GMPを利用したアプリケーションによるWebベースボランティアコンピューティングの性能評価

梶谷 翔馬[1]　渡邊 寛[2,a)]　福士 将[3]　野上 保之[2]　天野 憲樹[4]

概要：本稿では，Webベースのボランティアコンピューティング（VC）環境において，GMPを利用してC/C++で記述された既存アプリケーションを実行した場合の性能評価を行う．WebベースのVCでは，Webブラウザにより誰でも容易に計算資源の提供が可能になる反面，ライブラリを含む全てのコードをWebベースVC用の形式に変換して実行するため（PNaCl方式），端末上での実行（ネイティブ方式）と比較して性能が劣化する．本稿では，科学技術計算で特によく用いられるライブラリとしてGMPに着目し，これを利用した円周率計算などのアプリケーションを，ネイティブ方式とPNaCl方式の2つの方法で実行した．実験の結果，両方式の実行時間の比率は，GMPの各演算の種類やL3キャッシュの容量に依存して，約1倍〜8.9倍と大きく変化することを明らかにした．

キーワード：HTML5，LLVM，並列分散処理，デスクトップグリッド

A Performance Evaluation of Web-based Volunteer Computing using Applications with GMP

Shoma KAJITANI[1]　Kan WATANABE[2,a)]　Masaru FUKUSHI[3]　Yasuyuki NOGAMI[2]
Noriki AMANO[4]

Abstract: This paper presents the performance of GMP-based applications written in C/C++ on Web-based Volunteer Computing (VC) systems. The Web-based VC allows any Internet users to easily join a VC by using Web browsers, which attracts more volunteer participants. However, the performance of existing applications may be degraded on the Web-based VC because applications and libraries written in C/C++ must be converted to executable files on Web browsers. In this paper, we focus on GMP, an often-used library for scientific computations, and evaluate the performance of GMP-based applications on the Web-based VC. Experimental results show that the computation time of those applications mainly depends on GMP functions and L3 cache size, and it increases by a factor of 1 to 8.9 times.

Keywords: HTML5, LLVM, Parallel Computing, Desktop Grids

1. まえがき

Webベースのボランティアコンピューティング (VC) [7] は，参加者がWebブラウザで特定のURLを開くだけで手軽にVCに参加することを可能とする，ボランティアベースの並列計算システム構築手法である．SETI@home [1] などに代表される既存のVCと異なり，WebベースのVC（WebベースVC）では，クライアントソフトウェア導入・ユーザ登録といった煩わしいステップを参加者に対して強制する必要がないため，多数の参加者を集めやすいと考えられる．VCシステムの性能は参加者数に比例するため，WebベースVCは，[6] で報告されているような「AmazonEC2クラウドよりも低コストなVCシステム」をより容易に実現するための手段として期待されている．

[1] 岡山大学　工学部
[2] 岡山大学　大学院自然科学研究科
[3] 山口大学　大学院理工学研究科
[4] 埼玉大学　基盤教育研究センター
a) kan.watanabe@okayama-u.ac.jp

ⓒ 2014 Information Processing Society of Japan

Web ベース VC では，参加者が行う全ての処理（計算問題の実行ファイル取得，実行ファイルを用いた計算実行，計算結果の返却）を Web ブラウザ上で実現する必要がある．ここで，計算問題の実行ファイルを VC システムの管理者が用意する際，既存の C/C++で記述されたアプリケーションのソースコードを流用することで，その手間を大幅に削減することが可能である．このようなソースコードの流用技術として近年，LLVM を用いて C/C++のコードを Web ブラウザ上で直接実行可能な形式に変換する，PNaCl [3] や asm.js [4] といった技術が注目されている．

先行研究 [7] では，姫野ベンチマークをアプリケーションとして端末上で直接実行した場合（ネイティブ方式）に対して，PNaCl を用いて Web ブラウザ上で実行した場合（PNaCl 方式）では約 1.0 倍，asm.js で実行した場合は約 3 倍，JavaScript に書き換えて実行した場合は約 4 倍の実行時間が必要となり，特に PNaCl 方式を用いた場合の性能が高い事が示されている．しかし先行研究の評価では，Web ベース VC を実際の科学技術計算で使用することを想定した場合，PNaCl 方式で実行可能なアプリケーションの規模や，外部ライブラリを使用した際の性能に関する議論が不十分であった．

そこで本研究では，科学技術計算を行う際に特によく用いられる GMP に着目し，これを利用した円周率計算や RSA などの各種アプリケーションを用いて性能評価を行う．GMP の基本演算である mpz_add 等について，(1) ネイティブ方式に対して PNaCl 方式でどの程度の性能が出るか，(2) 各種基本演算の間の性能差はどの程度か，(3) 実行環境に対する依存性はあるか，といった観点から，複数の PC 上で引数のビット長を様々に変えて評価を行う．さらに，実行可能なアプリケーションの規模を調べるため，最大 5000 万桁の円周率計算や 10000 ビットの RSA 等の演算を行い，演算性能やメモリ使用量等を明らかにする．

2. Web ベース VC

2.1 VC への参加方法

Web ベース VC は, 参加者が Web ブラウザを介して計算プロジェクトへ参加する形式の VC である [7]．現在主流となっている VC ミドルウェア BOINC [2] を用いた VC では，参加者が計算プロジェクトに参加する際，以下の手順で作業を行う必要がある．

(1) BOINC クライアントソフトのダウンロード
(2) クライアントソフトのインストールと PC の再起動
(3) クライアントソフトの起動
(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

これに対して，Web ベース VC では，参加者は次の 2 つ手順のみで容易に VC へ参加することが可能である．

(1) Web ブラウザの起動（以下の (2) と同時に実行可能）
(2) 参加する VC プロジェクトの指定 URL へのアクセス

2.2 計算モデル

Web ベース VC における計算モデルは, BOINC を利用した VC と同様，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) を構成要素とするマスタ・ワーカモデルである．本モデルの概要を図 1 に示す．本モデルにおける計算は次の手順で行われる．

- マスタは巨大な計算プロジェクトを独立した N 個の計算問題 (ジョブ) に分割し，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．

また，各ワーカは図 2 に示すように，ユーザの操作によりブラウザのタブが閉じられるなどしてプロジェクトから離脱するまで，ジョブの要求・計算の実行・リザルトの返却といった動作を自動的に繰り返す．

2.3 信頼度に基づく多数決法による計算の高信頼化

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とは限らず, 悪戯やプロ

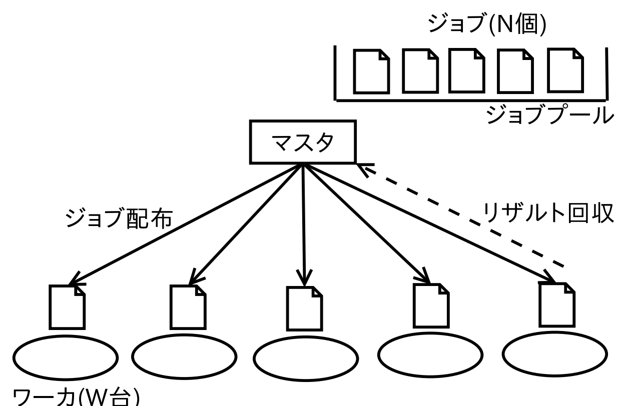


図 1 Web ベース VC の計算モデル

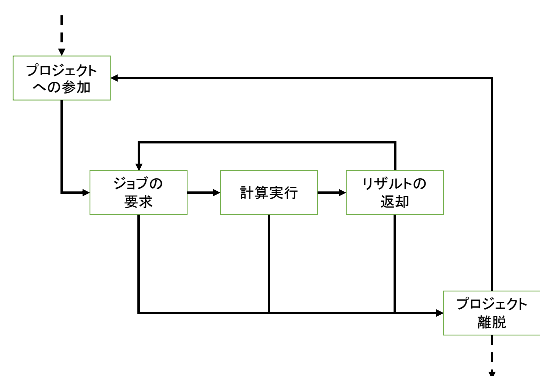


図 2 Web ベース VC におけるワーカの動作フロー

ジェクトの妨害等を目的とした，悪意ある参加者 (妨害者) が少なからず存在する [5]．このため BOINC では, 全ての参加者に対して「メールアドレスによるユーザ（ワーカ ID）登録」の手間を課しているが，この手間が参加への敷居の高さにつながってしまうことが懸念される．

Web ベース VC では, このような手間を課すことなく参加を可能とするために，ブラウザの cookie 情報等を自動的にワーカ ID に利用する．この際，何度もワーカ ID を変えて誤ったリザルトを返すような，悪意ある妨害者が計算結果の信頼性を大きく損ねてしまうという問題がある．

この問題に対し本研究グループでは，ワーカそれぞれに対して過去の計算実績に基づいて信頼度を計算し，信頼度を重みとした多数決を行う，信頼度に基づく多数決法 [9] を提案している．この手法を用いると，再参加を繰り返すような妨害者は他のワーカより信頼度が低くなるため，計算結果の信頼性に与える影響を低減させることが可能となる．ただし本稿では，1 台のワーカで 1 つのジョブを実行した際の実行性能を測定するため，多数決による冗長化で生ずる性能低下や計算信頼性に関する性能評価は，今後の課題となる．

2.4 LLVM による計算問題の Web アプリケーション化

Web ベース VC では，ジョブの実行を含むワーカ側の処理は全て Web ブラウザ上で行われる．しかし，既存の VC システムで扱われていたジョブは C/C++で記述されたアプリケーションが多く，これらを直接 Web ブラウザ上で実行することは困難である．また，JavaScript 等で記述したアプリケーションでは, 機能が制限されたり，十分な計算性能が出なかったりする場合が多いといった問題がある．

この問題に対し本研究グループでは，LLVM [8] を用いた Web アプリケーション化手法を提案している [7]．この手法では，既存の C/C++コードの先頭に短い定型コードを追加するだけで，容易に Web ベース VC のジョブへ変換することが可能である．現在までに，姫野ベンチマークを対象とした実験により，Chrome ブラウザ上で LLVM ビットコードを直接実行する PNaCl [3] や JavaScript に対して静的な型情報を付加する asm.js [4] を用いることで，ほぼ同等の性能を維持しつつネイティブプログラムを Web アプリケーション化できることが分かっている．

しかし，従来の評価 [7] では，姫野ベンチマーク以外のアプリケーション，特に外部ライブラリを使用するような複雑なソースコードを Web アプリケーション化した場合の性能については，これまで明らかにされていなかった．また，Web ベース VC のアプリケーションは Web ブラウザ上で動作させることになるため，Web ブラウザによる制約が存在すると考えられるが，従来の評価では使用可能メモリ量の上限などは明らかにされていなかった．

3. 関連研究

3.1 NaCl に関する性能評価

PNaCl は，Software Fault Isolation (SFI) による安全で高速なコード実行を可能とするための NaCl (Google Native Client) という技術を，CPU 非依存で，Web ブラウザ上で動作させるために発展させた技術である．

PNaCl の前身である NaCl に関する性能評価 [10] では，SFI によるオーバーヘッドが平均して 5%程度であり，その主な原因は，SFI のためのコードサイズの増大によって発生する，キャッシュミスの影響であると報告されている．

これに対し，PNaCl に関する詳細な性能報告は現時点までに行われておらず，また外部ライブラリを使用した場合の性能等については明らかにされていない．

3.2 PNaCl 用ツール

PNaCl を用いて様々なアプリケーションを動作させるためのプロジェクトとして，naclports [11] がある．naclports では，各種のアプリケーションを PNaCl を用いて Web ブラウザ上で実行できる形式に変換するためのビルドツールが公開されており，gclient というスクリプトファイルを使用することで，それらのツールを取得することができる．gclient を利用するには，chromium プロジェクトが公開している depot_tools [12] をあらかじめインストールしておく必要がある．

本研究では，naclports に含まれる，GMP を PNaCl 用にコンパイルするツールを利用して，PNaCl 用の GMP ライブラリファイルを生成する．naclports で利用可能なライブラリとしては，GMP の他に，Zlib，Python，SDL などがある．

3.3 実行性能分析ソフトウェア valgrind

valgrind [13] は，アプリケーションをネイティブで（端末上で）実行した際にプロファイリングを行うことができるフリーソフトウェアである．valgrind では，用途に応じて様々なツールが用意されており，例として，malloc 等で動的に割り当てられているメモリ量を測定できる massif や，キャッシュ参照数等を測定できる cachegrind などがある．

valgrind を利用する際の注意点として，プログラムを実行した際の実行時間が，通常の実行時間と比較して 5 倍～10 倍程度に増大してしまう．これは，valgrind が仮想化技術を利用しており，測定対象のプログラムを中間表現に変換し，さらに機械語に変換して実行しているためである．

本研究では，キャッシュミスやメモリ使用量を測定する場合にのみ valgrid を用い，実行時間を評価する場合には valgrid は用いないこととした．

ⓒ 2014 Information Processing Society of Japan

4. GMPを利用したアプリケーションによる性能評価方法

4.1 概要

GMPライブラリを利用してC/C++で記述されたアプリケーションを，(1) Cコンパイラを用いてコンパイルし端末上で直接実行した場合（ネイティブ方式）と，(2) WebベースVCのジョブとしてPNaCl形式に変換しWebブラウザ上で実行した場合（PNaCl方式）の2つの場合で性能評価を行う方法を説明する．

以下に，C/C++で記述された元のソースコードを“source.c”という名称であるとし，性能評価を行うための準備と評価方法を説明する．性能評価を行う環境としては，OSはUbuntu 14.04 LTS 32bit，CPUアーキテクチャはx86 (x86_64) を想定した．また，使用するGMPのバージョンはgmp-5.1.3とした．

まず4.2節で，source.cの例を用いて実行時間の測定方法を述べる．次に4.3節で，ネイティブ方式のコンパイルおよび実行方法と，3.3節で述べたvalgrindを用いた分析方法について説明する．最後に4.4節で，PNaCl用のGMPライブラリのコンパイル方法と，これを用いたジョブの生成及びWebベースVC上での実行方法を説明する．

4.2 実行時間の測定方法

ネイティブ方式とPNaCl方式のそれぞれで実行時間を測定して性能比較を行う．この際，PNaCl方式ではCPU時間での詳細な測定が困難であるため，両方式に共通して使用可能なgettimeofdayを使用し，プログラムの実行に要した実時間を実行時間とした．

図3に，性能評価に利用したソースコード (source.c) の例を示す．この図に示されるように，指定した関数funcをL回実行するのに要した時間を標準出力に出力し，この値を実行時間とした．

```
gettimeofday(&start, NULL);
for(i = 0; i < L; i++){
  func();
}
gettimeofday(&end, NULL);

end.tv_sec -= start.tv_sec;
end.tv_usec -= start.tv_usec;

printf("time: %f[sec]¥n",end.tv_sec + end.tv_usec * pow(10,-6));
```

図 3 関数func() をL回実行するソースコードの例

4.3 ネイティブ方式

ネイティブ方式では，端末上でsource.cをコンパイルし，生成した実行ファイルをそのまま端末上で実行する．コンパイラにはMac OS X，iOS，Free BSDなどの標準コンパイラとして採用されているC/C++向けコンパイラのClangや，Linuxの標準コンパイラであるGCCを使用した．以下では，Clangを用いたコンパイルと実行の方法を説明する．コンパイル時の最適化オプションとして -O1, -O2, -O3, -Osなどを使うことができるが，今回は-O3を用いている．また，GMPライブラリはapt-getにより事前に取得しインストールを行っておく．

図4に，source.cをコンパイルし実行した場合の例を示す．この場合，実行時間は約12秒である．

また，ネイティブ方式では，3.3節で述べたvalgrindを用いてプロファイリングを行う．valgrindを使用する際には，端末上で次のように実行する．この際，ツール名には，メモリ確保量を測定する場合はmassifを，キャッシュミス数を測定する場合はcachegrindを指定する．

valgrind –tool=[ツール名] [実行ファイル名]

図5に，cachegrindをツールとして指定した場合のプロファイリング結果の例を示す．この図から，例えばL1データキャッシュのミス数 (D1 misses: 13,418,923) や最終レベルのデータキャッシュミス数 (LLd misses: 260,474) などが測定できていることが分かる．

```
shoma@shoma:~/VC$ clang source.c -O3 -o source -lgmp -lm
shoma@shoma:~/VC$ ./source
time: 12.005146[sec]
```

図 4 ネイティブ方式でのコンパイル方法と実行例

```
==31264==
==31264== I   refs:      8,056,990,112
==31264== I1  misses:          590,868
==31264== LLi misses:            3,430
==31264== I1  miss rate:          0.00%
==31264== LLi miss rate:          0.00%
==31264==
==31264== D   refs:      4,047,045,319
==31264== D1  misses:       13,418,923
==31264== LLd misses:          260,474
==31264== D1  miss rate:           0.3%
==31264== LLd miss rate:           0.0%
==31264==
==31264== LL refs:          14,009,791
==31264== LL misses:           263,904
==31264== LL miss rate:            0.0%
```

図 5 valgrindによるプロファイリング結果の例

4.4 PNaCl方式

PNaCl方式では，以下の3つの手順でsource.cをコンパイルし実行する．

(1) PNaCl用GMPライブラリファイル (libgmp.a) の生成
(2) libgmp.aを使用した実行ファイル (source.pexe) の生成
(3) Webブラウザ上でのsource.pexeの実行

まず，PNaCl用GMPライブラリファイルlibgmp.aの生成手順を図6に示す．この手順では，$URL1からdepot_toolsを，$URL2からnaclportsを取得した後に，パスを設定しmakeすることでライブラリファイルを生成し

ⓒ 2014 Information Processing Society of Japan

ている．$URL1 と$URL2 には，それぞれ depot_tools と naclports のリポジトリの URL [14][15] を指定すればよい．

また，使用する GMP のバージョンを指定するには，図 6 で make gmp を行う前に，src/ports/gmp の中にある pkg_info というファイルを図 7 のように書き換える．図中の SHA1 の値には，使用したいアーカイブファイル (この場合 gmp-5.1.3.tar.bz2) の sha1 の値を調べて入力する．

次に，source.pexe の生成方法として，nacl_sdk を用いた手順を示す．まず，ネイティブ方式での実行に使用した source.c に必要な前処理 [7] を加え，source.cc を生成する．次に，手順 1 で生成した GMP ライブラリファイルを使用するため，nacl_sdk/pepper35 ディレクトリ内の Makefile を図 8 のように編集する（LDFLAGS 変数に -lgmp を付け加えるだけでよい）．さらに，Makefile 内でコンパイル対象に source.cc を指定する．Makefile の編集後，nacl_sdk/pepper35 ディレクトリ内に source.cc を配置し，make を行う事で実行ファイル source.pexe が生成される．

最後に，実行ファイルである source.pexe を，ジョブとして Web ベース VC のマスタに登録する [7]．登録後，ワーカとなる PC 上で Web ブラウザを起動しマスタにアクセスすることで，実行ファイルのダウンロードと実行が自動的に行われる．実行後，図 9 のように，標準出力への出力内容を記録したリザルトファイルが生成されマスタに返却されるため，このファイル内に記述された実行時間を性能評価に用いる．

```
~$ git clone $URL1
~$ export PATH=$PATH:$(pwd)/depot_tools
//depot_toolsのインストール，およびパスの設定

~$ gclient config --name=src $URL2
~$ gclient sync
//naclportsの取得

~$ cd src
~$ NACL_ARCH=pnacl make gmp
//PNaCl用GMPライブラリの生成
```

図 6 GMP ライブラリファイルの生成手順

```
NAME=gmp
VERSION=5.1.3
URL=ftp://ftp.gmplib.org/pub/gmp/gmp-5.1.3.tar.bz2
LICENSE=LGPL3
SHA1=b35928e2927b272711fdfbf71b7cfd5f86a6b165
```

図 7 GMP のバージョン指定方法（pkg_info 内の記述）

```
LDFLAGS := -L$(NACL_SDK_ROOT)/lib/pnacl/Release -lppapi_cpp -lppapi  -lgmp
```

図 8 Makefile の編集による GMP ライブラリファイルの指定

```
result=time: 12.909549[sec]
```

図 9 Web ベース VC 上で生成されるリザルトファイルの例

表 1 実験に用いた PC の性能と Web ブラウザのバージョン

	実験環境 1	実験環境 2
OS	Ubuntu 14.04 LTS 32bit	Ubuntu 12.04 LTS 32bit
CPU	Intel Core i7 870	Intel Core2 Quad Q6700
メモリ	8GB	4GB
L1 キャッシュサイズ	256kB	
L3 キャッシュサイズ	8MB	3MB
Web ブラウザ	Chrome バージョン 37.0.2062.94	

5. GMP を利用したアプリケーションによる性能評価結果

5.1 実験環境と使用したアプリケーション

前節で示した方法を用いて，ネイティブ方式と PNaCl 方式の両方式でアプリケーションを実行した際の実行時間を比較する．表 1 に，評価に用いた PC の性能を示す．今回の実験では，L3 キャッシュサイズの異なる 2 台の PC を用いて評価を行った．

また，GMP を用いたアプリケーションとして，まず，基本的な関数である mpz_add などを指定回数分だけ繰り返すものを作成し，実験に用いた．次に，これらの関数を組み合わせて作成したアプリケーションとして，GMP のホームページで公開されているベンチマークソース [16] からダウンロードした円周率計算プログラム Pi（chudnovsky のアルゴリズム）と RSA の計算プログラムの 2 種類を用いた．

5.2 実験結果

5.2.1 基本的な関数の速度比較

表 2 に，GMP ライブラリで用意された基本的な関数を L 回実行した場合の実行時間の一覧を示す．表 2 の上 2 行は $L=10^8$，下 2 行は $L=10^6$ 回，それぞれの関数を実行した場合の実行時間であり，単位は秒である．ビット長は，各関数に与える引数のビット長を意味しており，mpz_add/sub/mul/mod/tdiv_q はそれぞれ，与えられた 2 つの引数の加算，減算，乗算，除算 (剰余)，除算 (商) の結果を返す関数である．また，mpz_urandomb は引数で指定したビット長の乱数を返す関数である．表中の（）内の数値は，ネイティブ方式の実行時間に対する，PNaCl 方式の実行時間の比率を示している．

この表から，ビット長が 10^2bit 程度の場合，mpz_mul 以外の関数では実行時間比率が約 1 倍となり，ネイティブ方式と PNaCl 方式の実行時間の差がほとんど無いことが分かる．一方，mpz_mul ではビット長が 10^2bit の場合でもネイティブ方式の約 3 倍の時間がかかってしまう．これは，10^2bit の引数 2 つを掛け合わせた結果の積が 10^4bit と長くなってしまう事が原因と考えられる．また，mpz_tdiv_q では，ビット長が 10^6bit まではほぼ比率が 1 であり，ビット長が 10^8bit と大きくなっても比率が 2.5 程度と，比率が比

表 2 Web ベース VC 上で GMP の基本関数を L 回実行した際の実行時間（単位：秒）
（実験環境 1 で測定．() 内はネイティブ方式の実行時間に対する比率）

ビット長	ループ数	mpz_add	mpz_sub	mpz_mul	mpz_mod	mpz_tdiv_q	mpz_urandomb
10^2bit	L=10^8	*1 2.07 (1.20)	*1 1.55 (0.83)	14.41 (3.08)	16.78 (1.28)	10.01 (1.23)	7.21 (1.22)
10^4bit		63.98 (3.33)	106.54 (5.53)	*2 18,556.67 (5.27)	435.11 (7.47)	11.94 (1.22)	287.60 (1.01)
10^6bit	L=10^6	65.80 (3.14)	106.10 (5.09)	*4 46,245.67 (3.66)	421.11 (8.44)	*3 0.15 (0.94)	283.27 (1.01)
10^8bit		6,697.23 (1.86)	10,819.67 (2.98)	*5 9,915,449.80 (3.80)	*4 44,572.66 (4.99)	11.28 (2.64)	*3 28,601.13 (0.97)

較的小さくなる．この結果から，ネイティブ方式と PNaCl 方式の実行速度の比率は，演算における引数と計算結果のビット長のうち最も長いものに依存して変化すると推測できる．

次に，引数のビット長に対する実行時間比率の変化を詳しく見るため，図 10 に，ネイティブ方式と PNaCl 方式の実行時間の比率をグラフ化したものを示す．この図から，mpz_add/sub/mod はビット長に対してほぼ同じような変化をしていることが分かる．また，それぞれ 10^4bit から 10^5bit あたりで比率の増加が頭打ちとなり，特にビット長が大きな場合は，mpz_add であればネイティブ方式の約 2 倍，mpz_sub であれば約 3 倍と，PNaCl 方式が比較的高速であることが分かる．また，ビット長が 10^4bit から 10^7bit 程度の場合，mpz_sub は mpz_add の約 1.5 倍の比率となっており，同じような演算でありながら実行速度が大きく異なる．このことから，GMP を利用したアプリケーションの高速化法として，mpz_sub の代わりに mpz_add を用いる（例えば暗号理論等で用いられる有限体上では，n を法とした a の減算は，n-a の加算と同じである）といった方法が考えられる．このような置き換えによる Web ベース VC 上での実行速度の高速化は，今後の課題の 1 つである．

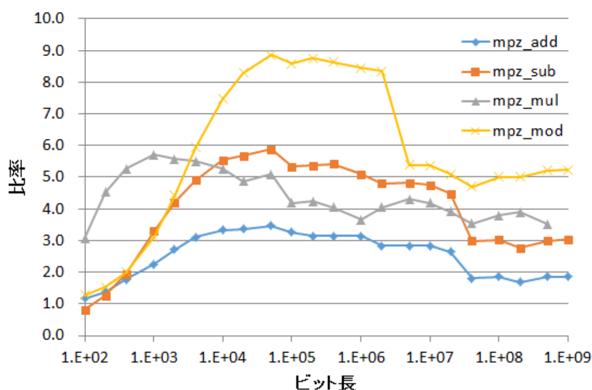


図 10 ビット長に対する各種基本関数の実行時間比率（実験環境 1）

*1 L=10^9 回実行して 10^{-1} 倍した時間
*2 L=10^5 回実行して 10^3 倍した時間
*3 L=10^8 回実行して 10^{-2} 倍した時間
*4 L=10^3 回実行して 10^3 倍した時間
*5 L=10 回実行して 10^5 倍した時間

5.2.2 ビット長に対する実行時間の比率とキャッシュミス数の変化

図 11 に，mpz_add を異なる実験環境で実行した場合における，引数のビット長に対する実行時間の比率とキャッシュミス数を示す．キャッシュミス数は，3.3 節で述べた valgrind を使用して，ネイティブ方式で実行した際の L1/L3 キャッシュにおけるデータキャッシュミスの数値を元に，ループ数を L=1 と L=10 として実行した際のキャッシュミス数の差を 9 で割った値をプロットした．これは，ループ内で発生するキャッシュミス数の平均値を示している．

この図から，実験環境 1（L3=8MB）の場合は引数が 2.5×10^7bit で比率が急減しているのに対し，実験環境 2（L3=3MB）の場合は，そのビット長の約 3/8 倍の 10^7bit で比率が急減している事が分かる．これは，それぞれのビット長においてネイティブ方式の実行時間が急増するために起きる現象である．この結果から，L3 キャッシュの容量は，ネイティブ方式の実行時間に大きく影響を与えるのに対し，PNaCl 方式の実行時間にはそれほど影響を与えない事が分かる．

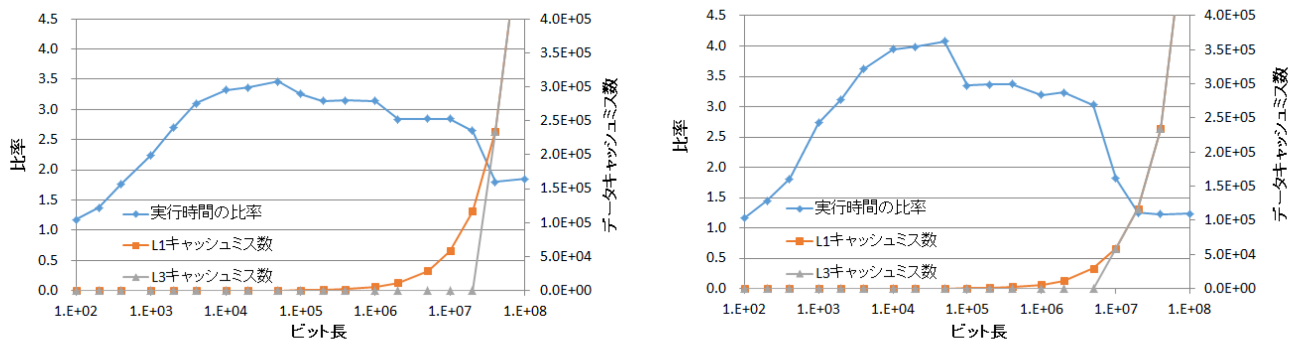
この原因に対する推測として，PNaCl 方式におけるキャッシュミス隠蔽の発生が考えられる．3.1 節で述べたように，PNaCl 方式では SFI 処理などのオーバヘッドのためネイティブ方式と比較して速度が低下するため，データキャッシュミスが発生しても実行速度にそれほど大きな影響を与えない（キャッシュミスの隠蔽が発生）という可能性がある．この可能性に関する詳細な考察と原因の究明は，今後の課題の 1 つである．

5.2.3 複数種類の関数を用いたアプリケーションの動的メモリ確保量と実行時間

表 3 に，問題サイズを変化させた場合に対する各アプリケーションの動的メモリ確保量と実行時間を示す．それぞれ，左側は問題サイズおよびネイティブ方式の動的メモリ確保量（malloc 等で動的に確保されたメモリの最大確保量）であり，右側は PNaCl 方式の実行時間と実行時間比率である．ただし，ネイティブ方式で Clang を用いて Pi をコンパイルした際にエラーが発生したため，コンパイル

*6 10^2 回実行して平均化した時間
*7 10^4 回実行して平均化した時間
*8 10^6 回実行して平均化した時間

© 2014 Information Processing Society of Japan



(a) 実験環境 1（L3 : 8MB）　(b) 実験環境 2（L3 : 3MB）

図 11　mpz_add の実行時間の比率とデータキャッシュミス数

表 3　複数種類の関数を用いたアプリケーションの動的メモリ確保量と実行時間（単位：秒）（実験環境 1）

問題サイズ	Pi (GMP 使用)		RSA (GMP 使用)		姫野ベンチ (GMP 未使用)	
小	10^5 桁 (916KB)	*6 0.24 (3.76)	10^2bit (15.8KB)	*8 0.000124 (2.20)	33×33×65 (0KB)	0.34 (1.09)
中	10^6 桁 (9.27MB)	4.97 (4.01)	10^3bit (15.8KB)	*7 0.033 (5.29)	65×65×129 (0KB)	3.16 (1.10)
大	10^7 桁 (91.4MB)	85.61 (3.93)	10^4bit (93.5KB)	1.65 (5.40)	129×129×257 (0KB)	39.71 (1.13)
特大	5×10^7 桁 (462MB)	578.87 (3.66)	10^5bit (3.30MB)	395.48 (4.75)		

には gcc を用いている．また，各方式の実行時間は，基本的には 3 回実行した際の平均値を秒で示している．この結果から，メモリを約 500MB 使用するような大きなアプリケーションでも，Web ベース VC 上で実行可能であることが確認できる．

また，図 12 に，Pi と RSA の問題サイズに対する実行時間の比率をグラフ化したものを示す．この結果から，GMP を使用したアプリケーションでは，基本的な関数の実験結果と同様に，実行時間の比率がある程度の値で頭打ちになり，問題サイズを大きくすれば比率が下がることが確認できる．また，L3 キャッシュ容量が小さい実験環境 2 の方が比率が低い．これは，5.2.2 節で述べたように，L3 キャッシュの容量がネイティブ方式の実行時間に大きく影響を与えているためであると推測できる．

6.　まとめと今後の課題

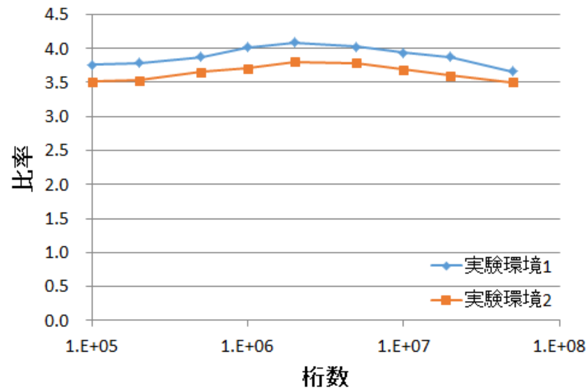
本稿では，GMP を用いて C/C++で記述されたアプリケーションを Web ベース VC 上で実行した場合の性能を評価した．性能評価のための準備として，GMP ライブラリファイルの生成方法と PNaCl 用実行ファイルへの変換方法を詳細に説明し，任意のアプリケーションを PNaCl 方式で実行する方法を説明した．ネイティブ方式と PNaCl 方式の実行時間の比較結果から，同じ GMP の基本演算でも，PNaCl 方式で高速に実行できる mpz_add などと，実行に大きな時間がかかってしまう mpz_mod などに大別できることを明らかにした．また，両方式の実行時間の比率には，引数となる変数に対して頭打ちとなるビット長が存在することや，その比率が L3 キャッシュの容量に大きく依存することを示した．さらに，円周率と RSA を計算するアプリケーションを用いてメモリ使用量等に関する調査を行う事で，Web ベース VC 上でどの程度の規模のアプリケーションが実行できるかを示した．

今後の課題として，性能評価の節に示した通り，mpz_sub を mpz_add に置き換えるなどしてアプリケーションを高速化することや，GMP 以外の様々なライブラリを用いた場合の性能評価などが挙げられる．

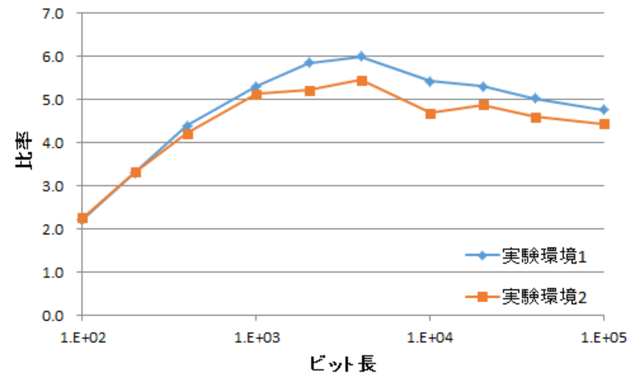
参考文献

[1] SETI@home: http://setiathome.berkeley.edu.
[2] BOINC: http://boinc.berkeley.edu.
[3] PNaCl : https://developers.google.com/native-client/dev/.
[4] asm.js: http://asmjs.org.
[5] 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.
[6] D. Kondo, Javadi, B., Malecot, P., Cappello, F. and Anderson, D.P., "Cost-benefit analysis of Cloud Computing

ⓒ 2014 Information Processing Society of Japan



(a) Pi



(b) RSA

図 12 複数種類の関数を用いたアプリケーションにおける問題サイズに対する実行時間比率の変化

versus desktop grids", *IPDPS*, pp. 1–12 (online), 2009.

[7] 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", 情報処理学会研究報告 2014-HPC-143(29), pp. 1–8, 2014.

[8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), pp.75–86, 2004.

[9] 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.

[10] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar "Native Client: A Sandbox for Portable, Untrusted x86 Native Code" Will appear in the 2009 IEEE Symposium on Security and Privacy

[11] naclports: https://code.google.com/p/naclports/

[12] depot_tools: http://www.chromium.org/developers/how-tos/depottools

[13] valgrind: http://valgrind.org/

[14] depot_tools.git: https://chromium.googlesource.com/chromium/tools/depot_tools.git

[15] naclports.git: https://chromium.googlesource.com/external/naclports.git

[16] GMPbenchmark: https://gmplib.org/download/misc/