社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

分散処理を用いた自動プログラム修正の高速化

松本淳之介† 肥後 芳樹† 松尾 裕幸†
有馬 諒† 柗本 真佑† 楠本 真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{j-matumt,higo,h-matuo,r-arima,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　GenProg は遺伝的アルゴリズムを利用した自動プログラム修正ツールである．GenProg はソースコードを個体とみなし，各個体に対してプログラム文の挿入・削除・置換といった加工を行う．GenProg は加工した個体を多数生成し，その全てをビルドし，テストする．全てのテストを通過する個体がない場合，生成した個体から優秀な個体を選択し，選択した個体を加工し，ビルドし，テストする．全てのテストを通過する個体が生成されるまでこのループを繰り返す．そこで問題となるのがパフォーマンスである．GenProg は生成した全ての個体に対してビルドとテストを行う．ビルドやテストは大きな計算コストを必要とする処理であるため，全てのテストを通過するプログラムを生成するまでのループの回数が多い場合に GenProg の実行時間は非常に長くなってしまう．このことにより，GenProg の改良を目指している研究者が思いついたアイデアを試す際に長い時間が必要になる等の問題が生じる．本論文では GenProg のパフォーマンスを上げる方法として，各世代において生成された個体のビルドとテストを並列化するツールを提案する．GenProg のビルドとテストを行うクラスタを構築し，それらの処理を分散させることでパフォーマンスを向上させる．提案ツールのパフォーマンスを評価するために，クラスタを構成するマシンの台数を変えて実験を行い，台数に応じてパフォーマンスが向上したことを確認した．

キーワード　自動プログラム修正, 分散処理, GenProg, Kubernetes

1. まえがき

ソフトウェア開発において，開発者はデバッグに多くの時間を費やしていると報告されている [1], [2]．そのため，プログラムの欠陥を自動で修正する，自動プログラム修正の研究が活発に行われている．

自動プログラム修正ツールである GenProg [3] は自動プログラム修正の研究にブレークスルーを起こした．オープンソースソフトウェア (以下 OSS と呼ぶ) に含まれるプログラムの欠陥を対象に実験を行い，GenProg は開発者の手を必要とせずに 55 個の欠陥を修正することができたと報告されている [4]．GenProg は遺伝的アルゴリズムを利用しており，ソースコードを個体とみなし，各個体に対してプログラム文の挿入・削除・置換といった加工を行う．GenProg は入力された個体に対して加工した個体を多数生成し，その全てに対してビルドとテストを行う．全てのテストを通過する個体がある場合，その個体を修正済みプログラムとして出力する．全てのテストを通過する個体がない場合，生成した個体からテスト通過率の高い個体を選択し，その個体に対して加工した個体を多数生成し，その全てに対してビルドとテストを行う．欠陥を修正できるか，事前に決めた終了条件 (タイムアウトや最大世代数) を満たすまで，この加工・ビルド・テスト・選択のループを繰り返す．

遺伝的アルゴリズムを利用している GenProg にとって，各世代で生成する個体の数が多いほど探索範囲は広がる．そこで問題となるのがパフォーマンスである．GenProg は生成した全ての個体に対してビルドとテストを行う．ビルドやテストは大きな計算コストを必要とする処理であり，修正済みプログラムを生成するまでのループの回数が多い場合に GenProg の実行時間は非常に長くなってしまう．著者らが GenProg の Java 実装である kGenProg [5] を Apache Commons Math の欠陥に適用したところ，実行時間の 90%をビルドとテストが占めていた．このようにビルドとテストの時間が長いため，GenProg の実行時間が長くなってしまい，GenProg の改良を目指している研究者が思いついたアイデアを試す際に，長い時間が必要となる等の問題が生じる．

本論文では GenProg のパフォーマンスを上げる方法として，各世代において生成された個体のビルドとテストを並列化するツールを提案する．ただし，単に並列化するだけでは実行するコンピュータのスペックが上限となってしまい，すぐに頭打ちとなってしまう．そこで提案ツールは，より高いパフォーマンスを実現するため，クラスタコンピューティングを行う．GenProg のビルドとテストを行うクラスタを構築し，それらの処理を分

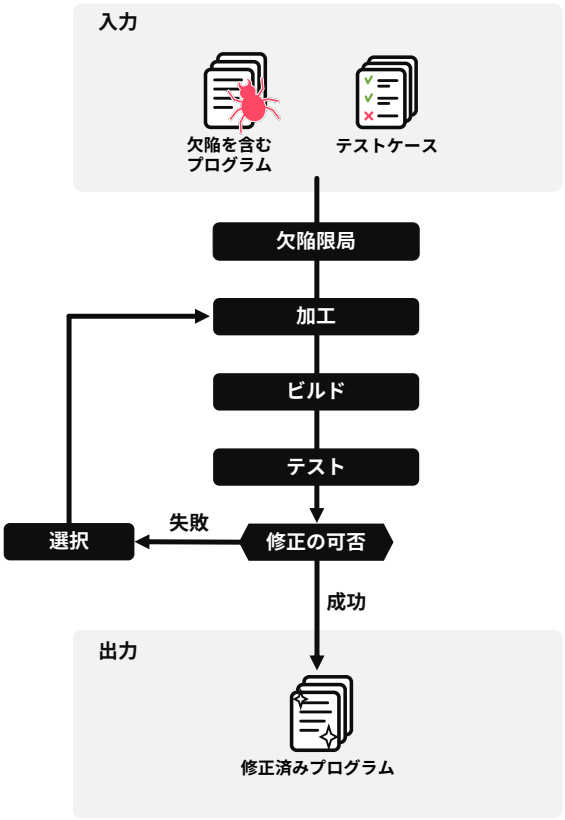


図 1　GenProg の処理のフロー

散・並列化させることで GenProg の高速化を行う．

提案ツールのパフォーマンスを評価するために，クラスタを構成するマシンの台数を変えて実験を行い，実際にパフォーマンスが向上したことを確認した．

なお提案ツールは OSS として，GitHub(注1)に公開されている．

2. 準　　備

2.1 欠 陥 限 局

デバッグを支援する手法として欠陥限局の研究が行われている．欠陥限局とは，入力としてプログラムとテストケースを受け取り，テストケースを実行した際の実行経路情報であるカバレッジからプログラムのどの箇所に欠陥が存在するのか推測する技術である．Aberu らは 7 つの欠陥限局の手法を比較し，Ochiai [6] が優れた手法であると結論づけている [7]．

2.2 GenProg

GenProg は遺伝的アルゴリズムを用いた自動プログラム修正の手法であり，ソースコードを個体として扱う．図 1 に GenProg の処理の概要を示す．まず，入力で与えられたプログラムとテストケースを用いて欠陥限局を行う．欠陥限局によって欠陥があると推測された箇所に対して，プログラム文の挿入・削除・置換といった加工を行い，新たな個体を複数生成する．生成された個体に対して，ビルドとテストを行い，全てのテストに通過する個体があればそれを修正済みプログラムとして出力する．生成された個体が全てのテストに通過できなかった場合，テスト通過率の高い個体を選択し，それらの個体に対して加工，ビルド，テストを行う．このループをテストに通過する個体が生成されるまで行う．

挿入もしくは置換を行う場合は，利用するプログラム文を入力として与えられたプログラムから選択する．また，どの操作を適用するか，どのプログラム文を利用するかはランダムに選択する．

2.3 kGenProg

kGenProg は GenProg を Java で実装したツール(注2)である．kGenProg は高拡張性，高処理効率性，高可搬性を備えている．欠陥限局やプログラムのビルドとテスト，個体の加工やその評価などといったコンポーネントがインターフェースとして切り出されているので，柔軟に処理を拡張できる特徴がある．また，ビルドとテストをメモリ上で行い，差分ビルドを行うため実行効率も高いのが特徴である．

GenProg とは異なり，kGenProg は加工するたびに欠陥限局を行うため，kGneProg は欠陥限局，加工，ビルド，テスト，選択を繰り返す．

2.4 Kubernetes

Kubernetes(注3)(以降 k8s と呼ぶ) は，自動デプロイ，スケーリング，ヘルスチェックなどを行うことができる．k8s を利用するには，まず master ノードとなるマシンを用意し，その master ノードに複数台のマシンを slave ノードとして登録する必要がある．k8s は slave ノード上に Docker のコンテナを設定された数だけ展開し，アプリケーションを実行することができる．例えば 10GB のメモリを持った slave ノードが 3 台登録された時に，k8s は 2GB のメモリを必要とするアプリケーションを 15 台起動することができる．また，k8s は各コンテナのヘルスチェックを行い，各コンテナが正常に実行されているか監視する．正常に実行されていないコンテナがあった場合，それを自動で破棄し，新しいコンテナを立ちあげる．

3. 提案ツール

クラスタコンピューティングを利用しビルドとテストの処理を分散させることで，自動プログラム修正のパフォーマンスを上げるツールを提案する．提案ツールは kGenProg に対して拡張を行う．kGenProg はビルドとテストの実行がインターフェースとして切り出されているため，ビルドとテストの実行を分散させるクラスタコンピューティングを組み込みやすいことが kGenProg を用いた主な理由である．

提案ツールの概要を図 2 に示す．提案ツールは，三種類のノードから構成される．

Client:　ビルドとテストを kGenProg から受け付ける．

Worker:　ビルドとテストを実際に行う．Worker はクラスタ構成の中に複数存在する．

Coordinator:　Client と Worker の中継，および負荷分散を行

(注1)：https://github.com/kusumotolab/clustered-kGenProg

(注2)：https://github.com/kusumotolab/kGenProg

(注3)：https://kubernetes.io

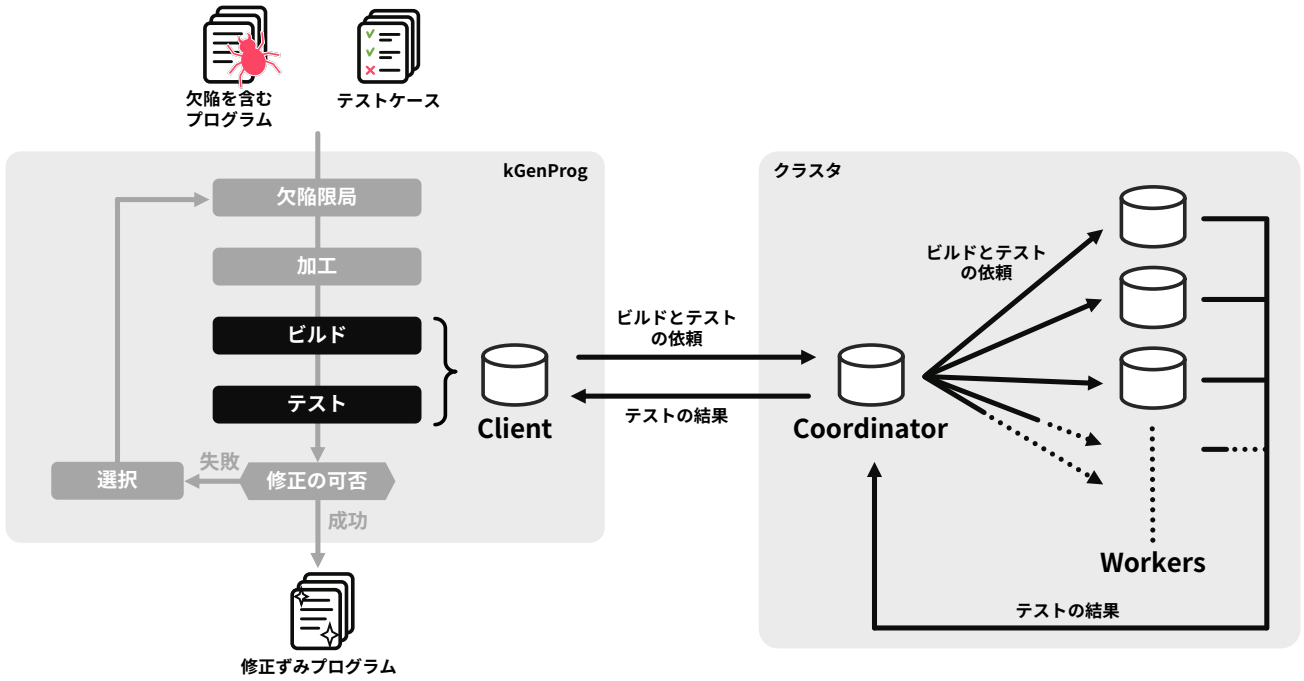


図 2　提案ツールの概要

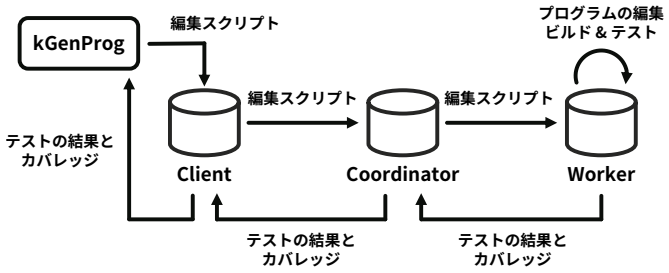


図 3　各個体の流れ

う．Coordinator はクラスタ構成の中に 1 つのみ存在する．三種類のノードは TCP/IP プロトコルに則って通信を行う．

kGenProg が編集スクリプトを生成し，それを Worker に伝え，テストの結果とカバレッジが返ってくる流れを図 3 に示す．編集スクリプトとは，入力されたプログラムからテスト対象のプログラムを得るまでの編集操作の列のことである．kGenProg はプログラムを加工する際，入力されたプログラムに対する編集スクリプトを計算する．その編集スクリプトを Worker に伝え，編集スクリプトに基づいてプログラムを編集することで，Worker はテストするべきプログラムを得る．その後，Worker はプログラムに対してビルドとテストを行い，テストの結果とカバレッジを Coordinator を経由して kGenProg に返す．

3.1　通信の流れ

図 4 に通信のフローを示す．

（1） Worker は起動時に Coordinator に通知を送り Coordinator に Worker の IP アドレスと Port 番号を認識させる．

(1) はクラスタの構築時に実行される．

（2） Client は修正するプログラムのビルドとテストに必要なファイルを Coordinator に転送する．

（3） Coordinator は受け取ったファイルを登録されている

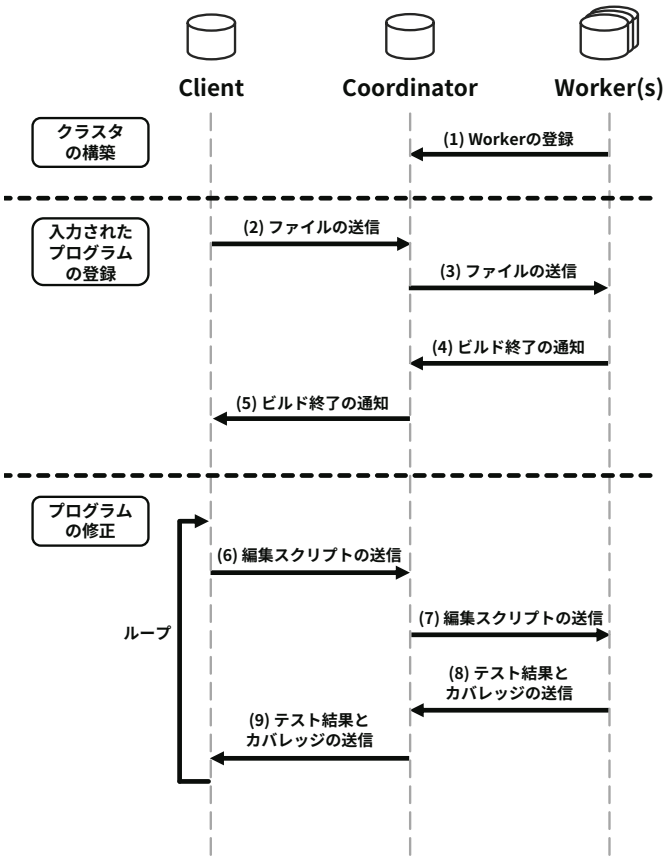


図 4　通信のフロー

Worker に配布する．

（4） Worker は受け取ったファイルに対して初回ビルドを行う．

（5） 全ての Worker が初回ビルドを終われば，Coordinator が Client に対してビルドを終わったことを通知する．

(2)〜(5) は kGenProg の起動時に実行される．(5) が終わった後，kGenProg は複数の編集スクリプトを計算し，プログラムの修正を行う．

（6） kGenProg から編集スクリプトを受け取った Client はそれを Coordinator に送る．

（7） 編集スクリプトを受け取った Coordinator は処理をしていない Worker に対して，編集スクリプトを送る．

（8） 編集スクリプトを受け取った Worker はビルドとテストを行い，その結果を Coordinator に送る．

（9） テスト結果を受け取った Coordinator はその結果を Client に送る．

(6)〜(9) は複数の Worker で並列して行う．kGenProg が修正済みプログラムを得るまで，修正対象のプログラムに対して編集スクリプトを計算し，(6)〜(9) の処理を繰り返し行う．

ネットワークの通信には gRPC(注4) を用いて行う．gRPC とは，Google が開発している Remote Procedure Call である．gRPC を用いた主な理由は，HTTP 通信と比べて高速である上，kGenProg に柔軟に取り入れることができるためである．

3.2 Client

ビルドとテストを行うインターフェースとして Client を kGenProg に組み込む．kGenProg がビルドとテストを行う際には，Client に処理を依頼する．Client が用いる情報は以下の二種類である．

- 修正対象のプログラムをビルドし，テストするために必要なファイル
- 修正対象のプログラムに対する編集スクリプト

Client はビルドに必要なファイルを zip にまとめて Coordinator に送信する．送信したファイルに対応する ID を付与したレスポンスを Coordinator から受け取るので，以降の通信ではその ID を添えて行う．

kGenProg から編集スクリプトを受け取った Client はそれを Coordinator に送信すると，レスポンスとしてビルドの結果とカバレッジを受け取る．

3.3 Coordinator

Coordinator はビルドとテストを依頼する Client とそれを実行する Worker との中継，および負荷分散をするノードである．Coordinator の機能は以下の通りである．

- Worker の登録
- 入力されたプログラムの登録
- ビルドとテストの依頼の受け付け

3.3.1 Worker の登録

Coordinator は常に Worker の登録を受け付ける．Worker から起動の通知を受け取れば，その Worker に対してコネクションを繋ぐ．

3.3.2 入力されたプログラムの登録

Coordinator は常に Client からの入力されたプログラムの登録を受け付ける．送信されたプログラムに対して Coordinator は ID を付与して，現在 Coordinator に登録されている全ての Worker に対してそのプログラムに関するファイルと ID を送る．全ての Worker に対してファイルを送り終えると，そのプログラムに対応する ID をレスポンスとして Client に返す．

なお，プログラムを登録した後に Worker が登録されることもあるため，Coordinator は ID とプログラムの対応関係を保持し続ける．

3.3.3 ビルドとテストの受付け

Coordinator は Client からのビルドとテストの依頼を受け付ける．Client からプログラムと対応する ID，及び編集スクリプトを含んだリクエストを受け取り，そのリクエストを処理をしていない Worker に送信する．

Coordinator はどの Worker にビルドとテストを依頼したかを把握し，ビルドとテストを終えた Worker に対して次のリクエストを送信する．全ての Worker がビルドとテストをしている場合，Client から受け取ったリクエストを保持し，Worker の処理が終わるのを待つ．Worker の処理が終わったらリクエストを送信し，ビルドとテストを依頼する．また，ビルドとテストの実行中に Worker との接続が切れてしまった場合は，処理をしていない別の Worker に依頼をする．

3.4 Worker

Worker はビルドとテストを実行するノードである．Coordinator からプログラムに関する ID と編集スクリプトを受け取る．Worker は送られてきた編集スクリプトに基づいてプログラムを編集し，ビルドとテストを行う．初回のビルドは全てのファイルに対して行うが，2 回目以降のビルドでは差分ビルドを行うことで，ビルドの高速化を図る．

ビルドに成功し，テストが終われば，全てのファイルに関してのカバレッジを Coordinator に送る．カバレッジを送るのは，kGenProg が欠陥限局する際に必要な情報だからである．

3.5 デプロイ方法

それぞれのノードを手動で起動し，提案ツールを実行することはもちろん可能であるが，Worker の台数が多いと，その管理は煩雑になってしまう．そこで今回はデプロイをする方法として，k8s を用いた．Coordinator と Worker を Docker のイメージとして用意し，k8s の設定ファイルを変更するだけで，ノードの数を増やしたり減らしたりすることができる．k8s によって各 Worker はヘルスチェックを定期的に行われ，予期せぬ障害 (例えばメモリが不足する，ネットワークが途絶える等) が発生して Worker が処理を続けることができなければ，その Worker が動いているコンテナを再起動させ，Worker と Coordinator を再接続させる．こうすることで，常に Worker の台数を設定した台数に保つことが可能である．

3.6 使 用 方 法

k8s を使わずに提案ツールを実行するコマンドを図 5 (a) に示す．提案ツールを利用するには，まず Coordinator を立ち上げる必要がある．その後，Worker を立ち上げて Coordinator の登録をする．Worker の起動は起動する台数分実行する必要がある．

k8s を導入する場合，よりシンプルに実行することができる．k8s を用いた実行のコマンドを図 5 (b) に示す．k8s の設定が終わった状態であれば，起動する worker の台数に関わらず，コマ

(注4)：https://grpc.io

```
# Coordinatorを実行するマシンで実行
$ coordinator --port 50051

# Workerを実行するマシンで実行
# (必要な回数実行)
$ worker --host <Coordinator IP> --port 50051

# クライアントを実行するマシンで実行
$ client --host <Coordinato IP> --port 50051 \
         --kgp-args '--config kGP.toml'
```

(a) 全て手動での実行方法

```
# k8sのmasterノードで実行
$ kubectl apply -f kubernetes/deploy.yml

# クライアントを実行するマシンで実行
$ client --host <k8s master IP> --port 50051 \
         --kgp-args '--config kGP.toml'
```

(b) k8s を用いた実行方法

図 5　提案ツールの利用の流れ

ンド一つクラスタを起動することができる．起動したい Worker の台数や，各ノードの CPU やメモリのスペック，どのポートを使うかなどは全て deploy.yml に記入されている．Worker の台数を変更したい場合はこのファイルを編集して，コマンドを実行するだけで良い．

4. 評　　価

OSS に対して提案ツールを適用し，そのパフォーマンスを評価する．提案ツールは Worker の台数を増やすことでパフォーマンスが向上すると考えられる．そのため，本実験では Worker の台数を変えて，パフォーマンスがどのように変化するか確かめる．

4.1 実 験 対 象

本実験では Defects4J [8] に含まれる Apache Commons Math プロジェクト（以降，Math）を実験題材として用いる．Math には 106 個の欠陥の情報が含まれており，いずれも実際の Math 開発の中で発生した欠陥である．Math を題材として選定した理由は，自動プログラム修正の論文の多くでベンチマークとして利用されているからである [9] [10].

4.2 実 験 環 境

6 台の IaaS サーバーを用いて実験を行なった．この IasS サーバーは全て 12 個の CPU，100GB のメモリを持つ．IasS サーバーと同様のスペックを持つ仮想マシンを各 IaaS サーバー上に立ち上げ，6 台の仮想マシンを slave ノードとして k8s に登録した．この slave ノード上に Coordinator，及び複数台の Worker を立ち上げ，クラスタを構築した．

実験を行う際の各ノードのスペックを表 1 に示す．Client は kGenProg のプログラムにインターフェースとして組み込むため，Client のスペックは示していない．

表 1　各ノードのスペック

Node	CPU	Memory
kGenProg	2	32GB
Coordinator	2	16GB
Worker	1	6GB

4.3 実 験 設 定

Worker の台数が増えれば増えるほど，同時にビルドとテストできる数が増え，パフォーマンスの向上が見込める．そこで，本実験では kGenProg 単体，及び 1, 4, 16, 32 台の Worker を起動した状態の提案ツールで実験を行い，そのパフォーマンスの比較を行なっていく．

提案ツールは各世代に生成する個体のビルドとテストを分散させる．各世代において生成するプログラムの数が少ないと処理を分散させることのメリットが低く，提案ツールを十分に評価できない．そのため，実験の設定として各世代に生成する数を kGenProg のデフォルトの値である 10 よりも大きい 1,000 とした．また，実験のタイムアウトを 30 分、最大世代数を無制限とした．

kGenProg の個体の生成と選択は乱数に基づいて行われる．この乱数はシード値に基づいて決定されるため，シード値が変われば実験結果も変わり，修正済みプログラムの個数が変わる．修正能力を評価するためには複数のシード値で実験を行う必要がある．しかし本実験は kGenProg の修正能力を評価するための実験ではなく，パフォーマンスを評価するための実験であるため，このシード値は 1 に固定して実験を行った．

4.4 実 験 結 果

4.4.1 修正済みプログラムの数

kGenProg 単体、および Worker の台数を変えて実験し，修正済みプログラムを求めることができた数を表 2 に示す．kGenProg 単体でプログラムの修正を試みるより，複数台の Worker を起動した方が多くのプログラムを修正ができていることが分かる．ただし 4 台以上の Worker を起動した状態では修正できた数が全て同じとなった．

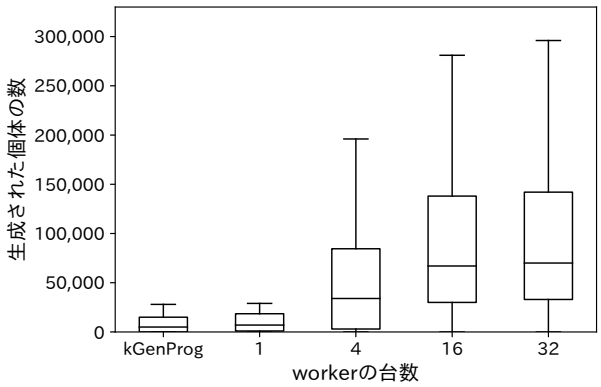
4.4.2 タイムアウトまでに生成した個体数

図 6 (a) は修正済みプログラムを見つけることができなかった欠陥に対して，タイムアウトまでに生成した個体の数を箱ひげ図で表している，1 台より 4 台，4 台よりも 16 台の方が多くの個体を生成できていることがわかる．

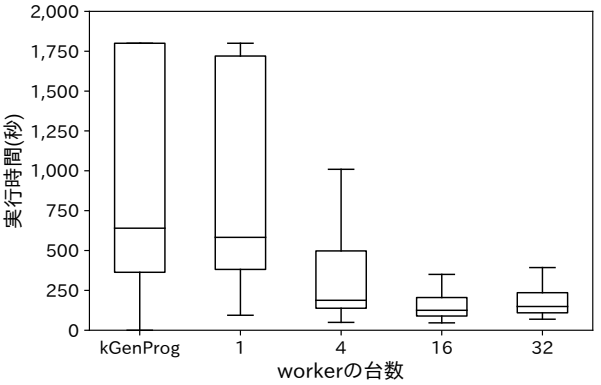
一方，16 台と 32 台の間で大きな差は見られなかった．16 台以上で実行した場合，分散していない処理が支配的になってきたことが原因と考えられる．特に欠陥限局は，全てのテストケースに対して全てのプログラムの全ての行に対して計算を行う必要があり，計算量が膨大である．ビルドとテストが分散されることによってビルドとテストの実行時間が短くなり，結果としてその

表 2　見つけた修正済みプログラムの数

kGenProg 単体	1 worker	4 workers	16 workers	32 workers
25	29	36	36	36



(a) 生成した個体の数



(b) 修正済みプログラムが見つかるまでの時間

図 6　実 験 結 果

他の処理が支配的になり，16 台と 32 台では差が明確に出なかったと考えられる．

4.4.3　修正済みプログラムを見つけるまでの時間

図 6 (b) は 1 ～ 32 台のどれか一つでも，修正済みプログラムを生成した欠陥に対しての実行時間を箱ひげ図で表している．1 台よりも 4 台，4 台よりも 16 台の方が実行時間が短くなっていることがわかる．

また，修正済みプログラムを見つけるまでの時間は 16 台よりも 32 台の方が長い結果となった．実行の最初にプログラムを転送するには 16 台より 32 台の方が長い時間必要である．このことから，16 台から 32 台に分散させて短縮された実行時間より，16 台から 32 台にした際に伸びた転送時間の方が長いことが原因と考えられる．

4.4.4　kGenPorg 単体と 1 台の Worker との比較

kGenProg を単体で動かしている時より，1 台の Worker で処理をする方が通信のオーバーヘッドが発生する．しかし，kGenProg を単体で実行した結果と提案ツールを 1 台の Worker で実行した結果との間に，実行時間・生成した個体の数について明確な差は見られなかった．そのため，本実験の環境では通信のオーバーヘッドが発生する欠点より，処理を分散することでパフォーマンスを上げる利点の方が大きいことがわかる．

5. ま と め

遺伝的アルゴリズムに基づいた自動プログラム修正ツールである GenProg のビルドとテストのパフォーマンスを上げるツールを提案した．提案ツールは，ビルドとテストの処理を分散させることで，パフォーマンスの向上を図る．実際のプロジェクトに対して提案ツールを適用させ，kGenProg を単体で実行した時より，高速に実行することができた．

今後の課題として，この提案ツールを利用したより優れた個体の加工や選択の手法を提案していきたい．提案ツールを利用すれば実行時間を短くすることができるため，GenProg のような文単位の加工ではなくて，AST ノード単位の加工なども可能になるのではないかと考えている．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：17H01725）の助成を得て行われた．

文　献

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software - quantify the time and cost saved using reversible debuggers,” 2013.

[2] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, no.1, pp.4–12, 2002.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” In Proc. International Conference on Software Engineering, pp.364–374, 2009.

[4] C. Le Goues, M. Dewey Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” In Proc. International Conference on Software Engineering, pp.3–13, 2012.

[5] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, “kgenprog: A high-performance, high-extensibility and high-portability apr system,” the 25th Asia-Pacific Software Engineering Conference, pp.697–698, Dec. 2018.

[6] A. daSilva, Meyer, A. Augusto, Franco Garcia, A. Pereira, de Souza, and C.L. deSouza, Jr., “Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l),” Genetics and Molecular Biology, vol.27, no.1, pp.83–91, 2004.

[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. vanGemund, “A practical evaluation of spectrum-based fault localization,” Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.

[8] R. Just, D. Jalali, and M.D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” In Proc. International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[9] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset,” Springer Empirical Software Engineering, vol.22, no.4, pp.1936–1964, 2016.

[10] M. Martinez and M. Monperrus, “ASTOR: A program repair library for java,” In Proc. International Symposium on Software Testing and Analysis, pp.441–444, 2016.