# Madoop: MapReduceとWebAssemblyを用いたWebブラウザベース分散処理フレームワーク

松尾 裕幸[1,a)]　柗本 真佑[1,b)]　楠本 真二[1,c)]

**概要**：自由参加型の分散処理パラダイムである従来の VC: Volunteer Computing を Web ブラウザ上に拡張した，BBVC: Browser-Based Volunteer Computing というアイディアが登場している．BBVC では，コードを埋め込んだページにブラウザからアクセスするだけで参加可能という手軽さに加え，世界中のインターネットユーザが対象になるという凄まじい潜在的処理能力が存在する．しかしながら，BBVC には幾つかの課題が存在する．課題の一つ目に分散処理プログラムの開発容易性の低さが挙げられる．分散処理の実現には，入力データの適切な配信やクライアント間の同期など様々な事柄を考慮しなければならず，開発容易性を下げる一因となっている．課題の二つ目に実行時性能の低さが挙げられる．BBVC ではその特性上 JavaScript で処理を行うが，JavaScript は実行時にソースコード解析を必要とするため，コンパイラ型言語と比べて性能が低く数値計算には向いていない．本稿では，これらの課題の解決を目的とした新しい BBVC フレームワーク Madoop を提案する．Madoop では MapReduce と WebAssembly を用いることで，開発容易性の確保と実行時性能の改善を実現する．また評価実験として，提案手法を用いて実際に分散処理を実行し，実行時間の長さを計測・比較することで，従来手法からの性能の改善度合いを確かめる．実験の結果，最も性能が向上した実験対象では，実行時間の長さが 60 %以上改善した．

**キーワード**：Volunteer computing, web browser, MapReduce, WebAssembly, JavaScript

## 1. はじめに

技術の発展に伴い，個人が所有する計算機の処理性能は飛躍的に向上した．また，世界中で稼働している計算機の数も非常に多く，控えめに見積もった場合でも現在 1 億台以上の計算機が使用されている [5]．世界中で稼働している全計算機を束ねた総処理能力は凄まじいが，一般的な計算機はその起動時間の多くがアイドル状態であり，その余剰処理能力の殆どが無駄になっている [5]．

科学技術の分野において，この余剰処理能力に着目した分散処理手法が提案されている．これを Volunteer Computing（以下，VC）という．近年，この VC を Web ブラウザ上で実行するように拡張した Browser-Based Volunteer Computing（以下，BBVC）という手法も登場している．従来，VC に参加するためには特別なクライアントアプリケーションのインストールが必要不可欠であった．一方，BBVC はユーザにクライアントアプリケーションのインストールを要求せず，多くのコンピュータに予めインストールされている Web ブラウザで特定の URL にアクセスするだけで動作させることができる．従って，BBVC は世界中のネットサーフィンユーザが対象であり，そこには莫大な潜在的処理能力が存在する [11]．

しかしながら，BBVC には幾つかの課題が存在する．一つ目の課題として，分散処理プログラムそのものの開発容易性の低さが挙げられる．一般に分散処理を行うためには，入力データの適切な配信や，クライアントノード間の同期，計算結果の統合などを行う仕組みが必要となる [5], [11]．開発者は実際に処理対象となるプログラムに加えてこれらの仕組みを用意する必要があり，その実装は開発者にとって非常に大きな負担となる．二つ目の課題に，処理言語である JavaScript の実行時性能の低さが挙げられる．JavaScript は実行時に動的なソースコードの解析を必要とするため，これがオーバヘッドとなりコンパイル型言語と比べてパフォーマンスに劣る．とりわけ，JavaScript は数値計算には向いていないことが指摘されている [9]．

本稿では，これらの課題を解決するための新しい BBVC フレームワーク Madoop を提案する．Madoop では，一つ目の課題の解決のために MapReduce を，二つ目の課題の解決のために WebAssembly をそれぞれ導入する．MapRe-

[1] 大阪大学 大学院情報科学研究科
[a)] h-matsuo@ist.osaka-u.ac.jp
[b)] shinsuke@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

duce [3] は分散処理パラダイムの一つであり，このパラダイムに沿ってプログラム開発を行うことで，開発者の負担が大きく削減される．WebAssembly [16] は Web ブラウザ上で実行可能なバイナリフォーマットであり，JavaScript が必要とした実行時解析によるオーバヘッドが大きく軽減されるため，パフォーマンスの向上が期待できる．

評価実験として，Madoop が従来手法と比べてどの程度パフォーマンスが向上しているのかを，計算量の大きさが異なる 2 つの実験対象を用いて，それぞれの実行時間の長さを計測することで確かめた．実験の結果，計算量が非常に大きい実験対象では，Madoop を用いた提案手法は従来手法よりも約 50 〜 64 %以上パフォーマンスが向上した．一方，計算量が小さい実験対象では，提案手法が従来手法よりも約 4 〜 8 %遅くなった．この実験結果から，提案手法は計算量が大きな分散処理を行う際に非常に有効であることが分かった．また，Madoop はオープンソースソフトウェアとして公開しており，第三者が自由に利用できる．現在 BBVC の処理系として公開されているソフトウェアは殆ど存在しておらず，BBVC プログラムを開発したい開発者にとって非常に有用であると考える．

## 2. 準備

### 2.1 VC: Volunteer Computing

VC は分散処理を行うための手法の一つである．現在広く用いられている分散処理システムでは，同じスペックの計算機を複数台用意し，それらを LAN で相互接続した上でクラスタを組むのが一般的である．これに対して，VC では計算機やそれらの構成が大きく異なる．VC における計算機（ノード）はインターネットに接続された不特定多数の様々な計算機である．一般的な計算機はその起動時間の多くがアイドル状態であり，この際の余剰計算能力をユーザ（ボランティア）から提供してもらうことでリソースを確保する．また，VC では各ノードが物理的に離れており，LAN ではなくインターネット越しに相互接続してシステムを構成する．

VC のアイディアは古くから存在しており，実際に VC を大規模活用した有名な例として SETI@home [13] が挙げられる．VC を用いて宇宙から飛来した電波を解析し，地球外知的生命体が発信した信号を検出する試みである．SETI@home ではこれまでに 170 万人以上が計算に参加しており，2018 年 8 月現在でも常時 9 万人以上が参加している [14]．これは VC が数値計算プラットフォームとして有用であることを示した代表例である．

VC の問題点として，ユーザ層が限定的であることが挙げられる．ここ数年では，VC 全体のユーザ数は増加することなく約 50 万人を横這いで推移している．これはインターネットに接続された全デバイス数の 0.02 %を下回っており，ユーザ数の増加は頭打ちになっていると言える [11]．VC に参加するためには，ユーザは専用のクライアントアプリケーションをインストールする必要があり，ユーザに対するハードルとなっていると考えられる．また，複数のオペレーティングシステムの台頭やタブレット・スマートフォンの普及など，ユーザの計算環境はますます多様化している．VC では各ユーザの環境に合わせたクライアントアプリケーションを開発する必要があり，開発者に対するハードルとなっていると考えられる [18]．

### 2.2 BBVC: Browser-Based Volunteer Computing

VC を Web ブラウザ上に拡張したのが BBVC である．VC が専用のクライアントアプリケーション上で動作していたのに対して，BBVC ではユーザが Web ブラウザで特定の URL にアクセスしている間に，その Web ブラウザ上で処理が行われる．

一般的に，ユーザが利用する計算機には予め Web ブラウザがインストールされていることが多いため，BBVC に参加するために特別なアプリケーションをインストールする必要はない．よって，BBVC は世界中のネットサーフィンユーザが対象となり，その潜在的な計算能力は測り知れない．例えば，YouTube にアクセスしている各ユーザの Web ブラウザ上で，計算機の 25 %の処理能力を BBVC に利用できると仮定した場合，その総処理能力は 46.4 PFLOPS にも上ると試算されている [11]．2017 年 4 月時点での世界最速のスーパーコンピュータである Sunway TaihuLight の処理能力が 93 PFLOPS であることを考慮すると，YouTube への訪問者を束ねるだけでもスーパーコンピュータの約半分の処理能力に値すると言える [11]．

また，開発者はクライアントアプリケーションを Web アプリケーションとして開発すればよく，計算機のオペレーティングシステムやその他のアーキテクチャの多くは Web ブラウザが隠ぺいするため，VC に比べると多様化するユーザの計算環境に対応するコストも低くなる．このことから，BBVC では前節で挙げたような VC の課題を解決できると考えられる．

### 2.3 BBVC の課題

前節で述べたように，BBVC は VC の課題のいくつかを解決した一方で，以下に挙げる BBVC 固有の課題も存在する．

#### 2.3.1 $P_1$: 分散処理プログラムの開発容易性の低さ

分散処理を実現するためには，処理対象データの適切な分割・配信や，クライアントノード間の同期，処理結果の統合，処理失敗時の復帰を始めとする，拡張性や可用性，耐障害性などを確保するための複雑な仕組みを要する [5], [11]．これらは分散処理そのものを実現するために必要な仕組みであるが，その実装は開発者にとって非常に大きな負担と

なる．分散処理の実装を支援する技術として，次章で述べる MapReduce パラダイムが有名だが，Web ブラウザ上で行う分散処理のために利用した例は少なく，またライブラリとして利用できる形に公開されているものも少ない．結果として，BBVC を行うためのプログラムを開発者が実装する際，現状では利用できる分散処理フレームワークの選択肢がないため，前述したような考慮事項を念頭に置きながら自分で開発しなければならない．これは開発者にとって大きな負担であり，BBVC の普及を妨げる要因となっていると言える．

#### 2.3.2 $P_2$: JavaScript の実行時性能の低さ

BBVC では，Web ブラウザ上で処理を行うという性質上，これまで処理言語としては実質的に JavaScript 以外の選択肢が存在しなかった．しかしながら，JavaScript は動的型付けのスクリプト言語であり，静的型付けのコンパイル言語と比べると，実行時解析のオーバヘッドのためにパフォーマンスが劣る．2008 年に Google Chrome の JavaScript エンジンである V8 に Just-In-Time（JIT）コンパイラが導入されてからパフォーマンスは改善されてきているものの，依然としてコンパイル言語と比べると処理速度は遅いと言える．例えば，数値計算タスクの実行速度を Java と JavaScript とで比べた場合，JavaScript の速度は Java よりも約 30 %遅いことが実験から指摘されている [9]．

## 3. 提案手法: Madoop

### 3.1 概要

本稿では，前節で挙げた BBVC の課題の解決を目的とした新しい BBVC フレームワーク Madoop[*1] を提案する．本提案手法では，課題 $P_1$ の解決に MapReduce を，課題 $P_2$ の解決に WebAssembly をそれぞれ導入する．これにより，開発容易性を確保しつつ，クライアントノード上での実行時性能が向上することが期待できる．

### 3.2 採用技術

#### 3.2.1 MapReduce

Madoop では，2.3.1 項で述べた課題 $P_1$: 分散処理プログラムの開発容易性の低さを解決するため，MapReduce を導入する．

MapReduce [3], [6] は Google により提起された分散処理パラダイムの一つである．MapReduce は分散処理における一連の操作を map と reduce の 2 つの関数に抽象化している．各ノードは入力データと map/reduce 関数を受け取り，それぞれ処理する．処理の概要は次の通りである．

(1) MapReduce の処理系が入力データを適切な単位に分割する．
(2) 各ノードが入力データと map 関数を受け取る．map 関数は中間値である (key, value) のペアの集合を出力する．
(3) MapReduce の処理系が各 key に対して，同一の key を持つすべての value を集め，中間値である (key, value 列) のペアの集合を作成する．
(4) 各ノードが (key, value 列) のペアと reduce 関数を受け取る．reduce 関数は value 列を集約し，その結果を出力する．

開発者が実際に開発するのは map/reduce 関数のみであり，2.3.1 項で挙げた分散処理に伴う諸考慮事項は処理系に任せればよい．これにより，MapReduce は分散処理プログラムの開発を容易にしたと言える．また MapReduce のアイディアは有名であるため，Madoop がこれに対応することで，これまで MapReduce を用いて分散処理プログラムを開発してきた開発者は，低い学習コストで Madoop を使い始めることができる．

MapReduce の実装としては Apache Hadoop [1] が有名だが，Hadoop は Java で記述されており，そのまま Web ブラウザ上で動作させることはできない．また JavaScript で記述された MapReduce 実装の例としては MRJS [12] や JSMapReduce [8] などが挙げられるが，いずれも実装を公開しておらず，第三者が利用することができない．Madoop はオープンソースソフトウェアとして公開しているため，第三者が利用することができ，BBVC を行いたい開発者にとって有用であると考える．

#### 3.2.2 WebAssembly

2.3.2 項で述べた課題 $P_2$: JavaScript の実行時性能の低さを解決するため，WebAssembly を導入する．

WebAssembly [16] は Web ブラウザ上で実行可能なバイナリフォーマットである．厳密にはスタックベースの仮想マシン用バイナリフォーマットとして設計されており，この仮想マシンが各種 Web ブラウザに搭載されていると考えればよい．WebAssembly 形式へのコンパイルにはいくつかの言語が対応しており，主要なものでは C/C++が挙げられる．2018 年 8 月現在，主要なモダンブラウザはすべて WebAssembly を実行可能であり[*2]，全ユーザが利用するブラウザの約 75 %がサポートしている [2] ため，多くのユーザの環境で利用できると考えてよい．

WebAssembly は Java に置き換えて考えると分かりやすい．**表 1** は WebAssembly と Java を比較したものである．Java では，記述したプログラムを `javac` でコンパイルす

[*1] 著者名の **M**atsuo と MapReduce の実装で有名な Apache **Hadoop** から命名した．Madoop はオープンソースソフトウェアとして公開している．
`https://github.com/h-matsuo/madoop/`

[*2] Microsoft Edge, Mozilla Firefox, Google Chrome, Safari, Opera のすべてがサポートしている．また，モバイルブラウザでも iOS Safari, Chrome Android の両方がサポートしている．

表 1　WebAssembly と Java の比較

| 項目 | WebAssembly | Java |
|---|---|---|
| プログラミング言語 | C/C++など | Java |
| コンパイラ | Emscripten など | `javac` |
| コンパイラによる生成物 | `.wasm` ファイル | `.class` ファイル |
| 実行環境 | Web ブラウザ | JVM |

ると，中間言語で記述されたクラスファイル（`.class`）に変換される．コンパイル済みのクラスファイルを実行するのは仮想マシン JVM（Java Virtual Machine）であり，JVM を様々な環境に合わせて用意することで，同じ Java プログラムを多様な環境で動作させることができる．同様に，WebAssembly では，C/C++などで記述されたプログラムを Emscripten [4] と呼ばれるツールなどを用いてコンパイルすることで，中間形式であるバイナリフォーマット（`.wasm`）が得られる．このバイナリフォーマットは Java のクラスファイルに相当する．変換後のバイナリフォーマットは各種 Web ブラウザが実行する．ここでは，Web ブラウザが Java の JVM に相当する．このように，バイナリフォーマットを Web ブラウザが直接解釈するため，C/C++プログラムを様々な環境に合わせてコンパイルし直す必要がない．WebAssembly 向けに一度コンパイルすれば，多様な環境の Web ブラウザ上で実行させることができる．

WebAssembly を導入することの最大の利点は，パフォーマンスの改善にある．前述したように，WebAssembly を用いることで Web ブラウザはコンパイル済みのバイナリを解釈・実行すればよいため，コンパイラ型言語と同等の性能を期待することができる．実際，同じ内容の数値計算をネイティヴアプリケーションとして実装したものと，WebAssembly を用いて実装したものを実行時間の長さで比較した場合，WebAssembly はネイティヴアプリケーションと遜色ないパフォーマンスを発揮したというベンチマーク結果もある [7]．

### 3.3　アーキテクチャ

Madoop のアーキテクチャを**図 1** に示す．図には，太字で示した 3 種類のアクタ，および 2 種類のサーバが登場する．

**依頼者**　Madoop 上で分散処理を行いたい者．

**コンテンツ投稿者**　ブログ記事などの Web コンテンツを投稿する者．

**Web サイト訪問者**　Web ブラウザを用いて，コンテンツ投稿者が投稿した Web コンテンツにアクセスする者．

**メインサーバ**　Madoop がインストール・セットアップされたサーバ．

**コンテンツサーバ**　コンテンツ投稿者が投稿した Web コンテンツを保持・配信するサーバ．

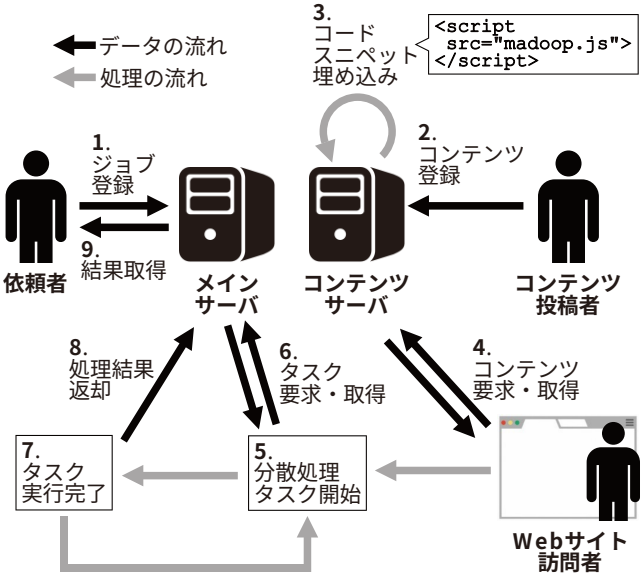


図 1　Madoop のアーキテクチャ

Madoop を用いて分散処理のジョブを登録してから処理結果が返却されるまでの一連の流れを説明する．なお，リストの番号は図中の番号と対応している．

(1) 依頼者が Madoop にジョブ（map/reduce 関数および処理対象データ）を登録する．このとき，map/reduce 関数は C/C++で記述し WebAssembly 形式にコンパイルしたものを登録することで，高パフォーマンスでの処理が期待できる[*3]．Madoop は登録されたジョブを適切な単位に分割し，タスクとして保持する．

(2) コンテンツ投稿者がブログ記事などの Web コンテンツを作成し，コンテンツサーバに投稿する．

(3) コンテンツサーバ上に保持されている Web コンテンツに，Madoop を用いた分散処理を行うためのコードスニペットが埋め込まれる．

(4) Web サイト訪問者が Web ブラウザでコンテンツサーバにアクセスし，HTML ファイルを始めとするコンテンツの閲覧に必要な各種データをダウンロードする．

(5) (3) で埋め込まれたコードスニペットにより分散処理が始まる．以降この Web ブラウザは，アクセスしたページに留まっている間はクライアントノードとして機能する．

(6) クライアントノードがメインサーバから処理対象タスク（map/reduce 関数および入力データ）を取得する．

(7) クライアントノード上でタスクを実行する．

(8) タスクの実行結果をクライアントノードから Madoop のメインサーバへ返却する．ページから離れるか，処理すべきタスクがなくなるまで (5)〜(8) を繰り返す．

(9) すべてのタスクが完了すると，依頼者は結果を取得できる．

[*3] 従来手法通り，JavaScript で記述した map/reduce 関数も登録できるように設計している．

```
1  void map(const char* data) {
2    // Count the number of occurrences of each word
3    std::unordered_map<std::string, long> hashmap;
4    const std::vector<std::string> &words =
5      split(data, '␣');
6    for (const auto &element: words) {
7      hashmap[element]++;
8    }
9    // Emit key-value pairs
10   for (const auto &element: hashmap) {
11     const std::string &key = element.first;
12     const std::string &value =
13       std::to_string(element.second);
14     emit(key, value);
15   }
16 }
```

図 2　ワードカウント用 map 関数の例（C++）

```
1  void reduce(const char* key,
2              const char* values_list) {
3    // Sum up values in the list
4    const std::vector<long> &values =
5      split(values_list, ',');
6    long sum = 0;
7    for (const auto value: values) {
8      sum += value;
9    }
10   // Emit key-value pair
11   emit(key, std::to_string(sum));
12 }
```

図 3　ワードカウント用 reduce 関数の例（C++）

上記 (1) で登録される map/reduce 関数の例をそれぞれ**図 2** と**図 3** に示す．ここでは，ジョブとしてワードカウントを想定し，それぞれの関数を C++で記述している．図 2 の map 関数は入力として文字列を受け取り，文字列に含まれる単語とその出現回数を出力する．4 行目で与えられた文字列をスペースで区切り，6 行目で各単語の出現回数をカウントする．その後 14 行目で，各単語について (key, value) = (単語, 出現回数) のペアを出力する．図 3 の reduce 関数では，入力として単語と出現回数のリストを受け取り，出現回数を足し合わせて単語と共に出力する．7 行目で与えられた出現回数のリストを足し合わせ，11 行目で (単語, 出現回数) のペアを出力する．このように，開発者は MapReduce パラダイムに沿って処理の本質となる部分のみを実装すればよく，それ以外の分散処理に必要な諸処理は Madoop が行うため，システム全体をフルスクラッチで実装する場合と比べて容易に開発することができる．

### 3.4　実装

Madoop のメインサーバは Node.js[*4] アプリケーションとして実装した．使用した言語は TypeScript[*5] である．また開発期間は約 8 ヶ月，コード行数は約 1,100 行である．

[*4] サーバサイド向けの JavaScript 実行環境．
https://nodejs.org/

表 2　メインサーバの環境

| 項目 | 説明 |
|---|---|
| インスタンスタイプ | t2.large[*7] |
| オペレーティングシステム | Ubuntu 16.04 LTS 64bit |
| Madoop のバージョン | 0.1.6 |
| Node.js のバージョン | 8.11.3 |
| TypeScript のバージョン | 2.9.2 |
| Emscripten のバージョン | 1.38.8 |

表 3　クライアントノードの環境

| 項目 | 説明 |
|---|---|
| インスタンスタイプ | m3.medium[*8] |
| オペレーティングシステム | Ubuntu 16.04 LTS 64bit |
| Google Chrome のバージョン | 67.0.3396.87 |
| Mozilla Firefox のバージョン | 60.0.2 |

## 4.　実験

### 4.1　概要

本実験の目的は，Madoop で WebAssembly を導入したことにより，従来の JavaScript で記述した場合と比べてパフォーマンスがどの程度向上するのかを確かめることである．具体的には，Madoop をインストール・セットアップしたメインサーバ，および Web ブラウザがインストールされた複数台の計算機を用意し，分散処理ジョブを実行してから完了するまでにかかった時間を計測・比較する．

なお，実験対象となるジョブには，計算量が大きく入出力データは小さい「レインボーテーブルの生成」と，計算量が小さく入出力データは大きい「ワードカウント」の 2 種類を用意した．これらの詳細は 4.3 節で述べる．

### 4.2　環境

#### 4.2.1　メインサーバ

**表 2** にメインサーバの環境の詳細を示す．メインサーバの計算機には Amazon EC2[*6] のインスタンスを用いる．インスタンス上に Madoop をインストールし，外部からアクセスできるように設定・公開する．また，簡単のため，本実験ではメインサーバがコンテンツサーバも兼ねる．

#### 4.2.2　クライアントノード

クライアントノードの環境の詳細を**表 3** に示す．クライ

[*5] Microsoft が開発したプログラミング言語．型の概念を追加した JavaScript のスーパセットであり，コンパイルすることで通常の JavaScript プログラムに変換できる．
https://www.typescriptlang.org/

[*6] Amazon Elastic Compute Cloud の略．Amazon.com, Inc. が提供する計算資源を利用して，クラウド上で仮想マシンを実行することができる．Amazon Web Services（AWS）のサービスの一つ．
https://aws.amazon.com/ec2/

[*7] CPU のコア数は 2，メモリサイズは 8 GiB である．

[*8] CPU のコア数は 1，メモリサイズは 3.75 GiB である．

表 4　実験対象ジョブのパラメータ

| | |
|---|---|
| 共通パラメータ | |
| クライアントノードの数 | 1, 2, 3, 5, 10 |
| 試行回数 | 10 回 |
| $E_1$: レインボーテーブルの生成 | |
| ハッシュ値の総計算回数 | 10,000,000 回 |
| 1 タスクあたりのハッシュ値の計算回数 | 1,000,000 回 |
| ハッシュアルゴリズム | MD5[*9] |
| $E_2$: ワードカウント | |
| テキストデータ[*10]の総容量 | 380 MB |
| 1 タスクあたりのテキストデータの容量 | 44 MB |

アントノードも，メインサーバと同様に Amazon EC2 のインスタンスを用いる．複数のインスタンスを同時に立ち上げて Web ブラウザを起動し，それぞれメインサーバへ接続する．

なお，パフォーマンスの計測においては，いかに計測時の誤差を削減するかが重要となる．今回の実験では Web ブラウザの操作が必要となるが，これを GUI で操作することは大きな誤差に繋がる．最近の Web ブラウザには，人手を介さずに操作を自動化することを目的とした「ヘッドレスモード」と呼ばれる実行モードが存在する．ヘッドレスモードは Web ブラウザを GUI なしに実行するモードであり，一般的には Web アプリケーション開発におけるテストの自動化などの用途で利用される．例えば，Google Chrome を用いて次のコマンドを実行することで，指定した`<URL>`のスクリーンショットがカレントディレクトリに保存される．

```
$ google-chrome --headless --screenshot <URL>
```

このように，ヘッドレスモードで実行した Web ブラウザでも，DOM の構築や要素の更新は通常通りに行われる．Madoop の要件は`<URL>`のリソースに含まれるプログラムを Web ブラウザ上で実行することであり，Web ページを GUI で表示する必要はないため，ヘッドレスモードで問題なく実験が可能である．今回は，ヘッドレスモードに対応している Google Chrome および Mozilla Firefox の 2 種類の Web ブラウザで実験を行う．

### 4.3　実験対象ジョブ

**表 4** に各実験対象のパラメータを示す．以下では各実験対象の詳細について述べる．

#### 4.3.1　$E_1$: レインボーテーブルの生成

レインボーテーブル [10] の生成は，Web サイトのログインパスワードに対する総当たり攻撃を効率良く行うための数値計算処理として知られる．セキュリティ上の理由から，Web サイト側のデータベースでは，ユーザのログインパスワードはハッシュ化した値を保存しておくことが多い．このハッシュ値が流出した場合，悪意のあるクラッカーが対応する元のパスワードを特定する方法として，パスワードとして有効な文字列を逐次ハッシュ化し，この値が入手したハッシュ値と一致するかどうかを総当たりで調べることが考えられる．レインボーテーブルは，このパスワード候補とそのハッシュ値の対応をメモリ効率が良くなるように計算して得られた表である．

map 関数は与えられた文字列（パスワード候補）をハッシュ化し，その値を還元関数と呼ばれる関数を用いることで，再びパスワードとして有効な文字列に戻す．さらにその文字列を再びハッシュ化する，といった処理を繰り返す．指定した回数だけこの処理を繰り返した後，最初の文字列と，一連の処理の結果として最終的に得られた文字列のペアを出力する．なお，reduce 関数の処理は特に存在せず，受け取った (key, value) の組をそのまま出力するだけである．

このジョブでは，メインサーバが配信する各ノードの入力データとしては，パスワードとして有効なある文字列と，ハッシュ値を計算する回数の 2 つのみであり，データ容量としては無視できるほど非常に小さい．一方で，各ノードは毎回大量のハッシュ値を計算するため，1 回あたりの計算量が非常に大きい．このように，1 タスクあたりの入力データが小さく，計算量は大きいのがこのジョブの特徴である．

#### 4.3.2　$E_2$: ワードカウント

ワードカウントでは，大容量のテキストファイルを解析し，テキスト内に存在する単語の一覧と，各単語の出現頻度を併せて出力する．MapReduce を用いたビッグデータ処理の題材として有名である．map/reduce 関数の内容は図 2，図 3 および 3.3 節で述べた内容と同一であるため，ここでは割愛する．

このジョブでは，各ノードの入力データは数 MB〜数十 MB と大きいが，実際に単語を数え上げる処理の計算量は少ないのが特徴であり，前項 $E_1$ のレインボーテーブル生成のジョブとは対照的である．

### 4.4　結果

#### 4.4.1　$E_1$: レインボーテーブルの生成

実験対象 $E_1$ の結果を**図 4** に示す．この箱ひげ図では，縦軸がジョブ全体の実行時間の長さ，横軸がクライアントノードの数をそれぞれ表す．黄色のデータが従来手法である JavaScript を用いた場合の結果を，紫色のデータが提案手法（Madoop）である WebAssembly を用いた場合の結

[*9] WebAssembly 形式の実装には，MD5 アルゴリズムの仕様が記述された RFC 1321 [15] の付録に記載されているプログラムを利用し，C++プログラムとして記述した．比較対象となる JavaScript の実装には，以下で公開されている RFC 1321 の仕様に忠実に従ったものを用いた．
`http://rocketeer.dip.jp/sanaki/free/javascript/freejs17.htm`

[*10] Yelp Open Dataset [17] のデータを用いた．

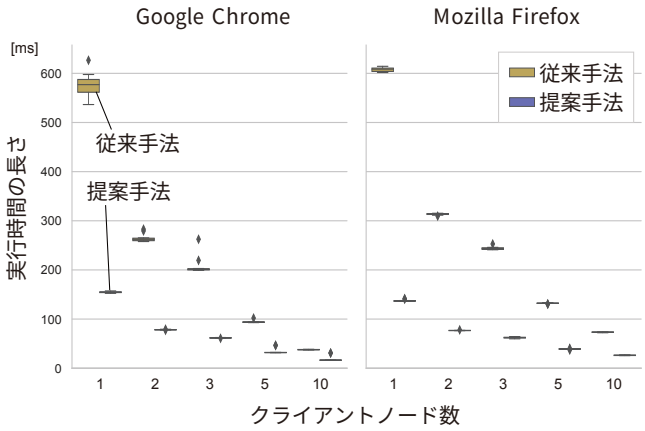


図 4　$E_1$: レインボーテーブル生成の結果

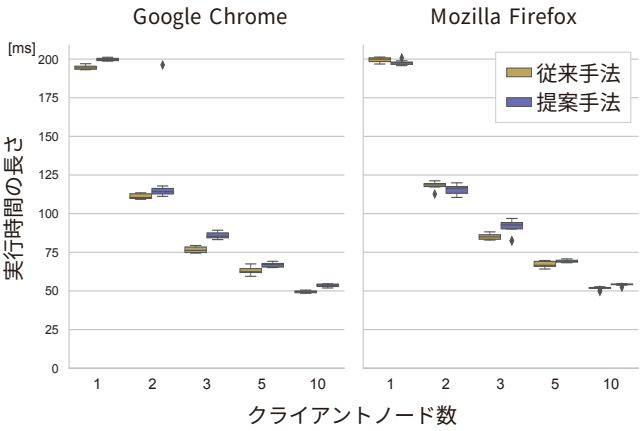


図 5　$E_2$: ワードカウントの結果

表 5　$E_1$: レインボーテーブルの生成における
ノード数 10 の結果の平均値

| 手法 | Google Chrome | Mozilla Firefox |
|---|---|---|
| 提案手法 | 19.2 s | 26.3 s |
| 既存手法 | 37.8 s | 73.2 s |
| 実行時間の削減比率 | 49.2 % | 64.1 % |

表 6　$E_2$: ワードカウントにおける
ノード数 10 の結果の平均値

| 手法 | Google Chrome | Mozilla Firefox |
|---|---|---|
| 提案手法 | 53.6 s | 54.0 s |
| 既存手法 | 49.5 s | 51.7 s |
| 実行時間の削減比率 | −8.3 % | −4.4 % |

果をそれぞれ示す．左側の図は Google Chrome での結果を，右側の図は Mozilla Firefox での結果をそれぞれ表す．図 4 より，クライアントノードの数 $n$ が増えるに従って，実行時間の長さはほぼ反比例して短くなっていることが分かる．また，いずれの結果においても，提案手法の方が従来手法よりも実行時間の長さが短くなっていることが見て取れる．さらに特筆すべき点として，$n = 5$ および $n = 10$ の結果に着目すると，従来手法における $n = 10$ のときよりも，提案手法における $n = 5$ のときの方が，実行時間の長さの方が短くなっていることが挙げられる．

**表 5** に，$n = 10$ の場合におけるジョブの実行時間の長さの平均値を示す．このように，提案手法と従来手法の実行時間の長さを比較した場合，Google Chrome では約 49 %，Firefox では約 64 %短くなった．

#### 4.4.2　$E_2$: ワードカウント

実験対象 $E_2$ の結果を**図 5** に示す．今回も同様に，ノードの数 $n$ に反比例して実行時間の長さは短くなっている．しかしながら，$E_1$ の結果とは異なり，今回の実験対象では，紫色の提案手法の方が黄色の従来手法よりも実行時間が長くなっている結果が多い．

**表 6** に，$n = 10$ の場合におけるジョブの実行時間の長さの平均値を示す．提案手法の方が従来手法よりも，実行時間の長さは Google Chrome で約 8 %，Mozilla Firefox で約 4 %増加した．

### 4.5　考察

実験対象 $E_1$: レインボーテーブルの生成において，提案手法を用いた場合 Google Chrome で約 49 %，Mozilla Firefox で約 64 % 実行時性能が改善した．このことから，

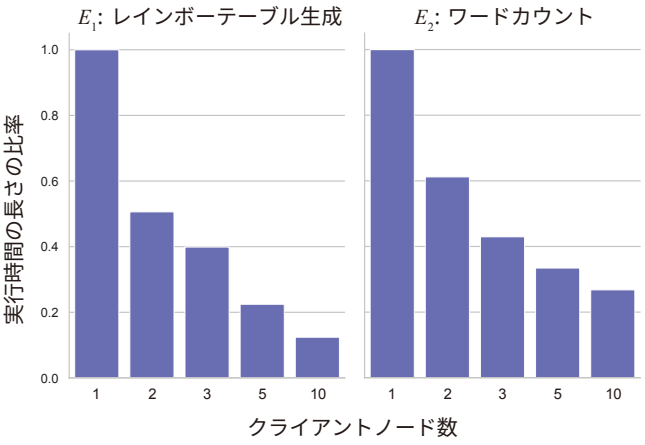


図 6　提案手法におけるクライアントノード数を増やした場合の実行時間の長さの比率

$E_1$ のような計算量が大きくデータ量が小さいジョブにおいて，提案手法が明らかに有利であることが分かる．

**図 6** は，提案手法において，クライアントノード数 $n = 1$ を基準とした場合における，$n$ を増加させたときの実行時間の長さの比率を表している．理論的には，$n = 5$ のときの比率は 1/5（0.2）となる．クライアントノードの数に応じて性能もスケールしており，特に実験対象 $E_1$ においては理論値に近い性能が出ていることが分かる．このことから，Madoop が分散処理システムのフレームワークとして正しく機能していることが見て取れる．

なお，実験対象 $E_2$: ワードカウントの結果の多くでは，提案手法を用いた場合の方が従来手法を用いた場合よりも実行時間の長さが増加するという結果になった．その増加比率は Google Chrome で約 8 %，Mozilla Firefox で約 4 %であった．この原因として，Web ブラウザが WebAssembly

バイナリを解釈・実行する際に発生するオーバヘッドが考えられる．Web ブラウザが WebAssembly バイナリを実行する手順の概要を以下に示す．

(1) バイナリデータ（.wasm ファイル）をサーバからダウンロードする．
(2) バイナリデータを Web ブラウザ内で再度コンパイルする．これにより，多様なアーキテクチャに合わせた最適化が可能となる．
(3) バイナリを実行するためのメモリ領域を確保する．
(4) バイナリを実行する．
(5) (3) で確保したメモリ領域を解放する．

上記の手順の中で，WebAssembly バイナリが実際に実行されているのは (4) のステップだけであり，それ以外のステップはバイナリを実行するための前後の処理である．これらは JavaScript を実行する場合から見てオーバヘッドとなる．よって，計算量が小さいタスクを実行した場合，WebAssembly を導入することにより削減された時間よりも，このオーバヘッドにより増加した時間の方が長くなる場合があると考えられる．4.3.2 項でも述べたように，実験対象 $E_2$ のワードカウントは 1 タスクあたりの計算量が小さいジョブであるため，WebAssembly を用いた方が遅くなったと考えられる．

## 5. おわりに

本研究では，Web ブラウザ上で行う分散処理手法 BBVC の課題である，開発容易性の低さおよび実行時性能の低さを解決するため，MapReduce と WebAssembly を導入した新しい BBVC フレームワーク Madoop を実装・提案した．また評価実験として，提案手法では従来手法と比べて実行時性能がどの程度改善するのかを，実行時間の長さを計測・比較することで確かめた．実験の結果，計算量が大きくデータ量が少ない実験対象では最大で約 64 %減少した．また計算量が小さくデータ量が多い実験対象では約 4 〜 8 %増加した．よって，計算量が大きくデータ量が少ないような分散処理では，提案手法が非常に有効であることが確かめられた．

今後の課題として，クライアントノードの特性を考慮して効率的にスケジューリングを行うことが挙げられる．現在 Madoop ではノードの通信状況や処理性能，滞在時間といった特性を考慮せず，すべてのノードに一律して同じサイズのタスクを配布している．BBVC では Web ブラウザ上で処理をするという特性上，VC に比べてノードの特性の影響を非常に受けやすいため，さらなる性能の改善にはスケジューリングアルゴリズムの改善が必要である．

**謝辞** 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：16H02908，18H03222）の助成を得て行われた．

### 参考文献

[1] Apache Hadoop: http://hadoop.apache.org/.
[2] Can I use... Support tables for HTML5, CSS3, etc: https://caniuse.com/#search=wasm.
[3] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pp. 137–150 (2004).
[4] Emscripten: http://kripken.github.io/emscripten-site/.
[5] Fabisiak, T. and Danilecki, A.: Browser-based harnessing of voluntary computational power, *Foundations of Computing and Decision Sciences*, Vol. 42, No. 1, pp. 3–42 (2017).
[6] Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 20–43 (2003).
[7] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.: Bringing the web up to speed with WebAssembly, *ACM SIGPLAN Notices*, Vol. 52, No. 6, ACM, pp. 185–200 (2017).
[8] Langhans, P., Wieser, C. and Bry, F.: Crowdsourcing MapReduce: JSMapReduce, *Proceedings of the 22nd International Conference on World Wide Web*, ACM, pp. 253–256 (2013).
[9] Merelo, J.-J., García-Valdez, M., Castillo, P. A., García-Sánchez, P., Cuevas, P. and Rico, N.: NodIO, a JavaScript framework for volunteer-based evolutionary algorithms: first results, *CoRR*, Vol. abs/1601.01607 (2016).
[10] Oechslin, P.: Making a faster cryptanalytic time-memory trade-off, *Annual International Cryptology Conference*, Springer, pp. 617–630 (2003).
[11] Pan, Y., White, J., Sun, Y. and Gray, J.: Gray Computing: A Framework for Computing with Background JavaScript Tasks, *IEEE Transactions on Software Engineering* (2017).
[12] Ryza, S. and Wall, T.: Mrjs: A javascript mapreduce framework for web browsers, http://www.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf (2010).
[13] SETI@home: https://setiathome.berkeley.edu/.
[14] SETI@Home - Detailed stats | BOINCstats/BAM!: https://boincstats.com/en/stats/0/project/detail/.
[15] The MD5 Message-Digest Algorithm: https://www.ietf.org/rfc/rfc1321.txt.
[16] WebAssembly: https://webassembly.org/.
[17] Yelp Open Dataset: https://www.yelp.com/dataset.
[18] 高木省吾， 渡邊寛， 福士将，天野憲樹，舩曳信生，中西透：Web ブラウザを用いたボランティアコンピューティングプラットフォームの提案，技術報告 29 (2014).