ip = new double[i];
5        // 内積を計算
6        for (int j = 0; j < i; j++) {
7          for (int k = 0; k < lengthVec; k++) {
8            ip[j] += vectors[i * lengthVec + k] * vectors[j * lengthVec + k];
9          }
10       }
11       // 正射影を減算
12       for (int j = 0; j < i; j++) {
13         for (int k = 0; k < lengthVec; k++) {
14           vectors[i * lengthVec + k] -= ip[j] * vectors[j * lengthVec + k];
15         }
16       }
17       normalize(vectors, numVec, lengthVec, i);
18     }
19   }
20
21   // 正規化
22   private void normalize(double[] vectors, int numVec, int lengthVec, int target)
         {
23     ..
24   }
25 }
```

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漏れや計算重複

計算空間の分割されたプログラムは, 本来行われるべき計算が抜け落ちてしまったり（計算漏れ）, 同じ計算を複数回行ってしまう（計算重複）ようなバグを引き起こしやすい. 一般的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. 例えば, 最適化以前の GS 法は, 図 2.1 の 12 行目がカーネル計算であったが, 最適化後の GS 法のカーネル計算は, BLAS.gemv メソッドと BLAS.gemm メソッド中に分割されている. カーネル計算が分割されると, 計算空間を捉えるためには, それぞれのカーネル計算が巡回した空間を考える必要性が出てくる. また, 計算空間が分割されたプログラムには, 分割による端数を処理するためのコードや部分空間の境界判定を行うコードが含まれる. そのようなコードの増加によって, 分岐や終了判定が複雑になり可読性や保守性が著しく低下する. 例えば, 最適化後の GS 法のプログラム（図 2.6）を見ると, 部分空間の大きさを端数を考慮しながら決定するコード（12, 13, 15, 16, 17 行目）や,

図 2.5. 行列ベクトル積を利用した GS 法プログラム

```
1  public class GemvGS {
2    public void calc(double[] vectors, int numVec, int lengthVec) {
3      for (int i = 0; i < numVec; i++) {
4        double[] ip = new double[i];
5
6        // 内積を計算
7        BLAS.dgemv('N', i, lengthVec, 1, vectors, 0, lengthVec, vectors, i
8            * lengthVec, 1, 0, ip, 0, 1);
9
10       // 正射影を減算
11       BLAS.dgemv('T', lengthVec, i, -1, vectors, 0, lengthVec, ip, 0, 1,
12           1, vectors, i * lengthVec, 1);
13
14       normalize(vectors, numVec, lengthVec, i);
15     }
16   }
17
18   // 正規化
19   private void normalize(double[] vectors, int numVec, int lengthVec, int target)
         {
20     ..
21   }
22 }
```

部分空間の境界を判定し, 行列積で計算するのか, 行列ベクトル積で計算するのか決定するコード（18 ~ 26 行目）が含まれおり, 最適化以前のプログラム（図 2.1）に比べると可読性が大きく低下していることがわかる. 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.

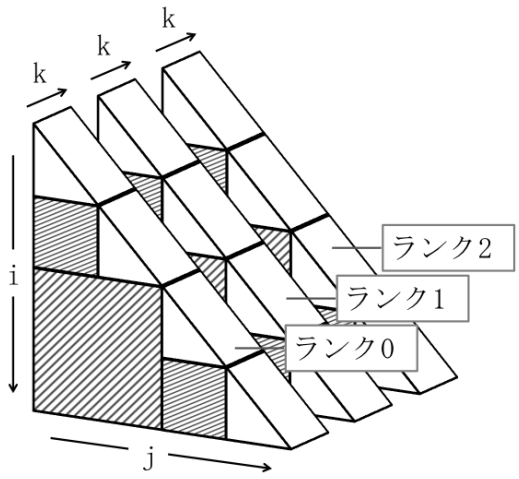
2.3 関連研究

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. ここでは, 一般的なユニットテスト手法に加え, 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

図 2.6. 行列積と行列ベクトル積を利用するようにした GS 法プログラム

```
public class OptimizedGS {
  final static int DIVIDED_AREA_SIZE = 4; //分割後の領域のサイズ

  public void calc(double[] vectors, int numVec, int lengthVec) {
    calcInner(vectors, numVec, lengthVec, 0, numVec - 1, 0, numVec - 1, numVec);
  }

  private void calcInner(double[] vectors, int numVec, int lengthVec,
      int iStart, int iEnd, int jStart, int jEnd, int areaSize) {
    int dividedIEnd, iLength, dividedJEnd, jLength;
    for (int i = iStart; i <= iEnd; i += areaSize) {
      dividedIEnd = min(i + areaSize - 1, iEnd);
      iLength = dividedIEnd - i + 1;
      for (int j = jStart; j <= jEnd; j += areaSize) {
        dividedJEnd = min(j + areaSize - 1, jEnd);
        dividedJEnd = min(dividedJEnd, dividedIEnd - 1);
        jLength = dividedJEnd - j + 1;
        if (jLength <= 0)
          continue;
        if (i >= dividedJEnd + 1)
          calcWithDgemm(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd
              );
        else if (iLength <= DIVIDED_AREA_SIZE)
          calcWithDgemv(vectors, numVec, lengthVec, i, dividedIEnd, j);
        else
          calcInner(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd,
              max(areaSize / 2, DIVIDED_AREA_SIZE));
      }
    }
  }

  // 行列積を用いて計算
  private void calcWithDgemm(double[] vectors, int numVec, int lengthVec,
                             int iStart, int iEnd, int jStart, int jEnd) {..}

  // 行列ベクトル積を用いて計算
  private void calcWithDgemv(double[] vectors, int numVec, int lengthVec,
                             int iStart, int iEnd, int jStart) {..}

  // 正規化
  private void normalize(double[] vectors, int numVec, int lengthVec, int target)
      {..}
}
```

図 2.7. MPI を利用した GS 法の計算空間



質を検査するモデル検査について述べる. そして, それら既存のアプローチでは, 計算漏れや計算重複の検出が難しいことを説明する. さらに, 実行時情報を取得するために有効なアスペクト指向について説明する.

2.3.1 ユニットテスト

ユニットテストとは, 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. ユニットテストを自動化するため, JUnit [8] や TestNG [9], pFUnit [10] などのフレームワークが開発されている. ここでは, 代表的なユニットテストフレームワークである JUnit と 科学技術計算向けの機能を有する pFUnit について述べる.

JUnit

JUnit は, Java 向けに開発されたユニットテストを自動化するためのフレームワークである. JUnit を利用すると, 一度定義したユニットテストを簡単に再実行することができ, 回帰テストを自動化できる. JUnit には複数のバージョンが存在するが, ここでは JUnit 4 について説明を行う.

JUnit では, テスト専用のクラス（テストクラス）を作成し, そのメソッドにテスト内容を記述することでテストを行う. 各々のメソッドには, @Test アノテーションを付加する必要がある. 実際に JUnit を用いたテストコード例を図 2.8 に示す. 図 2.8 中の StringUtilTest クラスは, StringUtil クラスをテストするために定義されたテストクラスである. StringUtilTest クラスには, StringUtil クラスの isBlank メソッドをテストするための, 2 つのテストメソッド blank と notblank が定義されており, それぞれのメソッドに@Test アノテーションが付加されている.

JUnit のテストを実行する際には, org.junit.runner.JUnitCore を実行クラスとし, その引数と

図 2.8. JUnit を用いたテストコード例

```
@RunWith(ExampleRunner.class)
public class StringUtil {
  public static boolean isBlank(String str1) {
    return str1 == null || str1.length() == 0;
  }
}

public class StringUtilTest {

  @Test
  public void blank() {
    assertThat(StringUtil.isBlank(""), is(true));
  }

  @Test
  public void notblank() {
    assertThat(StringUtil.isBlank("not blank!"), is(false));
  }
}
```

してテストクラス名を引き渡せばよい. 例えば, StringUtilTest のテストを実行する場合には以下のようなコマンドを実行すればよい. 以下のコマンドを実行すると, blank と notblank 両方のメソッドが実行される.

java -cp $CLASSPATH org.junit.runner.JUnitCore StringUtilTest

JUnit には, @Test 以外にも, 以下の様なユニットテストを助けるためのアノテーションが存在する. これらを活用することで, テストの実行をさらに簡易化できる.

@Ignore	テスト除外指定
@Before	テストメソッドごとに事前実行
@After	テストメソッドごとに事後実行
@BeforeClass	テストを通じて、一回だけ事前実行
@AfterClass	テストを通じて、一回だけ事後実行
@Rule	テスト時の一時的なルールの設定
@ClassRule	複数のテストを通じてのルール設定

JUnit は拡張性を考慮して設計されており, 独自のテストランナーを作成することもできる. 独自のテストランナーを作成することで, 特定の領域に特化したテストの記述を簡潔化できる. 作成したテストランナーを利用するときには, @RunWith アノテーションを利用する. 図 2.8 では, @RunWith アノテーションを用いて, テストランナーとして ExampleRunner クラスを

指定している.

JUnit のようなユニットテストフレームワーク利用すれば, ユニットテストを自動化し, 簡単に回帰テストを行うことができる. しかし, 一般的なユニットテストフレームワークでは, 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, 実行時の情報を利用しなければ検出できない計算漏れや計算重複のテストは直接サポートされていない.

pFUnit

pFUnit は, NASA が開発している Fortran 向けのユニットテストフレームワークである. Fortran は, 科学技術計算の分野で広く利用されている言語であり, Fortran で記述されるプログラムの多くは, MPI を利用した並列分散プログラムである. そのため, pFUnit は MPI の使用を想定したテストケースを提供している. このテストケースを利用することで, 複数プロセスで動作するテストを行う事ができる. しかし, pFUnit も JUnit と同様に, 実行時情報の利用は想定しておらず, 計算漏れや計算重複の検出は難しい.

2.3.2 実行時検査

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 [11, 12] や MOP [13, 14] では, 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, 計算漏れのテストのように, 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.

2.3.3 モデル検査

形式的モデル検査ツールである SPIN [15] や, Java のプログラム検査ツール Java Path Finder [16, 17] 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順序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. だが, 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, 科学技術計算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.

2.3.4 アスペクト指向

アスペクト指向プログラミング [18, 19] は, オブジェクト指向では分離しにくい特徴（横断的関心事）を分離して記述することを目的としたプログラミング手法である. オブジェクト指向プログラミングでは, 関心事をオブジェクトやクラスといった単位にまとめることで, プログラムのモジュール性を高めてきた. しかし, ロギング処理やエラー処理など, オブジェクト指向では綺麗にまとめることが難しい関心事が存在する. アスペクト指向を用いれば, それらの関

心事を分離して記述することができる.

アスペクト指向は, 計算漏れや計算重複といったバグの検出に役立つ. 計算漏れや計算重複を検査するためには, プログラムが巡回した計算空間を取得する必要がある. 計算空間を取得するためには, プログラム中にコードを埋め込む必要があるが, アスペクト指向を用いれば, このコードをテスト対象アプリケーションから分離して管理できる. アスペクト指向をサポートした言語は AspectJ [20] や GluonJ [21] などが開発されている. ここでは, AspectJ について述べる.

AspectJ

AspectJ は Java を拡張したアスペクト指向言語である. AspectJ を用いると, プログラムのある実行点の振る舞いを変更や拡張することができる. 振る舞いを変更可能な実行点のことをジョインポイントと呼ぶ. ジョインポイント には, メソッド呼び出し, メソッド実行, フィールドアクセスなどが存在する. 複数存在するジョインポイントから振る舞いを変更するジョインポイントを選択する方法をポイントカットと呼ぶ. AspectJ におけるポイントカットは, 以下のように, ポイントカット指定子と呼ばれる論理式によって表される.

```
call (void kernel(int, int, int)) && within(NormalGS)
```

上の例では, NormalGS クラス中に記述された void kernel(int, int, int) メソッドの呼び出しを選択している. AspectJ が持つ代表的なポイントカット指定子は以下の通り.

call	パターンにマッチするメソッドの呼び出しを選択
execution	パターンにマッチするメソッドの実行を選択
get	パターンにマッチするフィールドの参照を選択
set	パターンにマッチするフィールドへの代入を選択
within	パターンにマッチするクラス中に存在するジョインポイントを選択
withincode	パターンにマッチするメソッド中に存在するジョインポイントを選択
this	ジョインポイントにおいて, this が指定された型であるものを選択
target	ジョインポイントにおいて, target が指定された型であるものを選択
args	ジョインポイントにおいて, 引数が指定された型であるものを選択

ポイントカットによって選択されたジョインポイントの集合において, その振る舞いを変更するためのコードをアドバイスと呼ぶ. アドバイスには以下の 3 種類が存在する.

before	指定したジョインポイントの実行前にコードを実行する.
after	指定したジョインポイントの実行後にコードを実行する.
around	指定したジョインポイントを, 実行したいコードで置き換える.

例えば, void kernel(int, int, int) メソッドの呼び出しの前にその引数をロギングするコードは図 2.9 のように記述できる. このように, AspectJ を使えば, テスト対象プログラムから分離した形で, プログラムの実行時情報を取得/ロギングすることができる. これを応用すれば, 計算漏

図 2.9. kernel メソッドにおけるロギング

```
public aspect LoggingKernel {
  before(int i, int j, int k) :
    call(void kernel(int, int, int)) && args(i, j, k) {
      System.out.println(i + "," + j + "," + k);
  }
}
```

れや計算重複を検出することができる. だが, 一方で以下のような問題点もある.

1. 通常のテストとの共存が難しい.
2. 計算漏れや計算重複の検出に適した API を提供していない.

計算漏れや計算重複を検出するためには, プログラムに対しコードを埋め込み, プログラムの計算空間を取得する必要がある. だが場合によっては, 埋め込まれたコードがプログラムの挙動を変え, 通常のテストがうまくいかなくなる恐れがある. 通常のテストと計算漏れや計算重複のテストを共存させるためには, 計算漏れや計算重複のテストを行うときのみ, 計算空間取得コードが有効となることが望ましい. また, AspectJ は汎用的な言語であるため, 計算空間の取得に特化した API を提供しているとは言えない. 計算空間の取得のみに関心事を限定すれば, より良い API が構築できると考えられる.

第 3 章

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ログを元にした実行モデルを検査するようなユニットテストを可能にし, 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, プログラムが計算空間のどの部分を巡回したかを表現し, 実行ログから作られる. HPCUnit は, 実行ログの取得と検証のために，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.

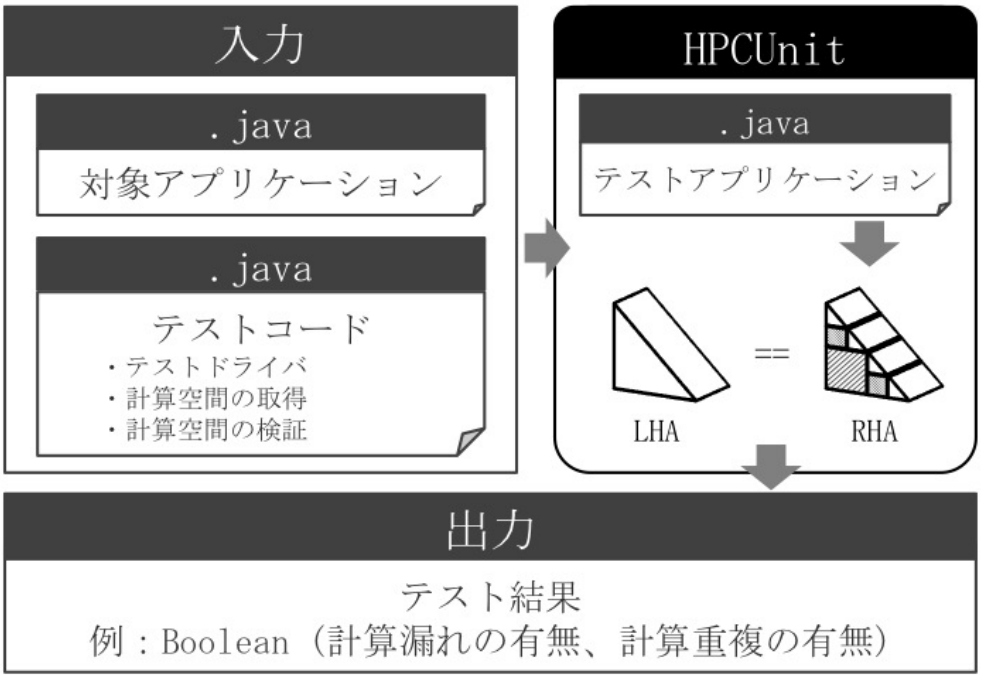
本章では, HPCUnit の全体像を示し, その後, HPCUnit における計算空間の取得方法と検証方法について述べる.

3.1 HPCUnit の全体像

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，従来のユニット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，計算カーネルを指定する専用言語によるアノテーション（計算空間の取得）と， テスト対象アプリケーションを起動するテストドライバ, 取得した計算空間の検証が含まれる．テストドライバによって起動されるアプリケーションには, クラス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, 正しい計算空間 (LHA: Left HandArea) と, 検証したい計算空間 (RHA: Right Hand Area) の比較で行われる.（図 3.1）

例として, 最適化前の GS 法プログラム（リスト 3.2）の計算空間を取得し, 検証することを考える. 最適化前の GS 法プログラム（図 3.2）は, 単純に 4 つの for 文を使って実装されており, カーネル計算に相当する部分が kernel メソッドとして切り出されている. この kernel メソッドが巡回した計算空間を取得し, LHA と比較するコードは図 3.3 のように記述できる. 図 3.3 の GSNormalTest クラスのように, HPCUnit を用いたテストを行う場合には, テストクラスに JUnit が提供する RunWith アノテーションを付加し, 引数として, “HUTestRunner.class” を指定

図 3.1. HPCUnit の全体像



する必要がある. GSNormalTest クラスにおけるテストドライバは, collectingCalculationArea メソッドであり, テストドライバであることを示す @HUBeforeClass アノテーションが付加されている. @HUBeforeClass アノテーションが付加される事で, テストドライバはテストメソッドよりも事前に実行される. collectingCalculationArea メソッドでは, テスト対象アプリケーションである GSNormal クラスのインスタンスを生成し, calc メソッドを呼び出している. 一方, kernelTest メソッドが計算空間の取得と検証を記述したテストメソッドである. HPCUnit のテストメソッドには, HPCUnit のテストメソッドである事を示すアノテーションを付加する必要があり, kernelTest メソッドには, @HUTest アノテーションが付加されている. kernelTest メソッドの仮引数部分が計算空間の取得, メソッドボディが計算空間の検証に相当する. 仮引数の calculationArea には, 計算空間の取得方法が引数として引き渡された @HUSet アノテーションが付加されており, kernelTest メソッド実行時に, 取得した計算空間が calculationArea に束縛される. メソッドボディでは, JUnit が提供する assertThat メソッドを使い, LHA と 取得した計算空間である calculationArea が等しいか検証している.

以降では, 計算空間の取得方法と取得した計算空間の検証方法についてより詳しく述べる.

3.2 計算空間の取得

計算空間の取得は, HPCUnit が提供する専用言語と @HUSet アノテーションを利用して宣言的に記述する. 専用言語には内包的記法を採用し，条件に AspectJ のポイントカットライク

図 3.2. 最適化前の GS 法プログラム

```
1  public class NormalGSWithKernel {
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          // 内積を計算
7          for (int k = 0; k < lengthVec; k++) {
8            ip += vectors[i * lengthVec + k] * vectors[j * lengthVec + k];
9          }
10         // 正射影を減算
11         for (int k = 0; k < lengthVec; k++) {
12           kernel(i, j, k, vectors, numVec, lengthVec, ip);
13         }
14       }
15       normalize(vectors, numVec, lengthVec, i);
16     }
17   }
18
19   // カーネル計算
20   private void kernel(int i, int j, int k, double[] vectors,
21                       int numVec, int lengthVec, double ip){
22     vectors[i * lengthVec + k] -= ip * vectors[j * lengthVec + k];
23   }
24
25   // 正規化
26   private void normalize(double[] vectors, int numVec, int lengthVec, int target)
         {
27     ..
28   }
29 }
```

なカーネル計算指定を記述できるように設計した．指定されるカーネル計算は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を呼び出した際の実引数とコンテキストを実行ログとして取得できる. この専用言語を，@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，実際に取得した計算空間が HUSet オブジェクトとして引数で受け渡される．最適化前の GS 法（図 3.2）を例にすると, kernel メソッドをカーネル計算として指定し, 取得した計算空間を calculationArea に束縛するコードは以下のように書ける.

```
@HUSet(``{(i, j, k) | call(void kernel(int i, int j, int k, ..))}'')
  HUSet<HUTuple3<Integer, Integer, Integer>> calculationArea
```

この例のように, 計算空間の取得は, { で始まり } で終わり, | を境に, 左側に取得する計算空間の形状, 右側に観測対象とするカーネル計算の指定を記述する.

図 3.3. 最適化前の 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 GSNormalTest {
3
4   /* テストドライバ */
5   @HUBeforeClass
6   public static void collectingCalculationArea() {
7     GSNormal gs = new GSNormal();
8     gs.calc(data, length, length) // テスト対象アプリケーションの呼び出し
9   }
10
11  @HUTest
12  public void kernelTest(
13   /* 計算空間の取得 */
14   @HUSet("{(i, j, k) | call(void kernel(int i, int j, int k, ..))}")
15    HUSet<HUTuple3<Integer, Integer, Integer>> calculationArea)
16  {
17    /* 計算空間の検証 */
18    assertThat(LHA, is(calculationArea));
19  }
20 }
```

3.2.1 観測対象とするカーネル計算の指定

観測対象とするカーネル計算の指定は, AspectJ のポイントカットライクな記法で記述することができる. 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用いる事で, 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. 具体的には, 以下に示すような予約語を用いて, 観測対象とするメソッド呼び出しを限定できる.

call	パターンにマッチするメソッドの呼び出しを選択
within	パターンにマッチするクラスもしくはメソッド中に存在するジョインポイントを選択
receiver	パターンにマッチするクラスをレシーバとするメソッド呼び出しを選択
cflow	パターンにマッチするメソッド呼び出し以下の制御フローに含まれるジョインポイントを選択

これらの予約語の使用法を GS 法を例に述べる. 最適化前の GS 法が図 3.2, 最適化後の GS 法が図 3.4 のように定義されている場合を考える. 最適化後の GS 法（図 3.4）は, 2 章において説明した行列積と行列ベクトル積を利用する最適化が施されており, BLAS クラスに定義され

図 3.4. 最適化後の GS 法プログラム

```
public class OptimizedGSWithKernel {
  final static int DIVIDED_AREA_SIZE = 4; //分割後の領域のサイズ

  public void calc(double[] vectors, int numVec, int lengthVec) {
    calcInner(vectors, numVec, lengthVec, 0, numVec - 1, 0, numVec - 1, numVec);
  }

  private void calcInner(double[] vectors, int numVec, int lengthVec,
      int iStart, int iEnd, int jStart, int jEnd, int areaSize) {
    int dividedIEnd, iLength, dividedJEnd, jLength;
    for (int i = iStart; i <= iEnd; i += areaSize) {
      dividedIEnd = min(i + areaSize - 1, iEnd);
      iLength = dividedIEnd - i + 1;
      for (int j = jStart; j <= jEnd; j += areaSize) {
        dividedJEnd = min(j + areaSize - 1, jEnd);
        dividedJEnd = min(dividedJEnd, dividedIEnd - 1);
        jLength = dividedJEnd - j + 1;
        if (jLength <= 0)
          continue;
        if (i >= dividedJEnd + 1)
          calcWithDgemm(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd
              );
        else if (iLength <= DIVIDED_AREA_SIZE)
          calcWithDgemv(vectors, numVec, lengthVec, i, dividedIEnd, j);
        else
          calcInner(vectors, numVec, lengthVec, i, dividedIEnd, j, dividedJEnd,
              max(areaSize / 2, DIVIDED_AREA_SIZE));
      }
    }
  }

  // 行列積を用いて計算
  private void calcWithDgemm(double[] vectors, int numVec, int lengthVec,
                         int iStart, int iEnd, int jStart, int jEnd) {
    .. BLAS.dgemm(..); ..
  }

  // 行列ベクトル積を用いて計算
  private void calcWithDgemv(double[] vectors, int numVec, int lengthVec,
                         int iStart, int iEnd, int jStart) {
    .. BLAS.dgemv(..); ..
  }

  // 正規化
  private void normalize(double[] vectors, int numVec, int lengthVec, int target)
      {..}
}
```

図 3.5. 行列積と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      ..
7      for (int i = 0; i < m; i++) {
8        for (int j = 0; j < n; j++) {
9          double sum = 0.0;
10         for (int..2 = 0; j2 < k; j2++) {
11           kernel(..);
12         }
13         ..
14       }
15     }
16   }
17
18   public static void dgemv(char trans, int m, int n,
19                            double alpha, double[] A, int offsetA, int ldA,
20                            double[] x, int offsetx, int incx,
21                            double beta, double[] y, int offsety, int incy) {
22     ..
23     for (int i = 0; i < m; i++) {
24       double sum = 0;
25       for (int j = 0; j < n; j++) {
26         kernel(..);
27       }
28       ..
29     }
30   }
31
32   private static void kernel (int i, int j, int k, ..) {
33     ..
34   }
35 }
```

た行列積（dgemm）と, 行列ベクトル積（dgemv）を利用している. dgemm と dgemv のカーネル計算は kernel メソッドとして切り出されている. このとき, kernel メソッドの呼び出しを観測し, 最初の 3 つの実引数の値を取得したい場合には以下のように記述すればよい.

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

しかし, この指定では void kernel(int, int, int, ..) というシグネチャーを持つ全てのメソッド呼び出しが観測対象になってしまう. 以下のように, within キーワードを使いクラスを指定すれば, NormalGS クラス中に記述されている kernel メソッドの呼び出しだけを観測対象にするこ

とができる. これにより, 最適化前の GS 法の計算空間を表現する HUSet インスタンスを取得することができる.

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

また, cflow キーワードを使えば, 制御フローによってメソッド呼び出しを限定することができる. BLAS クラスの dgemv メソッドの制御フロー下にある kernel メソッドを観測対象にしたい場合には, 次のように記述すればよい.

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

さらに, メソッド呼び出しのレシーバの型によってメソッド呼び出しを限定することもできる. NormalGS クラスのインスタンスをレシーバとする kernel メソッド呼び出しを観測対象にしたい場合には, 以下のように記述する.

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

3.2.2　計算空間の形状の指定

取得する計算空間は, 目的に合わせた任意の大きさ, 形状で取得できる. また, 計算空間には, 空間の各座標だけでなく, MPI のランクなどのコンテキスト情報を含められる. コンテキスト情報は以下に示す予約語を用いることで取得できる.

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

例えば, 以下のように count キーワードを使えば, メソッド呼び出しの回数を含める事ができる. メソッド呼び出しの回数を取得できれば, 計算空間を巡回した順序を知ることができ, 依存関係の検査や局所性の評価などが可能となる.

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

また, 科学技術計算でよく利用される MPI のコンテキスト情報を取得する予約語も用意した. 以下のように, mpiRank キーワードを使えば, すべてのプロセスを含むコミュニケータにおけ

るランク（ノード番号）を取得できる.

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

コンテキスト情報は, 複数の計算空間で共有することもできる. コンテキスト情報を共有するには, それぞれの HUset 仮引数に対し, @HUFriend アノテーションを付加すれば良い. 例えば, 以下の例では, area1 と area2 の間で count が共有される.

```
@HUFriend @HUSet(‘‘{(count, i, j, k) |
                    call(void kernel1(int i, int j, int k, ..))}’’)
  HUSet<HUTuple4<Integer, Integer, Integer, Integer>> area1,
@HUFriend @HUSet(‘‘{(count, i, j, k) |
                    call(void kernel2(int i, int j, int k, ..))}’’)
  HUSet<HUTuple4<Integer, Integer, Integer, Integer>> area2
```

この時, kernel1 メソッドと kernel2 メソッドの呼び出しが以下のような順序だとすると,

```
kernel1(..);
kernel1(..);
kernel2(..);
kernel1(..);
kernel1(..);
kernel2(..);
```

area1 と area2 には, それぞれ以下のような計算空間が束縛される.

area1 = { (0,..), (1,..), (3,..), (4,..) }

area2 = { (2,..), (5,..) }

3.3 計算空間の検証

取得した計算空間の検証や突合は, HUSet クラスに定義された集合演算と, その等価性の判定によって記述する. 集合演算を使えば, 取得した計算空間を比較に適した形に加工することができる. HUSet クラスに定義されている集合演算は以下の通りである.

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

例えば, 最適化後の GS 法プログラムに, 計算漏れや計算重複がないか検証するコードは, 図 3.6 のように記述できる. この例では, dgemv メソッドが作り出した計算空間（triangles）と, dgemm メソッドが作り出した計算空間（squares）の和集合を取り, 最適化後の GS 法の

図 3.6. 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    assertThat(LHA , is(triangles.union(squares))); // 計算漏れ
20    assertThat(NullArea, is(triangles.intersection(squares))); // 計算重複
21  }
22 }
```

計算空間を作成し, それを最適化前の GS 法から取得される計算空間（LHA）と比較することで, 計算漏れがないことをテストしている. また, triangles と squares の積集合をとり, 空集合 (NullArea) と比較することで, triangles と squares の間に計算重複がないことを検証している.

3.3.1　LHA の作成方法

HPCUnit は, LHA を生成する方法として以下の 3 つの方法を提供している.

- 最適化前の実装から生成
- HUSet クラスに定義された計算空間生成メソッドの利用
- 専用言語を用いた計算空間生成

1 つ目は, 図 3.6 のように, 最適化前の実装から生成する方法である. 科学技術計算のプログラムは, GS 法のように簡単な実装に始まり, 徐々に最適化を施すことが多く, 正しく計算空間の巡回を行っているプログラムがユーザの手元にある場合が多い. また, 最適化以前の実装から計算空間を生成することで, 計算空間が歪な場合にも対応できるという利点がある. GS 法の

計算空間は, 綺麗な三角柱の形をしていたが, 全てのプログラムが綺麗な計算空間を持つわけではない. 例えば, 疎行列を入力とする計算の計算空間は穴の空いた計算空間となることが想定される.

2つ目は, HUSet クラスに定義された計算空間生成メソッドを利用する方法である. HUSet クラスには, 以下のように, 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計算結果生成メソッドが定義されている. これらのメソッドを活用すれば, 簡単に LHA を作成することができる.

static ⟨U⟩ HPCList⟨U⟩ getNull ()
　　空集合を生成
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⟩⟩ getRightTriangle (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

3つ目は, 専用言語を用いる方法である. 集合の要素となる各変数の範囲を <, <=, >, >= で指定することで, 任意の計算空間をつくることができる. 例えば, 以下のように記述すれば, 10×30 の正方形の計算空間が生成され, calculationArea 変数に束縛される.

```
@HUSet(‘‘{(i, j)| 0 <= i <= 10 && 0 <= j <= 30}’’)
  HUSet<HUTuple2<Integer, Integer>> calculationArea
```

3.3.2 map メソッドを利用したテスト

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 2.7 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 3.7 のように

図 3.7. 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<Integer,Integer,Integer,Integer>,
                        HUTuple3<Integer,Integer,Integer>>(){
  HUTuple3<Integer,Integer,Integer>
            calc(HUTuple4<Integer,Integer,Integer,Integer> t) {
    return new HUTuple3<>(t.el1, t.el2, t.el3 + t.el0 * 30);
  }
});
```

map メソッドを利用して記述できる.

また, HUProjection クラスに定義されたメソッド群を活用すれば, 基本的な射影処理を簡潔に記述できる. 例えば, HUProjection クラスに定義された t012_t12 メソッドを利用すれば, 3 つの要素からなるタプルの集合を表現した HUSet インスタンス（calculationArea）の第 0 要素（0 が始点とする）の削除を以下のように記述できる.

```
calculationArea = HUProjection.t012_t12(calculationArea);
// HUProjection を static import すればより簡潔に記述できる
calculationArea = t012_t12(calculationArea);
```

t012_t12 メソッドのように, HUProjection クラスに定義されたメソッドは, _ を挟むような名前となっている. _ を境に左側が, 変換前の HUSet インスタンスの形状を表しており, _ の右側が変換後の形状を表している. つまり, t012_t12 メソッドは, HUSet インスタンスの第 0 要素を削除する射影処理になる.

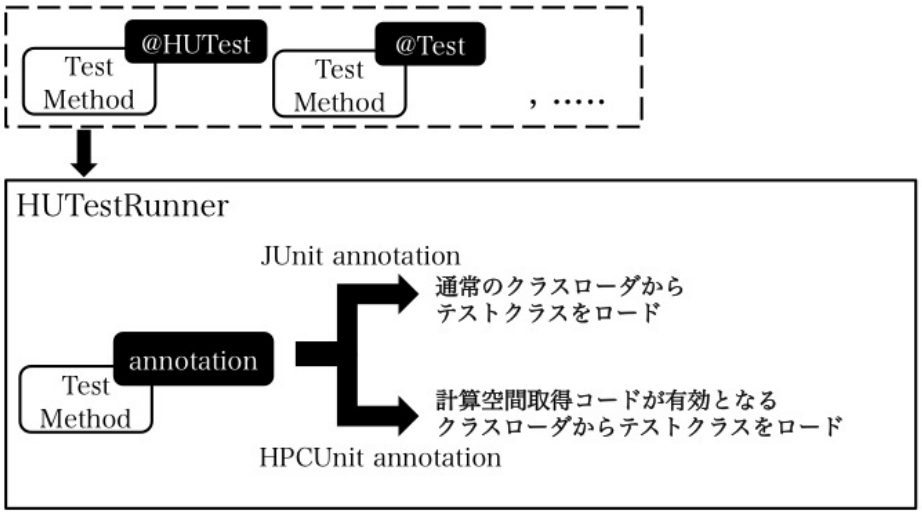
3.3.3　fold メソッドを利用したテスト

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の評価に利用できる. 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以下のように記述できる.

```
// 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
// decreasingOrder と origin はユーザ定義
gs.fold(decreasingOrder, origin);
```

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 3.8. アノテーションに応じたクラスローダの変更



プログラムの局所性の評価を助けるために, HUEvaluation クラスを準備した. HUEvaluation クラスに定義されているメソッド群を活用すれば, HUSet インスタンスに含まれるタプルの平均インターバルを求めることができる. 例として, HUSet インスタンス（calculationArea）が以下のような集合を表現している場合を考える. ここで, 0 番目の要素は巡回した順序を表している.

calculationArea = { (0, 0, 0), (1, 0, 3), (3, 1, 3), (4, 4, 5) }

このとき, calculationArea の平均インターバルを求めるコードは以下のように記述でき, 平均インターバルとして, (3 + 1 + 5)/3 = 3 が得られる.

```
int interval = HUEvaluation.intervalT3(calculationArea);
// HUEvaluation を static import すればより簡潔に記述できる
int interval = intervalT3(calculationArea);
```

3.4 効率的なテスト実行のためのサポート

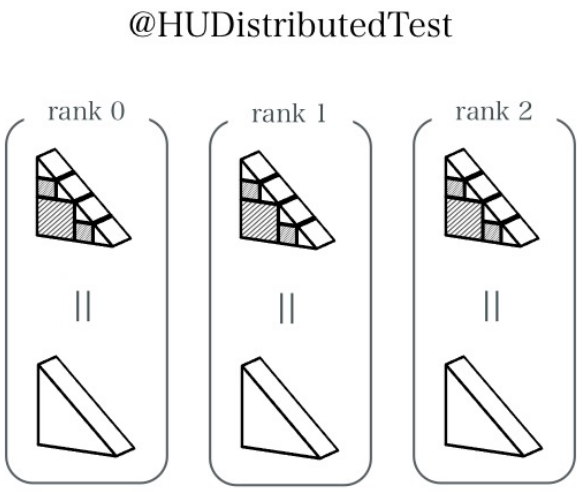
3.4.1 通常のテストとの共存

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

表 3.1. HPCUnit が提供しているアノテーション

JUnit が提供しているアノテーション	HPCUnit が提供しているアノテーション
@Test	@HUTest
@Before	@HUBefore
@After	@HUAfter
@BeforeClass	@HUBeforeClass
@AfterClass	@HUAfterClass

図 3.9. 分散テスト



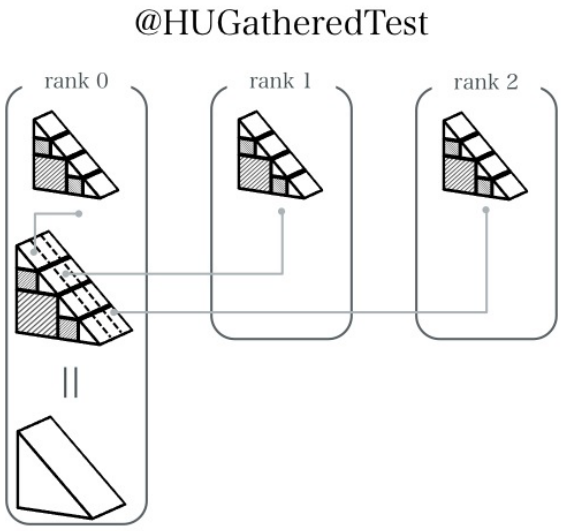
グラム変換によってログ取得コードを埋め込む．ロード時のプログラム変換は，@HU で始まるアノテーションの付加されたテストメソッドの呼び出しの際に行われる．したがって，通常の JUnit での機能テスト時には，アプリケーションのロードは JUnit 標準のクラスローダへ委譲され，ログ取得コードの埋め込まれないプログラムをテストできるようになる．（図 3.8）

アノテーションによるクラスローダの切り替えを可能にするために, HPCUnit は, JUnit が提供しているアノテーションに HU を付加したアノテーションを準備した．（表 3.1）これらのアノテーションが付加されたメソッド中では, 計算空間取得コードが有効となる．

3.4.2　MPI に特化したテスト方法

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

図 3.10. 集約テスト



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（図 3.9）と，集約テスト実行のための @HUGatheredTest アノテーション（図 3.10）をサポートしている．

3.4.3 カーネル計算の抑制

計算空間のテストを行う際には, カーネル計算を実際に行う必要がない場合がある．このときには, カーネル計算の実行を省略することで, テスト実行時間の短縮がはかれる．そこで, HPCUnit は, カーネル計算の実行を抑制するためのアノテーション @HUSkip を提供している．@HUSkip アノテーションは, @HUSet アノテーションに付加させることができる．例えば, kernel メソッドの実行を抑えつつ, その呼び出しの実引数を取得するコードは以下のように記述できる．

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

第 4 章

実装

我々は, HPCUnit を Java と Scala [22, 23] を用いて実装した. Scala は Java 仮想機械 [24] 上で動作するプログラミング言語であり, Java との間で相互呼び出しができる. 我々は, Scala のパーサコンビネータライブラリ [25] を活用し, 専用言語の処理系を実装した. パーサコンビネータライブラリは, 構文解析をおこなうためのライブラリであり, このライブラリを用いることで, 特定の用途に特化した設定ファイルや言語のパーザを手書きで書くよりも簡単に書けるようになる. HPCUnit の実装では, このライブラリを利用して, 専用言語の解析を行い抽象構文木（AST）を作成し, そこから AspectJ のソースコードを生成している. （図 4.1）

4.1 パーサコンビネータライブラリを活用した構文解析

構文解析を行うために, パーサコンビネータライブラリに定義されている JavaTokenParsers トレイトを利用した. JavaTokenParsers には, 基本的なトークンを表現する予約語と, 以下にあげるような構文解析を行うのに便利なメソッド群が定義されており, これらを使うことで, 構文解析を簡潔に記述できる. これらのメソッドの使用例として, 図 4.2 に専用言語の処理系の実装の一部を示す.

A ~ B	A と B を逐次合成
A <~ B	A と B を逐次合成、左の結果のみ保持
A ~> B	A と B を逐次合成、右の結果のみ保持
A | B	A と B の選択（順番に評価）
rep(A), A*	A をくりかえし
repsep(A, B)	B の成功を区切り位置として A
A ^^ f	解析結果の変換（A が成功したら f（関数）を実行）

4.2 AspectJ ソースコードの生成

HPCUnit では, 生成した抽象構文木から, AspectJ のソースコードを生成することで, テスト対象アプリケーションに計算空間取得コードを埋め込んでいる. 生成される AspectJ ソース

図 4.1. 専用言語の処理系の変換過程

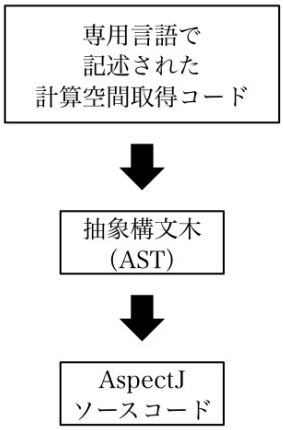


図 4.2. パーサコンビネータライブラリを用いた専用言語の解析

```
object HUParser extends JavaTokenParsers {
  def parse(str: String): ParseResult[Set] = {
    parse(set, str)
  }

  /** SET **/
  private def set = for {
    tuple <- "{" 〜> tuple
    condition <- "|" 〜> condition <〜 "}"
  } yield Set(tuple, condition)

  private def tuple = "(" 〜> repsep(element, ",") <〜 ")" ^^ {Tuple(_)}
  private def condition = makeArea ||| pointcut

  /** POINT CUT **/
  private def pointcut = repsep(joinPoint, "&&") ^^ {Pointcut(_)}
  private def joinPoint = receiver | call | cflow | within

  ..
}
```

図 4.3. 生成された AspectJ ソースコード

```
1 public aspect HUTraceRecipe$0 extends HUTraceRecipe {
2   before(int i,int j,int k) :
3     within(module.gramschmidt.GramSchmidtV0) &&
4     call( void kernelCalc (int,int,int,..))&&
5     args(i,j,k,..) {
6       add(new HUTuple3 (i,j,k));
7     }
8 }
```

コードは, 通常は before アドバイスを利用したものとなる. 一方, @HUSkip アノテーションが付加された計算空間取得コードからは, around アドバイスを利用したソースコードが生成される. 例えば, 以下のような計算空間取得コードがあった場合, 図 4.3 のようなソースコードが生成される.

```
{(i, j, k) | call(void kernelCalc(int i, int j, int k, ..)) &&
                        within(module.gramschmidt.GramSchmidtV0)}
```

生成されるアスペクトは, HUTraceRecipe（図 A.1） を拡張したものとなる. HUTraceRecipe には, 計算空間を保存するためのフィールドと, そのフィールドに要素を追加する add メソッドが定義されている. また, コンテキスト情報を取得するためのメソッド群も定義されている. 取得するコンテキスト情報が MPI に関するものを含む場合は, HUMPITraceRecipe を継承したアスペクトが生成される.

第 5 章

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. それぞれのソースコードは付録に添付した. また, @HUSkip の効果を検証するための実験も行った. 実験には, 以下に示す通り, 東京大学のスーパーコンピュータ FX10 [26] を用いた.

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

5.1 HPCUnit によるオーバーヘッドとメモリ使用量

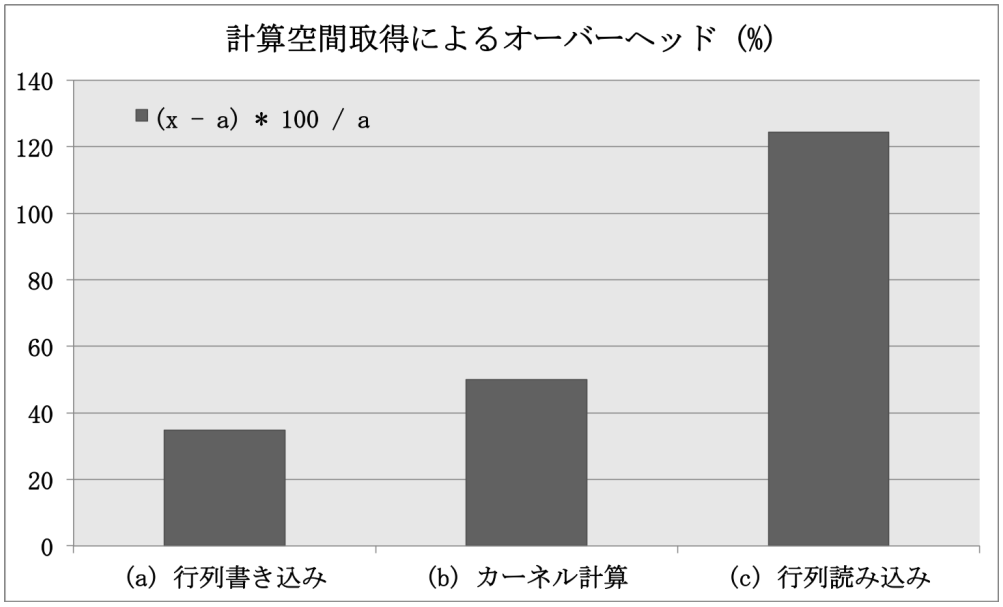
実行ログ取得コードのオーバーヘッドとメモリ使用量の増加を測定した. 実験結果として, 表 5.1 にそれぞれのプログラムを 7 回実行し最大と最小を除いた 5 回の平均実行時間とメモリ使用量, 取得した計算空間の大きさを示す. また, (a), (b), (c) の実行時間に含まれるオーバーヘッドの, 元の GS 法プログラムの実行時間に対する比率を図 5.1 に示す. GS 法の入力行列 (ベクトル数 L × ベクトルの長さ M) は, L を 128 , M を 4096 とした.

図 5.1 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

表 5.1. 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 5.1. 計算空間取得によるオーバーヘッド



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.

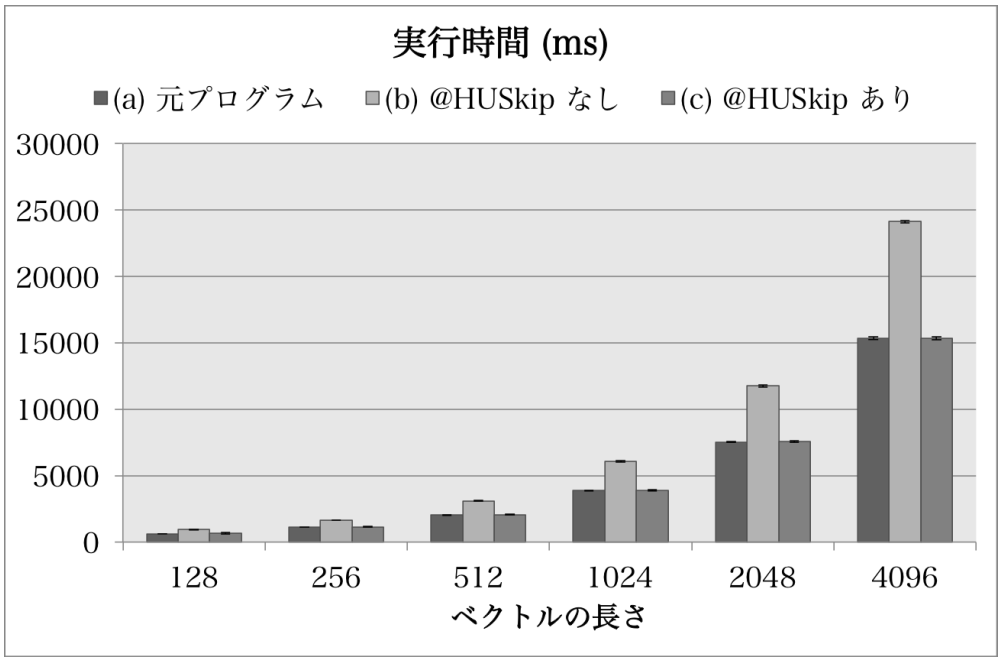
5.2　専用言語の生産性

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 5.2 に示す. なお生成され

表 5.2. 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

図 5.2. @HUSkip の効果



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.

5.3 @HUSkip の効果

@HUSkip による効果を確認するための実験を行った. @HUSkip が添付された計算空間取得コードは, カーネル計算を実際に行うことなく, 計算空間を取得する. そのため, カーネル計算の実行時間の分だけ, テスト実行時間が短くなると予想できる. この実験では, GS 法のプログラム (a), @HUSkip なしに計算空間を取得した GS 法プログラム (b), @HUSkip を付けて計

算空間を取得した GS 法プログラム (c) の 3 つのプログラムを実行し, それぞれの実行時間を計測した. GS 法のプログラムの入力行列は, ベクトルの本数を 128 で固定し, ベクトルの長さを 128 ~ 4096 の間で指数的に増加させた.

実験結果を図 5.2 に示す. 実験結果をみると, @HUSkip が付加されたプログラム (c) の方が, @HUSkip が付加されていないプログラム (b) よりも, 実行時間が短いことがわかる. したがって, @HUSkip によって, テスト実行時間が短縮されたと言える. @HUSkip による実行時間の短縮の幅は, カーネル計算の実行時間によって変化することが想定される. 全体の実行時間に占めるカーネル計算の実行時間の割合が大きいほど, @HUSkip の効果は大きくなる. @HUSkip を添付しテスト実行時間を短縮すれば, テスト実行する際の問題サイズを, より本番実行に近づけて行うことができる.

第 6 章

まとめと今後の課題

6.1 まとめ

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 を提案した. 科学技術計算のプログラムは種々の最適化が施され, クラスタコンピュータなどで並列分散実行される. 最適化や並列分散実行はプログラムの実行性能を高めるのに有効だが, 一方で計算空間の分割を招き, 計算漏れや計算重複といったバグを誘発する. HPCUnit は, プログラムの計算空間を取得し, 計算漏れや計算重複といったバグの検出を可能にする機能を提供する. 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. また, HPCUnit の有用性を示すために, グラムシュミットの正規直交化法プログラムの計算漏れテストを行った際のオーバーヘッドとコード行数を測定した.

6.2 今後の課題

今後考えられる改善として, 以下の 2 点があげられる.

- メモリ使用量の抑制
- GPGPU への対応

現状の HPCUnit は, 計算空間を点の集合として表現しているため, 取得する計算空間によっては大きなメモリ空間を消費する. メモリ使用量を抑える方法として, 点ではなく, 範囲で計算空間を表現する方法が考えられる. 範囲の集合として, 計算空間を表現することで, 集合の要素が減少し, メモリ使用量を抑制できる. 科学技術計算では, キャッシュヒット率をあげるために, 連続したアクセスを行うことが多い. そのため, 範囲として表現できる箇所が多いと想定される.

また, GPGPU への対応も考えたい. GPGPU のプログラミングモデルでは, カーネル計算をメソッドとして定義し, それを複数スレッドで並列実行する. このプログラミングモデルは, カーネル計算がメソッドとして切りだされている事を想定している HPCUnit と相性が良い.

発表文献と研究活動

(1) 穂積 俊平, 佐藤 芳樹, 千葉 滋. 登壇発表 2013 年並列／分散／協調処理に関する『北九州』サマー・ワークショップ（SWoPP 北九州 2013）2013.7.31-8.2
(2) ポスター発表 第 15 回プログラミングおよびプログラミング言語ワークショップ PPL 2013
(3) ポスター発表 MODULARITY aosd 2013

参考文献

[1] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[2] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 208. ACM, 2006.

[3] Yukihiro Hasegawa, Jun-Ichi Iwata, Miwako Tsuji, Daisuke Takahashi, Atsushi Oshiyama, Kazuo Minami, Taisuke Boku, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, Hikaru Inoue, Ikuo Miyoshi, and Mitsuo Yokokawa. First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the k computer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1:1–1:11, New York, NY, USA, 2011. ACM.

[4] Jun ichi Iwata, Daisuke Takahashi, Atsushi Oshiyama, Taisuke Boku, Kenji Shiraishi, Susumu Okada, and Kazuhiro Yabana. A massively-parallel electronic-structure calculations based on real-space density functional theory. *J. Comput. Physics*, pp. 2339–2363, 2010.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[6] 横澤拓弥, 高橋大介, 朴泰祐, 佐藤三久ほか. 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.

[7] CUDA Nvidia. Programming guide, 2008.

[8] Andrew Hunt, David Thomas, and Pragmatic Programmers. *Pragmatic unit testing in Java with JUnit*. Pragmatic Bookshelf, 2004.

[9] Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[10] NASA. pfunit. `http://opensource.gsfc.nasa.gov/projects/FUNIT/index.php`.

[11] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble.

Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 345–364, October 2005.

[12] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A Naeem. Collaborative runtime verification with tracematches. In *Runtime Verification*, pp. 22–37. Springer, 2007.

[13] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *ACM SIGPLAN Notices*, Vol. 42, pp. 569–588. ACM, 2007.

[14] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 546–550. Springer, 2005.

[15] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, Vol. 23, No. 5, pp. 279–295, May 1997.

[16] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 366–381, 2000.

[17] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 366–381, 2000.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[19] John Irwin, Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, and J Loingtier. Aspect-oriented programming. *Proceedings of ECOOP, IEEE, Finland*, pp. 220–242, 1997.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. ECOOP 2001—Object-Oriented Programming, pp. 327–354. Springer, 2001.

[21] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *ECOOP 2005-Object-Oriented Programming*, pp. 121–143. Springer, 2005.

[22] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[23] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[24] Tim Lindholm and Frank Yellin. *The Java virtual machine specification*, Vol. 297. Addison-Wesley Reading, 1997.

[25] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in scala. *CW Reports*, 2008.

[26] The University of Tokyo. Fx10. http://www.cc.u-tokyo.ac.jp/system/fx10/.

謝辞

本研究を進めるにあたり, 支えてくださった皆様へ感謝の意を示します. 指導教員の千葉滋教授には, 研究の方針や論文の書き方、研究発表の方法など数々の有用な助言をいただき, 大変感謝しております. また, 東京大学情報基盤センターの佐藤芳樹特任講師には, 研究の詳細を決めるにあたり適切な助言をいただきました. さらに, 千葉研究室の先輩方は, 情報科学に関する多くの知識を教えてくださいました. 特に, 伊尾木将之氏の指導のおかげで, 私の研究の土台ができました. とても感謝しております. 研究室の同期, 後輩とは, 時には情報科学を探求し, 時には笑って遊びました. その気分転換が私の研究生活を楽しいものとしてくれました. ありがとうございます. 最後に, 私の研究生活を経済的な面から, また精神的な面から支えてくれた穂積宣男氏, 洋子氏に感謝いたします.

付録 A

ソースコード

図 A.1. HUTraceRecipe.aj

```
1  public abstract aspect HUTraceRecipe<T extends Comparable<T>> implements
2      Comparable<HUTraceRecipe<?>> {
3    private HUSet<T> simulationArea;
4    private int id;
5    private static int cnt = 0;
6    private HUTraceRecipe<T>[] friendsCache;
7    private boolean friendsFlag = false;
8
9    public HUTraceRecipe() {
10     simulationArea = new HUSet<T>();
11     HUTracer.set(this, simulationArea);
12     id = cnt++;
13   }
14
15   protected void add(T tuple) {
16     simulationArea.add(tuple);
17   }
18
19   protected HUTraceRecipe<T>[] friends() {
20     return null;
21   }
22
23   private HUTraceRecipe<T>[] friendsInner() {
24     if (friendsFlag)
25       return friendsCache;
26     friendsFlag = true;
27     return friendsCache = friends();
28   }
29
30   protected int count() {
31     int num = simulationArea.size();
32     if (friendsInner() != null)
33       for (HUTraceRecipe<T> friend : friendsInner())
34         num += friend.simulationArea.size();
35     return num;
36   }
37
38   protected long time() {
39     return System.currentTimeMillis();
40   }
41
42   protected long threadID() {
43     return Thread.currentThread().getId();
44   }
45
46   @Override
47   public int compareTo(HUTraceRecipe<?> o) {
48     return this.id - o.id;
49   }
50   ..
51 }
```

図 A.2. 実験：(a) 行列書き込みの監視

```
@RunWith(HUTestRunner.class)
public class WriteTest
{
    @HUBeforeClass
    public static void collectingSA() throws MPIException, FileNotFoundException
    {
        start_mpi_parallel(new String[]{});  // MPI 初期化
        prepare(); // 並列処理の準備
        init_wf(); // サンプルデータの作成

        GramSchmidtV0 gs0 = new GramSchmidtV0();
        GramSchmidtV3 gs3 = new GramSchmidtV3();

        long start = System.currentTimeMillis();
        gs0.gram_schmidt(1, Nband, 1, 1); // 最適化前のグラムシュミット計算
        gs3.gram_schmidt(1, Nband, 1, 1); // 最適化後のグラムシュミット計算
        System.out.println("app_time:" + (System.currentTimeMillis() - start));
        System.out.println("app_mem:" + (Runtime.getRuntime().totalMemory()-Runtime
            .getRuntime().freeMemory()));
    }

    @HUAfterClass
    public static void mpiFinish() throws MPIException {
    end_mpi_parallel();  // MPI 解放
    }

    @HUTest
    public void writeTest(
           @HUSet("{(i, j, k, l, util.Complex m) | call(void array.Array4D.set$(
               int i, int j, int k, int l, * m)) && cflow(receiver(module.
               gramschmidt.GramSchmidtV0) && call(void normalize(..)))}")
             HUSet<HUTuple5<Integer,Integer,Integer,Integer,Complex>> normal,
           @HUSet("{(i, j, k, l, util.Complex m) | call(void array.Array4D.set$(
               int i, int j, int k, int l, * m)) && cflow(receiver(module.
               gramschmidt.GramSchmidtV3) && call(void normalize(..)))}")
            HUSet<HUTuple5<Integer,Integer,Integer,Integer,Complex>> optimized
       ) {
      long start = System.currentTimeMillis();
      assertThat(optimized, is(normal));
      System.out.println();
      System.out.println("test_time:" + (System.currentTimeMillis() - start));
        System.out.println("test_mem:" + (Runtime.getRuntime().totalMemory()-
            Runtime.getRuntime().freeMemory()));
    }
}
```

図 A.3. 実験：(b) カーネル計算の監視

```
@RunWith(HUTestRunner.class)
public class KernelTest
{
    @HUBeforeClass
    public static void collectingSA() throws MPIException, FileNotFoundException
    {
        start_mpi_parallel(new String[]{});  // MPI 初期化
        prepare(); // 並列処理の準備
        init_wf(); // サンプルデータの作成

        GramSchmidtV0 gs0 = new GramSchmidtV0();
        GramSchmidtV3 gs3 = new GramSchmidtV3();

        long start = System.currentTimeMillis();
        gs0.gram_schmidt(1, Nband, 1, 1); // 最適化前のグラムシュミット計算
        gs3.gram_schmidt(1, Nband, 1, 1); // 最適化後のグラムシュミット計算
        System.out.println("app_time:" + (System.currentTimeMillis() - start));
        System.out.println("app_mem:" + (Runtime.getRuntime().totalMemory()-Runtime
            .getRuntime().freeMemory()));
    }

    @HUAfterClass
    public static void mpiFinish() throws MPIException {
    end_mpi_parallel();  // MPI 解放
    }

    @HUTest
    public void kernelTest(
           @HUSet("{(i, j, k) | call(void kernelCalc(int i, int j, int k, ..)) &&
               within(module.gramschmidt.GramSchmidtV0)}")
             HUSet<HUTuple3<Integer,Integer,Integer>> normal,
           @HUSet("{(i, j, k) | call(void kernelCalc(int i, int j, int k, ..)) &&
               within(blas.BLAS)}")
             HUSet<HUTuple3<Integer,Integer,Integer>> optimized
       ) {
      long start = System.currentTimeMillis();
      assertThat(optimized, is(normal));
      System.out.println();
      System.out.println("test_time:" + (System.currentTimeMillis() - start));
        System.out.println("test_mem:" + (Runtime.getRuntime().totalMemory()-
            Runtime.getRuntime().freeMemory()));
    }
}
```

図 A.4. 実験：(c) 行列読み込みの監視

```
@RunWith(HUTestRunner.class)
public class ReadTest
{
    @HUBeforeClass
    public static void collectingSA() throws MPIException, FileNotFoundException
    {
        start_mpi_parallel(new String[]{});  // MPI 初期化
        prepare(); // 並列処理の準備
        init_wf(); // サンプルデータの作成

        GramSchmidtV0 gs0 = new GramSchmidtV0();
        GramSchmidtV3 gs3 = new GramSchmidtV3();

        long start = System.currentTimeMillis();
        gs0.gram_schmidt(1, Nband, 1, 1); // 最適化前のグラムシュミット計算
        gs3.gram_schmidt(1, Nband, 1, 1); // 最適化後のグラムシュミット計算
        System.out.println("app_time:" + (System.currentTimeMillis() - start));
        System.out.println("app_mem:" + (Runtime.getRuntime().totalMemory()-Runtime
            .getRuntime().freeMemory()));
    }

    @HUAfterClass
    public static void mpiFinish() throws MPIException {
    end_mpi_parallel();  // MPI 解放
    }

    @HUTest
    public void readTest(
           @HUSet("{(i, j, k) | order(i) && call(* array.Array4D.get(int j, int k,
                ..)) && within(void module.gramschmidt.GramSchmidtV0.kernelCalc
               (..))}")
             HUSet<HUTuple3<Integer,Integer,Integer>> normal,
           @HUSet("{(i, j, k) | order(i) && call(* array.Array4D.get(int j, int k,
                ..)) && within(void blas.BLAS.kernelCalc(..))}")
             HUSet<HUTuple3<Integer,Integer,Integer>> optimized
       ) {
      long start = System.currentTimeMillis();
      assertThat(HUEvaluation.intervalT3(optimized), lessThanOrEqualTo(HUEvaluation
          .intervalT3(normal)));
      System.out.println();
      System.out.println("test_time:" + (System.currentTimeMillis() - start));
        System.out.println("test_mem:" + (Runtime.getRuntime().totalMemory()-
            Runtime.getRuntime().freeMemory()));
    }
}
```