Vol.2014-HPC-143 No.29
2014/3/4

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

高木 省吾[1]　渡邊 寛[1,a)]　福士 将[2]　天野 憲樹[3]　舩曳 信生[1]　中西 透[1]

概要：ボランティアコンピューティング (VC) で高い性能を実現するためには，多数の参加者を集める事が重要である．しかし，既存の VC システムでは，参加に際して専用ソフトウェアのインストール等の手間がかかり，参加の障害となっている．そこで本研究では，参加者が Web ブラウザで指定の URL にアクセスするだけで即座に VC に参加することができるような，Web ベースの VC プラットフォームを提案する．提案するプラットフォームでは，PNaCl 等の LLVM 技術を用いることで，C/C++で記述された計算問題を Web アプリ化し，ブラウザ上で高速に実行することができる．性能評価として，PNaCl を用いて姫野ベンチマークを Web アプリ化した場合，ネィティブアプリと同等の実行性能を実現できることを確認した．

キーワード：Web，HTML5, LLVM, 並列分散処理, デスクトップグリッド

A proposal of Web Browser-based Volunteer Computing Platform

Shogo TAKAKI[1]　Kan WATANABE[1,a)]　Masaru FUKUSHI[2]　Noriki AMANO[3]
Nobuo FUNABIKI[1]　Toru NAKANISHI[1]

Abstract: In Volunteer Computing (VC) systems, volunteer participants can contribute their idle computing resources by computing a piece of the computation (job) in their idle time. In existing VC systems, participants must put an extra effort, e.g. installing a dedicated software to their computers and register their personal E-mail addresses, which will be barriers to join as a participant. This paper proposes a web-based VC platform, in which participants can join to VC system by just accessing a specified URL with Web browsers. By using LLVM techniques such as PNaCL, VC jobs become convertible as fast-acting Web applications. As the results of our experiments based on Himeno Benchmark, we show that the performance of PNaCl codes on a web browser is equivalent of native one.

Keywords: Web，HTML5, LLVM, Parallel Computing, Desktop Grids

1. まえがき

ボランティアコンピューティング (VC) は，一般の人々が普段 PC 等を利用する際に使い切れていない，余剰の計算能力やストレージ等をインターネットを通じて提供してもらうことで，高性能な大規模並列計算システムを構築する手法である．VC は，SETI@home[6] などに代表されるように，数百万人の参加者を集めることでスーパーコンピュータと同等以上の高い性能を実現可能である．また，ある一定以上の時間がかかる大規模な計算に対しては，AmazonEC2 クラウドより VC を用いる方が低コストであるという調査結果 [9] が報告されており，安価で高性能な計算プラットフォームとしての VC が近年注目されている．

VC システムの性能は，どれだけのボランティア参加者を集めることが出来るかによって大きく左右される．しかし，現在主流となっている VC ミドルウェア BOINC[2] を用いたシステムでは，参加者に対して専用ソフト (BOINC クライアント) のインストールやメールアドレスの登録と

[1] 岡山大学　大学院自然科学研究科
[2] 山口大学　大学院理工学研究科
[3] 岡山大学　教育開発センター
[a)] can@okayama-u.ac.jp

いったいくつかの作業を要求するため，これらが参加者に対するハードルになっていると考えられる．より多くの参加者を集める為には，より簡単で，参加者がもっと気軽にVC に参加できるような仕組みが必要不可欠である．また，近年，スマートフォンやタブレットといったデバイスの普及・高性能化に伴ない個人の所有する計算環境が多様化している．これらを BOINC ベースの VC で計算資源として利用するためには，各環境に応じた BOINC クライアントをそれぞれ準備する必要があり，その開発や保守に大きなコストがかかってしまう．

そこで本研究では，参加者がより気軽に，どのような計算環境からでも参加できるような，Web ブラウザを用いたVC プラットフォームを提案する．近年，HTML5 による Web 規格の標準化に伴い Web ブラウザの高機能化が進んでおり，旧来は実現できなかったような Web サービスが盛んに開発・提供されている．本研究では，近年発展の著しいこれらの Web 技術を利用し，参加者が Web ブラウザで指定の URL を開くだけで VC に参加できるような仕組みを用意する．

また，本研究では，一般的な科学技術計算等で用いられている C/C++で記述された計算問題（ジョブ）を高速にブラウザ上で実行することができるよう，PNaCl 等の LLVM 技術を用いてジョブの Web アプリケーション化を検討する．ジョブの実行時には，HTML5 を用いることでブラウザ上の別スレッドとして動作させ，参加者の Web ブラウザ体感速度が劣化することを防ぐ．性能評価として，姫野ベンチマークを Web アプリケーション化して用いた実験の結果，ネィティブアプリと同等の実行性能を実現できることを確認している．

以下では，2 章において VC のモデルを述べ，3 章では，関連技術として HTML5 及び VC における妨害者対策技術について説明する．4 章では，提案する Web ブラウザベースの VC プラットフォームについて，システムの構成要素や動作モデルを述べる．5 章では，LLVM 技術を用いたジョブの Web アプリケーション化の方法を検討する．6 章にて，Web アプリケーション化したジョブの性能や体感速度に与える影響を評価し，7 章にてむすびを述べる．

2. VC のモデル

2.1 計算モデル

BOINC 等に代表される既存の VC プラットフォームは，システム全体の管理を行う管理ノード（マスタ）と，参加者の PC 等（ワーカ）を要素とする，マスタ・ワーカモデルを構成する．このモデルでは，VC の計算は次のように行われる．

- 計算プロジェクトは独立した計算問題 (ジョブ)N 個から成り，マスタはワーカからのジョブ要求に応じて個々のワーカにジョブを配布する．
- ジョブを配布されたワーカはこれを実行し，生成された計算結果 (リザルト) をマスタへ返却する．
- 計算プロジェクトは N 個のジョブ全てが終了すれば完了となる．

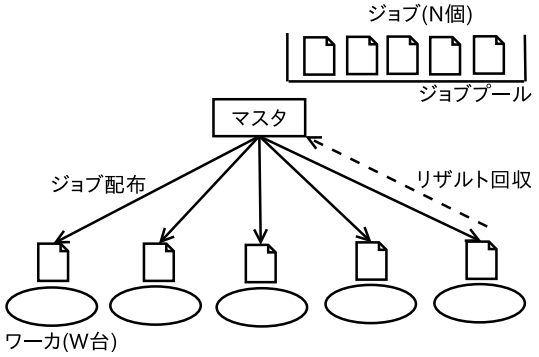

図 1　マスタ・ワーカモデル

2.2 マスタのモデル

VC におけるマスタは，VC システムの管理者が用意する，管理用のノードである．マスタが故障等をすることは想定されていないため，マスタが停止すると VC システム全体が停止することとなる．また，マスタは各ワーカからの要求（ジョブ要求・リザルト回収要求）に応じて動作するのみであり，マスタからワーカへ能動的にアクセス等をすることはない．

2.3 ワーカのモデル

VC におけるワーカは，各参加者が提供する計算資源である．例えば BOINC を用いた VC プラットフォームでは，参加者が以下の手順で作業を行うことで，自身の所有する PC 等をワーカとして動作させることが出来る．

(1) BOINC クライアントのダウンロード
(2) インストール実行と PC 等の再起動
(3) BOINC クライアントの起動
(4) 参加する VC プロジェクトの選択
(5) メールアドレスによるユーザ登録

これらの作業は，PC の操作に慣れた熟練者でも数分の時間を要する，多少手間のかかる作業である．よって，VC プロジェクトの内容（天文学・数学の問題や医薬品開発など多岐にわたる）に興味を持ち，自身の所有する計算機をワーカとして VC システムに参加したいと考えた積極的なユーザであっても，以上の作業を手間・苦痛に感じ，参加をためらってしまう可能性が十分に考えられる．

3. 関連研究

3.1 HTML5

HTML5 は，近年策定された新たな Web 標準規格である．

Web は，使用の容易さ・汎用性の高さから様々な用途で用いられており，多様なサービスが Web アプリケーション（HTML や HTTP/HTTPS 等を用いたネットワーク通信を用いて Web ブラウザ上で動作するアプリケーション）として提供されている．Web アプリケーションは，(1) インストール等の作業を行うことなくアプリケーションのサービスを利用できる．(2)Web ブラウザがインストールされていれば，OS や CPU といったプラットフォームの違いに関係なくサービスを利用できる．といったメリットが挙げられる反面，(3) アプリケーションの機能が Web ブラウザの提供する機能の枠の中に限定される．(4) ネイティブアプリケーションと比べて動作が遅いといったデメリットがあった．

このようなデメリットのため，Web の登場初期の Web アプリケーションは，サーバ側で処理を行い，Web ブラウザでは結果を表示するだけという手法が多かったが，HTML5 の策定により，(3)(4) のようなデメリットが解消されつつあり，より多様なサービスが Web アプリケーションとして提供されるようになっている．以下では，本研究で用いた HTML5 の機能を紹介する．

(1) WebWorker：
HTML5 以前では，シングルスレッドでの動作しかサポートされていなかったため，VC のジョブのような時間のかかる計算を実行した場合にブラウザが操作を受け付けなくなってしまうという問題があった．WebWorker を用いると，Web アプリケーションを複数のスレッドを使って動作させることが可能となるため，VC のジョブを実行しつつ他の複数の処理を同時に行うといった動作が可能となる．

(2) WebStrage/IndexedDB：
HTML5 以前では，一つの Web アプリケーションがクライアント側で保持できるデータ量は cookie の最大値として定義されている 4KB までであり，これを超えるデータ量を扱う場合は，Web アプリケーションの開始時に毎回サーバーからデータをダウンロードする必要があった．HTML5 では，WebStrage や IndexedDB といった機能により，ローカルディスクに大きなデータを保存しておくことが可能となったため，Web アプリケーション開始時の通信を減らすことが可能となっている．

(3) FileAPI:
従来の Web アプリケーションは，セキュリティ上の観点から Web ブラウザの外部のファイルには基本的にアクセスできないようになっていたが，FileAPI を用いると，ユーザから指定されたファイルに限り読み取りを行うことが可能になる．

3.2 VC における高信頼化手法

VC における参加者は，VC のプロジェクトに興味を持ち善意を持って参加する者ばかりであるとは限らず，中には悪戯等を目的とした悪意ある参加者（妨害者）が混じることが知られている [8]．このため BOINC では，*m*-first 多数決と呼ばれる，1 つのジョブに対して一定個数のリザルトを集め多数決を行うことで，誤ったリザルトを排除する高信頼化手法が用いられている．

BOINC では，参加者に対して「メールアドレスの登録」などの手間を課すことで，妨害者の攻撃をある程度抑制することができる．一方，本研究で提案するような，Web ブラウザでアクセスするだけで参加できる VC では，妨害者を抑制することが難しく，例えば何度もワーカ ID を変えつつ，誤りを生成し再参加を繰り返すような攻撃が想定される．

このような攻撃に対して有効な高信頼化手法として，信頼度に基づく多数決 [11] がある．この手法では，ワーカそれぞれに対して過去の計算実績に基づき信頼度という重みを付けて多数決を行うため，再参加を繰り返すような妨害者は信頼度が低くなり，計算結果の信頼性に与える影響力を低減させることが可能となる．

4. Web ブラウザを用いた VC プラットフォーム

4.1 概要

前章で述べた様に，近年の Web 技術の進歩により，最新の Web ブラウザは様々なサービスの実行基盤として耐えうる程に高機能化されている．本研究では，これらの技術を活用した，Web ブラウザを用いた VC プラットフォームの提案を行う．本提案により，参加者は，PC にインストール済のブラウザで特定の URL を開くだけで容易に VC に参加が可能となり，また，VC 管理者も，OS 等の環境の違いを意識する必要のない Web のクロスプラットフォーム性を生かし，これまで以上に多くの参加者を集めることができるといったメリットが考えられる．

4.2 システム全体の構成

提案する VC プラットフォームでは，システム全体の構成は図 2 のようになる．システムは，クライアント（ワーカが動作する：図左）と，Web サーバ（図中央），既存 VC システム（図右）の三要素と，Web サーバや既存 VC システムが利用するジョブプールやリザルトプールから構成される．

クライアントとしては，PC やスマートフォン・タブレット等の端末が利用されることを想定し，これらの端末の上で HTML5 に準拠したモダンブラウザ，もしくはこれに類似した Android アプリケーションの WebView 機能などがワーカとなる．Web サーバとしては，apache などの一般

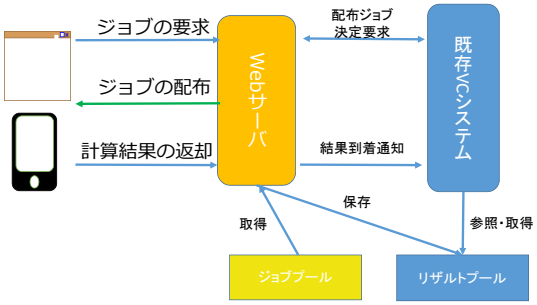

図 2　Web を用いた VC システムの構成

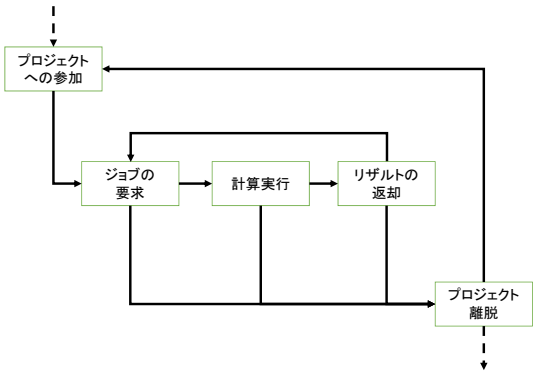

図 3　クライアントの動作フロー

的な Web サーバソフトウェアが動作しているサーバを想定する．また，ジョブの進捗管理やジョブスケジューリング，多数決による高信頼化機能を備えた既存 VC システムを利用し，従来の VC システムからの変更点を最小限に留める．

既存 VC システムは，Web サーバからの通知に応じて，信頼度に基づく多数決手法等を用いた計算プロジェクトの管理機能を提供する．具体的には，ワーカから計算結果が到着した際の多数決処理や，ワーカからジョブが要求された際の，対象ワーカに配布するべき最適な計算問題の決定等の機能が利用される．Web サーバはクライアントからの全ての操作の受け入れ窓口となり，クライアントからの要求に基づき，既存 VC システムに対して通知や要求を行った後に返答を受け，適切なデータをクライアントに対して返却する．Web サーバは，クライアント側から見た場合には Web のインターフェイスとして動作し，既存 VC システム側からみると Web 対応のためのラッパーとしての役割を担う．

4.3　動作モデル

本研究で提案する VC プラットフォームの動作モデルは，BOINC 等と同様にマスタ・ワーカモデルに基づいている．クライアント（ワーカ）から見た場合のマスタは，図 2 の Web サーバに相当する．

以下に，VC システムで発生する動作イベントの一覧を示す．

1. プロジェクトへの参加
2. ジョブの要求
3. リザルトの返却
4. ワーカの離脱・復帰

これらのイベントは，図 3 のような順序で発生する（計算実行処理の詳細については 5 章で述べる）．

クライアントは，VC プロジェクトへの参加後，ジョブ要求・計算問題の実行・リザルトの返却を繰り返す．また，任意のタイミングで VC への参加をやめ，システムから離脱することが可能である．以下では，それぞれのイベント時に発生する具体的な処理を説明する．

4.3.1　プロジェクトへの参加処理

プロジェクトへの参加する際．クライアントは Web ブラウザを用いて参加プロジェクトを一意に特定する URL へアクセスすることでプロジェクトへの参加を行う．クライアントからの参加要求が発生した際，マスタは要求に対応するプロジェクトの計算実行ページを返却する．さらに，クライアントに対して cookie を用いてセッション ID と呼ばれるユニークな ID の割り当てを行う．以降，セッションの有効期限の続く限りこの ID を用いてクライアントの追跡を行い，信頼度情報の管理等を行う．クライアントに返却されたプロジェクト用の計算ページには初期化処理が組み込まれており，実行可能な計算方式のチェック，計算用プログラムのロード，利用可能な API の確認処理等が行われる．これらのデータは WebStrage 機能を用いてクライアントのローカルストレージに保存される．

4.3.2　ジョブの要求処理

プロジェクトページのロードが完了すると，クライアントは Web サーバのスケジューリング用 WebAPI に対してアクセスを行う．Web サーバと既存 VC システムは，セッション情報を基に，そのワーカに対してどのジョブを割り当てるべきかのスケジューリング計算を行い，決定したジョブへの URL へリダイレクトを発生させる．実際にジョブを送信する際は，ブラウザが対応して場合には gz 圧縮されたジョブデータを送信することで帯域幅を節約することができる．また，ジョブに返却時間の期限がある場合には，キャッシュ期限の指定を行うことで，ワーカの離脱・再参加時にキャッシュを用いた通信の省略が可能である．ワーカはジョブの受け取りを完了すると，計算を開始する．

4.3.3　リザルトの返却処理

計算が完了すると，ワーカはその計算結果 (リザルト) をマスタに返却する．ワーカから Web サーバを通じて計算結果が返却された場合．既存 VC システムは多数決等の処理を行う．処理の完了後，受け取りの完了をワーカへ通知し，通知を受けたワーカは，再度ジョブの要求処理へと遷移する．

4.3.4 ワーカの離脱・復帰処理

ワーカは，Web ブラウザで開いているページを閉じることで，いつでも VC システムを離脱することができる．また，システムに再度復帰したい場合は，計算プロジェクトのページに再度アクセスする．復帰時，WebStorage 機能によりローカルにデータが残されている場合には，そちらを利用することができる．離脱時に計算実行中であった場合には，ジョブの期限を確認し，期限内であればローカルに保存されていたジョブを実行し，期限を過ぎていればジョブ要求処理へと遷移する．また，ワーカ自身が管理するセッション期限が過ぎてしまっていた場合には，新たなセッション ID が発行され，別のワーカとしてシステムに復帰することとなる．

5. ジョブの Web アプリケーション化

5.1 概要

前章で提案した Web ブラウザを用いた VC プラットフォームでは，ジョブの実行を含む，クライアント側の全ての処理は Web ブラウザ上で行われる．一方，既存の VC システムで扱われていたようなジョブは，C/C++で記述された科学技術計算問題などが多く，これらを直接 Web ブラウザ上で実行することはできない．ジョブをブラウザ外で実行する外部プロトコル方式や単純な Javascript 等で記述したジョブでは，機能が制限されたり，十分な計算性能が出ない場合が多いという問題がある．

そこで本章では，LLVM 技術を用いたジョブの Web アプリケーション化を行い，Web ブラウザ上でジョブを高速に実行する方法を検討する．高速な実行が期待できる asm.js と PNaCl については，C/C++で記述されたソースコードから自動的に Web アプリケーション化するスクリプトを作成した．また，Web アプリケーション化したジョブを実行する際の画面構成について，実際に本研究で実装した画面を用いて説明する．

5.2 外部プログラムの直接実行

Web ブラウザを用いた VC プラットフォームにおいて，C/C++で記述されたジョブを実行する方法として最も簡単なものが，ブラウザから直接外部プログラムを呼び出す「外部プロトコル方式」である．この動作モデルを図 4 に示す．この方式では，Web ブラウザはローカルストレージに存在しているネイティブプログラムを外部プロトコルリクエストという手法を用いて起動する．ブラウザは FileAPI 機能を用いてローカルファイルを監視し，そこに出力された計算結果を読み取りサーバへ返却を行うことができる．

この方式を用いると，提案システム上でネイティブプログラムを走らせることが可能であるが，以下の様ないくつかの問題点が存在する．(1) ブラウザに対して外部プロトコルの登録作業が必要，(2) 予め実行ファイルをローカルに用意して，プロトコル登録時に指定したパス上への配置が必要，(3) ブラウザから渡せるデータは，URL 内に組み込まれた文字列パラメータのみ，(4) プロジェクトの参加時に毎回，結果の出力ファイルをブラウザ上で指定する作業が必要．このため本研究では，外部プロトコル方式以外の方法について以下のように検討を行った．

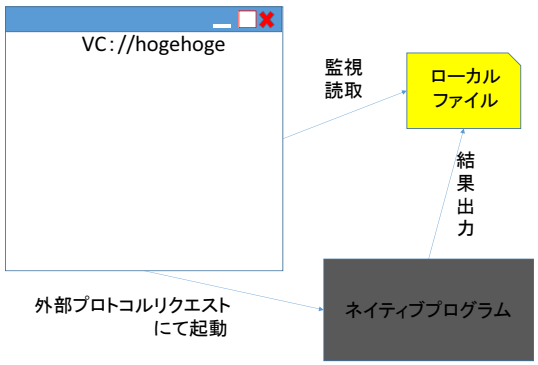

図 4 外部プロトコル方式の動作

5.3 LLVM 技術を用いた Web アプリケーション化

近年の Web の技術の向上により，様々なサービスが Web アプリケーションとして実装されるようになった．しかし，Web アプリケーションのクライアント側で実行されるプログラムはほとんど JavaScript に限定されているという問題がある．JavaScript の実行速度は JIT コンパイラの発展等によってますます高速になってはいるが，それでも C 言語の様な静的な型付け言語と比べると劣ってしまうため，どうしても速度が必要なサービスは Web アプリケーションと相性が悪いという問題があった．しかし近年，Web アプリケーションをより高速に実行するための技術として asm.js や PNaCl などの開発が進んでいる．

本研究では Web の高速化手法として，上述の PNaCl や asm.js 等を用いる．これらはどちらも静的な型付け等の最適化を行いやすい特徴を持っており，通常 Web ブラウザ上で用いらている JavaScript プログラムよりも高速な実行が可能である．asm.js[1] 及び PNaCl[5] では LLVM[10] と呼ばれるコンパイル基盤フレームワークの技術を活用することでクロスプラットフォームで高速なプログラムの実行を実現している．LLVM とは任意のプログラミング言語のコンパイル基盤として開発されたフレームワークである．LLVM は LLVM ビットコードと呼ばれる独立した命令セットを持っている．プログラミング言語からこの命令セットへの変換時や，この命令セットからマシン語への変換時等の様々な段階での最適化を行うことができるように設計されている．

5.3.1 asm.js 方式

asm.js とは JavaScript に静的な型情報を付与することの出来るサブセット言語である．これにより，asm.js に対応

している環境に置いては型情報を用いた高速な処理を行うことが出来る．また，対応していない環境においても通常の JavaScript として認識されるため，一般的なの全てのブラウザで動作させることが可能である．また，JavaScript であるため Web ブラウザの持っている機能にアクセスしやすいというメリットもある．asm.js 形式のプログラムは emscripten[3] と呼ばれるツールを用いて，C/C++のコードから生成することが出来る．emscripten は C/C++のコードを LLVM ビットコードに変換し，そのビットコードから asm.js 形式のコードを生成するという手法が取られている．

asm.js 方式の計算実行モデルを図 5 に示す．asm.js 方式では計算の実行は WebWorker と呼ばれる機能を用いて作られたサブスレッドの中で動作を行う．これにより，Web アプリケーションのその他の動作を阻害することなく計算が実行される．サブスレッドの管理やデータのやり取りは HTML5 の WebWorker 機能によって定められた方法で，JavaScript によって制御される．asm.js 機能は Mozilla により積極的に開発が進められており，今後ますます高速に動作するようになることが予測される．

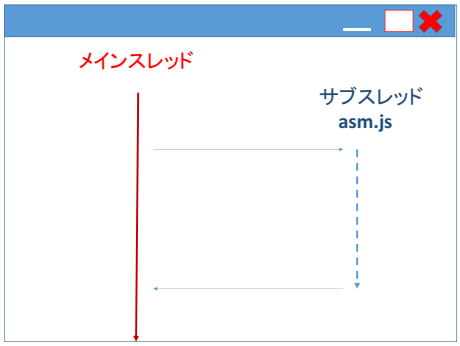

図 5　asm.js 方式の動作

5.3.2　PNaCl 方式

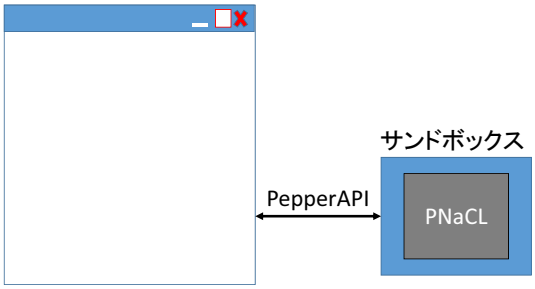

図 6　PNaCL 方式の動作

PNaCl とは，Chrome ブラウザ上で LLVM ビットコードを直接実行する機能である．CPU のアーキテクチャに依存せず動作させることが可能であるが，一般的なブラウザでは Chrome しか対応していない．その代わり，現状 asm.js 方式よりも高速に動作することが確認されている．ブラウザへの持つ機能へのアクセスやブラウザからの制御は Pepper と呼ばれる専用の API を通して行われる．PNaCl の実行プログラムは nacl_sdk を用いて，C/C++のコードから生成することが出来る．

PNaCl 方式の計算実行モデルを図 6 に示す．PNaCL 方式では計算の実行は，セキュリティ上の観点から専用に用意されたサンドボックスの中で行われる．ブラウザとは別のプロセスとして動作するため，この方式でも Web アプリケーションのその他の動作を阻害することなく計算を実行することができる．

5.4　計算プログラムの生成の自動化

asm.js と PNaCl の 2 方式を比較した場合．asm.js は全てのブラウザ上で動作させることが可能であるが PNaCl と比べて低速，PNaCl は特定のブラウザでしかの動作しない代わりに asm.js より高速に動作する，という特徴がある．そのため，計算プロジェクトでは両方の方式の実行プログラムを用意しておくことが望ましい．

PNaCl と asm.js はどちらも LLVM ビットコードから生成されるため，C/C++の同一ソースコードから変換することが可能である．ただし，本システムで動作させる場合には，WebWorkerAPI と PepperAPI という別々の手法で VC システムと連携する処理を追加する必要がある．計算実行プログラム生成までの簡単な流れを図 7 に示す．

asm.js 方式では，emscripten を用いて asm.js 形式のコードを生成した後，JavaScript で WebWorkerAPI を使った連携処理を埋め込む必要がある．また，PNaCl 方式では，C/C++コードの変換前に PepperAPI を用いた連携部分の埋め込みが必要となる．

実行プログラムの生成過程は，与えるデータ型・呼び出し関数名・返却されるデータ型によって決定される．これらが一致している場合，内部の C/C++の処理にかかわらず，まったく同じ処理によって asm.js 方式プログラムと PNaCl 方式プログラムに変換することが可能である．この特徴を用いて，実行プログラムの自動生成スクリプトの作成を行った．

以下，int 型の引数を 1 つ持ち int 型の戻り値を持つ function という名前の関数を呼び出す場合の変換処理を説明する．

asm.js 方式ではまず emscripten を利用してコンパイルを行う．この際，-s EXPORTED_FUNCTIONS = “[’_function’]” というコンパイルオプションを付与することにより，C で書かれた function 関数を javascript から呼び出す仕組みが組み込まれる．その後，用意しておいた WebWorkerAPI のコードに，function 関数の型情報を使った，バックグラウンドスレッドでの呼び出し処理と，関数の戻り値をメインスレッドに返却する処理を埋め込むことによって asm.js 方式プログラムを生成できる．

PNaCl 方式では，コンパイル前に JavaScript との連携処理の組み込みを行う．まず，計算処理が記述されたファイルと PepperAPI の雛形となる 2 つの C++プログラムを用意する．雛形ファイルの中に，JavaScript から PNaCl へのメッセージ送信メソッド HandleMessage と応答用メソッド PostMessage が存在しているため，そこに javascript から送られてくるデータの型変換と function 関数の呼び出しを組み込むだけでよい．その後これらのファイルをコンパイル・リンクしてやれば PNaCl 方式計算実行プログラムが完成する．

この様な処理によって，計算実行プログラムを生成することが可能である．ただし，C/C++コードで利用しているライブラリ等のプログラムの処理内容によっては，PNaCl/asm.js が対応していない場合がある．例えば，gmp ライブラリが利用されていた場合などがそれに当たる．その様な場合には特別な対応が必要となる．

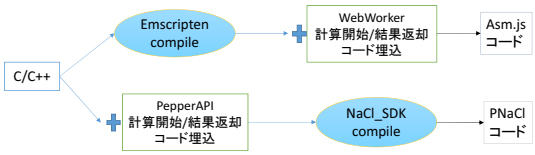

図 7　ジョブプログラムの作成

5.5　クライアントの画面構成

図 8 に，Web アプリケーション化したジョブを実行する場合の計算実行ページの例を示す．ユーザは，画面中央下部のボタンを使って，好きなときに計算の中止と再開を行うことが可能となっている．ただし，処理の一時的に停止させるのではなく完全に実行を終了させてしまっているため，再開時には計算問題は最初から計算し直しとなる．画面左では，現在までの処理の結果等の情報をユーザに対して開示している．

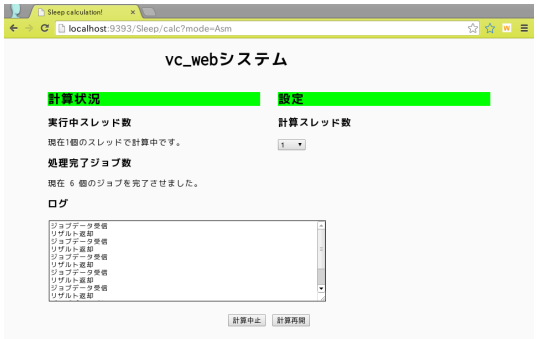

図 8　計算実行ページ

6.　実験と評価

6.1　実験の概要

今回，Web を用いた VC システムの評価として，ジョブの実行性能に関するテストを行った．さらに，ジョブ実行時のブラウザの動作速度の検証を行った．具体的には，今回提案した asm.js/PNaCl/外部プロトコルの各計算方式を用いてベンチマークプログラムを動作させ，その結果の比較を行った．さらに，ジョブの実行時に同時にブラウザのベンチマークソフトを走らせ，通常時の性能値との比較を行った．ジョブを動作させた PC の性能と動作環境を表 1 に示す．

表 1　性能諸元

	asm.js	PNaCL	外部プロトコル
CPU	Core i3 560 @3.33GHz		
Memory	4GB DDR3-1333		
OS	Ubuntu12.04 32bit		
compiler	clang3.2 emscripten1.8	pnacl-clang3.3	clang3.2
browser	firefox26	chrome32	

今回，ジョブ用のベンチマークプログラムとして姫野ベンチマーク [7] を利用し，各計算方式による浮動小数演算の性能の比較を行った．また，ブラウザのベンチマークとして，Octane2.0[4] を利用した．

6.2　姫野ベンチマークを用いた性能評価

まず，各計算方式自体の性能差を比較するために，姫野ベンチマークをそれぞれ対応するコンパイラの $O3$ オプションを用いて実行性能の比較を行った．配列サイズとして，S(33 * 33 * 65), M(65 * 65 * 129), L(129 * 129 * 257) の 3 種類を用いた．また，今回は asm.js の最適化によってパフォーマンスがどの程度改善しているのかを比較するために，asm.js に対応していない JavaScript インタプリタを用いてベンチマークを行った結果も載せている．計測結果を表 2 に示す．

表 2　姫野ベンチマークによる性能値 [MFLOPS]

配列サイズ [float]	S	M	L
外部プロトコル (ネイティブ)	2981	2578	2550
PNaCl	2898	2556	2498
asm.js	1050	error	error
JavaScript	710	error	error

結果から，PNaCl 方式はネイティブコードを走らせている外部プロトコル方式と比較しても遜色のない実行性能が得られていることがわかった．対して，asm.js 方式では，通常の JavaScript からの性能の向上は見られたが，ネイティブコード・PNaCL の性能と比較すると半分以下の性

能しか出ないという結果となった．

また，メモリ上に確保する配列のサイズを大きくしていった場合，メモリ上にキャッシュしきれないため徐々に性能が下がっていくことが確認できる．この低下の仕方についても, ネイティブと PNaCl の間でほとんど差が見られなかった．しかし，asm.js ではそもそも実行時にエラーが発生してしまい性能を確認することが出来ないという結果となった．

asm.js の実行時エラーに関して，コンパイル時のオプションを *O2*/*O1*/最適化無しに，それぞれ変更して実行してみたが，全く同じエラーが発生した．エラーのメッセージから，何らかの理由でメモリ関連のエラーが発生していると推測される．

また，asm.js 方式が特に遅かった原因として，姫野ベンチマークでは配列等のデータに float 型を利用していることが考えられる．JavaScript では内部的に float 型を持っておらず，メモリ確保や演算は全て double 型で行われている．このため，asm.js 方式では無駄な計算の発生や確保メモリの増加等の理由により，これだけの性能差が出たのではないかと考えられる．そこで，次の実験として，姫野ベンチマークで使われている float 型を全て double 型に変換して，実行時のデータ型の差のない状態での性能比較を行なった．結果を表 3 に示す．

表 3 姫野ベンチマーク (double) による性能値 [MFLOPS]

配列サイズ [double]	S	M
外部プロトコル (ネイティブ)	1920	1780
PNaCl	1820	1713
asm.js	930	error
JavaScript	630	error

この結果から，どちらの実行方式でも性能の低下が見られたが，ネイティブ/PNaCl の方でより顕著な性能低下が見られ，PNaCl と asm.js の性能差が約 2:1 に縮まったことが確認できた．

6.3 ジョブ実行時のブラウザの挙動

次に，ブラウザでのジョブの実行時に，別のタブでブラウザのベンチマークを同時に走らせた場合の結果を以下に示す．同時に実行するジョブとして，先程利用した姫野ベンチマークの配列サイズ S の場合プログラムを利用し，実験を行った．

表 4 別スレッドへの影響評価:octane 実行スコア

条件	Chrome(PNaCl)	firefox(asm.js)
octane のみを実行	16360	12214
ジョブと同時実行	14529	11310

ベンチマークを同時に走らせた場合に，10%程度のベンチマークスコアの低下が確認された．さらに，asm.js 方式では，稀にブラウザタブの切替時にレンダリングの遅れを感じるなど，体感速度への影響が見られた．

今回，ジョブプログラム・ベンチマークプログラムともにシングルスレッドで実行されているため，CPU 使用率の面で逼迫的な状況は確認されなかった．

7. むすび

本研究では，Web の機能を利用した VC システムの提案を行った．そして，asm.js や PNaCl と呼ばれる Web アプリケーションの高速化技術を利用した Web ブラウザでの高速な計算実行方式を提案し，その性能評価を行った．評価の結果，PNaCl 方式ではネイティブプログラムと遜色のない計算速度を確認できたが，asm.js 方式では，その半分程度の性能しか出ないという結果が得られた．また，計算問題の実行時にブラウザ上の別のタブへの動作の影響を確認したが，影響はほとんど見られないことが確認できた．今後の課題としては，通信部分や多数決処理も含めたシステム全体の性能の評価，gmp ライブラリ等の外部ライブラリを利用した計算実行への対応が挙げられる．

参考文献

[1] asm.js: http://asmjs.org.
[2] BOINC: http://boinc.berkeley.edu.
[3] emscripten: https://github.com/kripken/emscripten/.
[4] Octane2.0: http://octane-benchmark.googlecode.com/.
[5] PNaCl: https://developers.google.com/native-client/dev/.
[6] SETI@home: http://setiathome.berkeley.edu.
[7] 姫野ベンチマーク: http://accc.riken.jp/2145.htm.
[8] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello, “Characterizing Error Rates in Internet Desktop Grids”, 13th European Conf. Parallel and Distributed Comput., pp. 361–371, 2007.
[9] D. Kondo, Javadi, B., Malecot, P., Cappello, F. and Anderson, D.P., “Cost-benefit analysis of Cloud Computing versus desktop grids”, *IPDPS*, pp. 1–12 (online), (2009).
[10] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04), 2004.
[11] K. Watanabe, M. Fukushi and S. Horiguchi, “Expected-credibility-based Job Scheduling for Reliable Volunteer Computing”, IEICE Trans. Inf.& Syst., Vol.E93-D, No.2, pp.306 – 314, 2010.