Vol.2022-OS-155 No.7
2022/5/27

# 異種プログラミング言語間での高速なデータ共有を実現する軽量データ変換技術

野澤 真伸[1]　今村 智史[2]　河野 健二[1]

**概要**：異種プログラミング言語を併用することで，それぞれの長所を活かしてアプリを実装する場合がある．その際，異種プログラミング言語間でデータ構造を共有する処理が必要となる．データ共有を行うためには，データ構造を特定のプログラミング言語やマシンアーキテクチャに依存しない中立なフォーマットを経由してやりとりすることが一般的である．中立なフォーマットへの/からの変換をシリアライズ/デシリアライズといい，特に不揮発性メモリのような高速ストレージを媒介してデータ共有を行う場合，そのオーバーヘッドは無視できない．本論文では異言語間でのデータ共有高速化の手法を提案する．異言語間でのデータ共有を行う場合，あらかじめデータ共有を行う言語やアーキテクチャが特定できる場合も多く，そのような状況に特化した方式を示す．例えば，数値計算の前処理を Julia で行い，その後の計算処理を Python で行う場合などがある．このような状況では，1) 汎用性の高い中立なフォーマットを用いる必要がないため，シリアライズ/デシリアライズが簡略化できることに加え，2) ランダムアクセスが高速であるという永続メモリの特性を活かしたシリアライズ/デシリアライズ処理が可能となる．提案手法を Python と Julia 間のデータ共有に用い，Python と Julia に実装された従来のシリアライズ/デシリアライズ処理と比較したところ， Python から Julia にデータを送信する場合は，配列のシリアライズとデシリアライズがそれぞれ 1.57 倍と $3.03 \times 10^6$ 倍，辞書のシリアライズとデシリアライズが最大でそれぞれ 2.38 倍と 1.23 倍の高速化が可能であることがことが確認できた．また Julia から Python にデータを送信する場合は，配列のシリアライズとデシリアライズはそれぞれ 3.26 倍と $4.06 \times 10^5$ 倍，辞書のシリアライズとデシリアライズは最大でそれぞれ 1.78 倍と 14.4 倍の高速化が可能であることが確認できた．

## 1. はじめに

プログラミング言語を跨いだデータ共有は多くの場面で行われている．データ共有をするためには，シリアライズ・デシリアライズという処理が用いる必要がある．シリアライズとはプログラミング言語で使用されるデータ構造をプログラミング言語やマシンアーキテクチャに依存しないフォーマットに基づいたバイト列に変換する処理のことで，デシリアライズはその逆変換である．フォーマットの中でも，JSON や Pickle といった処理できるデータ構造・データ型に関して汎用性を持つものが広く用いられる．

汎用性の高いフォーマットを使うことが常に適切であるとは限らない．特に相手の環境が既知である場合，そのようなフォーマットを使うことがオーバーヘッドになることもある．例えば，同じマシンでデシリアライズすることがわかっている場合，エンディアンを考慮したフォーマットに変換することは明らかに無駄な処理と言える．そのような状況では，逆に相手の環境に特化したフォーマットを用いることで，より高速なシリアライズ・デシリアライズが可能になると考えられる．

データ共有の際，相手の環境が既知である場面がある．例えば，数値計算において前処理を Julia で行い，その後の処理を Python で行う場合などである．このような既知の相手の環境に特化したフォーマットを用いることでシリアライズ・デシリアライズを単純化でき，高速なデータ共有が可能になると考えられる．

本研究は既知の相手の環境に特化することで，汎用性が高いフォーマットよりも高速なデータ共有が可能となる手法を提案する．具体的には「デシリアライズ側のプログラミング言語」,「シリアライズとデシリアライズが行われるマシン」,「扱うデータ構造・データ型」を限定することで，実装とフォーマットを簡略化し，処理を高速化する．共有を行うプログラミング言語は Python と Julia を採用した．

提案手法の評価のために，インメモリのデータをシリアライズ・デシリアライズする時にかかった時間を計測し，汎用性が高い従来手法のそれと比較した．本実験はシリア

[1] 慶應義塾大学
Keio University
[2] 富士通株式会社 コンピューティング研究所

ライズ/デシリアライズの純粋なオーバーヘッドを計測するために， I/O 処理は計測対象外とし， Garbage Collector は無効化した．比較対象には Python のシリアライズ用モジュールである Pickle を採用した．

まず， Python 上の要素数が $4 \times 10^6$ のデータ構造をシリアライズするのにかかった時間と，そのデータ構造を Julia 側でデシリアライズした時にかかった時間を比較した結果を述べる．配列のシリアライズは 2.61 倍，デシリアライズでは $3.03 \times 10^6$ 倍高速化できた．辞書のシリアライズは 2.38 倍，デシリアライズでは 1.23 倍高速化できた．

Julia 上の要素数が $4 \times 10^6$ のデータ構造をシリアライズするのにかかった時間と，そのデータ構造を Python 側でデシリアライズした時にかかった時間を比較した結果を述べる．配列のシリアライズは 3.26 倍，デシリアライズでは $4.06 \times 10^5$ 倍高速化できた．辞書のシリアライズは 1.78 倍，デシリアライズでは 14.4 倍高速化できた．

本論文の構成を以下に示す．2 章ではプログラミング言語間のデータ共有を支えるシリアライズとデシリアライズについて説明する．3 章では本研究のアプローチを述べる．4 章では提案手法の実装を説明する．5 章では実験を通して，提案手法と環境に非依存的な手法とで純粋なシリアライズ・デシリアライズのオーバーヘッドを比較する．6 章では関連研究を紹介する．7 章ではまとめを述べる．

## 2. プログラミング言語間のデータ共有

### 2.1 シリアライズとデシリアライズ

プログラミング言語間でのデータ共有は多くのアプリケーションで行われる処理である．例えば図 1 のように，数値計算の前処理を Julia [8] で行い，その後の処理を Python [3] で行うアプリケーションがあるとする．このアプリを動かすためには， Python と Julia でデータを共有する必要があり，そのためにシリアライズ/デシリアライズという処理が用いられる．

ここで， Julia と Python を組み合わせる意義について簡単に説明する．Julia は科学技術計算のために開発されたプログラミング言語である．特徴としては数式に近く簡潔な文法ありながら，プログラマが細かく明示しなくともがコンパイラが最適化を行うため，プログラマが最適化のために気をつけてプログラミングをせずとも，高速な処理が可能という点が挙げられる [6], [7]．ただし， Julia は 2012 年に登場した新しい言語であるため， Python と比べるとコミュニティの規模が小さく，パッケージの数も劣っている [13], [9]．一方， Python で機械学習のような行列演算を含む重い計算処理を行う場合は， NumPy [10] や PyTorch [15] といった C/C++ で実装されたパッケージが豊富に提供されている．しかし Julia がパッケージを用いなくとも高速に処理できる処理も， Python は外部パッケージの機能を活用しないと処理が遅くなるため，プログラミングの容易さには難がある．そこで，簡単な記述で高速に処理できる Julia を前処理に用いて，それ以降の複雑な計算処理は Python を使うことが考えられる．


図 1: Julia と Python 間でデータを共有するアプリケーション

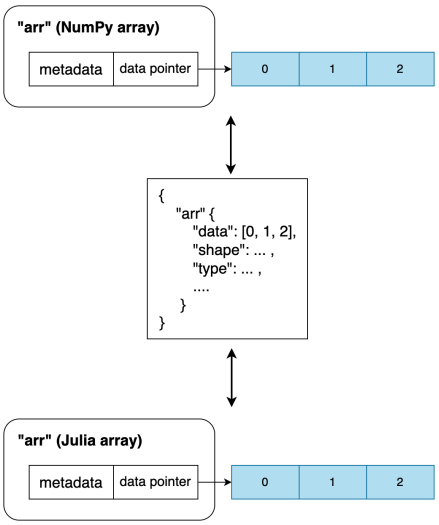



図 2: 配列オブジェクトのシリアライズ/デシリアライズ

### 2.2 シリアライズとデシリアライズの仕組み

シリアライズとは「メモリ上のオブジェクト・データ構造を特定のプログラミング言語やマシンアーキテクチャに依存しないフォーマットのバイト列に変換する処理」で，デシリアライズとはシリアライズの逆変換を行う処理である．中立なフォーマットに変換する目的は，シリアライズする段階ではデシリアライズする相手の環境が分からなくても，そのようなフォーマットを用いることで，相手が正しくデータをデシリアライズできることを期待できるためである．シリアライズ・デシリアライズ時に用いられるバイト列のフォーマットは様々だが，広く使われるものは「様々な種類のデータ構造を処理できる」という汎用性を持つ．そのようなフォーマットが広く使われるのはプログラミングが簡単になるためである．中立的なフォーマットを介して， Python と Julia で配列オブジェクト arr を共有する様子を図 2 に示す．中立的なフォーマットには，メタデータの他に配列本体のデータ [0, 1, 2] をコピーする必要がある．

広く使われるフォーマットの例として Python のシリアライズ用モジュールである Pickle [11] を取り上げる．このモジュール は Python のオブジェクトしか扱うことが

できないが，その中であればほとんど全てのオブジェクトを処理することができる汎用性を備えており，さらに古い Python が使う昔のバージョンの Pickle の出力をデシリアライズできるという後方互換性を持つ．Pickle は仮想スタックマシンを動かしてオブジェクトをデシリアライズする仕組みになっており，そのフォーマットはその仮想スタックマシンを動かすためのバイトコードになっている．

プログラミング言語間でのデータを共有をファイルを介して行う場合，基本的にパフォーマンスを律速するのは I/O であるが， I/O デバイスが高速になるとシリアライズ/デシリアライズのオーバヘッドが次第に顕在化してくる．例えば，近年登場した Persistent Memory [5] は DRAM に近いレイテンシでランダムアクセスできる不揮発性のメモリであり，このデバイスを用いてデータを共有する際は，シリアライズのオーバヘッドがパフォーマンスに与える影響がより支配的になる．よって，高速なストレージを用いたデータ共有の性能を高めるには，シリアライズ/デシリアライズの高速化が不可欠だと考えられる．

## 3. アプローチ

データを受ける相手の環境が未知の時にプログラミング言語間でデータ共有を行う場合は，必ず中立的なフォーマットを使う必要がある．しかし相手の環境が既知である場合，そのようなフォーマットを用いることでオーバーヘッドを生むことは明らかである．

実際，前もって相手の環境がわかっている状況でデータをシリアライズする場面は多く，このような場面で相手の環境に非依存的なフォーマットを使う必要はない．例えば 2.1 で挙げたアプリケーションや，ストリーム処理基盤が行われるスナップショットを用いた障害回復は，前もって用いる言語や用いるデータ構造が特定できる [1].

本研究では既知である相手の環境に特化することで，中立的なフォーマットを用いる場合に比べて高速なデータ共有が可能となる手法を提案する．具体的には以下に示す条件を満たすように実装とフォーマットを簡略化することで，シリアライズとデシリアライズを高速化する．

(1) デシリアライズする言語を前もって指定する
(2) 同じ命令セットアーキテクチャを持つマシンでシリアライズとデシリアライズを行う
(3) 使うデータ構造とデータ型を限定する

1 番目の条件はデシリアライズする言語に親和性がある形にデータ構造をシリアライズすることを意図している．これによってデシリアライズを高速化できると考えられる．2 番目の条件は実装を簡略化することが主な意図であるが，さらに，エンディアンを無視できるようにためシリアライズ・デシリアライズの処理が簡略化することができる．3 番目の条件は，汎用性の高い手法に比べてフォーマットに書き込む量を削減することと実装の簡略化を意図している．


図 3: 定義したフォーマット

表 1: 提案手法が変換をサポートするデータ構造
（o は変換可， x は変換不可を表す）

| | 配列 | リスト | 辞書 |
|---|---|---|---|
| Python | o | o | o |
| Julia | o | x | o |

具体的にどのデータ構造とデータ型に対応するかは 4.1 節で述べる．

## 4. 実装

### 4.1 概要

独自に定義したフォーマットを介した CPython , Julia 間のシリアライズ・デシリアライズの処理を実装した．この内，Python 側は CPython が提供する C-API [14] を用いたモジュールとして実装した．定義したフォーマットのイメージを図 3 に示す．フォーマットのヘッダ部分は以下の 4 つの情報を含んでおり，それ以降はデータ構造・データ型に応じたフォーマットでオブジェクトをシリアライズする．

- ELEMENT TYPE: データ構造に含まれる要素の型
- STRUCTURE: データ構造の種類
- DEST LANG: デシリアライズする側のプログラミング言語
- LENGTH: 要素数

ここで，提案手法が変換をサポートするデータ構造・データ型について説明する．まず，表 1 に提案手法がサポートするデータ構造を言語ごとに示す．Python 側の配列には，一次元配列のみをサポートしている組み込みの array モジュール [2] ではなく，より広く使われている NumPy モジュールの ndarray を採用した．次にリストだが，シリアライズ/デシリアライズが可能なのは Python のリストだけである．Julia 側の提案手法がリストに対応していない理由は， Python と Julia でリストの取り扱いに差があるためである，Python のリストは要素間のポインタによって繋がれる構造ではなく，内部では各要素へのポインタの動的配列として実装されているのに対し， Julia ではそもそも組み込み型としてリストが提供されていない．Julia からリストを使う場合は，例えば DataStructures.jl [4] などの外部モジュールで定義されている単/双方向連結リストを用いる必要がある．Julia からリストを用いることも可能だが，このようにデータ構造の名称は同じでも内部表現が大きく異なる．さらに，実験で提案手法の比較対象と

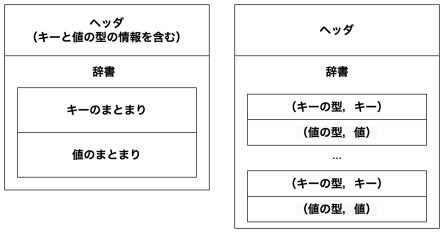



図 4: （左）キーと値の型が統一されているため，コンパクトにシリアライズできている辞書のフォーマット
（右）キーと値の型が任意であるため，キーと値の型をエントリごとに示す必要があるフォーマット

して用いた Pickle が外部モジュールで定義された Julia のリストをシリアライズすることができない．このような経緯から，提案手法ではプログラミング言語の組み込み型として実装されている Python のリストにのみ対応することになった．辞書は両言語の組み込み型で実装されているものに対応する．次にデータ型については， Python, Julia 共に「64 bit 符号付き整数」・「倍精度浮動小数点数」・「文字列」の 3 種類に対応する．さらに，「配列」・「リスト」・「辞書のキー」・「辞書の値」に含まれる型はそれぞれ統一されている必要がある．

データ構造に含まれるデータ型を統一する理由は，変換の高速化のためである．例えば辞書をシリアライズするとき，図 4 に示すように，型を統一した辞書をシリアライズするときはキーと値の型の情報を一回ヘッダに書き込めば良いのに対して，型を統一していない場合は各アイテムについて型の情報を書き込む必要がある．このため，提案手法は型を統一したデータ構造にのみ対応することで，変換処理の高速化を図る．

これ以降の節では，各データ構造・データ型に対するシリアライズ/デシリアライズの実装を説明する．

### 4.2 シリアライズ

どのデータ構造・データ型にも共通するシリアライズ全体の手順は以下の通りである．

(1) 入力に「シリアライズするオブジェクト」と「デシリアライズする言語」を取る．
(2) 入力からヘッダを作成する
(3) シリアライズに必要なサイズを持つメモリを確保する
(4) 確保した領域にシリアライズする

手順 3, 4 におけるヘッダ部以外（図 3 の data 部）の処理は，データ構造・データ型・デシリアライズする言語ごとに決められたフォーマットに従って行われる必要がある．各データ構造・データ型のフォーマット及びシリアライズの方法は，4.2.1 節と 4.2.2 節で説明する．

#### 4.2.1 各データ型のシリアライズ

**64 bit 符号付き整数, 倍精度浮動小数点数**　64 bit 符号付き整数と倍精度浮動小数点数はサイズが固定長であるため，メモリ上の表現をそのまま書き込む．なお，Python の整数型オブジェクトは多倍長整数をサポートするオブジェクトとして実装されているため，一度 C 言語の int64_t 型の変数に変換する必要がある．

**文字列**　Python, Julia で採用しているエンコーディングが異なっていたり，Python の中でも NumPy の ndarray と他のオブジェクトとで異なるエンコーディングになっている．この違いに対応するためにデシリアライズする言語やシリアイズするデータ構造に応じて異なるフォーマットでシリアライズをする必要がある．

Python は 組み込み型の文字列オブジェクトが消費するメモリを必要最小限に抑えるために， ASCII, Latin 1, UCS-2, UCS-4 の 4 種類のエンコーディングを文字列ごとに使い分けている．どのエンコーディングも 1 文字が占めるメモリサイズが固定長であり， ASCII と Latin 1 は 1 byte , UCS-2 は 2 bytes , UCS-4 は 4 bytes である．

よって， Python の文字列オブジェクトをシリアライズする際には，「エンコーディングの種類」・「文字列の長さ」・「文字列の実体」を表現する必要がある．

一方 NumPy の配列クラス ndarray は全ての文字列が UCS-4 でエンコーディングされており，さらに配列の各要素が占めるメモリサイズを「最長の要素が占めるメモリサイズ」に統一している．これは配列の各要素へのランダムアクセスの実装を容易にするための仕様である．例えば，配列 a = numpy.array([”he”, ”llo”, ”w”, ”orld”]) の最長の要素は “orld” で， 16 bytes を占める．よって他の要素にも 16 bytes が割り当てられることになる．

全ての要素が同じメモリサイズを占めていることから，フォーマットには「1 つの要素が占めるメモリサイズ」と「配列」を書き出す必要がある．

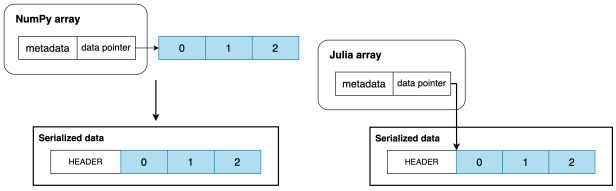
Julia ではどのデータ構造においても文字列のエンコーディングに UTF-8 が用いられている．そのため，フォーマットには「文字列の長さ」と「文字列」を含める必要がある．

さらに， Julia の文字列の配列は文字列へのポインタの配列で実装されている．よって NumPy の場合とは異なり，文字列の配列をシリアライズする際は要素ごとにシリアライズしなければならない．

#### 4.2.2 各データ構造のシリアライズ

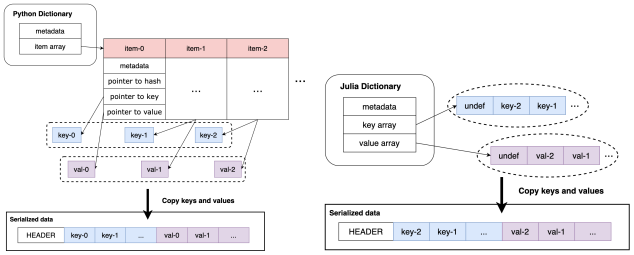
**配列**　配列をシリアライズする様子を図 5a に示す．要素の型が 64 bit 符号付き整数か倍精度浮動小数点数の場合，そのまま配列全体を図 3 の data 部に書き込む．要素の型が文字列の場合，先述したようにデシリアライズする言語に合わせたフォーマットでシリアライズする必要がある．

**リスト**　配列と同じフォーマットを用いて，リストの各




(a) シリアライズ　　(b) デシリアライズ

図 5: 提案手法を用いた配列 [0,1,2] のシリアライズとデシリアライズ




(a) Python　　(b) Julia

図 6: 提案手法を用いた辞書のシリアライズ

表 2: 提案手法がデータコピーせずにデシリアライズできるデータ構造
（o はデータコピー無し， x は有り， - は提案手法が対応していないデータ構造を表す）

| | 配列 | リスト | 辞書 |
|---|---|---|---|
| Python | o | x | x |
| Julia | o (文字列配列はコピー有り) | - | x |

要素を先頭から順に書き込む．

**辞書**　Python の辞書は基本的に各アイテムの「キーのハッシュ値」・「キー」・「値」を一つの構造体でまとめて管理しているのに対し，Julia の辞書はキーと値を別々の配列でまとめて管理している．

辞書のフォーマットは，Julia の辞書のようにキーと値を別々のまとまりにした後，それらをキー，値の順に書き出したものとする．注意すべき点として，空のスロットをシリアライズしないことが挙げられる．図 6 に提案手法を用いて Python と Julia の辞書をシリアライズする様子を示す．

### 4.3　デシリアライズ

どのデータ構造・データ型にも共通するデシリアライズの全体的な手順は以下の通りである．

(1) 入力に「シリアライズされたデータへのパス」を取る．（ディスク上にデータがある場合）
(2) 入力のパス上にあるデータをメモリマップ（プライベートマッピング）する
(3) メモリマップした領域を使ってデシリアライズする

メモリマッピングしたデータを用いて，各データ構造をデシリアライズする方法を 4.3.1 節で説明する．また，提案手法を用いて各データ構造をデシリアライズする際，データ構造によってメモリコピーを起こす場合と起こさない場合が存在する．そこで，表 2 に各データ構造についてデータコピーが起きるかどうかを示す．各データ型をデシリアライズする方法は，シリアライズの方法から明らかであるため省略する．

#### 4.3.1　各データ構造のデシリアライズ

**配列**　要素の型が 64 bit 符号付き整数か倍精度浮動小数点数の配列をシリアライズする場合，ヘッダの直後に配列本体がそのまま書き込まれる．このことを利用して，メモリマップしたデータうち，ヘッダの直後のアドレスを参照するポインタを取得すると，定数時間かつ高速に配列をデシリアライズすることができる．図 5b に配列をデシリアライズした結果， numpy.array([0,1,2]) が得られた様子を示す．

配列の要素が文字列だった場合は，エンコーディングに従って文字列をデシリアライズする必要がある．NumPy ndarray の場合は，各要素が占めるメモリサイズを示すフィールドの直後に配列本体が書き込まれている．よって，ヘッダの直後のアドレスを参照するポインタを取得することで，配列を定数時間かつ高速にデシリアライズできる．Julia の文字列の配列はポインタの配列であるため，1 文字列ずつデシリアライズして中身にゴミが入っている配列に代入する．

**リスト**　リストのフォーマットは配列と同じである．よって，リストの要素を一つ一つデシリアライズして，空のリストに代入すれば良い．

**辞書**　シリアライズされた辞書はキーと値がそれぞれ分けてあるため，デシリアライズの際はキーと値をリストないし配列でまとめ，空の辞書にエントリを追加するように実装した．

#### 4.3.2　デシリアライズ時のデータコピー

Python の配列もしくは Julia の 64 bit 符号付き整数・倍精度浮動小数点数の配列をデシリアライズする時，メモリマップした領域上のアドレスを指すポインタをそのまま配列の先頭ポインタとして活用している（図 5b 参照）．これによって，メモリマップ以外のデータコピーを無くし，高速かつ定数時間のデシリアライズが実現できる．

一方，配列以外のデータ構造をデシリアライズする時は，メモリマップした領域上のデータを再びコピーしてオブジェクトを生成する必要がある．これは，配列以外のデータ構造が，配列のように外部にあるデータ領域をポインタで参照する設計ではなく，データ構造の内部に値を持つ設計のデータ構造だからである．

しかし Julia の辞書はキーと値を配列で管理しており，一見メモリコピーが必要なさそうである．それでもコピー

表 3: 実験で用いたデータ構造に含まれるデータ型

| データ構造 | データ型 |
|---|---|
| 配列 | 倍精度浮動小数点数 |
| リスト | 倍精度浮動小数点数 |
| 辞書 | （キー，値） = （文字列，倍精度浮動小数点数） |

が必要な理由は，Julia の辞書のエントリの順番が Julia 内部で得られるハッシュ値に依存しており，メモリマップされた領域から作られるキーをまとめた配列と同じ順番になるとは限らないためである．また，空のエントリをシリアライズしないことも原因に挙げられる．

## 5. 評価

### 5.1 実験設定

実験として，相手の環境に特化した提案手法と汎用性の高い既存の手法の，インメモリでオブジェクトをシリアライズ・デシリアライズする際にかかる時間を計測した．具体的にはシリアライズ/デシリアライズをそれぞれ 1000 回繰り返し，各イテレーションにかかった時間の平均を算出し，両者の平均時間を比較した．この比較によって，汎用性の高い手法に比べて，既知の相手の環境に特化することでフォーマットが簡略化された提案手法がどれだけ高速になったかを確認する．また，なるべく純粋なシリアライズ/デシリアライズのオーバーヘッドを計測するために I/O（open, mmap）を計測対象から省き，計測中は GC を無効化した．

データ共有に用いる言語は Python と Julia の 2 種類であるため，シリアライズする側とデシリアライズする側の組み合わせは「Python から Julia へのデータ共有」,「Julia から Python へのデータ共有」の合計 2 通りである．スペースの都合上,「Python から Python へのデータ共有」,「Julia から Julia へのデータ共有」の結果は省略する．また 4.1 節で述べた理由から，Python と Julia 間のデータ共有においてリストの処理は行っていない．

実験に用いたそれぞれのデータ構造のデータ型を表 3 に示す．各データ構造ともに要素数が $4 \times 10^6$ のものを用いた．各データ構造の消費するメモリサイズは，5.3.1 と 5.4 に示す．

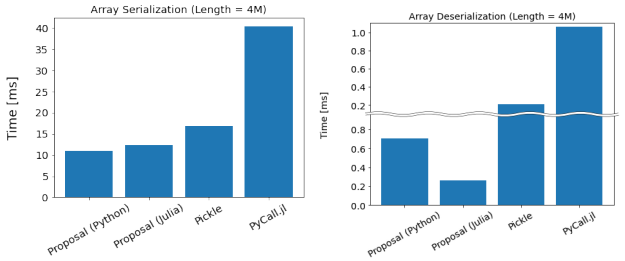
比較対象には 2.2 節で言及した Pickle を採用した．本来 Pickle は Python 固有のモジュールだが， PyCall.jl という Julia のモジュールを使うと Julia から Pickle を含む Python のプログラムを実行することができる．これにより， Pickle は「Python と Julia のオブジェクトならば言語によらず処理することができる」という環境に対する中立性を獲得することができる．

PyCall.jl [12] は Julia から Python のプログラムを実行することを可能にするモジュールである．このモジュールは Julia と Python 間の一部のデータ構造・データ型の変換をサポートしており，その中には配列，辞書， 64 bit 符号付き整数，倍精度浮動小数点数，文字列が含まれている．ここでいう変換とはシリアライズ/デシリアライズを用いたデータ共有を指すのではなく， Julia のアドレス空間内に，元となる Julia のデータ構造から Python のオブジェクトを生成することを意味している．

計測結果にはメモリ領域を確保する時間まで含まれている．これは， Pickle の計測において，関数内部でメモリの確保を行なっており分離ができなかったためである．

表 4: 実験環境

| | |
|---|---|
| OS | Ubuntu 20.04 |
| CPU | Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz |
| DRAM | 188 GB |
| Python | ver. 3.9.7 |
| Julia | ver. 1.6.2 |
| NumPy | ver. 1.21.4 |
| PyCall.jl | ver.1.92.5 |




(a) シリアライズ (b) デシリアライズ

図 7: 配列のシリアライズとデシリアライズにかかる平均時間
横軸は左から順に提案手法 (Python), 提案手法 (Julia), Pickle, PyCall.jl

### 5.2 実験環境

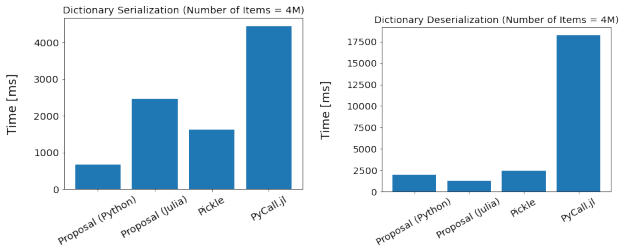
実験に使用したマシンの性能，プログラミング言語・ OS のバージョンを表 4 に示す．

### 5.3 Python， Julia 間のデータ共有

Python， Julia の配列と辞書をシリアライズ/デシリアライズするのにかかる平均時間をそれぞれ図 7，図 8 に示す．

#### 5.3.1 Python から Julia へのデータ共有

図 7 における，シリアライズ側が Python , デシリアライズ側が Julia の時の結果に注目する．配列と辞書のサイズは，それぞれおよそ 32 MB と 167 MB である．配列のシリアライズは 1.57 倍高速で，デシリアライズは $6.20 \times 10^5$ 倍高速であった．辞書のシリアライズは 2.40 倍高速で，デシリアライズは 14.4 倍高速であった．




(a) シリアライズ　　(b) デシリアライズ

図 8: 辞書のシリアライズとデシリアライズにかかる平均時間
横軸は左から順に提案手法 (Python), 提案手法 (Julia), Pickle, PyCall.jl

### 5.4 Julia から Python へのデータ共有

シリアライズ側が Julia ，デシリアライズ側が Python の時の結果に注目する．配列と辞書のサイズは，それぞれおよそ 32 MB と 209 MB である．配列のシリアライズは 3.26 倍高速で，デシリアライズは $3.03 \times 10^4$ 倍高速であった．辞書のシリアライズは 1.80 倍高速で，デシリアライズは 1.23 倍高速であった．以上の結果から，提案手法が汎用性の高い既存の手法に比べて高速に処理できていることが確認できた．

### 5.5 高速化の要因

配列のシリアライズが高速化できた原因は， Pickle に比べてフォーマットが簡潔であることが挙げられる．Pickle はフォーマット中にオブジェクトを再構築するために必要な情報を全て書き込むことで，ほとんど任意のオブジェクトをデシリアライズすることを可能にしている．基本的には，デシリアライズするオブジェクトを初期化するために呼ぶべき関数と，シリアライズしたオブジェクトが保持していた内部変数（初期化用の関数への引数として用いることが多い）が書き込まれる．よって，フォーマットに NumPy の配列を初期化するのに必要なメタデータを全て書き込む必要がある．これに対し提案手法は NumPy ndarray であることを示す必要はないため， Pickle のようなメタデータを書き出す処理を省略できる．このため，Picke に比べて比較的高速な処理が実現できる．

辞書のシリアライズ・デシリアライズが高速化できた原因は，図 4 にもあるようにフォーマットを簡略化したことによるものである．提案手法は，辞書のキーと値の型が統一していることを前提とするため，型の情報はヘッダ部に一度記せば良い．一方 Pickle は，任意の辞書をシリアライズできるようにするため，各アイテムについてキーと値の型の情報を書き込む必要がある．このように提案手法は型の情報を毎回取得する処理を省くことができるため，Pickle に比べて高速な処理が可能となる．

## 6. 関連研究

### 6.1 相手の環境が未知であるデータ共有

Network objects [16] は，オブジェクト指向プログラミング言語（OOP） Modula-3 を用いて設計や実装されたシンプルな分散システムを提案している．分散システムでは，シリアライズしたオブジェクトがデシリアライズされる環境を前もって知ることはできない．そのため Network objects が実装した手法では，シリアライズ側のインメモリなデータの表現をそのままデシリアライズ側に送信し，デシリアライズ側は受け取ったデータを自身にとって適切なインメモリの表現に置き換えるという仕組みになっている [17].

本研究との違いはシリアライズとデシリアライズが実行されるマシンが必ずしも一致していない点である．本研究は前提として相手の環境が既知であることを要求しているため，そこに特化したフォーマットを用いることができる．

### 6.2 インメモリの構造のままデータを永続化する手法を取る研究

2.2 節で触れたように， Persistent Memory はバイトアクセス可能なデバイスであるため，インメモリのデータ構造をそのままの形で永続化することが可能である．Twizzler [18] はポインタの永続化によってファイルの代わりにオブジェクトをそのまま永続化する仕組みを実装している．PyMM [20] は Persistent Memory に対応したメモリ管理機構を用いることで， Persistent Memory 上にオブジェクトを割り当てている．J-NVM [19] は Persistent Memory 上にオブジェクトを割り当てるが， JVM のヒープとは分けて管理することで実行時の Garbage Collection のオーバーヘッドを削減している．今村ら [21] は R と Python のバックエンドに Persistent Memory の管理機能を実装することで複雑な Persistent Memory を意識したプログラミングをプログラマから隠蔽する API を実装し，その API を SSD を用いた従来手法と比較したところ，オブジェクトの永続化および復元処理を最大 1/8 に短縮した．

これらの研究はインメモリなデータ構造を Persistent Memory にそのまま永続化するという手法をとっており，シリアライズ・デシリアライズのオーバーヘッドが無くなる分，提案手法に比べて Persistent Memory のレイテンシの低さをより発揮できていると考えられる．しかし本研究のように異言語間のデータ通信を行う場合，データのシリアライズ・デシリアライズは必要不可欠である．というのも，理論上は同じデータ構造であってもプログラミングごとに内部構造が異なるのは自然なことだからである．例えば本研究で取り扱った Python と Julia において，文字列オブジェクトの構造は全くの別物であった．Python と

Julia で文字列データを共有するならば，シリアライズ・デシリアライズが必要なのは明白である．

## 7. まとめ

プログラミング言語間のデータ通信は多くの場面で行われている．その際，オブジェクトをプログラミング言語やマシンアーキテクチャに依存しない中立的なバイト列に変換するシリアライズとその逆変換であるデシリアライズという処理が用いられる．中立的なフォーマットが使われるのは，デシリアライズ側の環境が未知な場合でも相手が正しくデシリアライズできるためである．また，中立的なフォーマットの中でも，処理できるデータ構造・データ型に関して汎用性を持つものが広く用いられる．しかし，デシリアライズ側の環境が既知である場合は少なくない．そのような場合，相手の環境に特化したフォーマットを用いることで，そのようなデータ共有を高速化するできると考えられる．

本研究では，既知の相手の環境に特化した Python と Julia 間のシリアライズとデシリアライズを実装した．提案手法と汎用性の高いフォーマットである Pickle を使って，インメモリのシリアライズとデシリアライズにかかった時間を測定し，両者の処理のオーバーヘッドを比較した．実験の結果，提案手法が Pickle より高速にシリアライズ/デシリアライズを行うことを確認した．

### 参考文献

[1] : Apache Flink, Apache (online), available from ⟨https://flink.apache.org/⟩ (accessed 2022-01-05).
[2] : array — Efficient arrays of numeric values, Python Software Foundation（オンライン），入手先 ⟨https://docs.python.org/3/library/array.html⟩ （参照 2022-04-28）.
[3] : CPython, Python Software Foundation (online), available from ⟨https://github.com/python/cpython⟩ (accessed 2022-04-18).
[4] : DataStructures.jl, julialang.org (online), available from ⟨https://juliacollections.github.io/DataStructures.jl/latest/⟩ (accessed 2022-04-28).
[5] : Intel® Optane™ Persistent Memory, (online), available from ⟨https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html⟩ (accessed 2022-04-18).
[6] : Julia 1.7 Documentation, julialang.org (online), available from ⟨https://docs.julialang.org/en/v1/⟩ (accessed 2022-04-28).
[7] : Julia Micro-Benchmarks, julialang.org (online), available from ⟨https://julialang.org/benchmarks/⟩ (accessed 2022-04-28).
[8] : The Julia Programming Language, julialang.org (online), available from ⟨https://julialang.org/⟩ (accessed 2022-04-26).
[9] : JuliaHub, julialang.org (online), available from ⟨https://juliahub.com/ui/Packages⟩ (accessed 2022-04-28).
[10] : NumPy, NumPy (online), available from ⟨https://numpy.org/⟩ (accessed 2022-04-26).
[11] : Pickle, Python Software Foundation (online), available from ⟨https://docs.python.org/3/library/pickle.html⟩ (accessed 2022-04-26).
[12] : PyCall.jl, JuliaPy (online), available from ⟨https://github.com/JuliaPy/PyCall.jl⟩ (accessed 2022-01-17).
[13] : PyPI - The Python Package Index, Python Software Foundation (online), available from ⟨https://pypi.org/⟩ (accessed 2022-04-28).
[14] : Python/C API Reference Manual, Python Software Foundation (online), available from ⟨https://docs.python.org/3/c-api/index.html⟩ (accessed 2022-04-26).
[15] : PyTorch, pytorch.org (online), available from ⟨https://pytorch.org/⟩ (accessed 2022-04-28).
[16] Birrell, A., Nelson, G., Owicki, S. and Wobber, E.: Network objects, *Software: Practice and Experience*, Vol. 25, No. S4, pp. 87–130 (online), DOI: https://doi.org/10.1002/spe.4380251305 (1995).
[17] Birrell, A., Nelson, G., Owicki, S. and Wobber, E.: Network Objects (1995).
[18] Bittman, D., Alvaro, P., Mehra, P., Long, D. D. E. and Miller, E. L.: Twizzler: a Data-Centric OS for Non-Volatile Memory, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, pp. 65–80 (online), available from ⟨`https://www.usenix.org/conference/atc20/presentation/bittman`⟩ (2020).
[19] Lefort, A., Pipereau, Y., Amponsem, K., Sutra, P. and Thomas, G.: J-NVM: Off-Heap Persistent Objects in Java, *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, Association for Computing Machinery, p. 408–423 (online), DOI: 10.1145/3477132.3483579 (2021).
[20] Waddington, D., Hershcovitch, M. and Dickey, C.: *PyMM: Heterogeneous Memory Programming for Python Data Science*, p. 31–37 (online), available from ⟨`https://doi.org/10.1145/3477113.3487266`⟩, Association for Computing Machinery (2021).
[21] 智史今村，英司吉田：Persistent Memory を用いたスクリプト言語のデータ永続化および復元処理の実装と評価，コンピュータシステム・シンポジウム論文集，Vol. 2020, pp. 72–77 (2020).