

# サーバ支援型通信に基づく並列ボランティアコン ピューティングに関する研究

A Study on Parallel Volunteer Computing Based on Server Assisted Communication

猪原 圭一  
Keiichi Inohara

2026 年 3 月

## 概要

コンピュータの性能向上に伴い、様々な研究分野で科学技術計算が不可欠になっており、計算問題の規模や取り扱われるデータ量が増加している。大規模な計算問題を実行するためのシステムの 1 つに、ボランティアコンピューティング (VC) がある。VC は、インターネットに接続されたパーソナルコンピュータ (PC) などに内在する余剰計算資源を提供してもらい、大規模な分散計算を行う手法である。VC には、高性能な分散計算環境を安価に構築できる利点があるため、大規模な計算問題の実行基盤として注目を集めている。

VC の計算ノード (ワーカ) は、一般ユーザが所有する PC などであるため、一般的には、ワーカ同士が直接通信を行うようには構成されておらず、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。このため、既存の VC では、個々の計算 (ジョブ) が独立した分散計算のみが対象とされ、計算中にワーカ間の通信を伴う並列計算は対象外とされてきた。このことが、VC の適用範囲を狭める要因となっており、言い換えれば、VC の環境で並列計算を実行できれば、VC の多様な分野への貢献が期待できる。

本研究では、並列計算を実行可能な VC (並列 VC) を実現することを目的とし、サーバ支援型通信のアイデアに基づく並列 VC システムを提案、実装し、並列 VC の実現可能性を示す。提案する並列 VC システムは、並列 VC サーバと並列 VC クライアントから構成される。並列 VC サーバは、サーバ支援型通信システムとジョブ配布システムから構成され、HTTP 通信により、ワーカ間の通信機能とジョブ配布機能を提供する。並列 VC クライアントには、一般的な Web ブラウザを使用し、WebAssembly [1] にコンパイルされたジョブを実行することにより、VC に参加するための難易度を軽減する。また、ワーカの離脱に対処するために、専用ソフトウェアのインストールが不要で特定のオペレーティングシステム (OS) に依存しないソースコードレベルのチェックポイント機能を提案し、実装する。

性能評価により、サーバ支援型通信を用いることで、VC で並列計算が実行可能であることを確認した。また、ジョブ配布機能や Web ブラウザでの計算実行、チェックポイント機能が、想定される計算時間に対して大きなオーバーヘッドにならないことを確認した。さらに、離脱が発生する環境において、チェックポイント機能が並列 VC に対して有効であることがわかった。これらの結果から、並列 VC の実現可能性を明らかにした。

# 目次

|       |                                           |    |
|-------|-------------------------------------------|----|
| 第 1 章 | 緒言                                        | 1  |
| 1.1   | 本研究の背景 . . . . .                          | 1  |
| 1.2   | 本論文の目的と構成 . . . . .                       | 2  |
| 第 2 章 | VC の種類と既存手法                               | 4  |
| 2.1   | まえがき . . . . .                            | 4  |
| 2.2   | 分散 VC . . . . .                           | 4  |
| 2.3   | 並列 VC . . . . .                           | 7  |
| 2.4   | 既存手法 . . . . .                            | 9  |
| 2.5   | むすび . . . . .                             | 11 |
| 第 3 章 | サーバ支援型通信を基にした並列 VC サーバ                    | 12 |
| 3.1   | まえがき . . . . .                            | 12 |
| 3.2   | 並列 VC サーバの概要 . . . . .                    | 12 |
| 3.3   | サーバ支援型通信システム . . . . .                    | 14 |
| 3.4   | ジョブ配布システム . . . . .                       | 22 |
| 3.5   | 性能評価 . . . . .                            | 27 |
| 3.6   | むすび . . . . .                             | 37 |
| 第 4 章 | WebAssembly を使用したブラウザベースの並列 VC クライアント     | 39 |
| 4.1   | まえがき . . . . .                            | 39 |
| 4.2   | 並列 VC クライアント . . . . .                    | 39 |
| 4.3   | ベンチマークを用いたブラウザベースの並列 VC クライアントの性能評価 . . . | 45 |
| 4.4   | むすび . . . . .                             | 46 |
| 第 5 章 | ソースコードレベルで実現するチェックポイント機能                  | 49 |
| 5.1   | まえがき . . . . .                            | 49 |
| 5.2   | チェックポイント機能 . . . . .                      | 50 |

|       |                                |    |
|-------|--------------------------------|----|
| 5.3   | チェックポイント機能の動作確認と性能評価 . . . . . | 55 |
| 5.4   | むすび . . . . .                  | 62 |
| 第 6 章 | 結言 . . . . .                   | 64 |
| 6.1   | 結論 . . . . .                   | 64 |
| 6.2   | 今後の課題 . . . . .                | 66 |
| 謝辞    |                                | 67 |
| 参考文献  |                                | 68 |



# 目次

|      |                                                            |    |
|------|------------------------------------------------------------|----|
| 2.1  | 分散計算モデル . . . . .                                          | 5  |
| 2.2  | 並列計算モデル . . . . .                                          | 8  |
| 3.1  | サーバ支援型通信の概要 . . . . .                                      | 13 |
| 3.2  | HTTP_Send 関数と HTTP_Recv 関数の実行の概要 . . . . .                 | 17 |
| 3.3  | HTTP_Gather 関数の実行の概要 . . . . .                             | 21 |
| 3.4  | 変換ツールの概要 . . . . .                                         | 22 |
| 3.5  | 配布の流れ . . . . .                                            | 23 |
| 3.6  | HTTP_Send/ HTTP_Recv 関数の実行時間 (W = 2) . . . . .             | 29 |
| 3.7  | HTTP_Send/ HTTP_Recv 関数の実行時間 (data size: 8 Byte) . . . . . | 29 |
| 3.8  | HTTP_Gather 関数の実行時間 (W = 2) . . . . .                      | 30 |
| 3.9  | HTTP_Gather 関数の実行時間 (data size: 8 Byte) . . . . .          | 30 |
| 3.10 | IS (クラス W) の実行時間 . . . . .                                 | 31 |
| 3.11 | EP (クラス A) の実行時間 . . . . .                                 | 32 |
| 3.12 | EP (クラス A) の実行時間 (MPI: 1PC, VC: 4PC) . . . . .             | 33 |
| 3.13 | 姫野ベンチマークの実行時間 . . . . .                                    | 34 |
| 3.14 | send.php と recv.php の実行時間 . . . . .                        | 36 |
| 3.15 | 簡易的なクライアントの実行時間 . . . . .                                  | 37 |
| 3.16 | ワークの離脱が発生した場合の EP (クラス W) の実行時間 . . . . .                  | 38 |
| 4.1  | Emscripten による C 言語のコンパイル . . . . .                        | 41 |
| 4.2  | ブラウザ上での WASM の実行 . . . . .                                 | 42 |
| 4.3  | Web ブラウザ上で実行可能な並列計算の実装 . . . . .                           | 44 |
| 4.4  | IS (クラス S) の実行時間 . . . . .                                 | 47 |
| 4.5  | IS (クラス W) の実行時間 . . . . .                                 | 48 |
| 5.1  | 提案手法の実装の概要 . . . . .                                       | 50 |

|     |                                            |    |
|-----|--------------------------------------------|----|
| 5.2 | チェックポイントファイルの形式 . . . . .                  | 52 |
| 5.3 | アドレステーブルの例 . . . . .                       | 53 |
| 5.4 | 構造体テーブルの例 . . . . .                        | 54 |
| 5.5 | 各配列における要素数に対するチェックポイントの保存時間 . . . . .      | 59 |
| 5.6 | 各配列における要素数に対するチェックポイントの復元時間 . . . . .      | 60 |
| 5.7 | 配列と複数の変数における要素数に対するチェックポイントの保存時間 . . . . . | 61 |
| 5.8 | 配列と複数変数における要素数に対するチェックポイントの復元時間 . . . . .  | 62 |
| 5.9 | 再実行とチェックポイント機能の IS の実行時間の比較 . . . . .      | 63 |

# 表目次

|     |                                               |    |
|-----|-----------------------------------------------|----|
| 2.1 | VC の実例 . . . . .                              | 6  |
| 3.1 | HTTP_Send 関数と HTTP_Recv 関数の DB テーブル . . . . . | 17 |
| 3.2 | HTTP_Gather 関数の DB テーブル . . . . .             | 20 |
| 3.3 | PC の仕様 . . . . .                              | 28 |
| 3.4 | 使用するソフトウェア . . . . .                          | 28 |
| 3.5 | 評価条件 . . . . .                                | 35 |
| 4.1 | PC の性能 . . . . .                              | 45 |
| 4.2 | 使用するソフトウェア . . . . .                          | 46 |
| 5.1 | 評価環境 . . . . .                                | 58 |
| 5.2 | PC の仕様 . . . . .                              | 58 |
| 5.3 | 使用するソフトウェア . . . . .                          | 59 |

# 第 1 章

## 緒言

### 1.1 本研究の背景

コンピュータの性能向上に伴い、様々な研究分野で科学技術計算が不可欠になっており、計算問題の規模や取り扱われるデータ量が増加している。このような大規模な計算問題は、一般的には、個々の計算が独立した分散計算や通信を行いながら計算を行う並列計算の形式になっており、スーパーコンピュータやグリッドシステムなどで実行されている。スーパーコンピュータとグリッドシステムには、以下のような違いがある。スーパーコンピュータとは、大規模な科学技術計算を実行することを前提に開発されたコンピュータである。そのため、高い計算能力が必要とされるリアルタイムの処理や高精度なシミュレーションなどを実行可能である。スーパーコンピュータの性能ランキングである Top500[2] によると、現在 1 位となっている El capitan [3] は、CPU と GPU を合わせて 11,039,616 コアを搭載し、LINPACK [4] ベンチマークで 1,742PFlops (Peta Floating-point Operations Per Second) もの計算性能を記録している。グリッドシステムとは、各地に散在する研究機関や計算センターが保有している高性能なコンピュータを専用のネットワークで接続し、1 台のコンピュータであるかのように、相互に計算資源を利用できるようにしたシステムである。既存の高性能な計算資源を利用することにより、さらに高性能な計算環境を低コストで実現可能である。これらのシステムは高い計算能力がある一方で、開発や運用に費用がかかることや専用のソフトウェアが必要となることが欠点となる。例えば、El capitan の開発には、およそ 6 億ドルもの費用が投じられており [5]、また、グリッドシステムを構築するためには UNICORE [6] のようなミドルウェアをインストールする必要がある。このため、これらのシステムは、誰もが気軽に利用できるものではない。

大規模な計算問題を実行するためのシステムの 1 つに、ボランティアコンピューティング (VC) がある。VC とは、インターネットで接続されたパーソナルコンピュータ (PC) などに内在する余剰計算資源を提供してもらい、大規模な分散計算を行う手法である。VC は、グ

リッドシステムの一つであるデスクトップグリッドとして分類されているが、組織が管理するコンピュータではなく、一般ユーザが管理する PC などを計算ノード（ワーカ）とする点が大きく異なる。一般ユーザが自身の PC などを用いて文書の作成や Web の閲覧を行っている間、CPU の計算資源は 10% 程度しか使われていないことが言われている。VC は一般ユーザから本来の処理には使用されない余剰計算資源を提供してもらうことで、高性能な分散計算環境を安価に構築できる利点がある。

VC は、様々な研究分野において、プロジェクトとして実際に運用されている。VC のプロジェクトの例として、地球外知的生命体を探索する SETI@home [7] やタンパク質の挙動をシミュレートし、がんなどの病気の治療法を開発する Folding@home [8]、がんや HIV の治療などの人道的な研究を行う World Community Grid [9] などが挙げられる。Folding@home は、28 万基の GPU と 480 万個の CPU コアを提供してもらい、約 1.01EFLOps (= 1,010PFLOps) を達成している [10]。これは、Top500 で 3 位にランクインしている Aurora [11] の約 1,012PFLOps に匹敵する性能である。このように、VC は、高性能な分散計算環境を安価に構築できるため、大規模な計算問題の実行基盤として注目を集めている。

VC のワーカは、一般ユーザが所有する PC などであるため、一般的には、ワーカ同士が直接通信を行うようには構成されておらず、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。このため、既存の VC では、個々の計算（ジョブ）が独立した分散計算のみを対象とし、計算中にワーカ間の通信を伴う並列計算は対象外とされてきた。このことが、VC の適用範囲を狭める要因となっており、言い換えれば、VC の環境で並列計算を実行できれば、VC の多様な分野への貢献が期待できる。

## 1.2 本論文の目的と構成

本研究では、並列計算を実行可能な VC（並列 VC）を実現することを目的とし、サーバ支援型通信のアイデアに基づく並列 VC システムを提案、実装し、並列 VC の実現可能性を示す。提案する並列 VC システムは、並列 VC サーバと並列 VC クライアントから構成される。並列 VC サーバは、サーバ支援型通信システムとジョブ配布システムから構成され、HTTP 通信により、ワーカ間の通信機能とジョブ配布機能を提供する。並列 VC クライアントには、一般的な Web ブラウザを使用し、WebAssembly [1] にコンパイルされたジョブを実行することにより、VC に参加するための難易度を軽減する。また、ワーカの離脱に対処するために、専用ソフトウェアのインストールが不要で特定のオペレーティングシステム (OS) に依存しないソースコードレベルのチェックポイント機能を提案し、実装する。

第 1 章では、緒言であり、本研究の背景と目的について述べる。

第 2 章では、VC の種類と既存手法について述べる。既存の VC の計算モデルと実例を述べ、

並列 VC の計算モデルと並列 VC を実現するために満たすべき要件を定義する。また、既存手法を挙げ、それらの問題点を述べる。

第 3 章では、サーバ支援型通信に基づく並列 VC システムを提案する。本システムは、2 つのサブシステムから構成されており、まず、ワーカ間の通信を実現するためのサーバ支援型通信システムについて説明する。次に、ジョブの配布・管理を行うジョブ配布システムについて説明する。また、既存の並列プログラムを本システムで実行可能な形式に変換するツールについて説明する。本手法を評価するために、各サブシステムの基本性能の評価と並列計算のベンチマークを使用した実行時間の評価を行う。

第 4 章では、WebAssembly を使用したブラウザベースの並列 VC クライアントを提案する。本章では、本手法の概要を述べ、Web ブラウザから第 3 章で説明したサーバ支援型通信システムと通信するための通信関数の実装について説明する。本手法の性能評価では、並列計算のベンチマークを使用し、本手法と C 言語で作成した簡易的なクライアントを比較し、実行時間のオーバーヘッドを評価する。

第 5 章では、ソースコードレベルで実現するチェックポイント機能を提案する。本章では、提案するチェックポイント機能の概要を説明し、チェックポイントの保存・復元を実行する関数の実装を説明する。また、チェックポイントで指定された位置から再開する機能について説明する。本手法の性能評価では、チェックポイントの保存/復元にかかる時間とワーカの離脱が発生した際に、チェックポイント機能を使用した場合の並列プログラムの実行時間を評価する。

最後に第 6 章で、サーバ支援型通信を基にした並列 VC について総括し、今後の課題について述べる。

## 第 2 章

# VC の種類と既存手法

### 2.1 まえがき

大規模な計算を実行するためのシステムとして、VC が注目されている。VC とは、インターネットで接続された PC に内在する余剰計算資源を提供してもらい、大規模な分散計算を行う手法である。VC は一般ユーザから本来の処理には使用されない余剰計算資源を提供してもらうことで、高性能な分散計算環境を安価に構築できる利点がある。

VC では、ユーザが所有する PC などをワーカとして使用するため、ワーカ同士が直接通信することが難しく、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。このため、既存の VC では、個々の計算が独立した分散計算のみが行われ、ワーカ同士が計算中に通信を行う並列計算はほとんど行われていない。しかし、現在、通信を伴わない計算はあまり多くなく、分散計算のみに対応する既存の VC が適用できる範囲は限定されている。

本章では、並列 VC を実現するために必要な要件を示すために、VC の種類と既存手法について述べる。2.2 節では、既存の分散 VC の計算モデルと実例を述べる。2.3 節では、並列 VC の計算モデルを定義し、並列 VC を実現するための要件を説明する。2.4 節では、既存の並列分散計算システムの構築手法と計算を途中から再開するための既存のチェックポイント手法を述べ、それらの欠点を述べる。

### 2.2 分散 VC

#### 2.2.1 分散計算モデル

既存の VC の計算モデルとして、マスタ・ワーカモデルに基づく分散計算モデルがある。このモデルは、VC において広く普及しているミドルウェアである BOINC [43] などで使用され

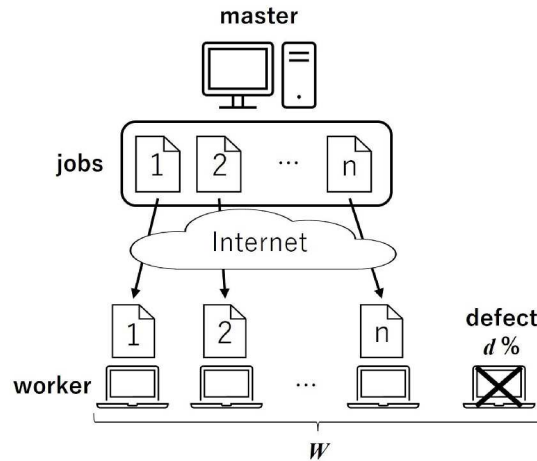


図 2.1: 分散計算モデル

ている。本モデルを図 2.1 に示し、詳細を以下に示す。

- VC は、マスタと  $W$  台のワーカにより構成される。
- マスタはワーカからのリクエストを受けて、1 個のジョブを 1 台のワーカへ配布する。
- 各ワーカは受信したジョブを計算し、この結果をマスタに返却する。
- $W$  台のワーカのうち  $d\%$  のワーカは、結果を返却せずにシステムから離脱する。
- 分散計算は、 $N$  個の独立したジョブからなり、全てのジョブに対する計算結果が確定すれば完了となる。

## 2.2.2 VC の実例

VC は、様々な研究分野において、プロジェクトとして実際に運用されている。既存の VC のプロジェクトの例として、地球外知的生命体を探索する SETI@home [7] やタンパク質の挙動をシミュレートし、がんなどの病気の治療法を開発する Folding@home [8], がんや HIV の治療などの人道的な研究を行う World Community Grid [9] などが挙げられる。表 2.1 に VC の実例を示す。この表で示すように、VC は、医学や数学、物理学など様々な分野で実際に活用されている。

各プロジェクトでは、様々な分野の計算が取り扱われるため、1 個のジョブの実行時間は異なる。例えば、CPU が Intel i7 10700 (8 コア)、メモリが 32GB の PC で実際に VC に参加したところ、1 個のジョブの実行時間は、Amicable Numbers では約 17 時間 1 分、Folding@home では 9 時間 50 分、Asteroids@home では 28 分 27 秒であった。Asteroids@home の 1 個の



表 2.1: VC の実例

| プロジェクト                     | 内容                                 |
|----------------------------|------------------------------------|
| Amicable Numbers [12]      | 友愛数を探す                             |
| Asteroids@home [13]        | 小惑星の物理的挙動を研究                       |
| Climateprediction.net [14] | 気候変動がインフラや健康, 天気, エネルギーに与える影響を調査   |
| DENIS@Home [15]            | 心血管疾患の治療法の研究                       |
| destrributed.net [16]      | 暗号解読や数学の未解決問題の証明                   |
| Einstein@home [17]         | 中性子星からの微弱な信号を探す                    |
| Folding@home [8]           | タンパク質の挙動をシミュレーションし, がんなどの病気の治療法を開発 |
| GPUGrid.net [18]           | gpu を利用して, 生命を構成する分子のシミュレーション      |
| LHC@home [19]              | 素粒および加速器物理学のシミュレーション               |
| LODA [20]                  | 整数列に対するアルゴリズムの探索                   |
| Milkyway@home [21]         | 天の川銀河の流れの解析                        |
| NFS@home [22]              | 巨大な整数の因数分解                         |
| NumberFields@home [23]     | 数の性質の研究                            |
| ODLK [24]                  | 魔方陣の探索                             |
| PrimeGrid [25]             | 素数の探索                              |
| Radioactive@Home [26]      | 放射線マップの作成                          |
| RakeSearch [27]            | 魔方陣の探索                             |
| Ramanujan Machine [28]     | 新しい数式の探索                           |
| RNA World [29]             | RNA 関係の研究を行う                       |
| Rosetta@home [30]          | タンパク質の 3 次元構造を解析                   |
| SETI@home [7]              | 地球外生知的命体の探索                        |
| SIDock@home [31]           | 新薬の発見                              |
| SRBase [32]                | シェルピンスキー/リーゼルの探索                   |
| World Community Grid [9]   | がんや HIV の治療などの人道的な研究               |
| YAFU [33]                  | 149 桁までの数の素因数分解                    |
| Yoyo@home [34]             | 既存の分散計算プロジェクトを BOINC で実行           |

ジョブは 30 分程度で完了したが, 多くのプロジェクトでは, 1 ジョブの実行時間は 10 時間を超えると言われている。

## 2.3 並列 VC

### 2.3.1 並列計算モデル

VC 上で並列計算を実行するために、並列計算モデルを以下のように定義する。並列計算は 1 個のジョブを並行して処理可能な複数の部分問題に分割し、それらを各ワーカに割り当て、ワーカ間で通信を行いながら、全体の計算を進める手法である。並列 VC では、マスタ・ワーカモデルで並列計算を行わせるために、1 個のジョブを互いに依存関係がある部分問題（サブジョブ）に分割する。本モデルを図 2.2 に示し、詳細を以下に示す。

- 並列 VC システムは、マスタと  $W$  台のワーカにより構成される。
- 並列計算は、 $N$  個の独立したジョブからなり、さらに、ジョブは  $P$  個の互いに依存関係のあるサブジョブに分割される。全てのサブジョブに対する計算結果が確定すればジョブは完了となる。
- マスタはワーカからのリクエストを受けて、 $P$  個のサブジョブを一度に  $P$  台のワーカへ配布する。 $P$  台のワーカ内で通信を行うため、これら  $P$  台のワーカを通信グループと呼ぶ。
- 各ワーカは受信したサブジョブを、必要に応じて他のワーカと通信を行いながら計算し、サブジョブ内で結果の返却が要求された場合は、この結果をマスタに返却する。したがって、全てのワーカが計算結果を返却するわけではない。
- $W$  台のワーカのうち  $d\%$  のワーカは、通信に応答せず結果を返却せずにシステムから離脱する。離脱が発生した場合、通信が行えなくなるため並列計算は完了しない。

上記の並列計算モデルは、ワーカ間の通信が発生することを除けば、2.2 節で説明した分散計算モデルと同じである。上記の並列計算では、ワーカの離脱に対処しなければ計算を完了させることは困難である。ワーカの離脱に対処するために、冗長計算に基づくいくつかの手法が研究されている [35, 36, 37]。

### 2.3.2 VC 上で並列計算を実行するための要件

本節では、並列 VC の実現に向けて、VC 上で並列計算を実行するための要件を定義する。まず、VC の特徴や分散 VC、既存の並列 VC を実現するための手法で一般的に言われている要件を述べる。次に、それらを踏まえて、VC 上で並列計算を実行するための要件を定義する。

VC の特徴として、次の 4 つが広く一般的に言われている。

特徴 1. PC のシャットダウンやネットワーク障害により、ワーカが離脱する。

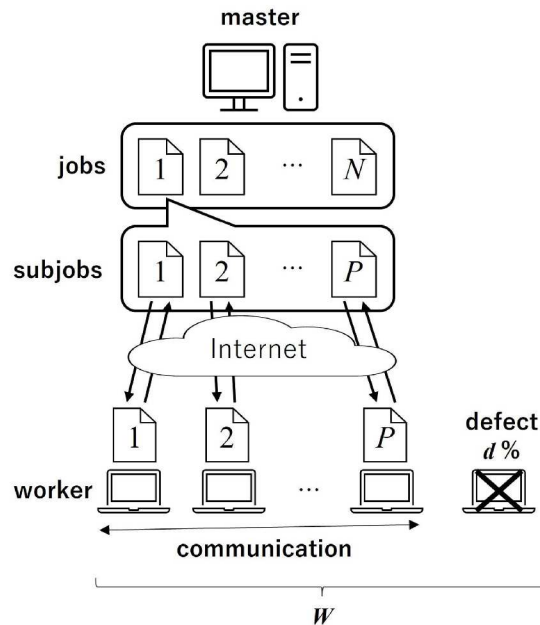


図 2.2: 並列計算モデル

特徴 2. 参加者の中に妨害者が存在する場合がある.

特徴 3. ワーカが一般ユーザの PC である.

特徴 4. VC の計算性能はワーカ数に依存する.

これらの特徴から、分散 VC の要件として次のことが言われている [36, 38, 39].

分散 1. ワーカの離脱への対処:

離脱が発生した場合、計算結果が返却されず、ジョブは完了しない. 既存の VC では、ジョブの完了を保証するために、ジョブの再配布や冗長化が行われている.

分散 2. 妨害者への対処:

妨害者は、誤った計算結果を返却する可能性がある. そのため、既存の VC では、ジョブの冗長化や多数決を行うことにより、計算結果の正しさを保証している.

分散 3. セキュリティの確保:

ワーカは一般ユーザの PC であるため、セキュリティを損なうソフトウェアのインストールや動作を可能な限りさせないようにする必要がある.

分散 4. リソースが少ないことへの対処:

一般ユーザの PC で実行できる程度のジョブにする必要がある.

分散 5. VC への参加の容易化:

VC へ参加するために、ソフトウェアのダウンロードやインストール、ユーザが行う設定などを可能な限り少なくする必要がある。

さらに、並列 VC を実現させるための既存手法では、並列 VC の要件として次のことが一般的に言われている [40, 41].

並列 1. インターネット上でワーカ間通信が可能である。

並列 2. ワーカの離脱・リザルトの誤りが発生する環境で高効率・高信頼な並列計算が可能である。

VC の特徴、分散 VC・並列 VC の特徴を踏まえ、本研究では、並列 VC を実現するために、次のように要件を定義する。

要件 1. インターネット上でワーカ間通信が可能である。

要件 2. PC 標準の環境で参加可能である。

要件 3. 離脱が発生する環境で効率よく計算可能である。

要件 1 と要件 3 は、並列 VC の要件として言われている並列 1 と並列 2 と同じものである。要件 2 は、分散 VC の要件として言われている分散 3 と分散 5 を組み合わせたものであるが、本研究では、要件を厳しくし、ソフトウェアのダウンロードやインストール、ユーザが行う設定を必要としない、PC 標準の環境で VC に参加可能であることを要件とする。また、本研究では、並列 VC を実現することを主な目的とするため、妨害者による計算誤りへの対処は対象外としている。

## 2.4 既存手法

### 2.4.1 既存の並列分散計算システムの構築手法

PC を用いて並列分散計算システムを構築するための有名なミドルウェアとして、HTCondor [42] と BOINC [43] がある。それぞれの詳細を以下に示す。

- HTCondor

HTCondor は主に、Windows, Mac OS, Linux といった異なるプラットフォームの PC を用いた PC クラスタ環境の構築を目的としている。並列計算では、一般的に、MPI (Message Passing Interface) ライブラリ [44] を用いて書かれた並列プログラムが使用される。HTCondor で並列プログラムを実行するためには、各ノードに他のノードから同一のユーザ名とパスワードでアクセス可能な MPI 環境が整備されている必要がある。このため、多数の一般ユーザが参加する VC には適していない。

- BOINC

BOINC は VC 環境を構築するための代表的なミドルウェアであり、主に分散計算を対象としている。現在の BOINC では、並列プログラムを実行できるものの、単一のワーカ上でのみ実行可能となっており、複数のワーカによる並列計算には対応していない。

VC 環境で並列計算を実行することを目的としたいくつかの手法も研究されている。代表的な手法を以下に示す。

- MPIWS

A. Calderón らは、MPIWS [40] と呼ばれるフレームワークを提案している。このフレームワークは、スタブクラスタとワーカから構成される。スタブクラスタは、サーバ内に構築され、並列プログラムの通信部分を実行し、計算部分をワーカに委託する。MPI プログラムは通信処理を gSOAP ライブラリ [45] を用いて実装したプログラムに変換されるため、ワーカ側では MPI 環境をインストールする必要がある。しかし、gSOAP ライブラリを用いた通信の詳細やスタブクラスタの実装については論文中で説明されておらず、実際の VC 上にこのフレームワークを実装するのは困難である。また、この手法ではワーカの離脱について考慮されていない。

- VolpexMPI

T. LeBlanc らは、ワーカの離脱に対処するための MPI 実装である VolpexMPI [41] を提案している。この手法では、並列プログラムの各プロセスを複数のワーカ上で冗長的に実行する。ワーカ間通信は、受信側ワーカが送信元ワーカを選択してデータを取得する方式で行われる。この手法では、MPI 環境のインストールが必要となり、MPI 通信を行うために、各ノードが他ノードからの接続を受け入れるサーバのような構成を取る必要がある。これはセキュリティの観点から望ましくなく、VC としては適していない。

## 2.4.2 既存のチェックポイント手法

チェックポイントとは、定期的にプログラムの実行状態をファイルに保存し、障害が起こったときに、保存したファイルからプログラムの実行状態を復元する手法である。これにより、チェックポイントに保存された実行状態からプログラムを再開可能になる。

チェックポイントの既存手法に CRIU [46], BLCR [47], FTI [48][49], BOINC API [50] などがある。詳細を以下に示す。

- CRIU, BLCR

これらの手法では、Linux の機能を使い、メモリやプロセスの情報をチェックポイントとして保存する。このため、ソースコードに変更を加えたり、ライブラリを使用したり

することなく、チェックポイントを実行可能である。しかし、メモリやプロセスの仕様は、実行するコンピュータによって変わるため、あるコンピュータで作成されたチェックポイントは別のコンピュータで復元できるとは限らない。

- FTI

本手法は、スーパーコンピュータのような高性能な計算環境でチェックポイントを実行するためライブラリである。しかし、利用するために、特殊なライブラリをインストールする必要がある。

- BOINC API

既存の VC ミドルウェアの BOINC では、BOINC クライアント上でジョブが実行されており、BOINC API を使ってクライアントとジョブの間で通信が行われる。提供されている機能は、チェックポイントの保存間隔をユーザが決める機能とチェックポイントを保存するときに、クライアントから強制終了などの操作を受け付けない機能である。しかし、チェックポイントの具体的な方法や作成するチェックポイントファイルの形式は、プログラマが実装する必要がある。

以上に述べたように、既存のチェックポイント手法には、OS に依存していたり、特殊なライブラリを必要としたりする問題があり、並列 VC には適用できない。

## 2.5 むすび

本章では、並列 VC を実現するための要件を示すために、VC の種類と既存手法について述べた。2.2 節では、既存の VC の計算モデルと実例を述べた。2.3 節では、並列 VC の計算モデルおよび、並列 VC を実現するための要件を定義した。2.4 節では、既存の計算環境の構築手法と計算を途中から再開するための既存のチェックポイント手法を述べ、それらの問題点を述べた。

本章で述べた並列 VC を実現するための要件を満たす並列 VC システムを 3 章、4 章、5 章で説明する。

## 第 3 章

# サーバ支援型通信を基にした並列 VC サーバ

### 3.1 まえがき

VC では、一般ユーザが所有する PC などをワーカとして使用するため、ワーカ同士が直接通信を行うようには構成されておらず、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。このため、既存の VC では、個々の計算（ジョブ）が独立した分散計算のみが対象とされ、計算中にワーカ間の通信を伴う並列計算は対象外とされてきた。

そこで、本章では、ワーカ間の通信を可能にし、VC 上で並列計算が実行可能な並列 VC サーバを提案する。提案する並列 VC サーバは、サーバ支援型通信システムとジョブ配布システムから構成される。各システムの実装には、Web アプリケーションで一般的に使用される、Apache Web サーバ [51] と MySQL データベース (DB) サーバ [52] を使用して実装する。これにより、ワーカに特別なソフトウェアをインストールすることなく、ワーカ間の通信を実現する。また、C 言語で必要最低限の VC の動作をする簡易的なワーカを作成し、実装した並列 VC サーバを用いた並列 VC が実行可能であることを実証する。

3.2 節では、提案する並列 VC サーバの概要を述べる。3.3 節では、サーバ支援型通信システムの詳細な実装について説明する。3.4 節では、ジョブ配布システムの詳細な実装について説明する。3.5 節では、サーバ支援型通信システムとジョブ配布システムの性能評価を行う。

### 3.2 並列 VC サーバの概要

本章では、サーバ支援型通信を基にした並列 VC サーバを提案する。サーバ支援型通信は、2.3.2 節で述べた要件 1 を満たすための、ワーカ同士の通信をサーバが支援するというアイディ

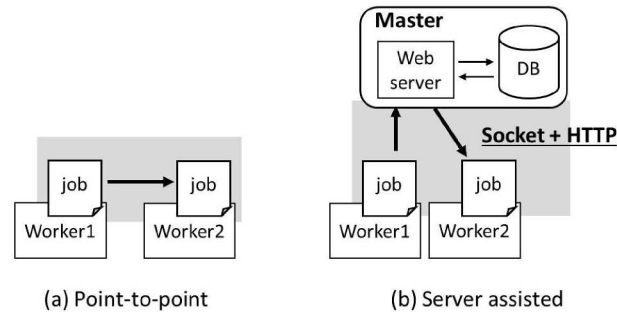


図 3.1: サーバ支援型通信の概要

アである。サーバ支援型通信では、ワーカ同士の通信が発生すると、マスタは通信を仲介するノードとして機能する。図 3.1 にサーバ支援型通信の概要を示す。サーバ支援型通信では、ワーカ 1 からワーカ 2 への直接通信をワーカ 1 からマスタ、マスタからワーカ 2 の 2 つの通信に置き換える。

ワーカにサーバの機能を持たせて、ワーカ間で直接通信する P2P を用いる手法 [41] も存在する。しかし、ファイアウォールで許可されなければ通信が遮断される問題があったり、ワーカがサーバとして動作するため、一般ユーザの PC のリソースを消費したり、セキュリティを低下させたりする問題がある。そのため、本研究では、VC の管理サーバであるマスタを使用したサーバ支援型通信を使用する。

本章で提案する並列 VC サーバは、サーバ支援型通信システムとジョブ配布システムで構成される。サーバ支援型通信システムは、ワーカ間の通信機能を提供するものである。また、ジョブ配布システムは、VC サーバとして必須機能であるジョブ配布機能を提供するとともに、離脱が発生した場合の対処を行うものである。

本章では、以下に示す基本方針に基づいて、並列 VC サーバを開発した。

- VC に参加するための要件や難易度を軽減するために、マスタとワーカ間の通信には、HTTP プロトコルを用いた標準的なソケットライブラリを使用する。ソケットライブラリは TCP 通信を使用する OS に標準的に備わっているため、ユーザは、追加で通信用のソフトウェアを PC にインストールする必要がない。
- ユーザが並列 VC システム用のプログラムを用意する手間を軽減するために、MPI 通信ライブラリで書かれたプログラムを並列 VC システムで実行可能にする。このために、並列 VC 用のプログラムに変換するツールを提供する。
- 安定したシステムを構築するために、マスタには、マスタとワーカ間の通信を処理する Web サーバと、通信とジョブの状態を管理する DB サーバを使用する。



現在の実装では、Web アプリケーションで一般的に使用される、Apache Web サーバと MySQL DB サーバを使用する。Web サーバは、マルチプロセス処理により複数のワーカと通信を行う機能を提供し、DB サーバは、ワーカから受信したデータの集中管理と高効率なデータアクセスを機能を提供する。

### 3.3 サーバ支援型通信システム

#### 3.3.1 通信関数の実装

本節では、サーバ支援型通信システムの通信の実装および詳細な通信手順を説明する。サーバ支援型通信を行うための機能は通信関数として実装され、ワーカが通信関数を実行することで通信が行われる。通信関数は、並列計算で一般的に使用される MPI 通信関数に対応するように実装される。通信関数の例として、一対一通信の代表的な関数である MPI の `MPI_Send/MPI_Recv` 関数に対応する `HTTP_Send/HTTP_Recv` 関数と集団通信の代表的な関数である `MPI_Gather` 関数に対応する `HTTP_Gather` 関数の実装について説明する。

##### `HTTP_Send/HTTP_Recv` 関数

`HTTP_Send/HTTP_Recv` 関数は、指定されたワーカとの間でデータを送受信する通信関数である。これらの関数は、MPI の `MPI_Send/MPI_Recv` 関数に対応する。`HTTP_Send/HTTP_Recv` 関数の疑似コードをアルゴリズム 3.1, 3.2 に示す。これらの関数の形式と引数は、以下に示すとおりである。

`HTTP_Send(myid, numprocs, comm_count, buf,count,datatype,dest)`

`HTTP_Recv(myid, numprocs, comm_count, buf,count,datatype,source)`

- `myid`: 関数を実行するワーカの ID
- `numprocs`: 並列計算に参加するワーカ数（プロセス数）
- `comm_count`: 通信の一意の ID
- `buf`: データの開始アドレス
- `count`: データの要素数
- `datatype`: データの型
- `source/ dest`: 送信元/ 受信元ノードの ID

ワーカは、以下の手順で、各 PHP ファイルの実行を要求する HTTP リクエストを発行する。

- I. マスタと TCP 接続を確立する。

II. 通信を識別するために必要なデータを含む HTTP リクエストを発行する. 必要なデータは, 以下のとおりである.

- (1) HTTP\_Send\_ID.php  
myid, comm\_count, dest
- (2) HTTP\_Send\_DATA.php  
buf に格納されているデータが書かれたファイル  
(ファイル名: comm\_count\_myid\_dest.txt)
- (3) HTTP\_Send\_FLAG.php  
myid, comm\_count, dest
- (4) HTTP\_Recv\_Rec.php  
myid, comm\_count, source

III. マスタから HTTP リクエストに対する応答を受信し, TCP 接続を閉じる.

上記の II-(2) では, buf 内のデータがテキスト形式に変換され, ファイルとして送信される. II-(4) では, マスタから受信したファイル内のデータを datatype で指定された型に変換し, buf に格納される. 一方, マスタでは, HTTP リクエストを受信すると以下の処理が行われる.

(4) HTTP\_Send\_ID.php

この PHP では, DB レコードを作成し, 送信者に \OK" を返信する. HTTP\_Send/ HTTP\_Recv 関数で使用する DB レコードを表 3.1 に示す. 表 3.1 において, (a), (b), (c), (f) は, それぞれ, myid, dest, データファイルの保存場所, comm\_count に対応している. また, (d), (e) は, それぞれ, ワークが送信/ 受信を完了したことを示すフラグである.

(4) HTTP\_Send\_DATA.php

この PHP では, 受信したデータファイルをマスタに保存し, 送信完了フラグ (d) を 1 に設定した後, 送信者に \OK" を返信する.

(4) HTTP\_Send\_FLAG.php

この PHP では, 受信完了フラグ (e) をチェックし, 1 であれば送信者に \OK" を返信し, そうでなければ \NG" を返信する.

(4) HTTP\_Recv\_Rec.php

この PHP では, (d) をチェックし, 1 であれば要求されたデータファイルと \OK" の文字を送信者に返信し, (e) を 1 に設定する. そうでない場合, \NG" を返信する.

HTTP\_Send/ HTTP\_Recv 関数では, マスタから \OK" の返信を受信するまで, 上記の (3) の HTTP\_Send\_FLAG.php と (4) の HTTP\_Recv\_REC.php が  $T_{wait}$  秒間隔で繰り返し実行される.

---



---

アルゴリズム 3.1 HTTP\_Send 関数の疑似コード

---



---

```

function HTTP_Send( )
    HTTP_Send_ID.php; // (1)
    HTTP_Send_DATA.php; // (2)
    while (1) do
        HTTP_Send_FLAG.php; // (3)
        if Response = \OK\ then
            break;
        else
            wait (Twait);
        end if
    end while
end function

```

---



---



---



---

アルゴリズム 3.2 HTTP\_Recv 関数の疑似コード

---



---

```

function HTTP_Recv( )
    while (1) do
        HTTP_Recv_REC.php; // (4)
        if Response = \OK\ then
            break;
        else
            wait (Twait);
        end if
    end while
end function

```

---



---

図 3.2 に、HTTP\_Send/ HTTP\_Recv 関数を使用した一対一通信の例を示す。この例では、ID 0 のワーカー (Worker0) と ID 1 のワーカー (Worker1) 間で 3 回目の通信 (comm.count = 3) を行う場合を示している。まず、Worker0 はマスタに PHP (1) の実行を要求する。PHP (1) は、DB にレコードを作成し、フィールド (a), (b), (f) をそれぞれ、0, 1, 3 に設定する。次に、Worker0 は、マスタに、3.0.1.txt というデータファイルとともに PHP (2) の実行を要求する。PHP (2) は受信したデータファイルを保存し、保存場所をフィールド (c) に設定する。また、フィールド (d) を 1 に設定する。一方で、Worker1 は、マスタに PHP (4) の実行を要求する。フィールド (d) のフラグが 1 の場合、PHP (4) は \OK" の文字と要求されたデータ

表 3.1: HTTP\_Send 関数と HTTP\_Recv 関数の DB テーブル

|                                    |             |            |
|------------------------------------|-------------|------------|
| (a) 送信者 ID                         |             | (b) 受信者 ID |
| (c) comm_count_myid_dest:txt の保存場所 |             |            |
| (d) 送信完了フラグ                        | (e) 受信完了フラグ | (f) 通信回数   |

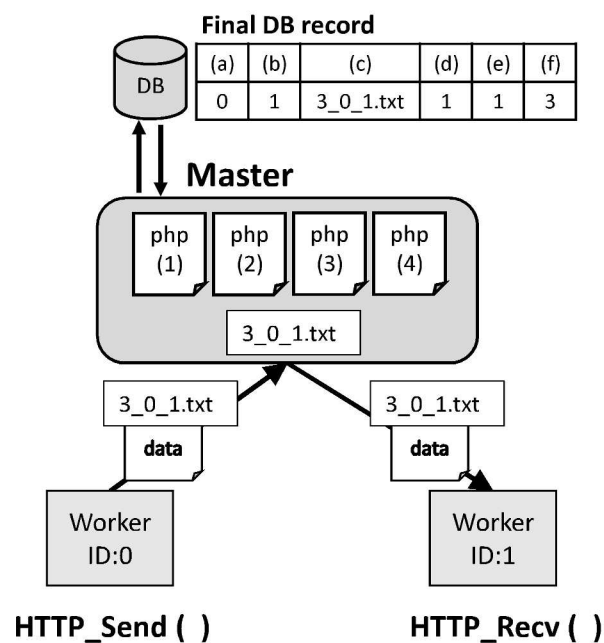


図 3.2: HTTP\_Send 関数と HTTP\_Recv 関数の実行の概要

ファイルを Worker1 に返信し、フィールド (e) を 1 に設定する。その後、Worker0 は、マスタに PHP (3) の実行を要求する。フィールド (e) のフラグが 1 の場合、PHP (3) は Worker0 に "OK" と返信し、通信が完了する。

#### HTTP\_Gather 関数

HTTP\_Gather 関数は、全てのワーカーからデータを集約し、集約したデータを指定されたワーカーに送信する通信関数である。この関数は、MPI の MPI\_Gather 関数に相当する。

並列計算に参加している全てのワーカーがアルゴリズム 3.3 に示す HTTP\_Gather 関数を実行する。この関数の実行中、マスタ側では PHP (1)-(4) が実行される。この関数の形式と引数を以下に示す。なお、説明は、前の節で説明していない引数についてのみ行う。

---

**アルゴリズム 3.3 HTTP\_Gather 関数の疑似コード**


---

```

function HTTP_Gather()
    HTTP_Gather_ID.php; // (1)
    HTTP_Gather_DATA.php; // (2)
    if myid = root then
        while (1) do
            HTTP_Gather_REC.php; // (4)
            if Response = \OK\ then
                break;
            else
                wait (Twait);
            end if
        end while
    else
        while (1) do
            HTTP_Gather_FLAG.php; // (3)
            if Response = \OK\ then
                break;
            else
                wait (Twait);
            end if
        end while
    end if
end function

```

---

HTTP\_Gather(myid, numprocs, comm\_count, sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root)

- sendbuf / recvbuf : 送信/受信データの先頭アドレス
- sendcount / recvcount: 送信/受信データの要素数
- sendtype / recvtype: 送信/受信データの型
- root: データを受信するノードの ID

全てのワークは、HTTP\_Send/HTTP\_Recv 関数と同じ 3つの手順 (I, II, III) に従って、マスタに HTTP リクエストを発行する。ただし、手順IIで実行する PHP が変更される。変

更後の手順Ⅱについて、以下に説明する。

Ⅱ. 通信を識別するために必要なデータを含む HTTP リクエストを発行する。必要なデータは、以下のとおりである。

- (1) HTTP\_Gather\_ID.php  
myid, numprocs, comm\_count
- (2) HTTP\_Gather\_DATA.php  
sendbuf に格納されているデータが書かれたファイル  
(ファイル名: comm\_count\_myid\_0.txt)
- (3) HTTP\_Gather\_FLAG.php  
myid, comm\_count
- (4) HTTP\_Recv\_Rec.php  
myid, comm\_count

HTTP\_Send/ HTTP\_Recv 関数と同様に、上記のⅡ-(2) では sendbuf 内のデータがテキスト形式に変換され、Ⅱ-(4) では、マスタから受信したファイル内のデータがテキストから recvtype で指定された型に変換され、recvbuf に格納される。

一方、マスタでは、ワーカーから HTTP リクエストを受信すると以下の処理が行われる。

- (4) HTTP\_Gather\_ID.php  
この PHP は、DB レコードを作成し、ワーカーに \OK" を返信する。HTTP\_Gather 関数で使用される DB レコードを表 3.2 に示す。表 3.2 の (c) は numprocs に対応し、その他は表 3.1 と同じである。(a) の受信者 ID は root に設定される。DB レコードは、最初の HTTP リクエストを受信したときのみ作成される。
- (4) HTTP\_Gather\_DATA.php  
この PHP は受信したデータファイルをマスタに保存し、送信カウンタ (d) を +1 した後、ワーカーに \OK" を返信する。(d) の値が (c) の値と等しい場合、送信完了フラグ (e) が 1 に設定される。
- (4) HTTP\_Gather\_FLAG.php  
この PHP は 受信完了フラグ (f) をチェックし、1 であれば、ワーカーに \OK" を返信し、0 の場合は \NG" を返信する。
- (4) HTTP\_Gather\_REC.php  
この PHP は、(e) をチェックし、1 の場合、保存されているデータファイルを集約し、\OK" の文字と集約したデータをファイルとしてワーカーに送信する。その後、(f) を 1 に設定する。(e) が 0 の場合、\NG" をワーカーに返信する。

表 3.2: HTTP\_Gather 関数の DB テーブル

|                                    |             |
|------------------------------------|-------------|
| (a) 受信者 ID                         |             |
| (b) comm_count_myid_root.txt の保存場所 |             |
| (c) プロセス数                          |             |
| (d) 送信カウンタ                         |             |
| (e) 送信完了フラグ                        | (f) 受信完了フラグ |
| (g) 通信数                            |             |

HTTP\_Gather 関数では、マスタから \OK" の返信を受信するまで、上記の PHP (3) と (4) が  $T_{wait}$  秒ごとに繰り返し実行される。

図 3.3 に、ID が 0, 1, 2 の 3 つのワーカ (Worker0, Worker1, Worker2) 間で 3 回目の HTTP\_Gather 関数を実行する例を示す。Worker0 はマスタに PHP (1) の実行を要求する。PHP (1) は DB レコードを作成し、フィールド (a), (c), (g) をそれぞれ 0, 3, 3 に設定する。次に、PHP (1) は Worker0 に \OK" を返信する。Worker1 と Worker2 も PHP (1) の実行を要求するが、この場合、PHP (1) は DB レコードを作成せず、\OK" のみを返信する。次に、Worker0 は 3.0.0.txt というデータファイルとともに PHP (2) の実行を要求する。PHP (2) は受信したデータファイルを保存し、フィールド (b) にその場所を追加する。次に、PHP (2) はフィールド (d) を +1 し、Worker0 に \OK" を返信する。Worker1 と Worker2 がそれぞれ 3.1.0.txt と 3.2.0.txt というデータファイルとともに PHP (2) の実行を要求すると、PHP (2) は Worker0 の場合と同様の動作を行う。この後、フィールド (d) と (c) の値が等しくなるため、送信フラグ (e) は 1 に設定される。Worker0 が PHP (4) の実行を要求した際、フィールド (e) の値が 1 の場合、PHP (4) は \OK" の文字列とデータファイルを Worker0 に送信する。Worker1 と Worker2 が PHP (3) を要求した際、フィールド (f) の値が 1 の場合、PHP (3) は \OK" を返信し、通信が完了する。

その他の全体通信も、HTTP\_Gather 関数と同様の方法で実装可能である。例えば、MPIReduce 関数に対応する HTTP\_Reduce 関数は、集約されたデータに対して合計など特定の操作を実行するように PHP (2) を変更することで実装される。さらに、MPIAllreduce 関数に対応する HTTP\_Allreduce 関数は、操作を実行した後にデータを全てのワーカが受信するように、PHP (3) を変更することで実装される。

提案する並列 VC システムでは、通信データはマスタ内の DB によって管理され、ワーカの動的な動作に対応可能である。つまり、ワーカがシステムから離脱しても、新たに追加されたワーカはマスタに保存された通信データを使用してジョブを引き継ぐことができる。

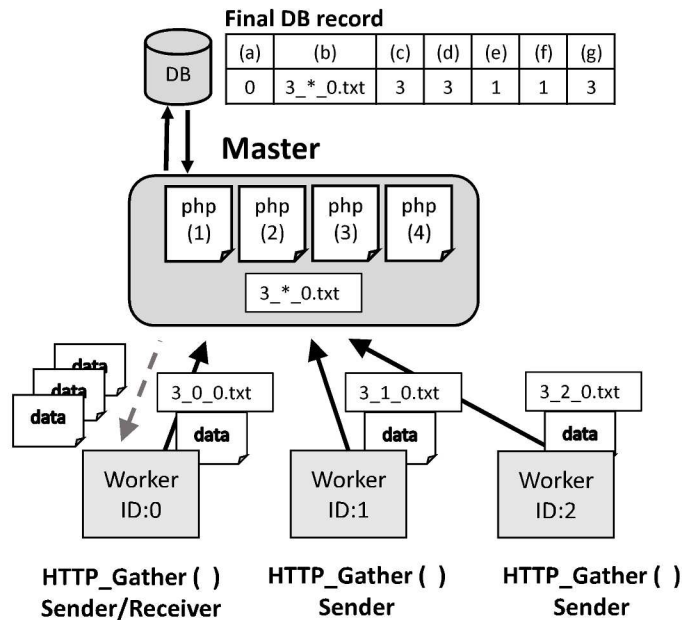


図 3.3: HTTP\_Gather 関数の実行の概要

### 3.3.2 プログラム変換ツール

並列プログラムは、通常、MPI 通信ライブラリを用いて記述される。インターネット上で MPI 通信をサポートする VC 環境を構築することは困難であるため、前節で紹介したサーバ支援型通信システムは、標準ソケットライブラリを用いた通信関数を提供する。ユーザが並列 VC システム用の通信関数を用いた並列プログラムを自ら作成する手間を省くため、並列 VC システム上で実行可能な並列プログラムを自動生成するプログラム変換ツールを開発した。変換ツールの概要を図 3.4 に示す。変換ツールの入力、MPI 通信関数を用いて記述された並列プログラムであり、出力は、前節で説明した動作をする通信関数を含むプログラムである。現在の変換ツールは、以下に挙げる一般的な MPI 関数をサポートしている。

- 一対一通信
  - { MPLSend, MPIRecv
  - { MPLIsend, MPIIrecv, MPIWait
- 集団通信
  - { MPLBarrier
  - { MPLBcast, MPLScatter



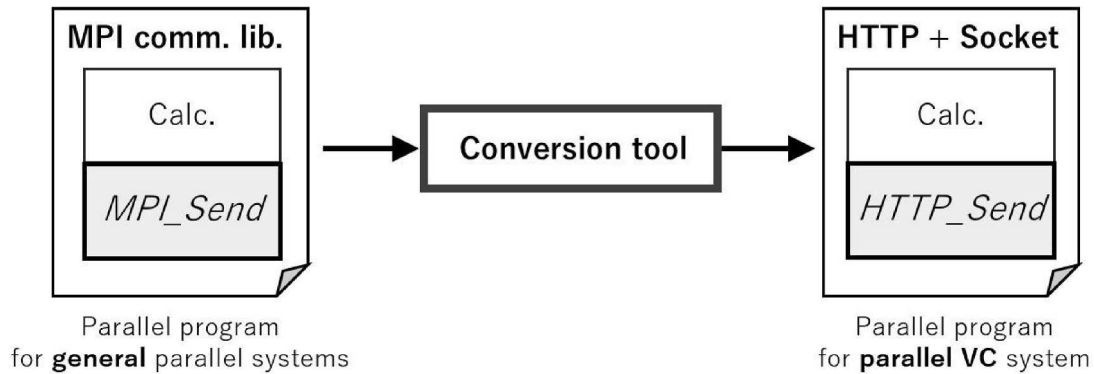


図 3.4: 変換ツールの概要

```

{ MPI_Gather, MPI_Allgather
{ MPI_Reduce, MPI_Allreduce
{ MPI_Alltoall, MPI_Alltoallv
  
```

この変換ツールは、上記の MPI 通信関数を、対応する HTTP\_ という名前の通信関数に置き換える。HTTP\_ 関数は、元の MPI 関数の引数に加え、`myid`, `numprocs`, `comm_count` の 3 つの引数が必要に応じて追加される。`myid` と `numprocs` は、MPI でも使用されるパラメータであり、ワーカの ID と通信グループ内のワーカ数 (P) に対応するものである。また、`comm_count` は、変換ツールによって通信関数の出現回数をカウントすることで管理される。この変換ツールでは、元のプログラムのうち、通信機能を除く計算部分はそのまま出力される。そのため、元のプログラムが C/ C++ 言語で記述されている場合、生成されたプログラムは標準的な C/ C++ コンパイラでコンパイル可能である。

## 3.4 ジョブ配布システム

### 3.4.1 ジョブ配布システムの実装

本節では、ジョブ配布システムの実装の詳細について説明する。ジョブ配布システムでは、VC の基本的な機能であるワーカへのジョブの配布処理とワーカの離脱の検知を行う。

まず、ワーカへのジョブ配布処理について説明する。ジョブ配布処理の流れを以下に示す。また、この流れを図 3.5 に示す。

1. ワーカが、マスタへジョブ配布リクエストを送信する。
2. マスタは配布するサブジョブを決定し、ワーカへその内容を送信する。

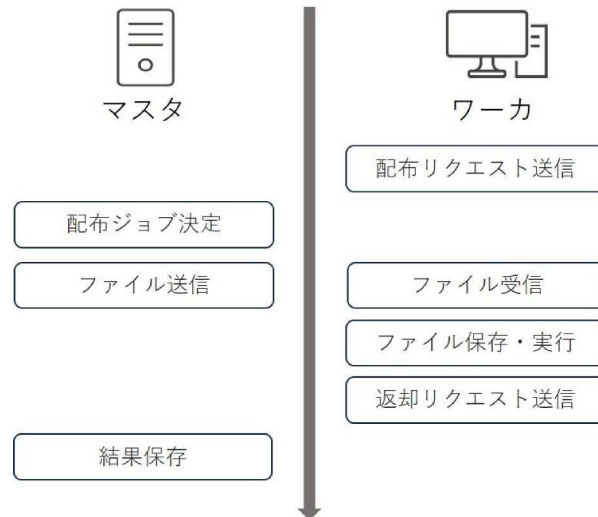


図 3.5: 配布の流れ

3. ワーカはサブジョブを保存し，実行する．
4. ワーカが，マスタへ計算結果とともに計算結果返却リクエストを送信する．
5. マスタは送信された計算結果をサーバ内に保存する．

サブジョブとは，2.3.1 節で述べたように，ジョブを構成する要素である．全てのサブジョブの計算が完了すれば，ジョブは完了となる．サブジョブの計算状態を管理するために，サブジョブの状態を以下のように定義する．

0. 未配布：まだワーカに配布されていない状態
1. 配布済み：ワーカに配布されたが，計算結果が返却されていない状態
2. 完了：ワーカが計算を完了し，結果がマスタに返却された状態

また，ジョブの状態を以下のように定義する．

0. 未配布：1 つもサブジョブがワーカに配布されていない状態
1. 配布中：未配布と配布済みのサブジョブが混在している状態
2. 配布済み：全てのサブジョブがワーカに配布され，かつ返却されていない計算結果がある状態
3. 完了：全てのサブジョブがワーカに配布され，かつ全ての計算結果が返却された状態

本システムでは、前述したサブジョブとジョブの状態は、DB によって管理される。それぞれの状態を管理するために、以下の 2 種類のテーブルを DB に作成する。

1. `table_registry`: ジョブ管理用テーブル

- `id`: ジョブ ID
- `table_name`: 各サブジョブテーブルの名前
- `rank_count`: ジョブの並列数
- `filepath`: ジョブファイルのパス
- `status`: ジョブの状態 (0: 未配布, 1: 配布中, 2: 配布済み, 3: 完了)

2. サブジョブテーブル: 各ジョブのサブジョブ管理用テーブル

- `id`: サブジョブ ID
- `rank`: サブジョブの担当番号 (並列計算において各ワーカが担当する計算部分を識別するための番号)
- `client_id`: サブジョブを配布したワーカ ID
- `status`: サブジョブの状態 (0: 未配布, 1: 配布済み, 2: 完了)

マスタが持つ全てのジョブの状態は、`table_registry` で管理され、各ジョブのサブジョブの状態は、それぞれ別のサブジョブテーブルで管理される。サブジョブテーブルでは、並列計算における各ワーカが担当する計算部分を示すために、各サブジョブがもつ固有の担当番号 (`rank`) を管理する。この番号により、各ワーカは自身が実行すべき計算内容を特定する。また、サブジョブテーブルでは、離脱の判定や計算結果の有効性を確認するために、サブジョブを配布したワーカ ID を記録する。ワーカ ID は、ワーカが VC システムに参加した際に、マスタがワーカに割り当てて一意の番号である。

ワーカからのジョブ配布/計算結果返却リクエストを処理するために、次の 2 つの PHP ファイルを作成する。ワーカからのジョブ配布リクエストを処理するために、`send.php` を作成する。`send.php` は、HTTP リクエストの受信時に以下の処理を順次実行する。

(1) 配布ジョブの決定

- DB にアクセスし、配布するサブジョブを決定する。

(2) HTTP ヘッダーの送信

- 決定したジョブに関する情報を HTTP ヘッダに格納しワーカへ送信する。

(3) ファイルの送信

- サブジョブの内容をワーカへ送信する。

配布するサブジョブは、ジョブ ID またはサブジョブ ID の小さい順に決定される。このようにすることで、ジョブごとにまとまった配布を行うことができ、ワーカ同士の通信待ち時間を

短縮することが可能である。send.php において、配布するサブジョブを決定する流れを以下に示す。

1. table\_registry テーブルをジョブ ID 順に走査し、未配布 (status=0) または配布中 (status=1) の最若番のジョブを特定する。
2. 特定したジョブのサブジョブテーブルをサブジョブ ID 順に走査し、未配布 (status=0) の最若番のサブジョブを配布するサブジョブに決定する。
3. 決定したサブジョブの状態を配布済み (status=1) に更新する。
4. ジョブが未配布状態 (status=0) の場合、配布中 (status=1) に更新する。

ワーカーからの計算結果返却リクエストを処理するために、receive.php を作成する。receive.php は、HTTP リクエストの受信時に以下の処理を順次実行する。

(1) 結果の有効性確認

- HTTP リクエストヘッダのワーカー ID とサブジョブテーブルに記録されたワーカー ID を照合する。
- ワーカー ID が一致しない場合は無効な結果としてリクエストを破棄する。

(2) 計算結果の受信

- ワーカーから計算結果が記録されたファイルを受信し、マスタ内に保存する。

(3) DB の更新

- 対応するサブジョブの状態を完了 (status=2) へ更新する。
- サブジョブテーブル内の全てのサブジョブの状態が完了の場合、対応するジョブの状態も完了 (status=3) へ更新する。

(4) 通信グループ処理の完了確認

- ジョブの状態が完了になった場合、計算結果の多数決を行い最終的なジョブの結果を決定する。

上記の (4) の通信グループ処理の完了確認は、並列計算を冗長実行する場合の処理である。これは、サブジョブの実行時に通信を行う通信グループ単位で冗長化を行う手法を想定したものである。本研究では、冗長化は行わず、1つの通信グループのみの実行を対象とする。

receive.php では、計算結果が返却された場合、以下の手順で DB 内のジョブとサブジョブの状態の更新を行う。

1. 該当サブジョブの状態を完了 (status=2) に更新する。
2. ジョブ内の全サブジョブの状態を確認し、以下のようにジョブの状態を更新する。
  - 全て完了 (status=2): ジョブを完了 (status=3) に更新する。
  - 配布済み (status=1) を含む: ジョブを配布済み (status=2) に更新する。

次に、ワーカの離脱の検知について説明する。VC システムでは、ワーカが予期せず離脱する可能性がある。本システムでは、制限時間  $T_{\text{limit}}$  を設定し、ジョブ配布から  $T_{\text{limit}}$  秒以上経過しても計算結果返却リクエストがない場合、ワーカの離脱と判定する。

並列 VC では 1 台のワーカの離脱が全体の計算を停止させる可能性があるため、離脱検知時には同一通信グループに属する全サブジョブの状態を未配布にリセットする。これにより、新たな通信グループに再配布することが可能となる。

具体的な処理は、以下に示すとおりである。

- 通信グループ内の最後のサブジョブを配布時、別プロセスで `timer.php` を起動する。
- `timer.php` は  $T_{\text{limit}}$  秒のスリープ後、通信グループ内の全サブジョブの状態を取得する。
- 取得した状態に未完了 (`status ≠ 2`) のものが含まれる場合、`timer.php` は通信グループ全体を未配布状態 (`status=0, client_id=null`) へリセットする。

### 3.4.2 簡易的なクライアントの実装

本システムの動作検証のため、マスタ・ワーカモデルに基づくクライアントプログラムも同時に実装した。このプログラムは C++ で記述され、HTTP リクエストを用いてマスタサーバと通信を行う。クライアントはマスタからジョブを受け取り、計算を実行した後、計算結果を返却する。各ワーカで実行されるクライアントプログラムは、以下の 2 つのフェーズで処理を実行する：

- ジョブ受信・実行フェーズ
  - (1) サーバへの接続確立
    - { マスタとのソケット通信による接続を確立する。
  - (2) ジョブ要求リクエスト送信
    - { `send.php` の実行を要求する HTTP リクエストをマスタに送信する。
    - { HTTP リクエストヘッダに自身のワーカ ID を含める。
  - (3) ジョブ受信
    - { HTTP レスポンスヘッダ内から、ジョブ実行に必要な情報を取得する。
    - { サブジョブを受信する。
  - (4) ジョブ保存
    - { サブジョブをローカルファイルへ保存する。
  - (5) ジョブ実行
    - { 保存したサブジョブを実行し、結果をファイルへ保存する。
- 結果返却フェーズ

- (6) サーバへの接続確立
  - { マスタとのソケット通信による接続を確立する.
- (7) 結果返却リクエスト送信
  - { `receive.php` の実行を要求する HTTP リクエストを送信する.
  - { HTTP リクエストヘッダにワーカ ID とジョブ情報を含める.
- (8) 結果送信
  - { 計算結果のファイルをマスタへ送信する.

## 3.5 性能評価

### 3.5.1 サーバ支援型通信システムの評価

#### 評価環境

並列 VC システムの実現可能性を示すために、サーバ支援型通信システムを実装し、その性能を評価した。本節では、基本的な通信関数の評価および並列プログラムの実行時間の評価を行った。基本的な通信関数の評価に使用した通信関数は、それぞれ一対一通信と集団通信の代表的な関数の `HTTP_Send/HTTP_Recv` 関数と `HTTP_Gather` 関数である。並列プログラムの実行時間の評価に使用した並列プログラムは、NAS Parallel Benchmarks [53] の Integer Sort (IS) のクラス W と Embarrassing Parallel (EP) のクラス A、姫野ベンチマークである。

これらの評価を行うために、ギガビットイーサネット LAN に接続された 5 台の PC を使用して、並列 VC システムを実装した。PC の仕様を表 3.3 に示す。並列 VC システムのマスタは PC1 上で動作させた。また、ワーカは、実際に数百台の PC を用意することができないため、PC2-PC5 上で複数のプロセスを使用して仮想的にワーカとして動作させた。ワーカのプロセスは、これら 4 台の PC に均等に割り当てられる。システムで使用したソフトウェアの仕様を表 3.4 に示す。Apache では、接続時間を短縮するためにプロセス数を可能な限り大きく設定した。本評価では、512 に設定した。また、MySQL では、実行速度を向上させるためにマスタのメモリ上にテーブルを作成した。各通信関数の  $T_{wait}$  は 0.1 秒に設定した。各通信関数には、通信データを一時的に格納する 1024 バイトのバッファを作成した。上記の条件で、測定を 10 回繰り返し、平均実行時間を測定した。並列プログラムの実行時間の評価では、参考として並列 VC システムと MPI プログラムを使用した PC クラスタで実行時間を比較した。比較の目的は、MPI に対する VC の優位性を示すものではなく、提案する並列 VC システムの動作を確認することである。

表 3.3: PC の仕様

| PC (role)           | OS              | CPU                                                  | Mem.  |
|---------------------|-----------------|------------------------------------------------------|-------|
| PC1<br>(master)     | Ubuntu<br>20.04 | Intel Xeon Silver 4216<br>×2, 2.1GHz<br>(16×2 cores) | 128GB |
| PC2{PC5<br>(worker) | Ubuntu<br>20.04 | Intel Core i9-10900K<br>3.7GHz (10 cores)            | 64GB  |

表 3.4: 使用するソフトウェア

| PC      | Name    | Version         |
|---------|---------|-----------------|
| PC1     | Apache  | 2.4.39          |
|         | Mysql   | 10.1.38-MariaDB |
|         | PHP     | 7.3.4           |
| All PCs | OpenMPI | 4.0.3           |

#### 基本の通信関数の通信性能の評価

図 3.6 は、ワーカー数  $W = 2$  の場合の `HTTP_Send` 関数と `HTTP_Recv` 関数の実行時間を示している。横軸の  $S$  と  $R$  はそれぞれ、`HTTP_Send` 関数と `HTTP_Recv` 関数を示している。各 PHP ファイルの実行時間も内訳に示している。図より、`HTTP_Send` 関数では、データサイズの増加とともに、データ送信時間が増加する。また、データサイズが比較的小さい場合、受信フラグのチェックにかかる時間が大部分を占めている。これは、 $T_{\text{wait}} = 0.1$  秒に設定しているからである。`HTTP_Recv` 関数の実行時間には、送信フラグのチェックとデータの受信にかかる時間が含まれているが、前者の割合が大多数を占める。データサイズが  $8 \times 10^7$  バイトの場合、`HTTP_Send` 関数と `HTTP_Recv` 関数の実行時間はそれぞれ 3 秒程度であった。

図 3.7 にデータサイズを 8 バイト (`double` 型の値一つ) に固定した場合の `HTTP_Send` 関数と `HTTP_Recv` 関数の実行時間を示す。実行時間は、 $W=2$  ペアの通信の平均時間であり、 $W$  ワーカーの半分は送信側で、残り半分は受信側になる。この図から、実行時間は  $W = 128$  あたりから増加し始めることがわかる。しかし、 $W$  が 2 から 1024 に増加しても、`HTTP_Send` 関数と `HTTP_Recv` 関数の実行時間の増加は 5 倍未満である。

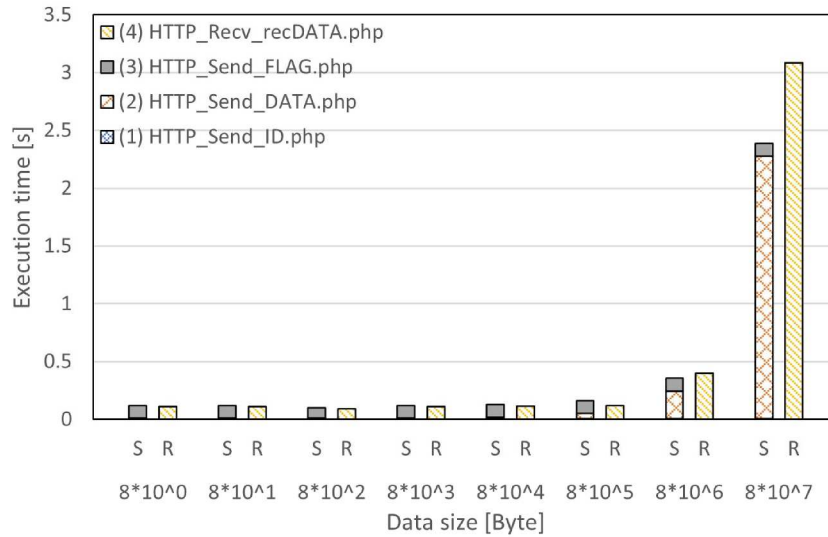


図 3.6: HTTP\_Send/ HTTP\_Recv 関数の実行時間 (W = 2)

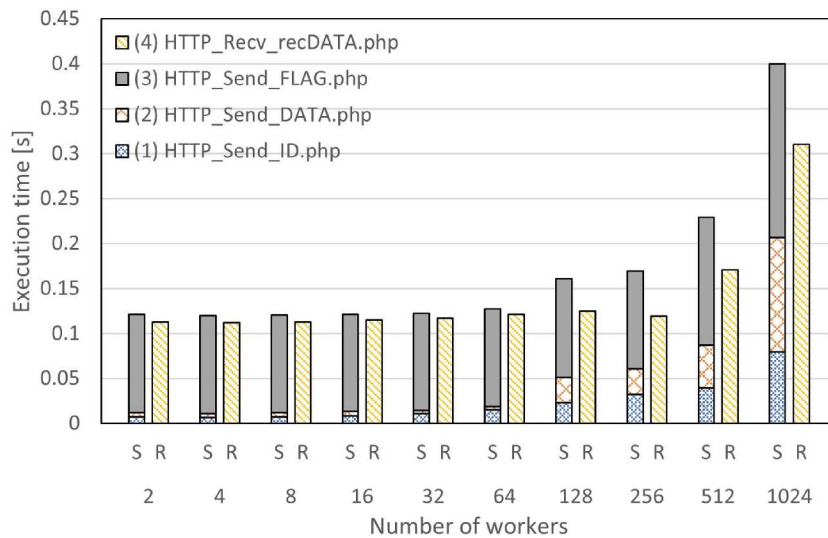


図 3.7: HTTP\_Send/ HTTP\_Recv 関数の実行時間 (data size: 8 Byte)

図 3.8 と図 3.9 は, HTTP\_Gather 関数の実行時間を示している. HTTP\_Send/ HTTP\_Recv 関数と同様に, データサイズと W の増加に伴い, データの送受信にかかる時間は増加している. データサイズが  $8 \times 10^7$  バイトの場合, HTTP\_Gather 関数の実行時間は約 3 秒であり, W = 1024 の場合, 約 0.8 秒であった. 図 3.6{3.9 から, HTTP\_Send/ HTTP\_Recv 関数およ



び HTTP\_Gather 関数の実行時間は、データサイズと  $W$  の増加に伴って、増加することがわかる。しかし、VC 環境では、様々な計算能力が存在し、各ワークで計算時間に差があると考えられることから、同期の待ち時間が大部分を占めると考えられるため、このような実行時間の増加は、許容範囲内である。

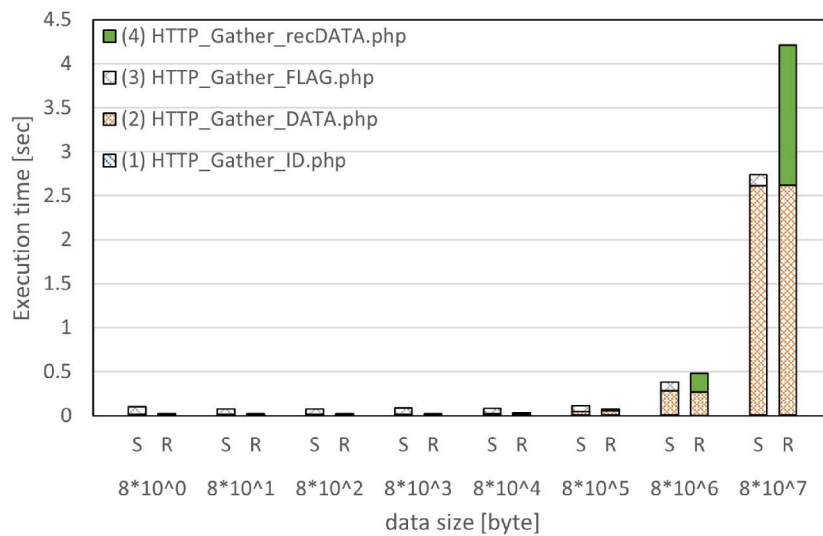


図 3.8: HTTP\_Gather 関数の実行時間 ( $W = 2$ )

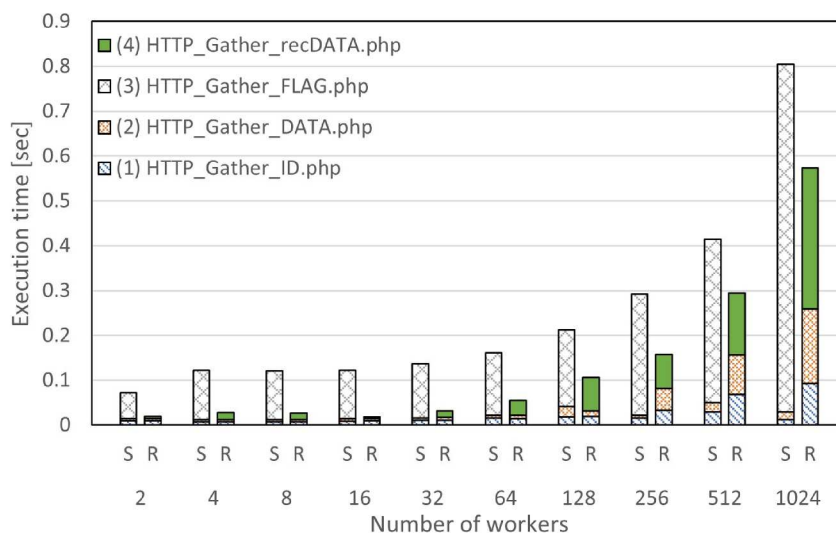


図 3.9: HTTP\_Gather 関数の実行時間 (data size: 8 Byte)

### ベンチマークを用いた並列 VC システムの性能評価

図 3.10 に、並列 VC システム上での IS の実行時間と PC クラスタ上での IS の実行時間を示す。横軸の M と V は、それぞれ、MPI 通信関数を使用した PC クラスタでの実行と並列 VC システムでの実行を示している。IS では、MPI/ HTTP\_Allreduce 関数, MPI/ HTTP\_Alltoall 関数, MPI/ HTTP\_Alltoallv 関数を使用し、整数ソートを並列実行し、同じ処理を 10 回繰り返す。W = 128 までは、VC は MPI よりも時間がかかる。しかし、W = 256 を超えると実行時間が短くなる。これは、全体全通信の実装によるものと考えられる。MPI では、全体全通信の実際の通信方式は、リング、ペアワイズ、一対一など [55, 56] 実装に依存する。そのため、総通信時間は最大で  $O(W^2)$  となる。提案する VC では、各ノードはアルゴリズム 3.3 と同様に 4 つの PHP を実行してマスタの DB と指定されたデータを送受信するため、総通信量は  $O(W)$  である。図 3.10 から、VC, MPI とも、このベンチマークでは、並列計算による効果、つまり、実行時間の短縮は得られないことがわかる。

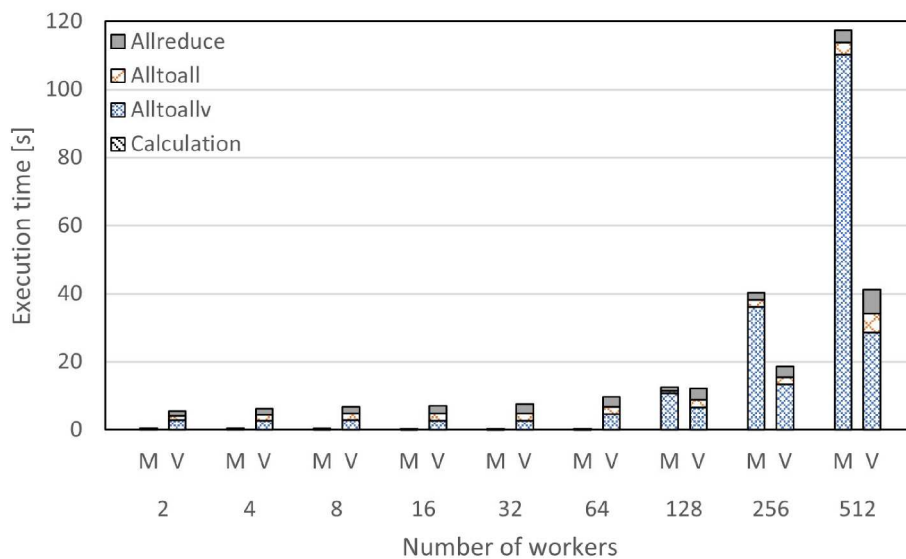


図 3.10: IS (クラス W) の実行時間

図 3.11 に、EP の実行時間を示す。EP では、乱数のペアを作成し、それを座標軸上に投影し、指定領域内のペアを MPI/ HTTP\_Allreduce 関数を 3 回使用し、カウントする。VC では、W = 16 の場合、W = 2 の場合と比較して実行時間が 62% 短縮された。どちらのシステムもワーカー数の増加にともない計算時間は短縮されたが、MPI/ HTTP\_Allreduce 関数による通信時間は増加した。MPI と比較すると、VC では通信時間が長くなった。これは、同期の待ち時

間によるものと考えられる。ワーカは、SEND\_FLAG または RECV\_FLAG が 1 になるまで、 $T_{\text{wait}}$  ごとにマスタにアクセスする必要があるからだと考えられる。

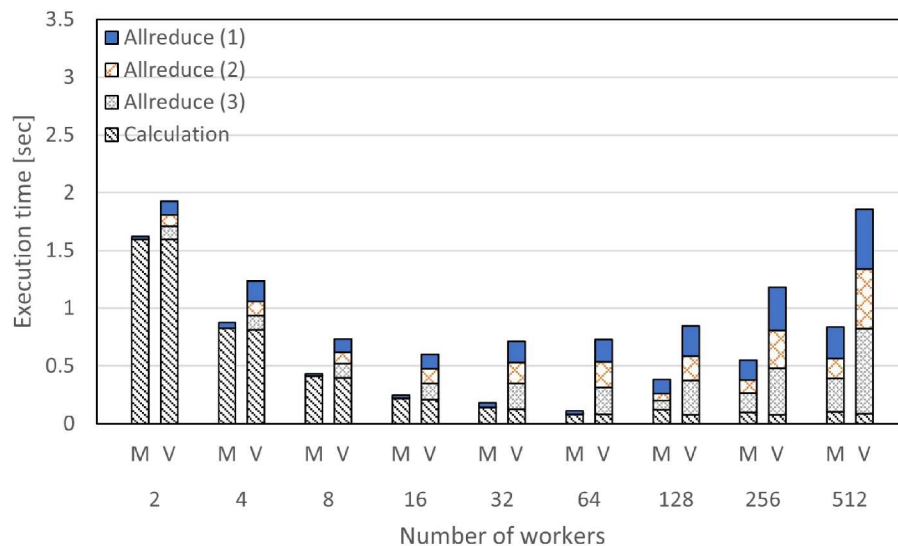


図 3.11: EP (クラス A) の実行時間

図 3.12 に、並列 VC システムでは 4 台の PC、MPI では 1 台の PC で EP を実行した場合の実行時間を示す。図より、VC は並列数  $P = 64$  まで実行時間が減少しているのに対し、MPI は、 $P = 16$  までしか実行時間が減少していない。また、VC での  $P = 64$  の計算時間は、MPI での  $P = 16$  の計算時間よりこれは、CPU のコア数が 10 で、各コアが 2 つのスレッドを同時に処理可能であるため、1 台の PC では 20 並列を同時に処理可能であるのに対し、4 台の PC では 80 並列を同時に処理可能であるからである。これより、VC で並列計算を実行することで並列数を増やし、計算時間を短縮可能であることが示された。

図 3.13 に、姫野ベンチマークの実行時間を示す。姫野ベンチマークは、非圧縮流体解析のベンチマークで、圧力の分布を求めるポアソン方程式をヤコビ法で解く場合の実行時間を測定するものである。この処理は、以下のような幅広い分野で使用する流体解析の基盤的な処理である。

- ・ 製造業: 乗り物（飛行機、船舶、自動車など）の形状設計と室内やエンジンの気流解析
- ・ 建設業: 建物（ビルや原子炉など）の空調・換気設計や安全性評価
- ・ スポーツ: ボールの軌道予測や競技用スーツの開発

このベンチマークでは、通信関数は、MPI.Send 関数と MPI.Recv 関数が使用される。今回の評価では、ヤコビ法を 50 回実行した場合の実行時間を測定した。グリッドは、X 軸、Y

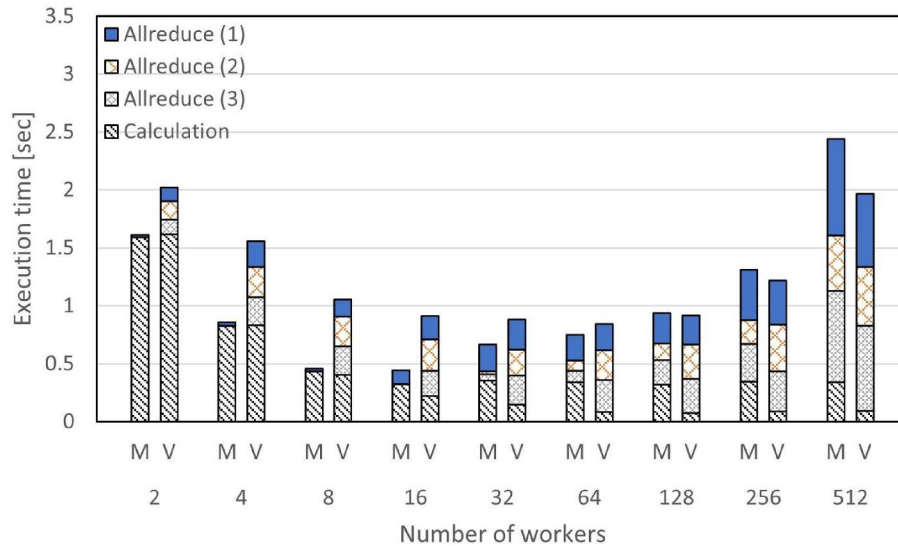


図 3.12: EP (クラス A) の実行時間 (MPI: 1PC, VC: 4PC)

軸, Z 軸をそれぞれ, 1024, 1024, 2048 と設定した. また, メモリアクセスを効率的にするために, X 軸を分割し, 並列化した.

図 3.13 より,  $W = 2, 4$  と増やした場合に, 実行時間が減少していることがわかる. これは,  $W = 1$  の場合, プログラム中の配列がワーカのメインメモリに収容可能な量を越えて swap 領域を使用し, メモリアクセスにかかる時間が増加しているのに対し, ワーカ数を増やすことで, 各ワーカが扱う配列の量が小さくなり, 各ワーカのメインメモリに配列が収容可能になったからである. また,  $W = 8$  の場合に実行時間が増加しているのは, 4 台の PC に, それぞれ, ワーカが 2 つ割り当てられており, それぞれのワーカがメモリにアクセスし, キャッシュが効かなくなっているためだと考えられる.  $W = 8$  の場合に 8 台の PC を使用すれば, 実行時間はさらに減少すると考えられる.

本評価において, 並列 VC の適用効果がある程度示すことができたが, 並列化の効果に関しては, さらなる検証が必要である. 一般的に, スーパーコンピュータや PC クラスタなどの専用機で並列計算を行う場合であっても, プロセス間の通信が, 全体の計算時間を左右する支配的な要因になる. そのため, 実運用においてシミュレーションを行う場合には, 通信の隠蔽や処理の近似による通信データ量の削減など, 対象の処理の特徴を考慮した, 様々な対策が取られるのが一般的である. 本論文は, 並列 VC の実現方法を提案することが主旨であるため, 特定のアプリケーションに対して, 並列計算の高速化に関する工夫までは行っていない. 提案手法の並列化の効果に関しては, VC の応用分野における研究者との共同研究などを通じてプログラムを入手し, 処理対象の特徴を考慮した高速化手法を検討することが, 今後の課題の一つと

なる。

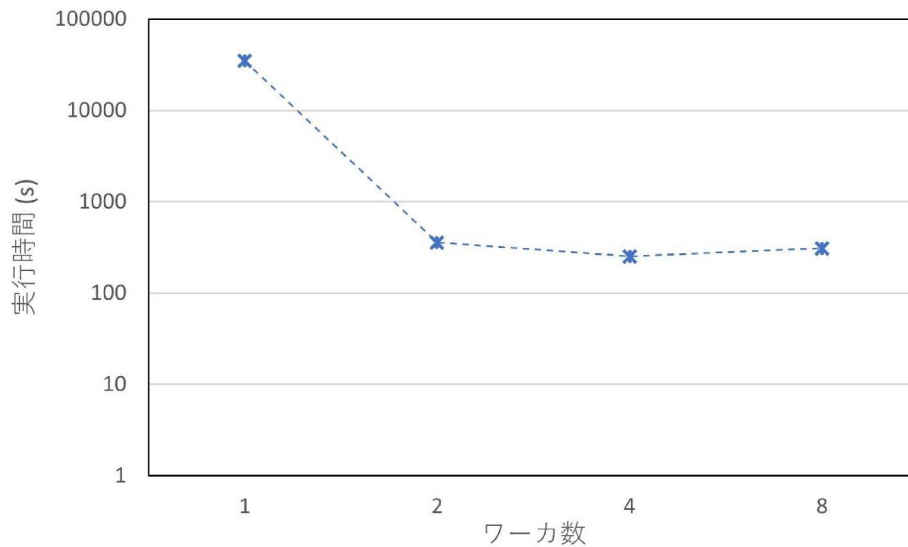


図 3.13: 姫野ベンチマークの実行時間

### 3.5.2 ジョブ配布システムの評価

#### 評価環境

本節では、並列プログラムを使用して、ジョブ配布システムの性能評価を行う。また、ワーカの離脱が発生した場合のシステムの実行時間の評価を行う。使用する並列プログラムは、Nas Parallel Benchmarks の EP のクラス W である。ジョブ配布システムの性能評価では、マスタで実行される PHP の実行時間とワーカで実行される簡易的なクライアントの実行時間についてそれぞれ評価を行う。評価条件を表 3.5 に示す。

ワーカの離脱が発生した場合のシステムの実行時間の評価では、並列数  $P = 4$ 、離脱判定時間  $T_{\text{limit}}$  を 5 秒に設定した。この評価は以下の 2 つのシナリオで実施した：

1. 離脱なし
  - 4 台のワーカが正常に実行を完了する。
2. 離脱あり
  - 初期状態で 4 台のワーカがジョブ配布リクエストを実行する。
  - 1 台のワーカが結果を返却せずに離脱する。
  - 初回リクエストから 6 秒後 ( $T_{\text{limit}} = 5$  秒に処理時間を考慮して +1 秒) に新たな 4

表 3.5: 評価条件

|           |                          |
|-----------|--------------------------|
| 配布ジョブ     | EP ベンチマーク（並列 VC 用），クラス W |
| プログラムの並列数 | 2～512                    |
| ワーカ数      | 2～512<br>(並列数に応じる)       |

台のワーカが配布リクエストを実行する。

並列 VC システムの構築に使用した PC やソフトウェアは、3.5.1 節と同じである。

#### ジョブ配布システムの性能評価

マスタで実行される PHP の実行時間を図 3.14 に示す。S は `send.php` の実行時間、R は `receive.php` の実行時間を表している。処理の内容ごとに時間を計測しており、凡例はそれぞれ 3.4.1 節で述べた内容に対応している。送信処理（`send.php`）の実行時間は、ワーカ数によらず一定のばらつきが見られる。これは主にジョブ配布時の DB 操作において、テーブルへの占有ロックが必要な処理が含まれており、複数のアクセスが競合した際に待機時間が発生するためである。一方、ファイル送信に要する時間は比較的小さく、ワーカ数による変動も少ない。これは、ソケット通信を用いた実装により、ファイルの出力処理のみを計測しているためである。

受信処理（`receive.php`）の実行時間は、送信処理と比較して長く、またワーカ数の増加に伴って延長する傾向が見られる。これは以下の 2 つの要因による。

##### 1. DB 状態更新

- `send.php` と比較して占有ロックを必要とする処理が多く、ワーカ数の増加にともないアクセス競合が発生しやすくなる。

##### 2. 受信ファイル保存

- ワーカ数の増加により 1 台あたりの処理プロセス数が増え、各ノードの処理負荷が上昇する。

簡易的なクライアントの実行時間を図 3.15 に示す。処理の内容ごとに時間を計測しており、凡例は 3.4.2 節で述べた内容に対応している。実行時間はワーカ数の増加に伴って増加する傾向が見られ、特にリクエスト送信からジョブファイル受信までの時間と、ジョブの実行時間が大きな割合を占めている。リクエスト送信からジョブファイル受信までの時間がワーカ数に応



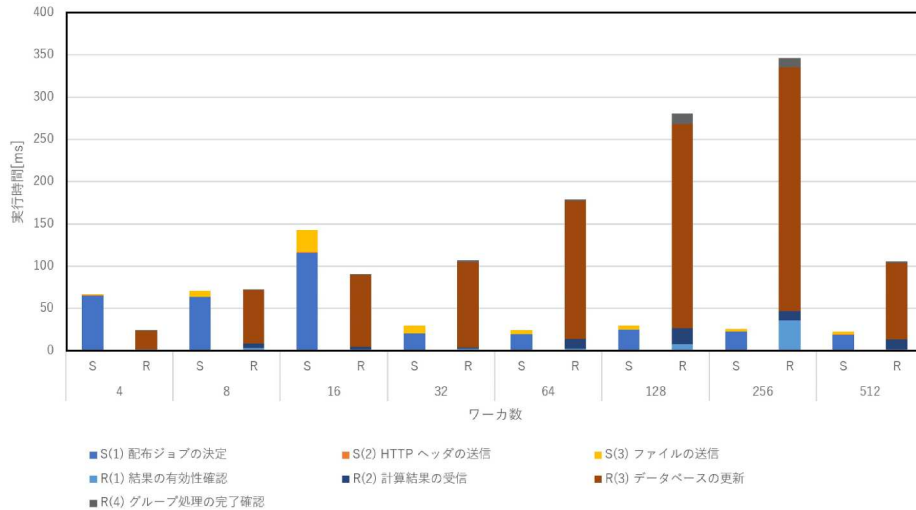


図 3.14: send.php と recv.php の実行時間

じて増加する原因として、Apache サーバへの同時接続数の増加が考えられる。特に、MPI による並列実行により各 PC 内の複数ワーカが同時にマスタにリクエストを送信するため、局所的な通信の集中が発生する。マスタで実行される send.php 自体の処理時間はワーカ数に依存しないことから、この遅延は主にサーバ負荷と通信の輻輳によるものと推測される。

また、ジョブの実行時間の増加は、使用したベンチマークプログラムの特性によるものであると考えられる。本実験では並列数がワーカ数に応じて増加するプログラムを使用しているため、ワーカ数の増加にともない、通信の輻輳やワーカ間の同期待ち時間が増加することで、全体の実行時間が増加したと考えられる。

図 3.15 では、ジョブの実行時間が 1~50 秒程度であるのに対し、図 3.14 では、マスタの PHP 実行時間は最大でも 0.35 秒程度であった。これは、ジョブ配布と計算結果の受信に要する時間が実際の計算時間と比較して十分に小さいことを示しており、本研究で実装したジョブ配布機能がシステム全体の性能に与える影響は軽微であると言える。

#### ワーカの離脱が発生した場合のシステムの実行時間の評価

図 3.16 に、ワーカの離脱が発生した場合の EP の実行時間を示す。離脱ありの場合の実行時間は、離脱なしの場合と比較して約 6 秒増加している。この時間は、離脱が発生した後、再配布が行われるまでの時間に等しい。このため、離脱検知と再配布が想定通りに機能している

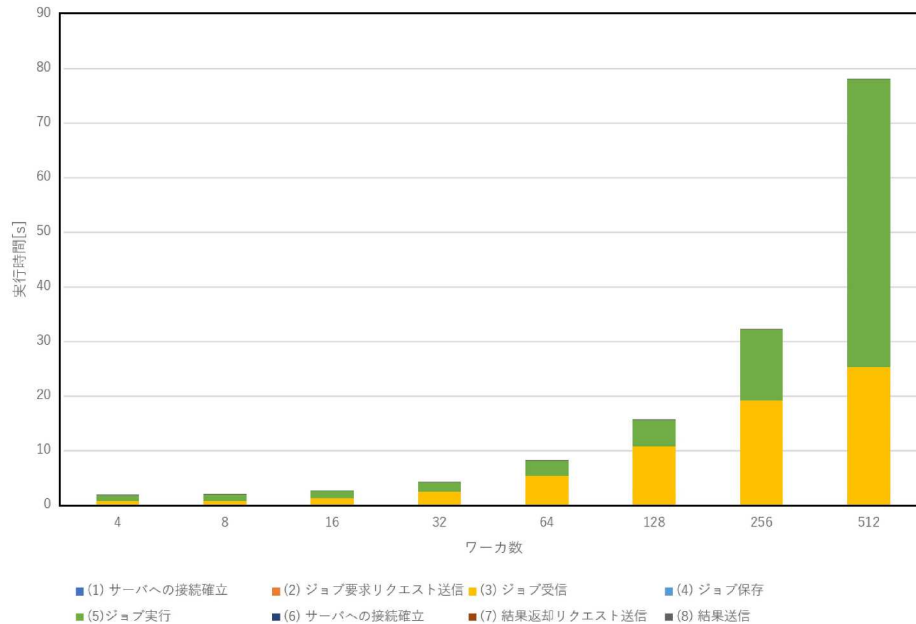


図 3.15: 簡易的なクライアントの実行時間

ことが確認できた。

### 3.6 むすび

本章では、サーバ支援型通信に基づく並列 VC を提案し、その実行に欠かせない並列 VC サーバの構成を提案し、実装した。本研究では、並列 VC サーバは、Web アプリケーションで一般的に使用される Apache Web サーバと Mysql DB サーバをベースに実装した。サーバ支援型通信システムとジョブ配布システムから構成される、各システムの動作確認と性能評価を行った。

サーバ支援型通信において、HTTP\_Send/ HTTP\_Recv 関数および HTTP\_Gather 関数の実行時間は、データサイズと  $W$  の増加に伴って、増加することがわかった。しかし、ワーカーごとに性能差がある VC 環境環境において許容範囲内であることがわかった。IS および EP の実行時間は、MPI と比較すると、ほとんどの場合で遅くなっていた。しかし、本章で提案したサーバ支援型通信システムを使用することで、VC 上で並列計算が可能であることを実証した。また、姫野ベンチマークにおいて、ワーカー数が 1 の場合に比べて、ワーカー数を 4 に増やした場



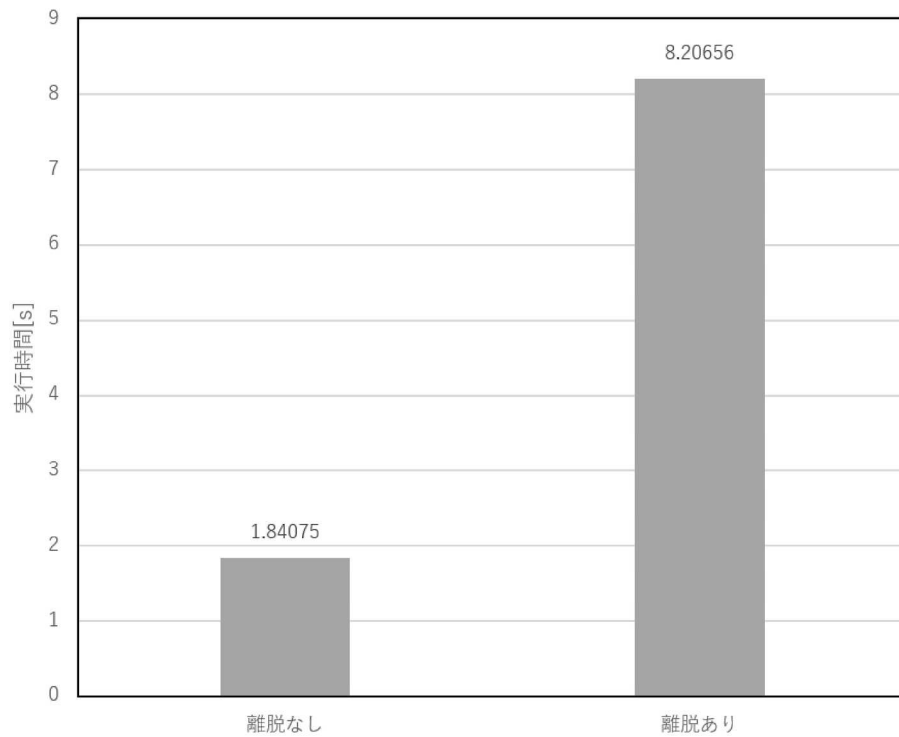


図 3.16: ワーカの離脱が発生した場合の EP (クラス W) の実行時間

合に実行時間が減少していることから、並列 VC の適用効果をある程度示すことができた。

ジョブ配布システムにおいて、実行時間の評価では、マスタプログラムにおける DB アクセスの競合がジョブ配布と結果受信の処理時間に影響を与えること、また、ワーカ数の増加にともないサーバ負荷と通信輻輳による遅延が発生することを明らかにした。ただし、マスタの PHP 実行時間はジョブの実行時間と比較して十分に小さく、ジョブ配布機能が実用に耐えうるものであることを確認した。さらに、ワーカ離脱時の再配布機能が期待通りに動作することを確認した。

## 第 4 章

# WebAssembly を使用したブラウザベースの並列 VC クライアント

### 4.1 まえがき

本章では、2.3.2 節で定義した要件 2 を満たす、PC 標準の環境で一般ユーザが参加可能な並列 VC の実現を目的として、Web ブラウザをプラットフォームとしたブラウザベースの並列 VC システムを提案する。並列プログラムを Web ブラウザ上で実行可能にするために、近年、多くの Web ブラウザで標準対応されるようになった WebAssembly を用いる。具体的には、並列計算における計算部分を WebAssembly、通信部分をブラウザ API の Fetch API を用いて実装し、WebAssembly と Fetch API 間のデータ共有を C/ C++ と Javascript の仲介役である EMLJS を用いて実装する。OS に初期から組み込まれている Web ブラウザを VC 参加者用のプラットフォームとすることで、VC への参加が容易になり、参加人数の増加による並列 VC の計算能力の向上が期待できる。実装したブラウザベースの並列 VC システムにおいて、NAS Parallel Benchmarks の Integer Sort (IS) を用いた性能評価を行い、動作確認および有用性を検証する。

4.2 節では、提案する並列 VC クライアントの概要と詳細な実装を説明する。4.3 節では、提案する並列 VC クライアントの性能評価を行う。

### 4.2 並列 VC クライアント

#### 4.2.1 並列 VC クライアントの概要

本章では、並列 VC システムを PC 標準の環境で一般ユーザが参加可能なものにするため、Web ブラウザをプラットフォームとした並列 VC クライアントを提案する。一般的な VC の

プラットフォームである BOINC のように、一般ユーザの参加環境として、専用のソフトウェアを開発するという手段もあるが、VC への参加のために一般ユーザに新たなソフトウェアをインストールさせる負担を強いることは、VC 参加のハードルを上げることになる。それに対し、OS に初期から組み込まれているソフトウェアである Web ブラウザを用いるという手段は、一般ユーザに負担をかけずに VC に参加させることができる。一般ユーザが VC に参加しやすくなることで、参加人数が増え、VC の計算能力の向上が期待できる。

3 章で提案した並列 VC サーバでは、C 言語で記述されたプログラムをコンパイルして得られる実行ファイルをジョブとして、ワークに配布していたが、Web ブラウザを VC 参加者のプラットフォームにするにあたり、ジョブをブラウザで処理可能な形にする必要がある。そこで、WebAssembly [1] を用いる。WebAssembly とは、Web ブラウザで実行可能なバイナリ形式の低レベルなアセンブリ風の言語である。WebAssembly は、コードの記述方法を知らずとも、C/C++ 等の言語のコードからツールを使って作成することができ、そのファイルを Web ブラウザ上で実行することができる。この技術を用いて、Web ブラウザを VC 参加者のプラットフォームとした、PC 標準の環境で一般ユーザが参加可能な並列 VC システムを実現可能にする。

#### 4.2.2 並列 VC クライアントの実装

提案手法では、Web ブラウザで並列計算を実行するために、WebAssembly (WASM) を使用する。WebAssembly とは、Web ブラウザで実行可能なバイナリ形式の低レベルなアセンブリ風の言語であり、Web ブラウザでの計算処理に一般的に使用される JavaScript より高速に計算処理を実行することが可能である。また、C/C++、C# や Rust などの言語で書かれたプログラムを Web ブラウザ上で実行することが可能である。WebAssembly は、多数の人が使用するほとんどのブラウザで実行可能である。現在 WebAssembly に対応している Web ブラウザを以下に示す。

- Google Chrome
- Mozilla Firefox
- Apple Safari
- Microsoft Edge
- Opera

C/C++ のコードを WebAssembly にコンパイルするには、Emscripten [57] を使用する。Emscripten とは、WebAssembly 用の完全なオープンソースコンパイラツールチェーンである。Emscripten を用いることで、C/C++ のコード、またはコンパイラの基盤である LLVM を使用するその他の言語を WebAssembly にコンパイルし、Web ブラウザ、Node.js、または

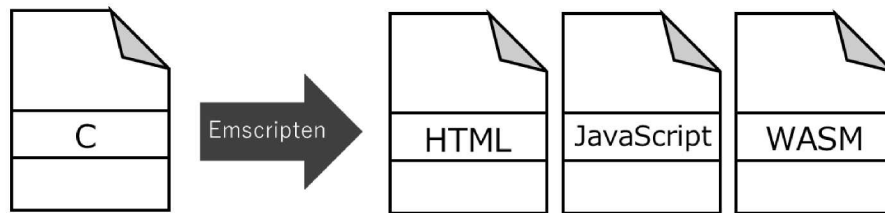


図 4.1: Emscripten による C 言語のコンパイル

その他の Wasm ランタイムで実行することができる。C 言語で記述されたプログラムを Web ブラウザ上で実行するまでの一連の流れを以下と図 4.1, 図 4.2 に示す。

1. C 言語で記述されたファイルを Emscripten を用いてコンパイルする。
2. コンパイルにより生成された HTML, JavaScript, WASM ファイルを Web サーバの適当な位置に配置する。
3. Web ブラウザを用いて, HTML ファイルの URL を入力し, Web サーバへ HTTP リクエストを送ることで, レスポンスとして, HTML ファイルを受信し, 実行する。
4. HTML が実行されることにより, 新たに HTTP リクエストが送信され, 対応する JavaScript ファイルを受信し, 実行する。
5. JavaScript ファイルが実行されることにより, 新たに HTTP リクエストが送信され, 対応する WASM ファイルを受信し, 実行する。

3.3 節で説明した通信関数をそのまま用いて, C 言語で記述された並列計算のジョブを WebAssembly にコンパイルし, 実行することはできない。なぜなら, 通信関数には, 標準的なソケットライブラリを使用した通信が使われているからである。ソケットライブラリを使用した通信では, OS の中心部分しか使用できない特権的な機能を用いてハードウェアを操作し, VC サーバと通信を行っている。しかし, WebAssembly では, セキュリティの観点から OS の特権的な機能を用いたプログラムを実行することができない。この問題を解決するために, ソケットライブラリを使用した通信に代わる通信手段をとる必要がある。そこで, ブラウザに

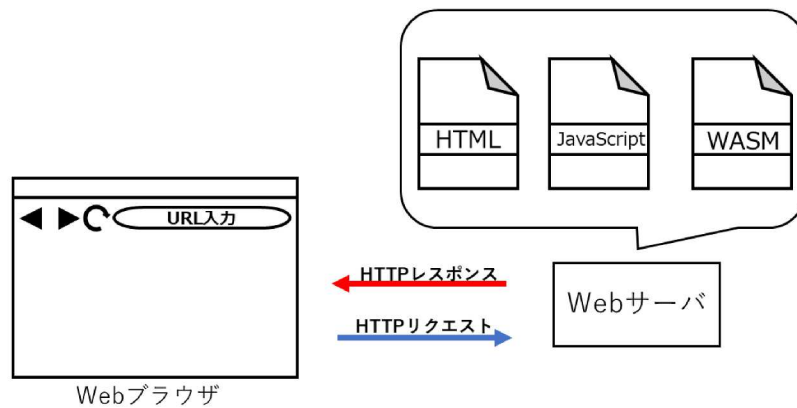


図 4.2: ブラウザ上での Wasm の実行

組み込まれている機能であるブラウザ API を用いる。

ブラウザ API [58] とは、Web ブラウザが提供する様々な機能やサービスを、より簡単に実装するために Web ブラウザに組み込まれた仕組みである。ブラウザ API を用いることで、サーバからデータを取得したり、ブラウザに読み込まれた文書进行操作したりすることができる。例えば、ウェブオーディオ API は、ブラウザ内で音声トラックの取得、音量の変更、エフェクトの適用などの操作を行うための JavaScript の仕組みを提供している。実際には、裏でブラウザは低レベルの複雑なコードを使って実際の音声処理を行っている。しかし、この複雑さを API が抽象化し、隠蔽している。

今回、サーバとの通信を行うプログラムを記述するために、ブラウザ API の一種である FetchAPI [59] を使用する。FetchAPI は、HTTP リクエストを行い、レスポンスを処理するための JavaScript インターフェースを提供している。FetchAPI を使用したコードの例をソースコード 4.1 に示す。ソースコード 4.1 は、FetchAPI を使用して、サーバから JSON データを取得し、取得した JSON データを表示するプログラムである。2 行目では、HTTP リクエストの送信先である URL を格納した文字列を宣言している。3-5 行目では、HTTP リクエストにおける Method, Header を設定したオブジェクトを宣言している。7 行目では、2 行目、3-5 行目で宣言した変数を引数として fetch 関数を実行することで、HTTP リクエストを送信しており、fetch 関数の戻り値を変数 response に代入している。8-10 行目では、レスポンスステータスが成功レスポンスである 200 番台であるかどうかを判定し、そうでなければエラーを表示している。12 行目では、レスポンスとして受信した JSON データを、宣言した変数に代

ソースコード 4.1: FetchAPI を用いたプログラム例

---

```

1 async function getData() {
2   const url = "https://example.org/products.json";
3   const options = {method: "GET",
4                     headers: {'Content-Type': 'application/x-www-form
                               urlencoded'}}
5
6   try {
7     const response = await fetch(url, options);
8     if (!response.ok) {
9       throw new Error(`response status: ${response.status}`);
10    }
11
12    const json = await response.json();
13    console.log(json);
14  } catch (error) {
15    console.error(error.message);
16  }
17 }

```

---

入している。13行目では、JSON データを表示している。

この FetchAPI を用いて、3.3 節で実装した通信関数と同様の機能を持つ通信関数を JavaScript で実装する。本提案の通信関数では、FetchAPI を使用し、マスタにある 4 つの PHP を実行する。この通信関数をワーカが実行することで、Web ブラウザでサーバ支援型通信が可能になる。本提案の通信関数を WebAssembly で実行するために、WebAssembly のコンパイラである Emscripten が提供しているマクロである EMLJS を用いる。

EMLJS [60] とは、Emscripten が提供するマクロで、C/ C++ のコードの中に直接 JavaScript のコードを埋め込むための仕組みである。EMLJS を使ったコードの例をソースコード 4.2 に示す。ソースコード 4.2 の例では、3 行目で EMLJS を用いて定義した関数を、main 関数内の 10 行目で呼び出している。EMLJS の第一引数が作成したい関数の戻り値、第二引数が関数名、第三引数が作成したい関数の引数、第四引数が関数の中身となっている。この例では、戻り値は void 型になっているが、char, int, double, ポインタ型などの様々な戻り値を設定することができる。また、引数も同様に、char, int, double, ポインタ型などを取ることができる。関数

## ソースコード 4.2: EMLJS を用いたプログラム例

---

```

1 #include <emscripten.h>
2
3 EMLJS(void, JS_function, (), {
4     // のプログラムを記述可能 JavaScript
5     console.log('hello world');
6 });
7
8 int main() {
9     // で記述された関数を呼び出し可能 JavaScript
10    JS_function();
11    return 0;
12 }

```

---

の中身は、JavaScript のプログラムを記述することができる。EMLJS を利用することで、非常に柔軟性に富んだプログラムを記述することができる。

これらの技術を利用し、Web ブラウザ上で実行可能な並列計算の実装を図 4.3 に示す。並列計算のソケットライブラリを用いた通信部分を Fetch API を用いた通信に変換し、EMLJS マクロを用いて C/C++ に埋め込んだプログラムを作成する。その後、このプログラムを Emscriptren を使用して、WebAssembly にコンパイルすることで、ブラウザ上で実行可能な並列計算を実装することができる。

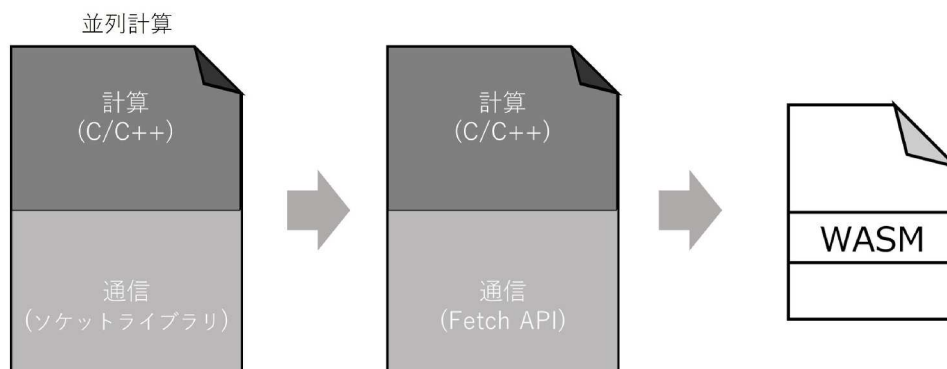


図 4.3: Web ブラウザ上で実行可能な並列計算の実装

表 4.1: PC の性能

| PC           | OS           | CPU                                | Memory |
|--------------|--------------|------------------------------------|--------|
| PC1 (master) | Ubuntu 20.04 | Intel Xeon Silver 4216*2 (16*2 コア) | 128GB  |
| PC2 (worker) | Ubuntu 20.04 | Intel Core i9 10900k (10 コア)       | 64GB   |
| PC3 (worker) | Ubuntu 20.04 | Intel Core i9 10900k (10 コア)       | 64GB   |
| PC4 (worker) | Ubuntu 20.04 | Intel Core i9 10900k (10 コア)       | 64GB   |
| PC5 (worker) | Ubuntu 20.04 | Intel Core i9 10900k (10 コア)       | 64GB   |

### 4.3 ベンチマークを用いたブラウザベースの並列 VC クライアントの性能評価

#### 4.3.1 評価環境

本章では, WebAssembly を用いたブラウザベースの並列 VC クライアント上で, 並列計算のベンチマークを実行した際の動作確認と実行時間について評価を行う. 動作確認では, 様々な OS でも動作可能か確認した. 実行時間の評価では, 参考性能として, 3 章で作成した簡易的なクライアント上で並列計算のベンチマークを行った際の実行時間と比較を行う. 本評価のベンチマークプログラムには, NAS Pararell Benchmarks の IS (クラス W) を用いる.

性能評価を行うにあたり, PC クラスタ上に並列 VC システムを作成した. PC クラスタは, 1 台の親ノードと 4 台の子ノードから成る. それぞれの PC のスペックを表 4.1 に示す. マスタとして 3 章の並列 VC サーバを PC1 に実装し, ワーカーとして本提案のブラウザベースの並列 VC クライアントを PC2{5 上で起動させる. 並列 VC サーバおよび本提案の並列 VC クライアントに用いたソフトウェアのスペックを表 4.2 に示す. 実行には, PC2{5 上に均等にブラウザを複数起動させ, 1 つのブラウザタブを 1 ワーカーとして扱った. 動作確認では, このクラスタに Windows11 がインストールされた PC と IOS 26.2 がインストールされたスマートフォンを接続し, 並列プログラムの IS を実行した. ブラウザには表 4.2 の Firefox を使用した.

#### 4.3.2 並列 VC クライアントの評価

まず, 本章で提案するブラウザベースの並列 VC クライアントの動作確認を行った. Linux がインストールされた PC2, PC3 と Windows がインストールされた PC, IOS がインストールされたスマートフォンの 4 台にそれぞれワーカーを 1 つずつ起動し, IS (クラス W) を並列数  $P = 4$  で実行した. その結果, ワーカーの OS が異なっているにもかかわらず, ブラウザベースのクライアン



表 4.2: 使用するソフトウェア

| PC      | Name            | Version  |
|---------|-----------------|----------|
| PC1     | Apache          | 2.4.46   |
|         | Mysql           | 8.0.40-0 |
|         | PHP             | 7.4.3    |
| All PCs | Mozilla Firefox | 134.0    |

トで並列計算を正しく実行可能であることを確認した。また、実行時間は、約 8.4 秒であった。

次に、提案するブラウザベースの並列 VC クライアントと、3.4.2 節で作成した C 言語の簡易的なクライアント上で実行した IS ベンチマークの平均実行時間を図 4.4、図 4.5 に示す。横軸の C と B は、それぞれ、簡易的なクライアント（C 言語で作成）とブラウザベースのクライアントに対応する。IS ベンチマークは、各ワークが HTTP\_Allreduce 関数, HTTP\_Alltoall 関数, HTTP\_Alltoallv 関数を用いた一連の通信を 10 回反復するベンチマークとなっており、図 4.4、図 4.5 に示す通信時間は 3 つの通信関数をそれぞれ 10 回行った合計の通信時間を表している。図 4.4 と図 4.5 より、IS ベンチマークは全体の実行時間のなかで通信の実行時間がほとんどを占めていることがわかる。また、ワーク数の増加にともない、実行時間も増加することがわかる。簡易的なクライアントとブラウザベースのクライアントを比較すると、いずれのクラス、ワーク数においても、簡易的なクライアントのほうが実行時間が短いことがわかる。これは、ブラウザベースのクライアントは、並列計算を実行する際、WASM と JavaScript で、それぞれ別のメモリ領域を確保しており、互いに必要なときに参照し合いながら処理を行っているため、メモリアクセスがオーバーヘッドになるからであると考えられる。ブラウザベースのクライアントの実行時間は、C 言語で作成された簡易的なクライアントと比較して、クラス S で最大約 3 秒、クラス W で最大約 5 秒ほどしか実行時間が増加しなかった。この差は、1 ジョブの実行時間が数十時間程度であることを考慮すると、無視できる程度の差であることが明らかとなった。

## 4.4 むすび

本章では、WebAssembly を用いたブラウザベースの並列 VC クライアントを提案した。また、並列 VC クライアント上で並列計算のベンチマークを実行した際の動作確認を行い、実行時間について評価を行った。その結果、様々な OS でブラウザベースの並列 VC クライアントを使用した並列計算が実行可能であることを確認した。また、実行時間は、C 言語で作成された簡易的なクライアントと比較して、クラス S で最大約 3 秒、クラス W で最大約 5 秒ほどし

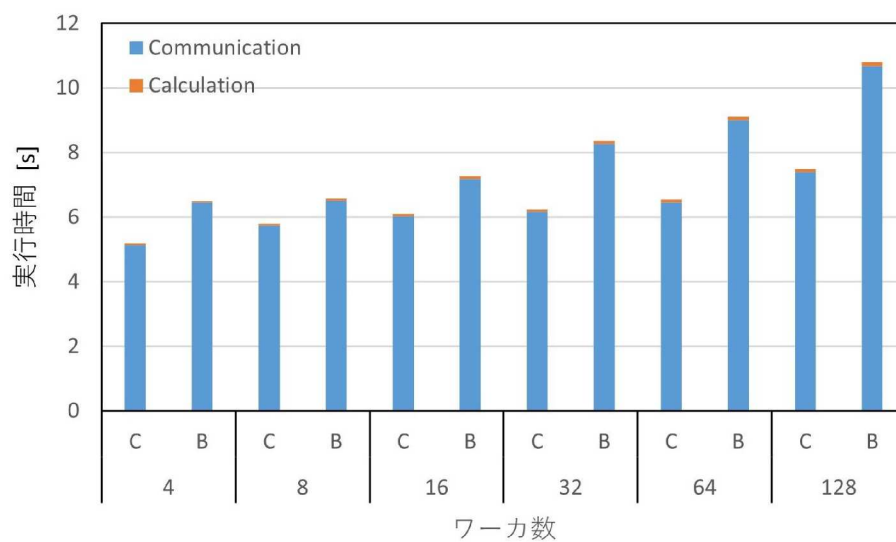


図 4.4: IS (クラス S) の実行時間

か実行時間が増加しないことがわかった。この差は、1 ジョブの実行時間が数十時間程度であることを考慮すると、無視できる程度の差であることが明らかとなった。

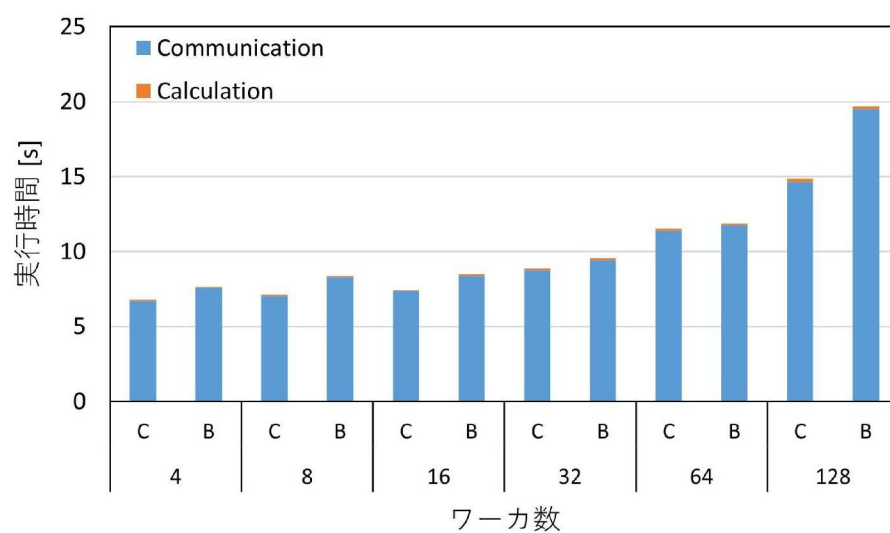


図 4.5: IS (クラス W) の実行時間

## 第 5 章

# ソースコードレベルで実現する チェックポイント機能

### 5.1 まえがき

VC のワーカは、一般ユーザが所有する PC などであるため、ワーカ同士が直接通信を行うようには構成されておらず、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。

3.4 節では、離脱が発生した場合、別のワーカにサブジョブを再配布を行うことにより、離脱に対処した。この場合、サブジョブを最初から再実行するため、離脱するまでに処理していた分が無駄になり、計算時間が増加する。さらに、あるワーカでジョブを再実行している最中に別のワーカが離脱し、再度、再配布を行うことになる可能性がある。このような悪循環を避け、3.5.1 節で定義した要件 3 を満たすために、サブジョブを途中から再開するためのチェックポイント機能が必要になる。

本章では、ワーカの離脱に対処するために、ソースコードレベルのチェックポイント手法を提案する。提案手法では、ソースコード自体にチェックポイント機能を埋め込み、テキストファイルによってプログラムの実行状態を保存する。そのため、チェックポイント保存時と異なる PC 環境で、チェックポイントを復元することが可能である。

5.2 節では、チェックポイント機能の概要と詳細な実装を述べる。5.3 節では、チェックポイント機能の動作確認と性能評価を行う。

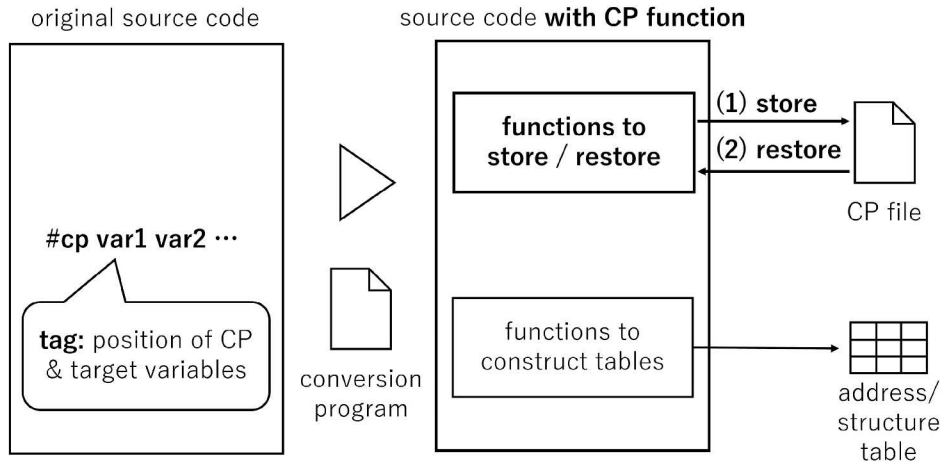


図 5.1: 提案手法の実装の概要

## 5.2 チェックポイント機能

### 5.2.1 チェックポイント機能の概要

チェックポイントとは、プログラムの実行状態を定期的に保存し、障害が起きたときに、最後に保存した状態からプログラムを再開する手法である。提案手法の実装の概要図を図 5.1 に示す。本研究では、プログラマがチェックポイントを行いたい位置に以下のタグを書くことで、それを指定するようにする。また、タグと一緒に、チェックポイントとして保存したい変数を指定するようにする。

# checkpoint 変数名 1 変数名 2 . . .

このタグを、変換プログラムを用いて、以下のように自動で置き換える。

- 指定された変数の値を保存・復元する関数
- 保存・復元関数を補助するテーブルを作成する関数
- タグが書かれた位置からプログラムを再開するための構造

これにより、チェックポイントのためのアプリケーションをインストールするユーザの手間をなくすることができる。この後、ソースコードをコンパイルすることで、チェックポイント機能付のプログラムを得ることができる。

## 5.2.2 チェックポイント関数の実装

本研究では、タグと共に指定された変数の値を保存・復元するために、`cp_write` と `cp_read` の 2 つの関数を作成した。 `cp_write` 関数は指定された変数の値をチェックポイントファイルに保存し、 `cp_read` 関数はファイルに保存された変数の値を復元する。次に詳細を示す。

### `cp_write` 関数

`cp_write` 関数は指定された変数の値をチェックポイントファイルに保存する関数である。`cp_write` 関数の形式と引数の詳細を以下に示す。

`cp_write(type, pointer, name, size, address)`

- `type`: 保存する変数の型
- `pointer`: 保存する変数の値を取得するまでのアクセス数（ポインタの場合、変数についているアスタリスク数）
- `name`: 保存する変数名
- `size`: 保存する変数の要素数
- `address`: 保存する変数の値が格納されているメモリのアドレス

例えば、保存対象の変数が、`\int b[3]` と定義されていれば、`cp_write` 関数の引数は以下のようになる。

- `type = \int`
- `pointer = 0`
- `name = b`
- `size = 3`
- `address = b`

また、保存対象の変数が、`\int *c` と定義されていれば、`cp_write` 関数の引数は以下のようになる。

- `type = \int`
- `pointer = 1`
- `name = c`
- `size = 1`
- `address = &c`

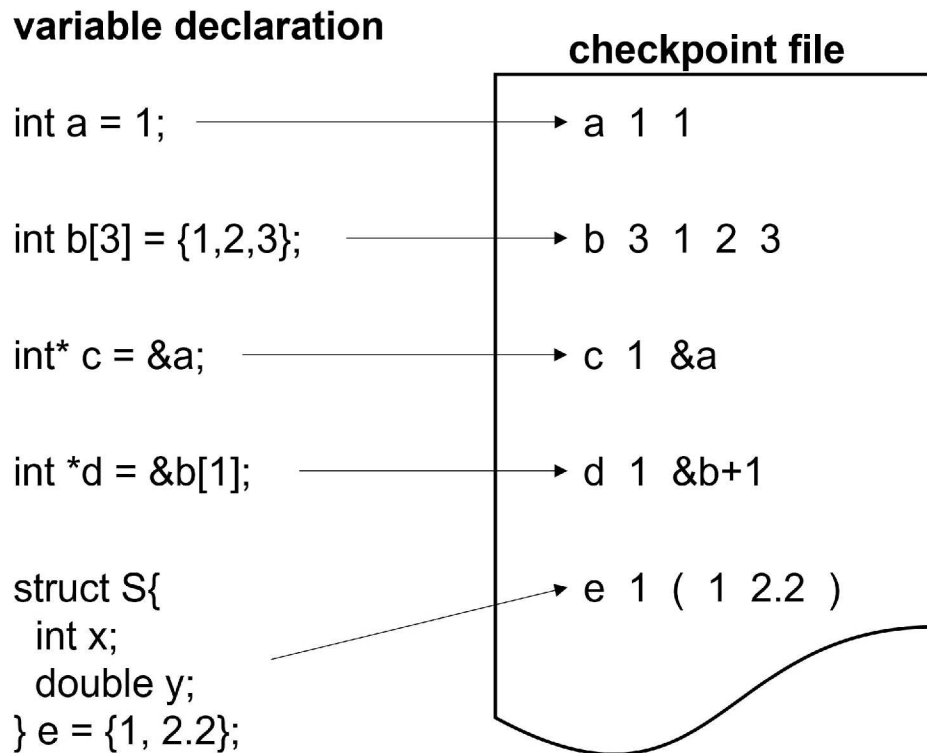


図 5.2: チェックポイントファイルの形式

図 5.2 に、`cp_write` 関数によって保存されるチェックポイントファイルの形式を示す。ファイルに書かれているのは、左から順に、変数名、変数のサイズ、変数の値となっている。それぞれは空白文字で区切られ、改行によって変数同士が区別される。

保存対象の変数がポインタの場合、アドレスはコンピュータの環境によって変化するため、ポインタの値をそのままチェックポイントで使うことはできない。そのため、ソースコード中の全変数に対して、変数名、アドレス、要素数、型のサイズなどの情報を記録するアドレステーブルを作成する。アドレステーブルの例を図 5.3 に示す。アドレステーブルに変数の情報を追加する関数として、`cp_addr` 関数を作成した。この関数は、変換プログラムによって、全ての変数の定義や `malloc` 関数が実行された次の行に自動的に挿入される。アドレステーブルで、アドレスをキーとして検索することにより、ポインタが指し示す先の変数名を取得し、図 5.2 の変数 `c` の例のように、`\&変数名` の形式で、チェックポイントファイルに保存する。

ポインタの指し示す先が配列の場合、配列のどの要素を指しているか示すため、図 5.2 の変数 `d` の例のように、ポインタが指し示す変数名の後に `\+ 配列のインデックス` の形式で、配列のインデックスも保存する。また、ポインタが指す先が `malloc` 関数などで動的に確保された場

```

int a = 1;
cp_addr("a", 1, &a, sizeof(int));
int b[3] = {1, 2, 3};
cp_addr("b", 3, b, sizeof(int));

```

**Address table**

| Name | # of elements | Address    | Size of type |
|------|---------------|------------|--------------|
| a    | 1             | 0x11111111 | 4            |
| b    | 3             | 0x22222222 | 4            |

図 5.3: アドレステーブルの例

合、変数名がなく、ファイルの形式通りに保存することができない。この場合、`MALLOC_X` (`X` はこれまで `malloc` 関数が実行された回数) と仮の変数名を付けることで、通常の変数と同じように保存する。

保存対象の変数が構造体の場合、構造体はプログラマが自由に定義でき、通常は複数の型の変数で構成されるため、他の変数と同じ方法で、保存・復元できない。そのため、構造体の構造を記録する構造体テーブルを作成する。構造体テーブルの例を図 5.4 に示す。構造体テーブルには、ソースコード内の全ての構造体について、タグ、サイズ、メンバの数、メンバのデータを記録しておく。メンバのデータには、全てのメンバ変数について、型、値を取得するまでのアクセス数、メンバ名、要素数、オフセット、型の大きさの情報を記録する。構造体テーブルに構造体の情報を追加する関数として、`st_add_structure` 関数と `st_add_member` 関数を作成した。これらの関数は、変換プログラムによって、`main` 関数の先頭に自動的に挿入される。構造体テーブルで、`type` として指定されるタグをキーとして検索することにより、メンバーの型を取得し、`address` として指定される構造体の先頭のアドレスと各メンバのオフセットの情報を使い、チェックポイントファイルに値を保存する。図 5.2 の変数 `e` の例のように、各メンバの値を括弧でくくる。このようにすることで、構造体が入れ子になった場合でも保存することが可能になる。



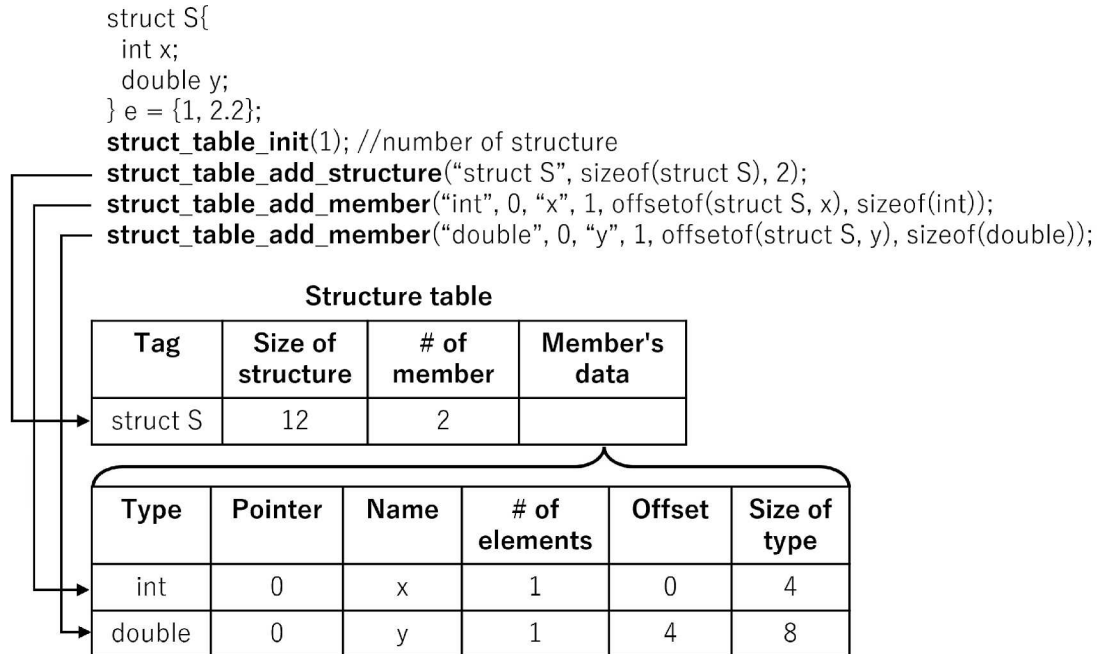


図 5.4: 構造体テーブルの例

### cp\_read 関数

cp\_read 関数は、チェックポイントファイルから変数の値を復元する関数である。cp\_read 関数の形式を以下に示す。

```
cp_read(type, pointer, name, size, address)
```

cp\_read 関数の引数は、cp\_write 関数の引数と全く同じである。

cp\_read 関数では、変数名をキーとして、チェックポイントファイルから該当する情報が書かれた行を特定する。次に、変数のサイズ分の長さの char 型の 2 次元配列を作成し、変数の値を文字列として読み取る。その後、それを復元する変数の型に変換し、元の変数に値を復元する。復元対象の変数がポインタの場合、アドレステーブルを使用し、読み取った変数名をその変数のアドレスに変換する。また、配列のインデックスが保存されている場合はインデックスで指定された位置にアドレスを移動させた後で、元の変数に値を復元する。復元対象の変数が構造体の場合、構造体テーブルから変数の型を取得し、読み取った値をその型に変換する。

ソースコード 5.1: 変換前のソースコード

---

```

1 codes 1
2 #checkpoi nt a
3 codes 2
4 #checkpoi nt b

```

---

### 5.2.3 指定された位置からの再開機能

本研究では、変換プログラムでソースコードにチェックポイント機能を埋め込む。変換前のソースコードの例をソースコード 5.1 に示し、それに対応する変換後のソースコードをソースコード 5.2 に示す。プログラムを途中から再開できるようにするために、タグが挿入された位置までの処理に対して if 文で囲む。各タグには ID が昇順に与えられており、これを `spot` 変数に外部から入力される再開番号と比較する。もし、ID が再開番号より小さければ処理をスキップする。例えば、ソースコード 5.2 において `spot = 2` の場合、7 行目の `\code 1` は実行されない。また、ID が再開番号と等しければ、この位置から再開することを意味するため、`cp_read` を実行し、復元を行う。例えば、ソースコード 5.2 において `spot = 1` の場合、3 - 5 行目の復元処理が実行される。さらに、ID が再開番号より大きければ、再開位置よりも先にある処理であることを意味するため、その処理を実行し、`cp_write` 関数を実行し、保存を行う。例えば、ソースコード 5.2 において `spot = 1` の場合、19 行目の `\code 2` と 20 - 22 行目の保存処理が実行される。なお、if 文の中に変数の宣言があるとコンパイル時にエラーとなる。そのため、変数宣言を if 文の前に移動し、`cp_addr` 関数を変数宣言の次行に挿入する。また、`st_add_structure` 関数と `st_add_membner` 関数は、`main` 関数の先頭に挿入する。

## 5.3 チェックポイント機能の動作確認と性能評価

### 5.3.1 チェックポイント機能の動作確認と性能評価

提案したチェックポイント機能を評価するために、`cp_write` 関数で配列の値をファイルに保存する時間と `cp_read` 関数で配列の値をファイルから復元する時間を計測した。提案手法は、`int`、`double` などの通常変数やポインタ、構造体変数に対してチェックポイントを実行することができる。それぞれの配列を保存・復元する時間を計測し、性能評価を行った。計測に使用

## ソースコード 5.2: 変換後のソースコード

---

```

1 if(spot<=1){
2   if(spot==1){
3     cp_fopen_r(checkpoint file directory);
4     cp_read(a's type, a's pointer, a's name, a's size, a's address);
5     cp_fclose_r();
6   } else{
7     codes 1
8     cp_fopen_w(checkpoint file directory);
9     cp_write(a's type, a's pointer, a's name, a's size, a's address);
10    cp_fclose_w();
11  }
12 }
13 if(spot<=2){
14   if(spot==2){
15     cp_fopen_r(checkpoint file directory);
16     cp_read(b's type, b's pointer, b's name, b's size, b's address);
17     cp_fclose_r();
18   } else{
19     codes 2
20     cp_fopen_w(checkpoint file directory);
21     cp_write(b's type, b's pointer, b's name, b's size, b's address);
22     cp_fclose_w();
23   }
24 }

```

---

したソースコードをソースコード 5.3・5.5 に示す。また、複数の変数を保存・復元する時間を計測し、性能評価を行った。計測に使用したソースコードをソースコード 5.6 に示す。ソースコード中の  $L$  は配列の要素数である。要素数を  $L = 1; 10; 100; 1; 000; 10; 000; 100; 000$  とした場合を計測した。この評価の評価環境を表 5.1 に示す。

また、並列プログラムを使用し、ワーカの離脱が発生した際に、プログラムを再実行した場合の実行時間とチェックポイント機能を使用してプログラムを途中から再開した場合の実行時間を比較した。並列プログラムには、NAS Parallel Benchmarks [53] の Integer Sort (IS) のクラス C を使用した。また、評価は、5 台の PC を使用した並列 VC システム上で実行した。

ソースコード 5.3: 評価で用いたソースコード (通常変数の配列)

---

```

1 int a[L];
2 Init(a); // initialization of a[]
3 #checkpoint a // execute cp_write or cp_read

```

---

ソースコード 5.4: 評価で用いたソースコード (ポインタ配列)

---

```

1 int *a;
2 for(int i; i<L; i++){
3   a[i]=(int*)malloc(sizeof(int));
4 }
5 Init(a); // initialization of a[]
6 #checkpoint a // execute cp_write or cp_read

```

---

ソースコード 5.5: 評価で用いたソースコード (構造体配列)

---

```

1 struct S{
2   int x;
3 } a[L];
4 Init(a); // initialization of a[]
5 #checkpoint a // execute cp_write or cp_read

```

---

使用した PC の仕様とソフトウェアをそれぞれ表 5.2, 表 5.3 に示す. 並列 VC システムのマスタは PC1 上で動作させ, ワーカーは PC2{PC5 上でそれぞれ 1 台ずつ動作させた.

提案したチェックポイント機能を評価するために, `cp_write` 関数で配列の値をファイルに保存する時間と `cp_read` 関数で配列の値をファイルから復元する時間を計測した. 時間計測には, `gettimeofday` 関数を使った. 保存・復元を 1 セットとして 10 回行い, それぞれの平均時間と最大, 最小時間を求めた. 保存・復元後に対象の変数の値をファイルに書き出し, 違いがないことを確かめることで, チェックポイント機能が正しく動作していることを確認した.

ソースコード 5.6: 評価で用いたソースコード (複数の変数)

---

```

1 int a_1;
2 int a_2;
3 ...
4 int a_L;
5 Init(a_1); // initialization
6 Init(a_2);
7 ...
8 Init(a_L);
9 #checkpoint a_1 a_2 ... a_L // execute cp_write or cp_read

```

---

表 5.1: 評価環境

|        |                   |
|--------|-------------------|
| CPU    | Core i7-9700      |
| Memory | 32GB              |
| OS     | Ubuntu 20.04.3LTS |

表 5.2: PC の仕様

| PC (role)           | OS              | CPU                                                  | Mem.  |
|---------------------|-----------------|------------------------------------------------------|-------|
| PC1<br>(master)     | Ubuntu<br>20.04 | Intel Xeon Silver 4216<br>×2, 2.1GHz<br>(16×2 cores) | 128GB |
| PC2{PC5<br>(worker) | Ubuntu<br>20.04 | Intel Core i9-10900K<br>3.7GHz (10 cores)            | 64GB  |

最初に、通常変数やポインタ、構造体の各配列を `cp.write` 関数で保存した時間の計測結果を図 5.5 に示す。図 5.5 より、3 種類の配列全てにおいて、配列の要素数が大きくなるほど保存にかかる時間は増加することがわかる。しかし、配列の要素数が 100,000 であっても、実行時間は、通常変数配列が約 18ms、ポインタ配列が約 54s、構造体配列が約 34ms であった。そのため、2.2.2 節で示したように、1 ジョブの実行時間が数十時間程度であることを考慮すると、

表 5.3: 使用するソフトウェア

| PC      | Name    | Version         |
|---------|---------|-----------------|
| PC1     | Apache  | 2.4.39          |
|         | Mysql   | 10.1.38-MariaDB |
|         | PHP     | 7.3.4           |
| All PCs | OpenMPI | 4.0.3           |

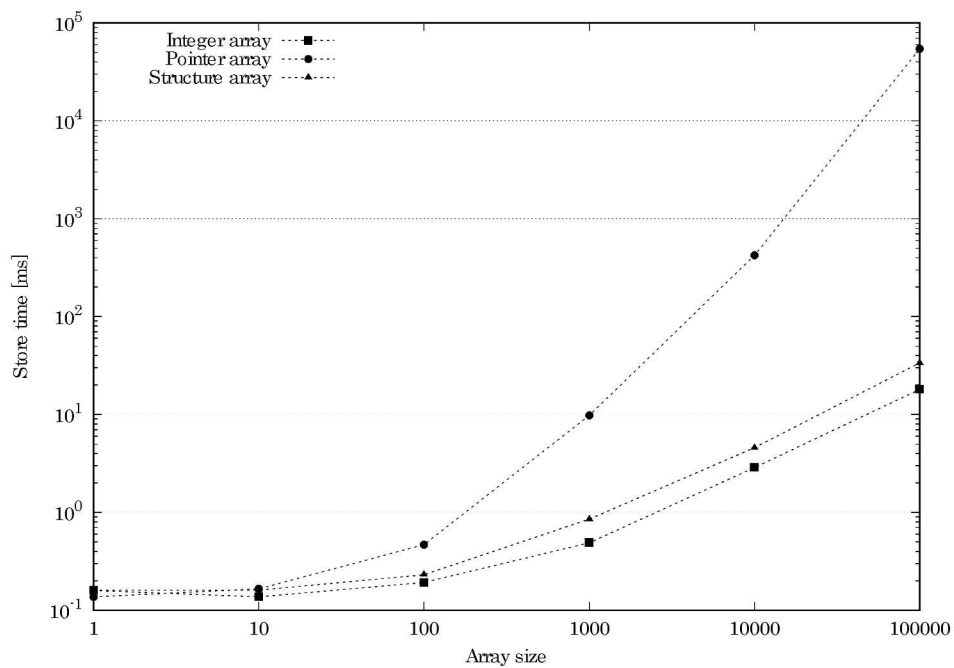


図 5.5: 各配列における要素数に対するチェックポイントの保存時間

ジョブの計算処理に対して大きなオーバーヘッドにはならないことを確認できた。また、配列の要素数が 100 ~ 100,000 のとき、ポインタ配列の時間が最も大きくなっていることがわかる。これは、通常変数と構造体の配列が各要素の値を保存するだけなのに対し、ポインタ変数は、各要素の値を保存するのに加え、各要素が指す `malloc` で確保された領域も 1 つの変数と同じように保存するためである。

次に、通常変数やポインタ、構造体の各配列を `cp_read` 関数で復元した時間の計測結果を図 5.6 に示す。図 5.6 より、3 種類の配列全てにおいて、配列の要素数が大きくなるほど復元に

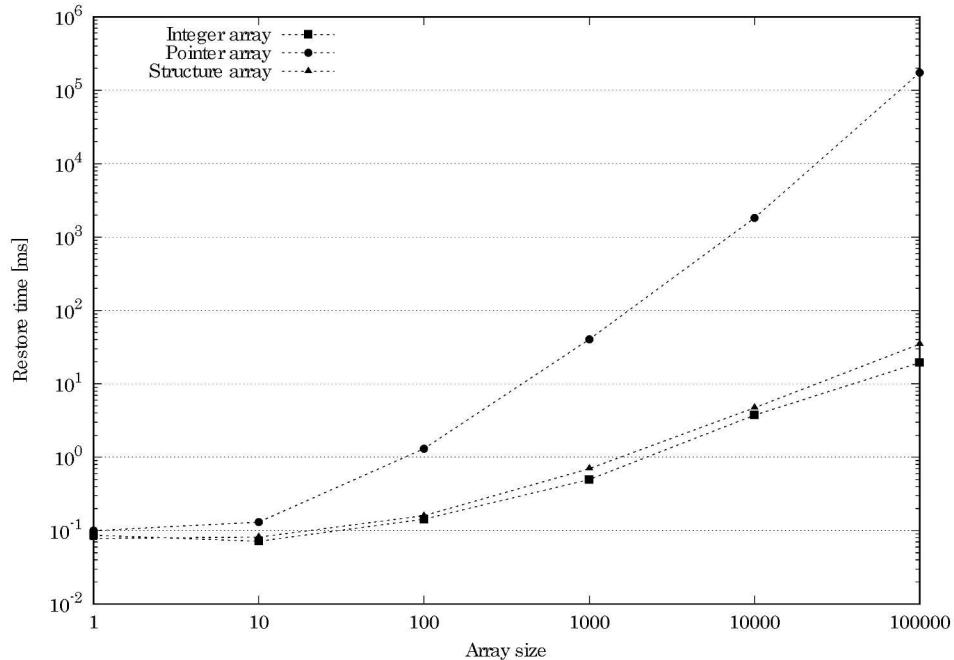


図 5.6: 各配列における要素数に対するチェックポイントの復元時間

かかる時間は増加することがわかる。しかし、配列の要素数が 100,000 であっても、実行時間は、通常変数配列が約 19ms、ポインタ配列が約 173s、構造体配列が約 35ms であった。そのため、本来の計算処理に対して大きなオーバーヘッドにはならないことを確認できた。また、配列の要素数が 10 ~ 100,000 のとき、ポインタ配列の時間が最も大きくなっている。これは、`cp_write` 関数の場合と同様に、通常変数と構造体の配列が各要素の値を復元するだけなのに対し、ポインタ変数は、各要素の値を復元するのに加え、各要素が指す `malloc` 関数で確保された領域を作成し、1 つの変数と同じように復元するためである。

次に、通常変数の配列とその要素数と同数の変数を `cp_write` 関数で保存した時間の計測結果を図 5.7 に示す。図 5.7 より、変数の数が大きくなるほど保存にかかる時間は増加することが分かる。しかし、変数の数が 100,000 であっても、実行時間は、約 17s 程度であり、本来の計算処理に対して大きなオーバーヘッドにはならないことを確認できた。また、変数の数が 100 ~ 100,000 のとき、通常変数の配列と比べて時間が大きくなっている。これは、チェックポイントファイルに変数を保存するときに、その変数がすでに保存されていないか確認するためである。保存する各変数ごとに、その確認を行うため、時間がかかっていると考えられる。

次に、通常変数の配列とその要素数と同数の変数を `cp_read` 関数で復元した時間の計測結果を図 5.8 に示す。図 5.8 より、変数の数が大きくなるほど復元にかかる時間は増加することが

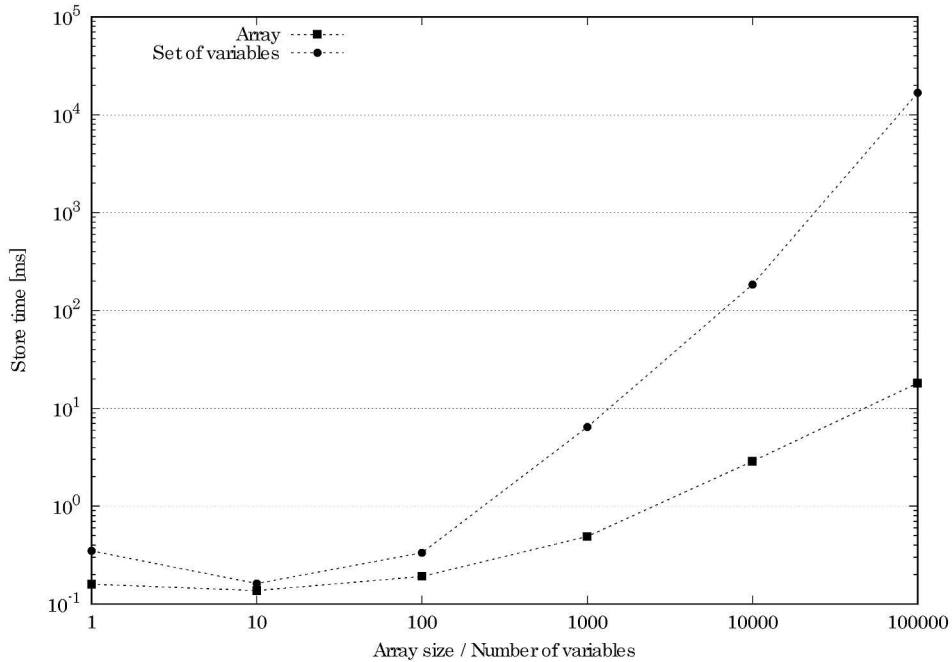


図 5.7: 配列と複数の変数における要素数に対するチェックポイントの保存時間

分かる．しかし、変数の数が 100,000 であっても、実行時間は、約 200s 程度であり、本来の計算処理に対して大きなオーバーヘッドにはならないことを確認できた．また、変数の数が 100 ～ 100,000 のとき、通常変数の配列と比べて時間が大きくなっている．これは、チェックポイントファイルから復元する変数を探すことに時間がかかっているためである．復元する各変数ごとに、チェックポイントファイルの先頭から復元する変数を探すため、時間がかかっていると考えられる．

図 5.9 に、ワーカの離脱が発生した際に、プログラムを再実行した場合とチェックポイント機能を使用してプログラムを途中から再開した場合の IS の実行時間を示す．IS は、バケットソートを 10 回のループで行う．今回の評価では、6 回目のループの終了後に 1 台のワーカが離脱すると想定した．チェックポイント機能を使用した場合、実行時間は、離脱が発生しない場合とほぼ変わらなかった．一方で、再実行の場合は、ワーカの離脱が発生しなかった場合と比較して、実行時間が約 200 秒増加している．これは、ワーカの離脱が発生するまでの処理時間に相当する．この結果から、提案したチェックポイント機能が並列 VC に対して有効であることがわかった．



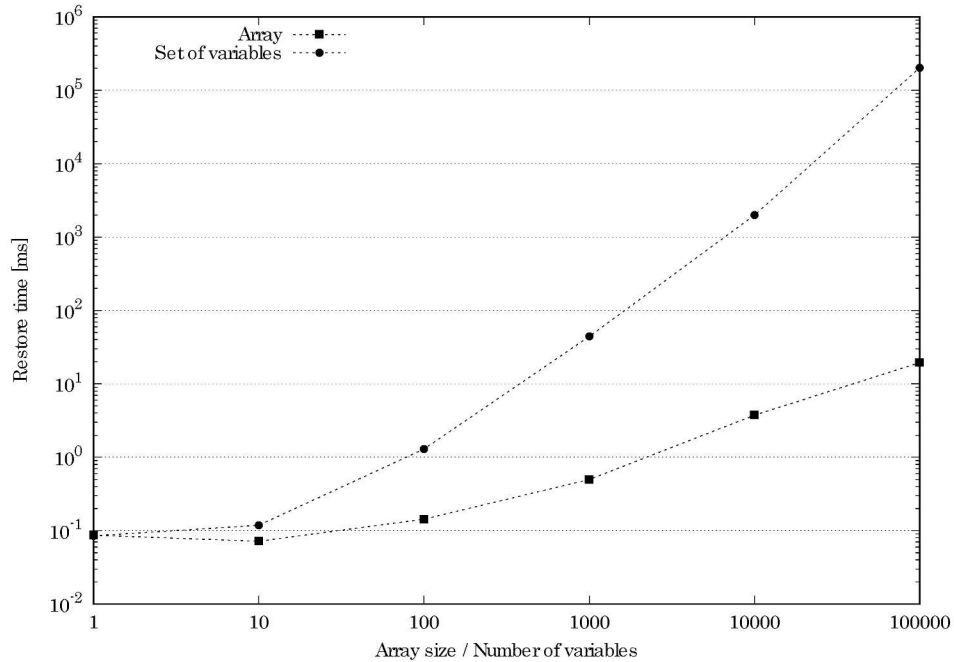


図 5.8: 配列と複数変数における要素数に対するチェックポイントの復元時間

## 5.4 むすび

本章では、ソースコードレベルのチェックポイント機能の概要と詳細な実装を説明した。変数の値をチェックポイントファイルに保存する関数 `cp write` とファイルから変数の値を復元する関数 `cp read` を作成した。また、作成した変換プログラムでこれらの関数を、ソースコードに自動的に埋め込むことでチェックポイント機能を実装した。さらに、チェックポイントを行う位置を指定するタグまでの処理を `if` 文で囲むことにより、プログラムを途中から再開することを可能にした。

性能評価では、`int`, `double` などの通常変数やポインタ、構造体の配列、複数の変数を保存・復元する時間を計測した。その結果、変数の要素数・変数の数が 100,000 であっても、保存する時間はそれぞれ、約 18ms, 約 54s, 約 34ms, 約 17s 程度であった。また、復元する時間はそれぞれ、約 19ms, 約 173s, 約 35ms, 約 200s 程度であった。このことから、ジョブの計算処理に対して大きなオーバーヘッドにはならないことを確認した。また、並列プログラムを使用した評価では、ワーカの離脱が発生した際に、チェックポイント機能を使用した場合の実行時間は、ワーカの離脱が発生しなかった場合の実行時間とほぼ変わらなかった。このことから、提案したチェックポイント機能が並列 VC に対して有効であることがわかった。

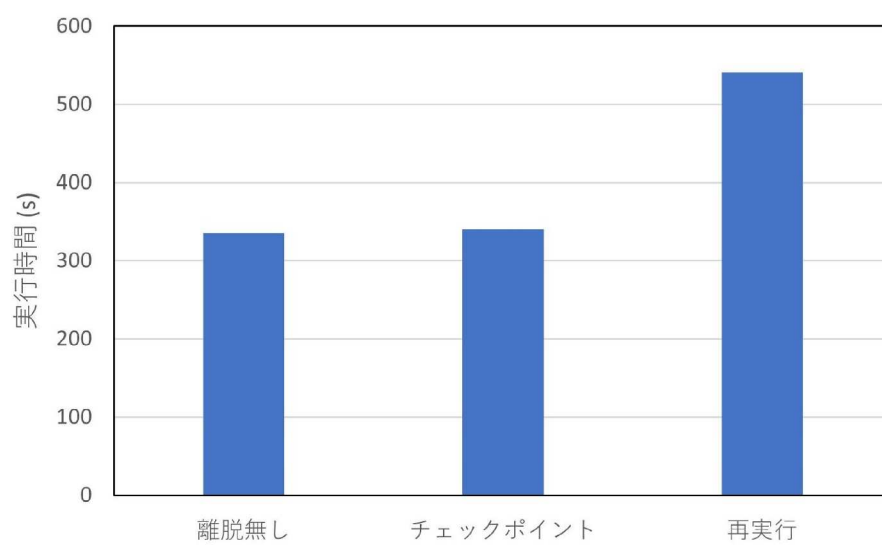


図 5.9: 再実行とチェックポイント機能の IS の実行時間の比較

## 第 6 章

# 結言

### 6.1 結論

コンピュータの性能向上にともない、様々な研究分野で科学技術計算が不可欠になっており、計算問題の規模や取り扱われるデータ量が増加している。VC には、高性能な分散計算環境を安価に構築できる利点があるため、大規模な計算問題の実行基盤として注目を集めている。

VC では、一般ユーザが所有する PC などをワーカとして使用するため、ワーカ同士が直接通信を行うようには構成されておらず、シャットダウンやネットワーク障害などにより、ワーカが任意のタイミングで計算から離脱するという特徴がある。このため、既存の VC では、個々の計算（ジョブ）が独立した分散計算のみが対象とされ、計算中にワーカ間の通信を伴う並列計算は対象外とされてきた。このことが、VC の適用範囲を狭める要因となっていた。

そこで、本研究では、並列 VC を実現することを目的とし、サーバ支援型通信を基にした並列 VC システムを提案、実装し、並列 VC の実現可能性を示した。

以下で、本論文の各章の結論をまとめる。

#### 第 2 章

並列 VC を実現するための要件を示すために、VC の種類と既存手法について述べた。まず、既存の VC の計算モデルと実例を述べた。次に並列 VC の計算モデルおよび、並列 VC を実現するための要件を定義した。その後、既存の並列分散計算システムの構築手法と計算を途中から再開するための既存のチェックポイント手法を述べ、それらの問題点を述べた。

#### 第 3 章

サーバ支援型通信システムとジョブ配布システムから構成される、並列 VC サーバを提案、実装した。また、各システムの動作確認と性能評価を行った。

サーバ支援型通信において、HTTP\_Send/HTTP\_Recv 関数および HTTP\_Gather 関数の

実行時間は、データサイズとワーカ数  $W$  の増加に伴って、増加することがわかった。しかし、ワーカごとに性能差がある VC 環境において許容範囲内であることがわかった。IS および EP の実行時間は、MPI と比較すると、ほとんどの場合で遅くなっていた。しかし、本章で提案したサーバ支援型通信システムを使用することで、VC 上で並列計算が可能であることを実証した。また、姫野ベンチマークにおいて、 $W = 1$  の場合に比べて、 $W = 4$  に増やした場合に実行時間が減少していることから、並列 VC の適用効果がある程度示すことができた。

ジョブ配布システムにおいて、実行時間の評価では、マスタプログラムにおける DB アクセスの競合がジョブ配布と結果受信の処理時間に影響を与えること、また、ワーカ数の増加にともないサーバ負荷と通信輻輳による遅延が発生することを明らかにした。ただし、マスタの PHP 実行時間はジョブの実行時間と比較して十分に小さく、ジョブ配布機能が実用に耐えうるものであることを確認した。さらに、ワーカ離脱時の再配布機能が期待通りに動作することを確認した。

#### 第 4 章

本章では、WebAssembly を用いたブラウザベースの並列 VC クライアントを提案した。また、並列 VC クライアント上で並列計算のベンチマークを実行した際の動作確認を行い、実行時間について評価を行った。その結果、様々な OS でブラウザベースの並列 VC クライアントを使用した並列計算が実行可能であることを確認した。また、実行時間は、C 言語で作成された簡易的なクライアントと比較して、クラス S で最大約 3 秒、クラス W で最大約 5 秒ほどしか実行時間が増加しなかった。この差は、1 ジョブの実行時間が数十時間程度であることを考慮すると、無視できる程度の差であることが明らかとなった。

#### 第 5 章

本章では、ソースコードレベルのチェックポイント機能の概要と詳細な実装を説明した。変数の値をチェックポイントファイルに保存する関数 `cp write` とファイルから変数の値を復元する関数 `cp read` を作成した。また、作成した変換プログラムで、これらの関数をソースコードに自動的に埋め込むことでチェックポイント機能を実装した。さらに、チェックポイントを行う位置を指定するタグまでの処理を `if` 文で囲むことにより、プログラムを途中から再開することを可能にした。

性能評価では、`int`、`double` などの通常変数やポインタ、構造体の配列、複数の変数を保存・復元する時間を計測した。その結果、変数の要素数・変数の数が 100,000 であっても、保存する時間はそれぞれ、約 18ms、約 54s、約 34ms、約 17s 程度であった。また、復元する時間はそれぞれ、約 19ms、約 173s、約 35ms、約 200s 程度であった。このことから、ジョブの計算処理に対して大きなオーバーヘッドにはならないことを確認した。また、並列プログラムを使用し

た評価では、ワーカの離脱が発生した際に、チェックポイント機能を使用した場合の実行時間は、ワーカの離脱が発生しなかった場合の実行時間とほぼ変わらなかった。このことから、提案したチェックポイント機能が並列 VC に対して有効であることがわかった。

## 6.2 今後の課題

本論文では、PC クラスタ内に構築したマスタと仮想的なワーカを使用し、提案した並列 VC システムの性能評価を行った。今後は、より実際の運用に近いインターネットなどの環境で性能評価を行い、実運用上での課題を明らかにすることが挙げられる。また、実際の環境では、悪意を持って計算を妨害するワーカやハードウェアの故障やソフトウェアの不具合などで誤った計算結果をマスタに返却するワーカが少なからず存在する。これらのワーカの対策として、計算を冗長実行し、今回提案したサーバ支援型通信の特性を考慮した新たな多数決手法を考案することが挙げられる。さらに、様々な分野の研究者との共同研究を通して、実際にシミュレーションなどで使用されるプログラムやデータを用いて、並列 VC の適用効果を検証することが挙げられる。

## 謝辞

本研究は、山口大学大学院創成科学研究科システム・デザイン工学系専攻在学中に、福士 将教授のもとで行われました。本研究を進めるにあたり、懇切丁寧かつ適切なご指導、ご鞭撻を賜りました同専攻 福士 将 教授に心から深く感謝申し上げます。また、本研究に対して、多くの助言やアドバイスをくださった、黒川 陽太 助教に深く感謝申し上げます。

本論文において副査を務めていただきました、山口 真悟 教授、爲末 隆弘 教授、井田 悠太 准教授に深く感謝申し上げます。

さらに、本論文をまとめるにあたって大変お世話になりました、ICT 基盤センターの皆様、計算機システム工学研究室の皆様に厚くお礼申し上げます。

## 参考文献

- [1] WebAssembly: <https://developer.mozilla.org/ja/docs/WebAssembly/Concepts> (最終閲覧日: 2025 年 11 月 5 日)
- [2] Top500: <https://top500.org/lists/top500/2025/06/> (最終閲覧日: 2025 年 11 月 5 日)
- [3] El Capitan: <https://asc.llnl.gov/exascale/el-capitan> (最終閲覧日: 2025 年 11 月 5 日)
- [4] LINPACK: <http://www.netlib.org/linpack/> (最終閲覧日: 2025 年 11 月 5 日)
- [5] U.S. Department of Energy: <https://www.energy.gov/articles/does-nnsa-signs-600-million-contract-build-its-first-exascale-supercomputer> (最終閲覧日: 2025 年 11 月 5 日)
- [6] UNICORE: <https://www.unicore.eu/> (最終閲覧日: 2025 年 11 月 5 日)
- [7] SETI@home: <https://setiathome.berkeley.edu/> (最終閲覧日: 2025 年 11 月 5 日)
- [8] Folding@home: <https://foldingathome.org/about/> (最終閲覧日: 2025 年 11 月 5 日)
- [9] World Community Grid: <https://www.worldcommunitygrid.org/> (最終閲覧日: 2025 年 11 月 5 日)
- [10] Maxwell I. Zimmerman, Justin R. Porter, Michael D. Ward, Sukrit Singh, Neha Vithani, Artur Meller, Upasana L. Mallimadugula, Catherine E. Kuhn, Jonathan H. Borowsky, Rafal P. Wiewiora, Matthew F. D. Hurley, Aoife M. Harbison, Carl A. Fogarty, Joseph E. Coffland, Elisa Fadda, Vincent A. Voelz, John D. Chodera and Gregory R. Bowman, "SARS-CoV-2 simulations go exascale to predict dramatic spike opening and cryptic pockets across the proteome," *Nature Chemistry*, vol. 23 ,pp. 651{659, July 2021.
- [11] Aurora: <https://www.anl.gov/aurora> (最終閲覧日: 2025 年 11 月 5 日)
- [12] Amicable Numbers: <https://sech.me/boinc/Amicable/> (最終閲覧日: 2025 年 11 月 5 日)
- [13] Asteroids@home: <https://asteroidsathome.net/boinc/> (最終閲覧日: 2025 年 11 月 5 日)

- [14] Climateprediction.net: <https://climateprediction.net/> (最終閲覧日: 2025 年 11 月 5 日)
- [15] DENIS@Home: <https://denis.usj.es/denisathome/> (最終閲覧日: 2025 年 11 月 5 日)
- [16] destributed.net: [https://www.distributed.net/Main\\_Page](https://www.distributed.net/Main_Page) (最終閲覧日: 2025 年 11 月 5 日)
- [17] Einstein@home: <https://einsteinathome.org/ja/home> (最終閲覧日: 2025 年 11 月 5 日)
- [18] GPUGrid.net: <https://gpugrid.net/gpugrid/> (最終閲覧日: 2025 年 11 月 5 日)
- [19] LHC@home: <https://lhathome.cern.ch/lhathome/index.php> (最終閲覧日: 2025 年 11 月 5 日)
- [20] LODA: <https://boinc.loda-lang.org/loda/> (最終閲覧日: 2025 年 11 月 5 日)
- [21] Milkyway@home: <https://milkyway.cs.rpi.edu/milkyway/> (最終閲覧日: 2025 年 11 月 5 日)
- [22] NFS@home: <https://escatter11.fullerton.edu/nfs/> (最終閲覧日: 2025 年 11 月 5 日)
- [23] NumberFields@home: <https://numberfields.asu.edu/NumberFields/> (最終閲覧日: 2025 年 11 月 5 日)
- [24] ODLK: <https://boinc.progger.info/odlk/> (最終閲覧日: 2025 年 11 月 5 日)
- [25] PrimeGrid: <https://www.primegrid.com/> (最終閲覧日: 2025 年 11 月 5 日)
- [26] Radioactive@Home: <http://radioactiveathome.org/boinc/> (最終閲覧日: 2025 年 11 月 5 日)
- [27] RakeSearch: <https://rake.boincfast.ru/rakesearch/> (最終閲覧日: 2025 年 11 月 5 日)
- [28] Ramanujan Machine: <https://www.ramanujanmachine.com/> (最終閲覧日: 2025 年 11 月 5 日)
- [29] RNA World: <https://www.rnaworld.de/rnaworld/> (最終閲覧日: 2025 年 11 月 5 日)
- [30] Rosetta@home: <https://boinc.bakerlab.org/rosetta/> (最終閲覧日: 2025 年 11 月 5 日)
- [31] SIDock@home: <https://www.sidock.si/sidock/> (最終閲覧日: 2025 年 11 月 5 日)
- [32] SRBase: <https://srbase.my-firewall.org/sr5/> (最終閲覧日: 2025 年 11 月 5 日)
- [33] YAFU: <https://yafu.myfirewall.org/yafu/> (最終閲覧日: 2025 年 11 月 5 日)
- [34] Yoyo@home: <https://www.rechenkraft.net/yoyo/> (最終閲覧日: 2025 年 11 月 5 日)
- [35] L. F. G. Sarmenta: "Sabotage-tolerance Mechanisms for Volunteer Computing Systems," Future Generation Computer Systems, Vol. 18, Issue 4, pp. 561-572, May 2001.
- [36] P. Domingues, B. Sousa, and L. M. Silva: "Sabotage-tolerance and trust management in desktop grid computing," Future Generation Computer Systems, Vol. 23, Issue 7,



- pp. 904{912, August 2007.
- [37] K. Ochi and M. Fukushi: "A Group-Based Job Scheduling Method for Parallel Volunteer Computing," Proceedings of International Symposium on Computing and Networking, pp. 571{575, December 2015.
  - [38] L.F.G. Sarmenta, S. Hirano, "Bayanihan: building and studying web-based volunteer computing systems using Java," Future Generation Computer Systems, Vol. 15, Issues 5{6, pp. 675{686, October 1999.
  - [39] D.P. Anderson, "BOINC: A Platform for Volunteer Computing," Journal of Grid Computing, Vol. 18, pp. 99{122, March 2020.
  - [40] A. Calderón, F. Garcí-Carballeira, B. Bergua et al., "Expanding the volunteer computing scenario: A novel approach to use parallel applications on volunteer computing," Future Generation Computer Systems, Vol. 28, Issue 6, pp. 881{889, June 2012.
  - [41] T. LeBlanc, R. Anand, Edgar Gabriel et al., "VolpexMPI: an MPI Library for Execution of Parallel Applications on Volatile Nodes," EuroPVM/ MPI 2009: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 5759, pp. 124{133, September 2009.
  - [42] HTCondor: [https:// htcondor.org/ index.html](https://htcondor.org/index.html) (最終閲覧日: 2025 年 11 月 5 日)
  - [43] BOINC: [https:// boinc.berkeley.edu/](https://boinc.berkeley.edu/) (最終閲覧日: 2025 年 11 月 5 日)
  - [44] MPI, [https:// www.mpi-forum.org/](https://www.mpi-forum.org/) (最終閲覧日: 2025 年 11 月 5 日)
  - [45] Genivia, [https:// www.genivia.com/ dev.html](https://www.genivia.com/dev.html) (最終閲覧日: 2025 年 11 月 5 日)
  - [46] CRIU: [https:// www.criu.org/](https://www.criu.org/) (最終閲覧日: 2025 年 11 月 5 日)
  - [47] BLCR: [https:// upc-bugs.lbl.gov/ blcr/ doc/ html/ BLCR\\_Admin\\_Guide.html](https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html) (最終閲覧日: 2025 年 11 月 5 日)
  - [48] FTI: [https:// github.com/ leobago/ fti](https://github.com/leobago/fti) (最終閲覧日: 2025 年 11 月 5 日)
  - [49] Marcos Maronas, Sergi Mateo, Kai Keller, Leonardo Bautista-Gomez, Eduard Ayguadé, and Vicenç Beltran, "Extending the OpenCHK Model with Advanced Checkpoint Features," Future Generation Computer Systems, Vol. 112, pp. 738{750, November 2020.
  - [50] BOINC API: [https:// boinc.berkeley.edu/ trac/ wiki/ BasicApi](https://boinc.berkeley.edu/trac/wiki/BasicApi) (最終閲覧日: 2025 年 11 月 5 日)
  - [51] Apache: [https:// httpd.apache.org/](https://httpd.apache.org/) (最終閲覧日: 2025 年 11 月 5 日)
  - [52] MySQL: [https:// www.mysql.com/](https://www.mysql.com/) (最終閲覧日: 2025 年 11 月 5 日)
  - [53] NAS Parallel Benchmarks: [https:// www.nas.nasa.gov/ software/ npb.html](https://www.nas.nasa.gov/software/npb.html) (最終閲覧

日: 2025 年 11 月 5 日)

- [54] 姫野ベンチマーク: <https://i.riken.jp/supercom/documents/himenobmt/> (最終閲覧日: 2026 年 2 月 4 日)
- [55] A. Naruse, K. Nakashima, S. Sumimoto, "A Proposal and Evaluation of All-to-all Algorithm for Multi-core PC Cluster Systems," IPSJ Transactions on Advanced Computing Systems, Vol.3, No. 3, pp. 166{177, September 2010.
- [56] Inter-connect simulator NSIM, [https://www.sskn.gr.jp/MAINSITE/download/wg\\_report/multicore/report\\_multicore\\_3-1.pdf](https://www.sskn.gr.jp/MAINSITE/download/wg_report/multicore/report_multicore_3-1.pdf) (最終閲覧日: 2025 年 11 月 5 日)
- [57] Emscripten: <https://emscripten.org/> (最終閲覧日: 2026 年 2 月 19 日)
- [58] Web API の紹介: [https://developer.mozilla.org/ja/docs/Learn\\_web\\_development/Extensions/Client-side\\_APIs/Introduction](https://developer.mozilla.org/ja/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction) (最終閲覧日: 2026 年 2 月 19 日)
- [59] フェッチ API の使用: [https://developer.mozilla.org/ja/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/ja/docs/Web/API/Fetch_API) (最終閲覧日: 2026 年 2 月 19 日)
- [60] emscripten.h: [https://emscripten.org/docs/api\\_reference/emscripten.h.html](https://emscripten.org/docs/api_reference/emscripten.h.html) (最終閲覧日: 2026 年 2 月 19 日)