特別研究報告

題目

分散処理による大規模ソフトウェアに対するコーディングパターン検出ツール

指導教員
井上 克郎 教授

報告者
悦田 翔悟

平成 21 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 20 年度 特別研究報告

分散処理による大規模ソフトウェアに対するコーディングパターン検出ツール

悦田 翔悟

内容梗概

コーディングパターンとは，ソースコード上の複数のモジュールに存在する定型的なコード片である．コーディングパターンには，ソフトウェアを理解する上で有用なコードや横断的関心事などが含まれており，複数の文字列から頻出する部分列を抽出する手法であるシーケンシャルパターンマイニングを用いて検出することができる．

しかし，大規模ソフトウェアを対象としてコーディングパターンを検出する場合は，計算量が大幅に増加することから，実行時間やメモリ不足などの問題があり，計算機 1 台でのパターン検出は難しい．

そこで本研究ではシーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan 法を複数台の計算機に分散処理させることにより，コーディングパターン検出の高速化を実現する．PrefixSpan 法を分散処理させる手法としては，マスタ・ワーカ法や，マスタ・タスク・スティール法が提案されているが，これらはアミノ酸データベースなどを実験対象としていた．本研究では，これらの分散手法が Java ソースコードを対象データとしたときの有効性を評価するために，ツールの実装を行い，実験を行った．その結果，Java ソースコードを対象とした場合でも，マスタ・タスク・スティール法による分散処理が有効であることを確認した．

主な用語

コーディングパターン
シーケンシャルパターンマイニング
PrefixSpan
分散処理
マスタ・ワーカ法
マスタ・タスク・スティール法

目 次

1 まえがき **3**

2 コーディングパターン **5**

2.1 コーディングパターン 5

2.2 コーディングパターンマイニング 5

2.2.1 ソースコードの正規化 7

2.2.2 シーケンシャルパターンマイニング 8

2.3 既存手法の問題点 9

3 PrefixSpan アルゴリズムの分散処理 **11**

3.1 マスタ・ワーカ法 11

3.2 マスタ・タスク・スティール法 11

4 設計と実装 **13**

5 評価実験 **15**

5.1 実験目的 15

5.2 実験環境 15

5.3 実験対象 15

5.4 実験方法 16

5.5 実験結果 16

5.5.1 マスタ・ワーカ法 16

5.5.2 マスタ・タスク・スティール法 19

5.6 考察 22

6 まとめと今後の課題 **24**

謝辞 **25**

参考文献 **26**

1　まえがき

近年，多くのソフトウェア開発にオブジェクト指向プログラミングが使用されている．オブジェクト指向プログラミングの利点として，継承や多態性など，モジュール化された部品を有効に活用するための機構がある．しかし，すべての機能がモジュール化されるわけではなく，ロギングや同期処理などは，複数のモジュールに分散した定型的なコード片，すなわち，コーディングパターンとして実装される．コーディングパターンが生じる原因としては，元となるソースコードを開発者が複製し，配置先の状況に応じて適宜変更を加えるという場合などが考えられる．

コーディングパターンを検出することにより，定型的なライブラリの使い方や，エラー処理などソフトウェアを理解する上で有益なコードや横断的関心事に関する情報が得られる [6]．我々の研究グループでは，コーディングパターンをメソッド呼び出し要素と制御構造要素で構成される定型的な文字列と捉え，Java のソースコードに対するコーディングパターン検出手法を実現している．具体的には Java ソースコードを対象とした正規化ルールを用意し，各メソッドをメソッド呼び出し要素と制御構造要素からなる特徴列へ変換する．得られた特徴列データベースに対して，シーケンシャルパターンマイニング手法の 1 つである PrefixSpan アルゴリズム [9] を用いて，頻出する部分列をコーディングパターンとして検出する．

コーディングパターン検出における課題として，対象ソースコードの規模が増大するにつれて，計算コストが大幅に増加することがあげられる．数百万行におよぶ大規模ソフトウェアに対しては，メモリ不足や計算時間などの問題から既存手法によるパターン検出が困難である．そのため大規模ソフトウェアに対してコーディングパターン検出を行う場合には，既存手法に改良を加える必要がある．そこで，本研究ではコーディングパターン検出における PrefixSpan アルゴリズムを分散処理させることにより，パターン検出の高速化を実現し，大規模ソフトウェアを対象とするコーディングパターン検出を可能にする．

具体的な分散処理の手法としてはマスタ・ワーカ法 [10] や，マスタ・タスク・スティール法 [11] などが提案されているが，従来の分散処理の実験が対象としているのはアミノ酸などのデータベースであり，本研究の対象データはソースコードのデータベースである．ソースコードから抽出された要素列はアミノ酸などのデータに比べて，要素の種類が多く，列の長さが短いという特徴があるため，分散処理による性能向上が得られるかどうか，実験による評価を行う必要があった．

コーディングパターン検出ツール Fung[4] に対して PrefixSpan の分散処理機能を組み込み，ANTLR[1] と Apache Ant[2] を実験対象とした評価実験を行った．その結果，Java ソースコードを対象とした場合であっても分散手法が有効であることを確認した．具体的には，

マスタ・タスク・スティール法を用いることで，ワーカを 6 台にすることで 4〜5.2 倍の性能向上比が確認できた.

本論文の構成は次の通りである．2 章ではコーディングパターンとその検出法を詳しく述べる．3 章では PrefixSpan アルゴリズムの分散処理法について説明し，4 章で具体的な設計と実装を述べる．5 章では Java ソースコードを対象とした分散処理法の適用実験を評価する．最後に 6 章で本研究のまとめと今後の課題を述べる.

2　コーディングパターン

2.1　コーディングパターン

コーディングパターンとは，ソースコードの複数個所に出現する類似した構造をもつコード片である．我々の研究グループでは，コーディングパターンをメソッド呼び出し要素と制御構造要素（条件分岐と繰り返し文）で構成される定型的な列と捉えたパターンマイニング手法を提案している．

図 1 は，画像編集ソフト JHotDraw 5.4b1 から検出されたコーディングパターンの例で，編集作業を「元に戻す」ことを可能とするための実装の一部である．各編集操作に対応するクラスごとに実行する処理は異なるが，下線で示されるメソッド呼び出し列が共通しており，パターンとして抽出されている．このようなメソッド呼び出し列を知ることで,「元に戻す」仕組みがどのように実装されているか理解しやすくなり，新たな機能を実装する際の手助けになる．

開発者が既存のコード記述に基づいて新たな機能を実装する場合，既存のコードを複製し，改変を加えて配置を行うということがしばしば起こる．そして，新しいコードが既存のコードと類似した機能を実装する場合は，新しいコードにおいても，元となったコードと同じメソッド名を使用し続けることが多いと考えられる．このようなメソッドや，継承関係にあるクラスでオーバライドされたメソッドを同一視するために，メソッド呼び出し要素を抽出するときは，メソッドのシグネチャのクラス名を無視して抽出を行う．また，コーディングパターンに対応する具体的なコード片，すなわちインスタンスは，ソースコード上では不連続でも良いものとする．

2.2　コーディングパターンマイニング

コーディングパターン抽出法は次のステップからなる．まず，入力されたソースコードをメソッド単位に分割したものに対して正規化を行い，メソッド呼び出し要素と制御構造要素で構成される特徴列データベースを作成する．得られた特徴列データベースに対してシーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan 法を適用し，コーディングパターンを抽出する．

シーケンシャルパターンマイニングとは，複数の文字列にユーザが指定した閾値以上の頻度で共通して現れる部分列を求める手法である．ソースコードを対象としたシーケンシャルパターンマイニングの利用例としては，API 理解支援 [14] やソフトウェアのパターン違反検出 [8][13] などが知られている．

図 2 はコーディングパターン検出の手順を表している．既存手法では入力として，解析対象の Java プログラムのソースコード，PrefixSpan の引数である最小サポート値，実際に提

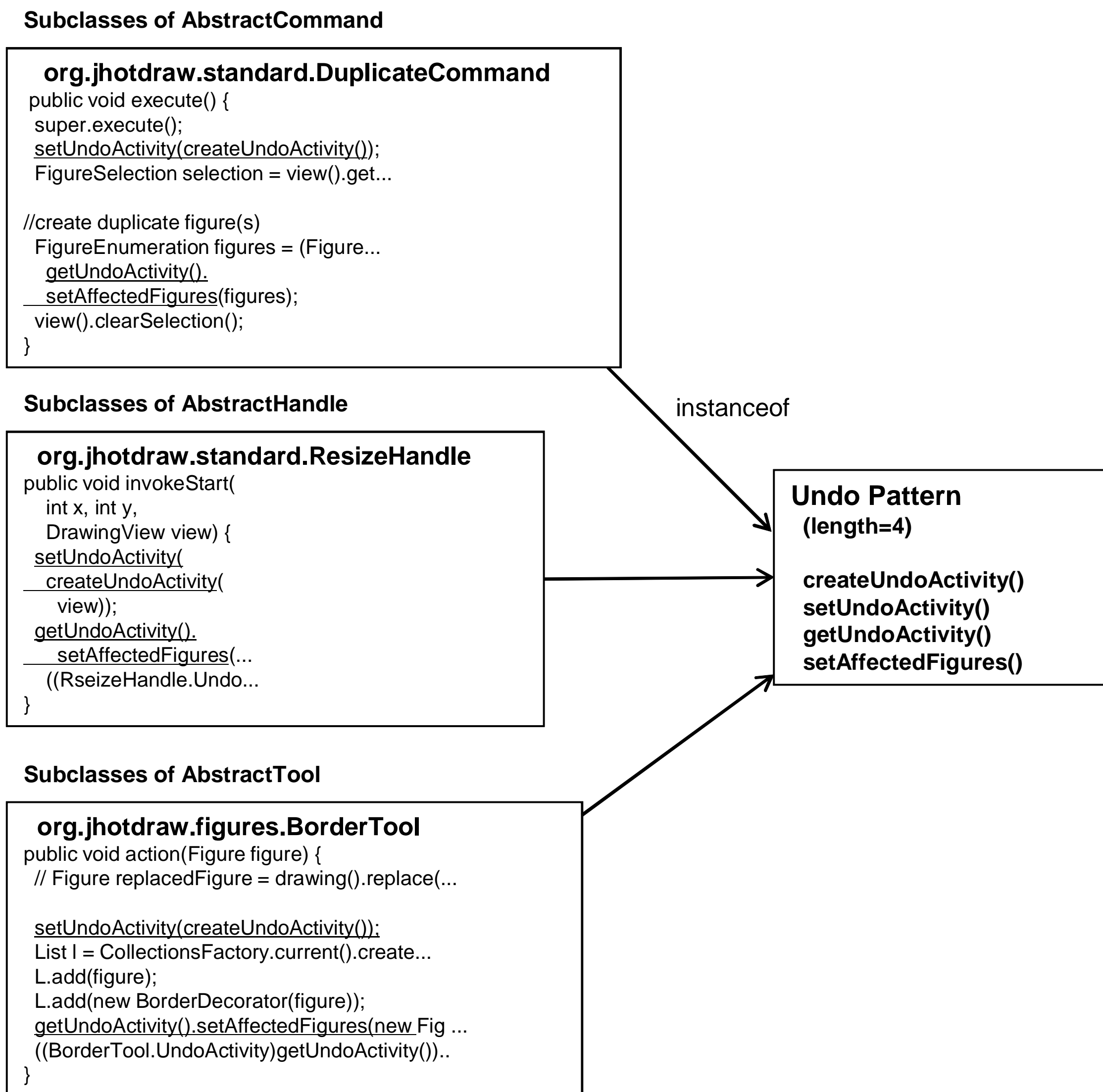

図 1: JHotDraw 5.4b1 における画像の編集作業を「元に戻す」実装パターン

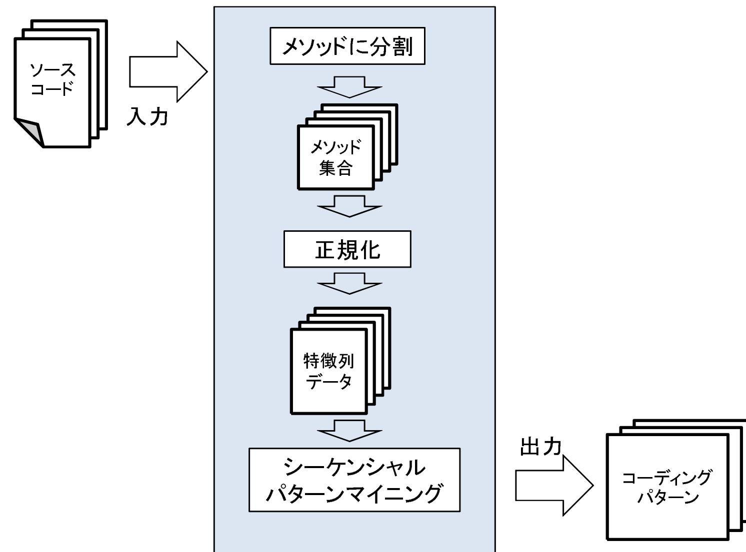

図 2: コーディングパターン検出の手順

```
Statement: if (<cond>) <then> else <else>;
Sequence:  <cond>, IF, <then>, ELSE, <else>, END-IF

Statement: for (<init>; <cond>; <inc>) <body>;
Sequence:  <init>, <cond>, LOOP, <body>,
           <inc>, <cond>, END-LOOP

Statement: while (<cond>) <body>;
Sequence:  <cond>, LOOP, <body>, <cond>, END-LOOP
```

図 3: 制御文の正規化ルール

示するコーディングパターンの要素数の最小値を指定する最小パターン長を与える．

2.2.1 ソースコードの正規化

正規化のステップではソースコードをメソッド単位に分割し，メソッド呼び出し要素と制御構造要素の特徴列として正規化する．ソースコード中に出現するメソッド呼び出し式は，シグネチャ，すなわちメソッド名，引数の型名の列，そして戻り値を持つメソッド呼び出し要素へと変換し，ソースコード上での呼び出し発生順序に従って一列に並べる．また，これらのメソッド呼び出しの実行順序を制御する if 文，for 文，while 文は図 3 に示す制御文の正規化ルールに従い，メソッド呼び出し要素と制御構造要素から構成される特徴列に変換する．正規化の例を図 4 に示す．

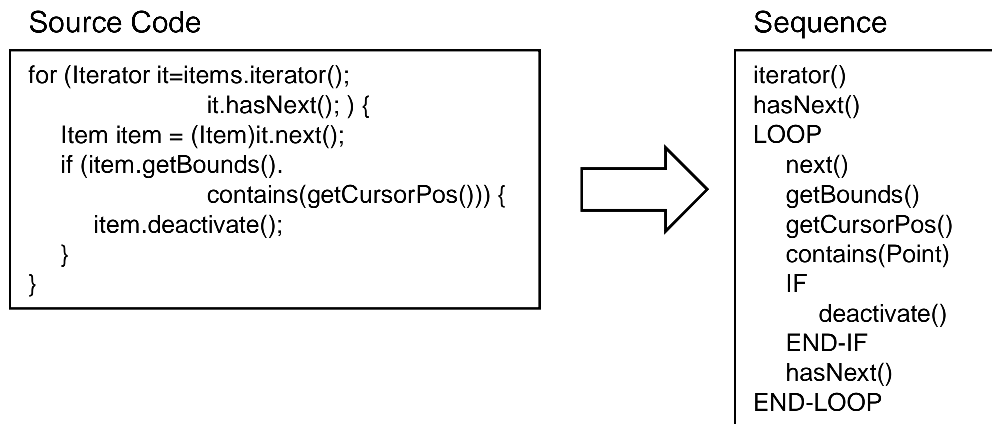

図 4: ソースコードの正規化例

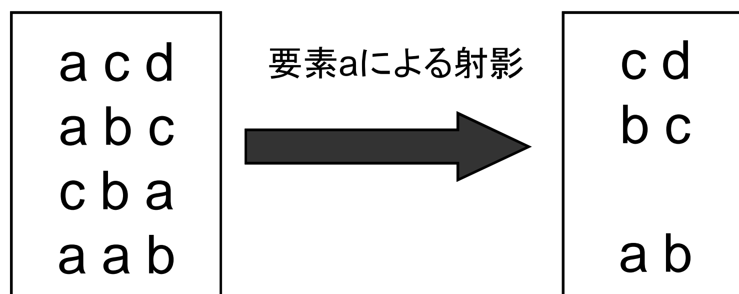

図 5: 要素 a についての射影

2.2.2 シーケンシャルパターンマイニング

正規化された要素列に対して，PrefixSpan を用いたシーケンシャルパターンマイニングを行い，コーディングパターンを抽出する．PrefixSpan は射影と呼ばれる操作を繰り返し行い，複数の特徴列の中から頻出する部分列，すなわちパターン候補を抽出していくアルゴリズムである．射影とは，すべての特徴列から，特定の要素に対しての接尾辞を取り出す操作であり，図 5 は要素 a について射影を行う様子を表している．

PrefixSpan では最小サポート値というパラメータを用いる．最小サポート値は，コーディングパターンとして検出する要素列の最低出現回数に関する閾値である．

PrefixSpan を用いたシーケンシャルパターンマイニングの処理の流れは次の通りである．

(1) 各特徴列に出現するすべての要素について，それぞれ，出現回数 (サポート値) を計算する．

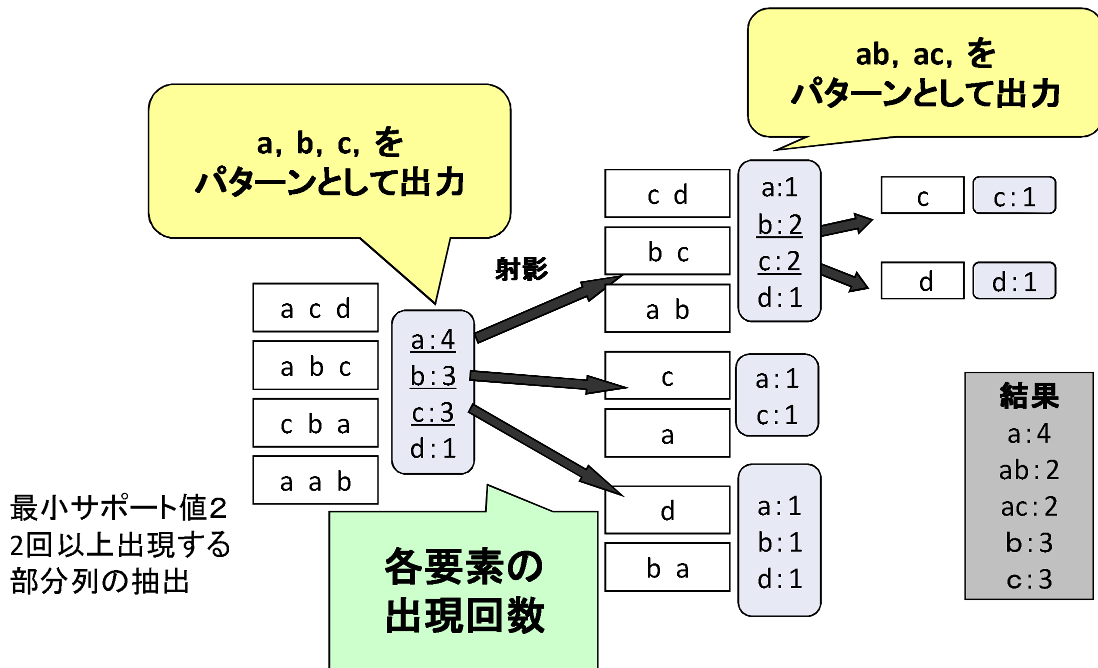

図 6: 最小サポート値 2 における PrefixSpan を用いたシーケンシャルパターンマイニング

(2) 出現回数が最小サポート値以上である要素を長さ 1 のパターン候補とする．

(3) パターン候補それぞれに対して射影を行う．

(4) 射影された特徴列における各要素の出現回数を計算する．

(5) 出現回数が最小サポート値以上である要素を，1 つ前の射影に使用したパターン候補の末尾に付け加えたもの新たなパターン候補とする．

(6) (3) に戻る．(3) において新たな接尾辞が取り出されなくなるか，(4) において最小サポート値以上の出現回数を持つ要素がなくなるまで (3)〜(6) を繰り返す．

図 6 は最小サポート値が 2，最小パターン長が 1 のときの PrefixSpan によるシーケンシャルパターンマイニングの様子を表している．パターンとして a，ab，ac，b，c が抽出されている．

2.3　既存手法の問題点

コーディングパターン検出における課題の 1 つが，解析対象の規模の拡大である．パターン抽出を行う際，解析対象のソースコードが大きくなると，計算コストが大幅に増加するため，

既存手法のまま大規模ソフトウェアを対象としたパターン検出を行うのは難しい．Eclipse のような大規模プロジェクトを対象とする場合や，あるソフトウェアの複数バージョンを同時に解析するためには，パターン検出の高速化が必要である [5].

3 PrefixSpan アルゴリズムの分散処理

本研究ではコーディングパターン検出におけるシーケンシャルパターンマイニングの計算を分散処理させることにより，パターン抽出の高速化と解析対象ソフトウェアの大規模化に対応する．

PrefixSpan アルゴリズムは，パターンの要素を 1 つ選んで射影を計算し，その要素に続く次のパターン要素を 1 つ選ぶというように探索を繰り返すものであるが，異なる要素に対しての射影以後のパターン検出は基本的に独立である．したがって，パターン候補ごとの計算を複数台の計算機に分散処理させることで，性能向上が期待できる．

3.1 マスタ・ワーカ法

PrefixSpan を分散処理させるための具体的な手法として，マスタ・ワーカ法が提案されている [10]．マスタ・ワーカ法はジョブを生成，管理するマスタプロセスとパターンを検出するワーカプロセスに分けて PrefixSpan 法の分散処理を実現している．具体的な手順は次の通りである．

まずマスタプロセスは，ユーザが与える任意の値の深さまでは単独で PrefixSpan の計算をを行い，それによって得られたパターン候補をジョブとして保持する．次に各ワーカがジョブを受け取り，パターン検出を続けていく．ワーカがすべてのジョブを処理したら，ワーカが抽出したコーディングパターンをマスタが回収し，結果として出力する．

3.2 マスタ・タスク・スティール法

マスタ・ワーカ法ではマスタがジョブを分割した後，新たなジョブの分割は行われない．PrefixSpan の特性から，各ジョブの処理時間を事前に予測することは難しく，ジョブの処理時間に偏りが生じると，処理時間が長いジョブを受け取ったワーカに負荷が偏ることになる．各ワーカの負荷を均一にするには動的に負荷分散を行う必要がある．

PrefixSpan 法を分散処理させる際に動的な負荷分散を行う手法として，マスタ・ワーカ法を拡張したマスタ・タスク・スティール法が提案されている [11]．マスタ・タスク・スティール法ではワーカはマスタからジョブを受け取った後，ジョブを細かく分割し，ローカルジョブとして保持する．すべてのジョブを終了したワーカが出現したとき，マスタは他のワーカから未処理のローカルジョブを回収し，すべてのワーカに対して再配布することで動的な負荷分散を実現している．具体的な手順は次の通りである．

まずマスタプロセスがユーザが与える任意の値の深さまで PrefixSpan の計算を行い，それによって得られたパターン候補をグローバルジョブとして保持する．次に各ワーカがグローバルジョブを受け取り，任意の値の深さまでの PrefixSpan の計算を行い，それによっ

て得られたパターン候補をローカルジョブとして保持する．ワーカはローカルジョブがなくなるまでパターン抽出を行い，ローカルジョブがなくなると新しくグローバルジョブを受け取る．グローバルジョブがなくなったときに，一部のワーカがローカルジョブを保持している場合は，マスタがローカルジョブを回収して，グローバルジョブとして割り当てを行っていく．

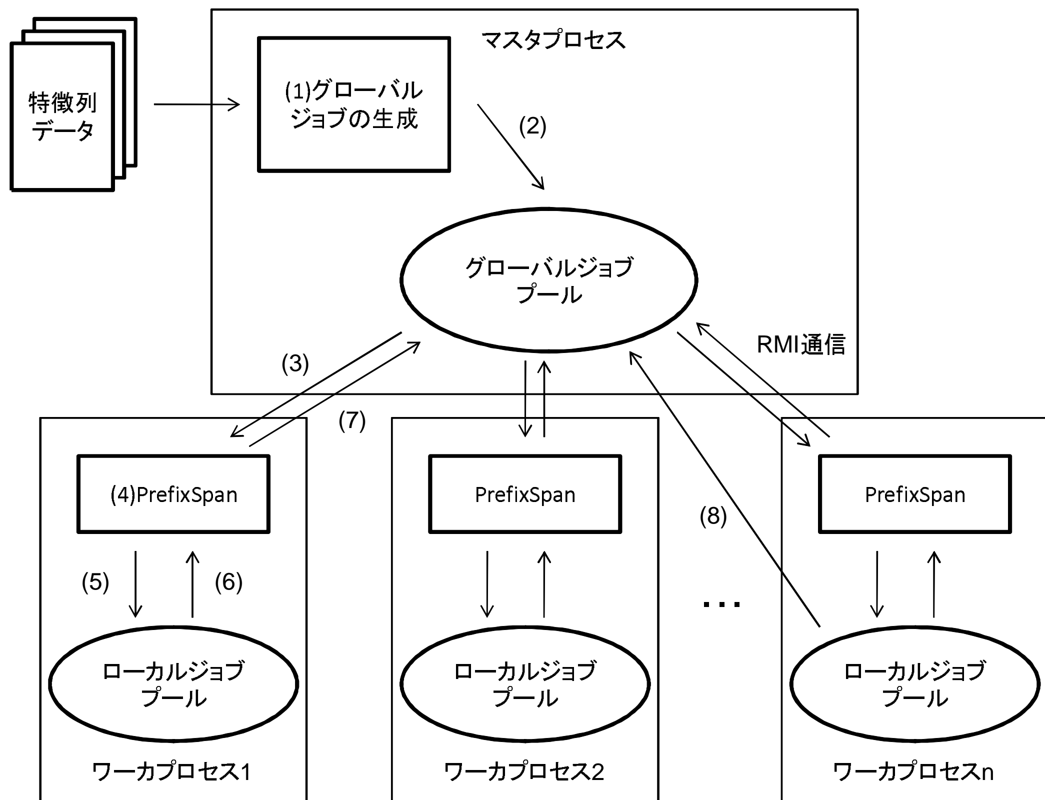

図 7: システム構成図

4 設計と実装

本研究では，我々の研究グループで開発しているコーディングパターン検出ツール Fung を拡張し，マスタ・タスク・スティール法を用いたパターンマイニングの分散実行を実現した．マスタとワーカの通信には Java RMI[7] を使用する．図 7 はシステムの構成図であり，以下のような手順で，処理を実行する．

(1) マスタが深さ 1 の射影を行い，特徴列データからグローバルジョブを生成する．

(2) グローバルジョブをグローバルジョブプールへ入れる．

(3) ワーカがグローバルジョブプールからジョブを受け取る．

(4) ワーカが受け取ったジョブに対して PrefixSpan 法を用いた深さ 1 の射影を行う．

(5) (4) によって得られたパターン候補をローカルジョブとみなし，ローカルジョブプールへ入れる．

(6) ローカルジョブプールからジョブを取り出し，パターンマイニングを行って，コーディングパターンを検出する．

(7) ローカルジョブが空になったら，グローバルジョブプールから新たなジョブを受け取る．

(8) ワーカは一定の間隔でグローバルジョブが空になっていないかチェックを行い，空になっていた場合は自分のすべてのローカルジョブをグローバルジョブプールへ入れる．

(9) ローカルジョブから回収されたジョブが 0 ならば (10) へ，ジョブが回収されれば (3) へ戻る．

(10) 各ワーカが検出したコーディングパターンをマスタが回収する．

本来 (1) や (4) ではワーカ台数に対して十分なジョブ数を確保するために，何度か射影を行う必要がある．しかし本研究の解析対象は要素が多く，一度の射影でおよそ 1000 以上のジョブを生成することから，射影は一度だけ行うものとした．

5　評価実験

実装したツールを使って Java ソフトウェアに対してコーディングパターンの検出を行い，パターン検出ににおける性能の向上を測定，評価する．

5.1　実験目的

従来の PrefixSpan アルゴリズムの分散処理法の実験ではアミノ酸配列を実験対象としており，パターンを構成する要素数は 20 種類程度だった．本研究では Java ソースコードを実験対象としており，要素数は呼び出されているメソッド呼び出しの数であり，プログラムの規模に比例して増加する．一方で，要素列の長さは 1 つのメソッドの長さによって決まるため，プログラムの規模が大きくなっても，適切に設計されているプログラムでは，大きく変動しないという特徴がある．このような特徴列を解析対象とした場合，分散処理によりどの程度の性能が向上するか調査することが，実験の目的である．

5.2　実験環境

本実験で用いたシステムの構成を以下に示す.

- マスタ PC：CPU Pentium4 3.2GHz，OS Windows XP 1 台
- ワーカ PC：CPU Pentium4 1.5GHz，OS Windows XP 6 台
- ネットワーク：Ethernet 100base-TX
- 通信方法：Java RMI[7]

5.3　実験対象

本実験の対象データとして選んだソフトウェアの詳細な情報を表 1 に示す．また本実験では最小サポート値 4，最小パターン長 10 という設定でコーディングパターンの検出を行った.

表 1: 対象データの詳細

実験対象	コード行数 (KLOC)	特徴列数	特徴列の平均長	特徴列の最大長	要素の種類数	検出したパターン数
ANTLR 3.0.1	56	1196	16	199	1370	58545
Apache Ant 1.7.0	195	2841	15	416	3847	134568

特徴列数とはJavaソースコードをメソッドごとに分割し，正規化して得られた特徴列の中で，パターン長が最小パターン長以上の列数である．要素の種類数とはメソッド呼び出し要素と制御構造要素のうち出現回数が最小サポート値以上の要素の種類数であり，本実験ではマスタが生成するジョブ数は要素の種類数と同じである．

5.4　実験方法

実験対象で示したソフトウェアに対して，以下の場合に関する計算時間を計測した．

1. マスタ・ワーカ法による計算時間 （ワーカ台数：1〜6台）
2. マスタ・タスク・スティール法による計算時間（ワーカ台数：1〜6台）

性能の比較は，PrefixSpanの計算時間における性能向上比を用いる．基準となる計算時間として，マスタ・ワーカ法でワーカ1台だけを用いた，つまり分散処理が行われない場合の処理時間を用いて，以下の式によって計算時間の比を求める．

$$\text{性能向上比} = \text{ワーカ1台での計算時間}/\text{分散計算での計算時間} \quad (1)$$

5.5　実験結果

5.5.1　マスタ・ワーカ法

マスタ・ワーカ法を用いて，計算機の台数を1台から6台まで増やしたときのシーケンシャルパターンマイニング部分の処理時間を表2に示す．前処理とはシーケンシャルパターンマイニングを行うまでに必要な処理時間であり，ソースコードのメソッド分割や，正規化などにかかった時間である．

次にワーカの台数を増やしていったときの性能向上比を示す．図8，図9は，それぞれANTLR，Apache Antにおける性能向上比を示したものである．図ではワーカ台数に比例した理想的な性能向上比 (N台の計算機によって性能がN倍になる) と，マスタ・ワーカ法を用いたときの実際の性能向上比を示す．

表2: ワーカ台数ごとのシーケンシャルパターンマイニング部分の処理時間

実験対象	前処理	ワーカ1台	ワーカ2台	ワーカ3台	ワーカ4台	ワーカ5台	ワーカ6台
ANTLR	35秒	2982秒	1637秒	1141秒	970秒	963秒	958秒
Apache Ant	69秒	6140秒	3358秒	3340秒	3375秒	3359秒	3349秒

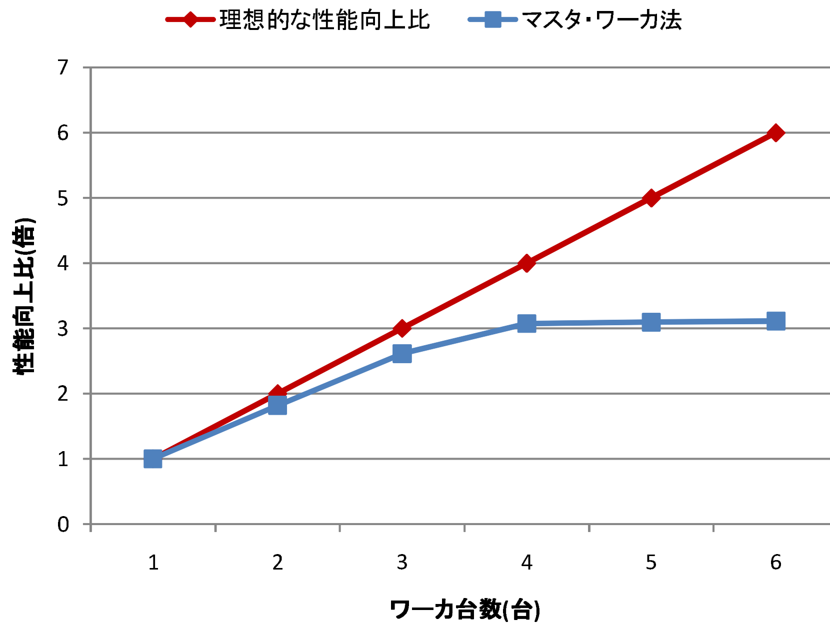

図 8: ANTLR 解析時の性能向上比

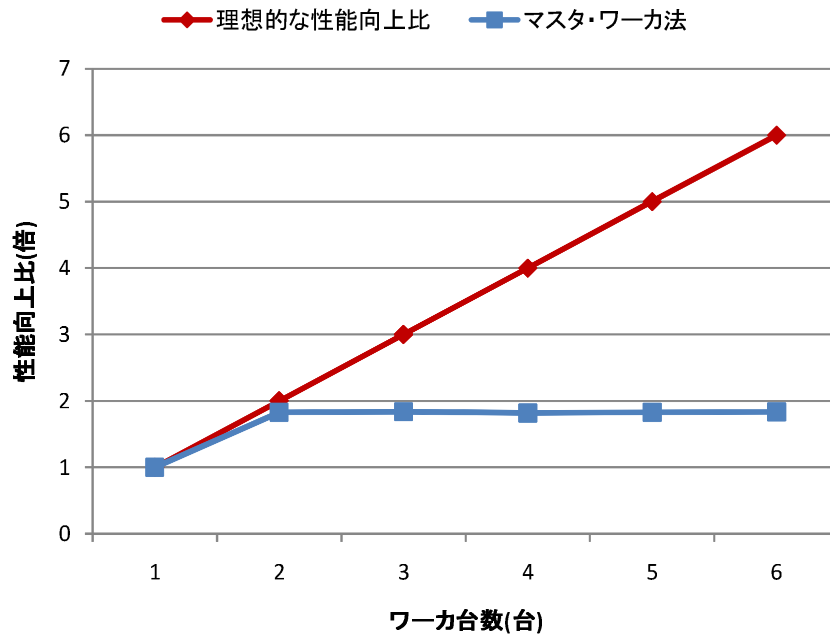

図 9: Apache Ant 解析時の性能向上比

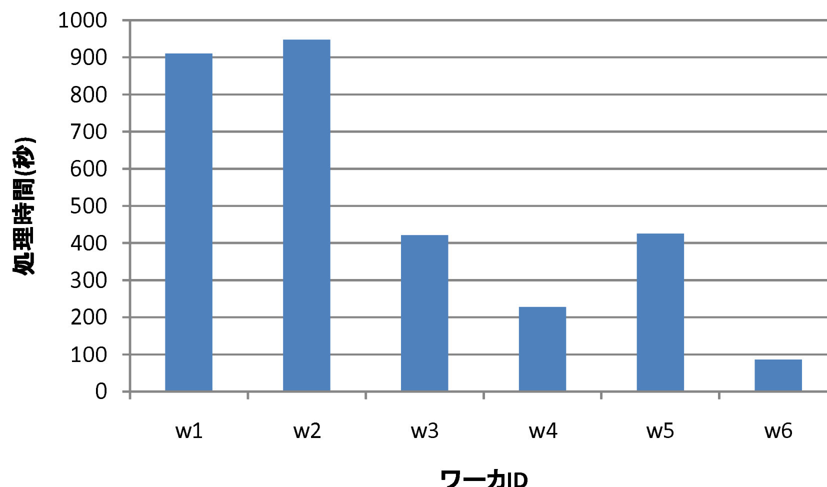

図 10: ワーカ台数 6 台における ANTLR 解析時の各ワーカの処理時間

ANTLR 解析時にはワーカ台数が 4 台のときに 3 倍の性能向上比が見られるが，5 台目からは性能向上比はほとんど変わらない．Apache Ant 解析時にはワーカ台数が 2 台のときに 1.9 倍とほぼ理想的な性能向上比が見られるが，3 台目からは性能向上比はほとんど変わらない．

ワーカの台数を増やしても性能向上比がほとんど変わらない理由は，マスタ・ワーカ法では一部のワーカに負荷が偏るためである．ワーカ台数を 6 台としたときの各ワーカの処理時間を図 10 と図 11 に示す．

ANTLR の解析では各ワーカの処理時間は最小で 86 秒，最大で 948 秒とワーカ w1 と w2 に負荷が偏っている．Apache Ant においても，各ワーカの処理時間は最小で 288 秒，最大で 3323 秒となっており，w1 に負荷が大きく偏っている．

一部のワーカに負荷が偏ってしまうのは，マスタよって生成された各グローバルジョブの処理時間に偏りがあるためである．全体の処理時間に対する各グローバルジョブの処理時間が占める割合を図 12 と図 13 に示す．図では処理時間が大きい上位 5 つのジョブについて示し，他はその他としてまとめている．円グラフの横に示しているのは，各グローバルジョブが探索するパターンの先頭要素を示している．

グローバルジョブ数は ANTR では 11370，Apache Ant では 3847 であるが，図からわかるように上位 5 つのジョブの処理時間が全体の処理時間に対して 90%近くを占めている．そのため処理時間が大きいグローバルジョブを受け取ったワーカに負荷が偏り，ワーカ台数を増やしても性能向上比は上がらない．また，Apache Ant では制御構造要素 IF という要素

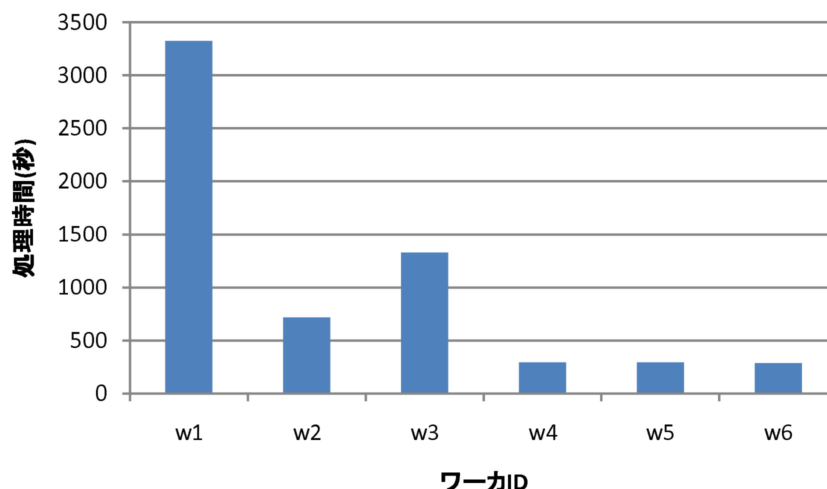

図 11: ワーカ台数 6 台における Apache Ant 解析時の各ワーカの処理時間

による射影から始まるグローバルジョブが全体の 50%以上を占めている．Apache Ant 解析時にワーカ台数が 2 台の時に，ほぼ理想的な性能向上比が見られ，3 台目から性能向上比が変わらないのは，この要素に対する計算がボトルネックとなっているためである．

5.5.2 マスタ・タスク・スティール法

マスタ・タスク・スティール法を用いて，計算機の台数を 1 台から 6 台まで増やしたときのシーケンシャルパターンマイニング部分の実行時間を表 3 に示す．

次にワーカの台数を増やしていったときの性能向上比を図 14 と図 15 に示す．図では理想的な性能向上比と，マスタ・ワーカ法およびマスタ・タスク・スティール法を用いたときの実際の性能向上比を示す．

性能向上比はワーカを増やすごとに上がっており，ANTLR ではワーカ台数 6 台に対しての性能向上比は 5.2 倍，Apache Ant ではワーカ台数 6 台に対しての性能向上比は 4.0 倍だっ

表 3: ワーカ台数ごとのシーケンシャルパターンマイニング部分の処理時間

実験対象	前処理	ワーカ 1 台	ワーカ 2 台	ワーカ 3 台	ワーカ 4 台	ワーカ 5 台	ワーカ 6 台
ANTLR	35 秒	3019 秒	1543 秒	1106 秒	833 秒	675 秒	570 秒
Apache Ant	69 秒	6231 秒	3233 秒	2847 秒	2156 秒	1822 秒	1531 秒

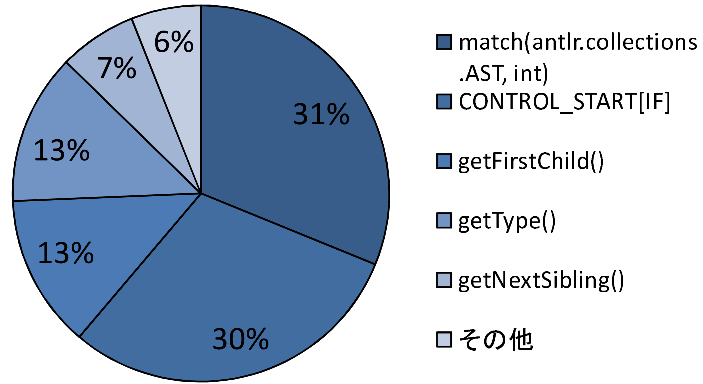

図 12: ANTLR の各グローバルジョブの処理時間が全ジョブの処理時間の合計に占める割合

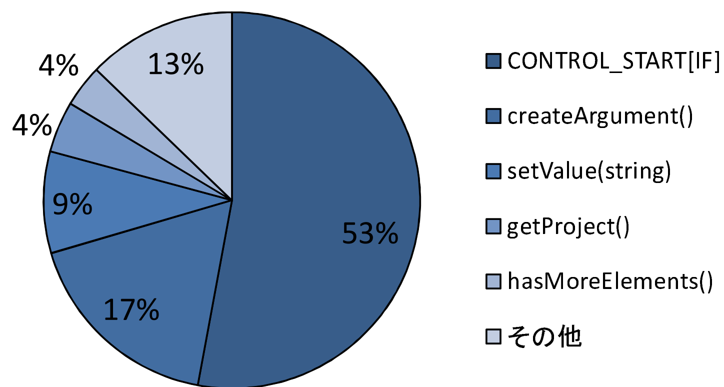

図 13: Apache Ant の各グローバルジョブの処理時間が全ジョブの処理時間の合計に占める割合

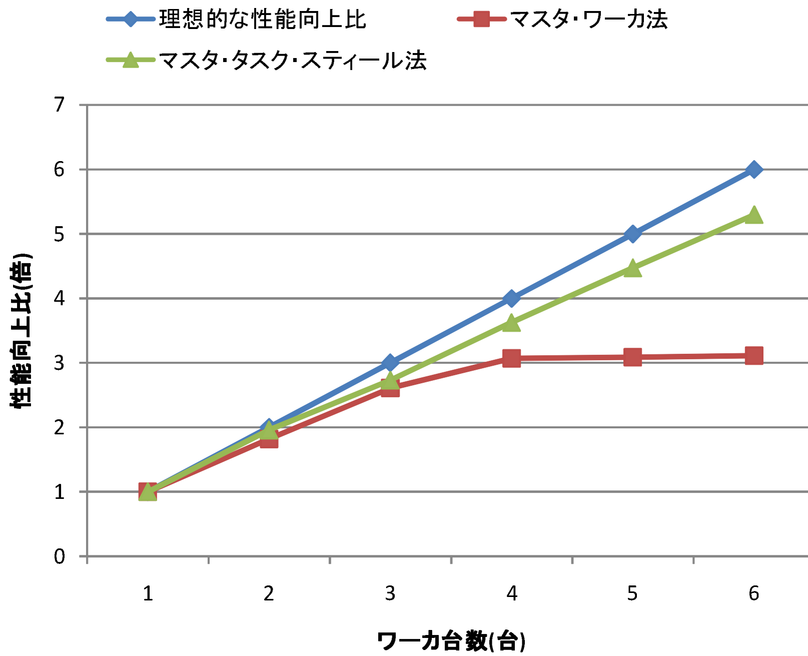

図 14: ANTLR 解析時の性能向上比

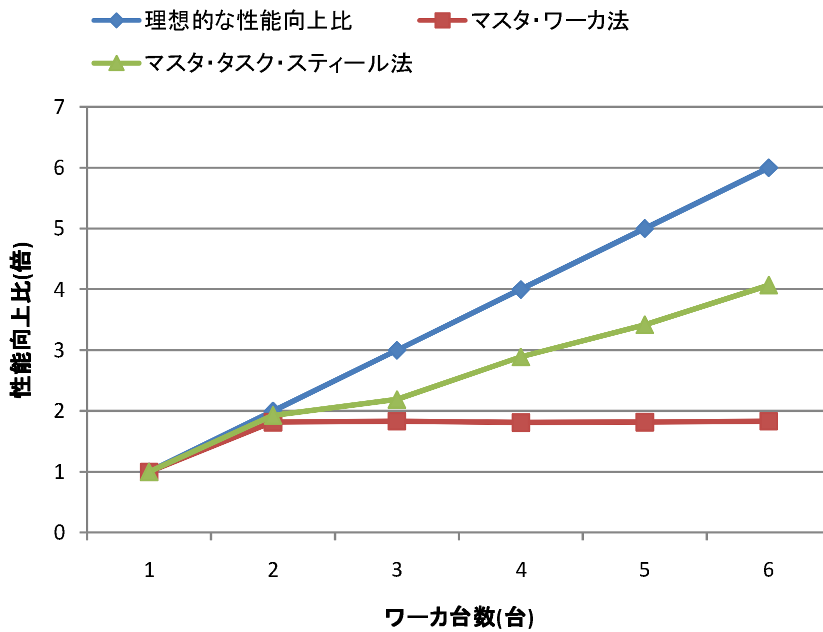

図 15: Apache Ant 解析時の性能向上比

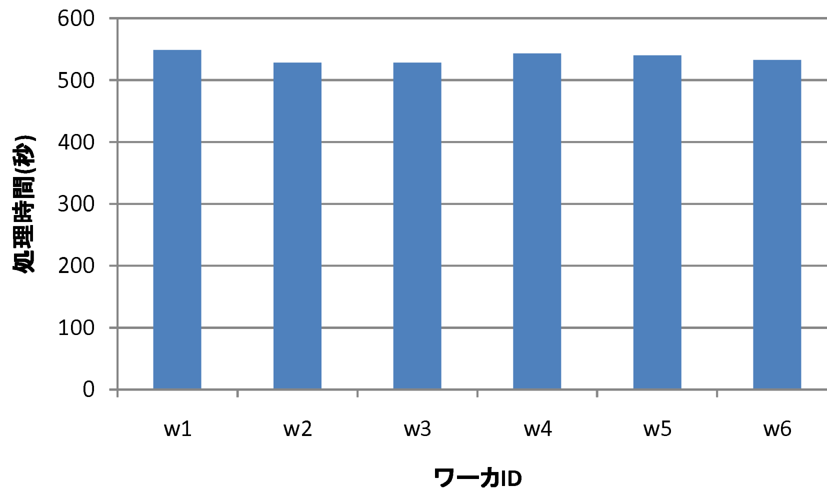

図 16: ワーカ台数 6 台における ANTLR 解析時の各ワーカの処理時間

た．ワーカ台数を 6 台としたときの各ワーカの処理時間を図 16 と図 17 に示す．

各ワーカの処理時間はほぼ同じになっており，マスタ・ワーカ法に比べて負荷分散が成功していることがわかる．

5.6　考察

本実験ではワーカの台数を増やしてもマスタ・ワーカ法による性能向上はあまり見られなかった．マスタが行う射影の深さを増やし，ジョブをより細かくすることはできるが，マスタが生成するジョブ数はワーカ台数に比べて既に十分大きいことから，マスタがさらにジョブを分割するのは，マスタに負荷が偏るため望ましくない．また，各グローバルジョブに大きな偏りがあり，一部のワーカに大きな負荷がかかることから，これ以上ワーカ台数を増やしても性能向上の効果は薄いと考えられる．

一方でマスタ・タスク・スティール法を用いた場合，ワーカ台数を増やすごとに性能向上比の増加が見られた．各ワーカの処理時間も均等になっており，負荷分散が成功している．よって，新たにワーカを増やしてもある程度の性能向上が期待できる．

ただし，台数が増加するに従って，ローカルジョブの回収における通信遅延の増加が生じる可能性がある．図 12 および図 13 で示したように多くのジョブは短時間で計算が終了するため，回収して再割り当てを行うコストのほうが高くなると予測される．これに対して，一部のワーカのローカルジョブだけを回収する手法を導入することを検討している．

今後，大規模ソフトウェアを解析する場合，マスタが一回の射影で生成するジョブ数は大

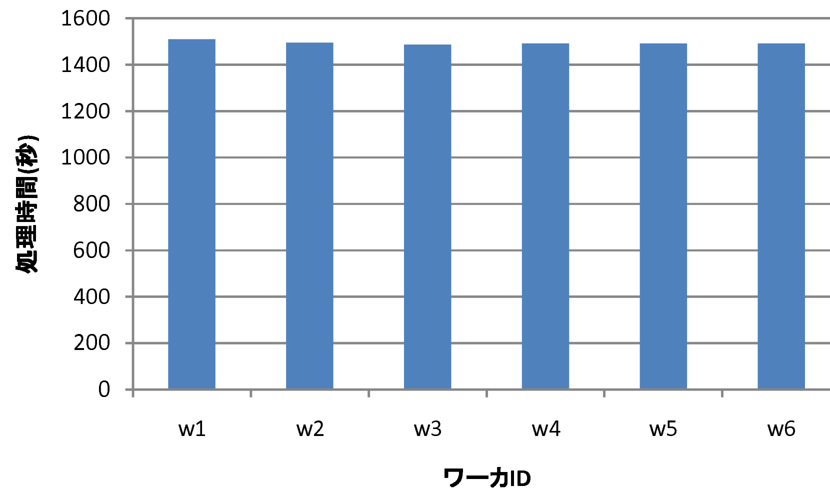

図 17: ワーカ台数 6 台における Apache Ant 解析時の各ワーカの処理時間

幅に増加すると考えれる．そのため，マスタによる必要以上のジョブの分割は避けるべきであり，ワーカから動的にジョブを回収するマスタ・タスク・スティール法が非常に有効である．

6　まとめと今後の課題

大規模ソフトウェアを対象としてコーディングパターンを検出する場合，大幅な計算量の増加により，実行時間やメモリ不足などの問題が生じる．そのため 1 台の計算機で大規模ソフトウェアを対象とするコーディングパターン検出は困難であった．そこで本研究では複数台の計算機を分散処理させることにより，コーディングパターン検出の高速化を実現した．

分散処理の手法としてはマスタ・ワーカ法とマスタ・タスク・スティール法が提案されているが，Java ソースコードを対象データとした場合，マスタ・ワーカ法ではワーカに対する負荷分散が難しく，動的な負荷分散手法を用いたマスタ・タスク・スティール法が有用であることがわかった．

今後の課題としては，より多くの計算機を使って分散処理を行う場合の，タスク・スティールによる通信の遅延への対処が挙げられる．マスタ・タスク・スティール法ではすべてのワーカがローカルタスクをマスタへ送信しているが，一部のワーカのローカルジョブのみを回収する手法などが有効とされている [12].

また今回検出されたコーディングパターンの数は，ANTLR で約 6 万，Apache Ant で約 13 万と非常に多く，コーディングパターンを有効活用するためには，検出されたパターンの中から有用なパターンを見つけるための支援が必要である [3].

謝辞

本研究において，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究において，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において，逐次適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究において，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 早瀬 康裕 特任助教に深く感謝いたします.

本研究において，様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊達 浩典 氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] ANTLR Parser Generator. `http://www.antlr.org/`.

[2] Apache Ant. `http://ant.apache.org/`.

[3] 伊達浩典, 石尾隆, 三宅達也, 井上克郎. コーディングパターンの分類に用いるソフトウェアメトリクスの検討. 平成 20 年度 情報処理学会関西支部 支部大会 講演論文集, pp. 59–62, 2008.

[4] Fung: A Pattern Mining Tool for Java method calls. `http://sel.ist.osaka-u.ac.jp/~ishio/fung/`.

[5] 石尾隆, 伊達浩典, 三宅達也, 井上克郎. 大規模パターンマイニングを用いた高品質ソースコードの検索. インターワークショップ 2009・イン・宮崎 論文集情報処理学会シンポジウムシリーズ, pp. 9–10, 2009.

[6] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. Mining coding patterns to detect crosscutting concerns in java programs. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pp. 123–132. Antwerp, Belgium, 2008.

[7] Java RMI. `http://java.sun.com/docs/books/tutorial/rmi/index.html`.

[8] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pp. 20–27. Minneapolis, USA, 2007.

[9] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl B, Helen Pinto, iming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224. Heidelberg, Germany, 2001.

[10] Toshihide Sutou, Keiichi Tamura, Yasuma Mori, and Hajime Kitakami. Design and implementation of parallel modified prefixspan method. In *Proceedings of the 5th International Symposium on High Performance Computing*, pp. 412–422. Tokyo, Japan, 2003.

[11] 高木允, 田村慶一, 周藤俊秀, 北上始. 並列 modified prefixspan 法における動的負荷分散手法. 情報処理学会研究報告 MPS 数理モデル化と問題解決研究報告, pp. 9–15, 2004.

[12] Makoto Takaki, Keiichi Tamura, Toshihide Sutou, and Hajime Kitakami. New dynamic load balancing for parallel modified prefixspan with distributed worker paradigm. In *21st International Conference on Data Engineering Workshops*, pp. 1243–1244. Tokyo, Japan, 2005.

[13] Andraej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 35–44. Dubrovnik, Croatia, 2007.

[14] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pp. 54–57. Shanghai, China, 2006.