Julia 言語を用いた高性能並列実行環境の構築

中田 秀基[1,a)]

概要：Julia 言語は科学技術計算向けに設計されたインタラクティブ言語として、機械学習や統計の分野を中心に広く注目を集めている。Julia はネイティブで分散計算をサポートしており、Remote Procedure Call に類した通信が可能である。また、Julia は MPI をサポートしており、高性能計算環境の高速なノード間通信を利用できる。さらにわれわれはすでに、Julia の Remote Procedure Call とリモート呼び出しを用いて、Julia に Actor を導入する手法について発表している。しかし、これらの分散実行機構の高性能計算環境での評価は行われていない。本発表では、高性能計算環境での Julia による並列実行環境の構築を目的とし、Julia の Remote Procedure Call および提案 Actor 機構の、高性能計算環境での性能評価を行った。その結果大幅な性能低下が見られたためこの改善を試み、一定の性能向上を達成した。また、Actor 機構の記述方法を見直し、Julia の通常関数と親和性の高い記法を導入した。

キーワード：Julia, Actor, 分散実行

High-performance parallel environment in Julia

Hidemoto Nakada[1,a)]

Abstract: Julia language is one of the most promising programming languages for the next generation scientific programming language because of its fast execution thanks to LLVM based JIT compilation. Julia supports distributed computing with remote function invocation, remote channel communication, and Future based synchronization, out of the shelf. It also supports MPI communication which is required to achive fast communication on HPC environments. We have already presented Actor based communication library, but it is not evaluated on the HPC environments. In this presentation we show the evaluation results on ABCI. Furthermore we discuss on the parallel language constract which is suitable Julia language.

Keywords: Julia, Actor, Distributed Computing

1. はじめに

Julia 言語は科学技術計算向けに設計されたインタラクティブ言語として広く注目を集めている。Julia はネイティブで分散計算をサポートしており、Remote Procedure Call(RPC) とリモートチャンネルを用いたマルチプロセス計算が可能となっている。また、Julia は MPI をサポートしており、高性能計算環境の高速なノード間通信を利用できる。われわれは、Julia の RPC ではリモートノードに状態を維持することが難しいことが問題であると考え、リモートチャンネル機構を用いた Actor 機構を提案した [1]。しかし、RPC 機構や Actor 機構の高性能計算環境での評価は行っていなかった。

本稿では、提案 Actor 機構の高性能計算環境での評価を行い、性能の改善を試みる。さらに、Actor の記述方法を再考し、記述量が少なく、Julia の文法と整合性の高い記法を導入する。

本稿の構成は以下のとおりである。2 節では、Julia 言語と Distributed.jl その MPI 実装、先行研究での Actor 実装についてのべ、文献 [1] で提案した Actor 実装について説明する。3 節では、今回行った高速化手法について説明する。4 節で評価結果を示す。5 節で Actor の記法について

[1] 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
[a)] hide-nakada@aist.go.jp

議論する。6 節はまとめである。

2. 背景

2.1 Julia 言語

Julia[2] は 2012 年に登場した比較的新しい言語である。いわゆるスクリプト言語的な使い勝手をもちながら、ユーザによる部分的な型付けと、LLVM[3] をバックエンドとして用いる強力な JIT コンパイラを用いることで、C 言語などの静的型付け言語に迫る実行効率を示す。JIT コンパイル時に、実行時に決定された変数型に最適化されたコードを出力する点に特徴がある。

Julia は、配列インデックスが 1 オリジンである、多次元配列の最内周インデックスが左側に来る、などの特徴を持ち、Fortran や R などと同様に数学的な記法との親和性が高く、統計計算の分野で広く用いられつつある。また、C 言語による拡張を用いずに複雑な数値演算を十分高速に実装できることから、機械学習の分野での利用が期待されている。

Julia は広く用いられるクラスとインスタンスメソッドに基づくオブジェクト指向ではなく、構造体とジェネリック関数に基づく CLOS[4] に類似したオブジェクト指向をサポートする。関数呼び出し時に、呼び出し引数列の型に対してもっとも「近い」シグネチャを持つ関数定義 (Julia では個々の定義をメソッドと呼ぶ) を選択し、実行する。ある関数呼び出しに対して、どのメソッドがより「近い」かが明確でない場合には、呼び出しはエラーとなる。この点では、厳密な関数選択ルールを定義することで、必ずいずれかの関数を選択する CLOS とはアプローチが異なる。

2.1.1 コルーチンとチャンネル

Julia はコルーチンを Task という名前でサポートしている。Julia のコルーチンは、文献 [5] の分類に従えば、対称で、一級オブジェクトで、スタックフルである。通常のスレッドに非常に近く、I/O や後述するチャンネルに対する操作の際に自動的にスケジューラに制御が戻るように設計されている。I/O の実装には libuv[6] が使用されている。

Julia にはもともと OS スレッドを利用する機能はなくコルーチンだけが用意されており、後になってスレッドが導入された経緯がある。このため、バージョン 1.8 の時点では、コルーチンはそれを作成した特定のスレッドに紐付けられており、スレッド間で負荷分散されることはない。

また、Julia にはコルーチン間の通信のためにチャンネルも用意されている。一方で、排他制御の機構は比較的貧弱で、基本的な排他ロックや条件変数、アトミック変数は用意されているものの、Java の `synchronized` のような言語的なサポートはない。

2.1.2 Julia 言語の分散並列実行機能

Julia の分散並列機構は Distributed.jl[7] と呼ばれるパッケージで実現されている。Julia のマルチプロセス実行は、

```
1 future = @spawnat 2 f(X)
2 value = fetch(future)
```

図 1 Distributed.jl を用いたコード例

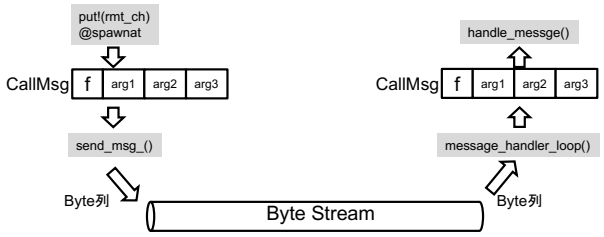

図 2 Distributed.jl の実装

一つのマスタプロセスと複数のワーカプロセスで構成される。Distributed.jl における分散並列プログラミングは、ワーカプロセス を指定して関数を呼び出すことで行われれる。プログラムの例を図 1 に示す。`@spawnat` は Julia のマクロで、直後の引数にワーカ ID を指定し、その後ろの式を指定したワーカで実行する。図 1 の 1 行目は、ワーカ 2 で `f(X)` を実行するという意味になる。この際に `X` が自動的にワーカ 2 へ転送される。`@spawnat` の返り値は Future 型のオブジェクトであり、これを用いて非同期な関数呼び出しが実現される。返り値の Future 型オブジェクトに対して `fetch` を行うことで、関数呼び出しに対する同期が行われる。すなわち、計算が終了して結果の値が得られていれば、`fetch` は即座に値を返す。まだ値が得られていなければ、`fetch` は計算が終了するまでブロックする。

分散ノードにまたがった実行のために RemoteChannel 機構が用意されている。RemoteChannel の実態は、リモートノード上に作成されたチャンネルであり、これに対してネットワーク越しに値をプッシュする。したがって、参照さえ取得していれば任意のノードから書き込むことが可能である。

2.1.3 分散並列実行の実装

図 2 に Distributed.jl の実装の概要を示す。図左側が呼び出し側、右側が呼び出される側である。リモート関数の呼び出し時には、呼び出しを表現した構造体 `CallMsg` が作成される。この構造には関数名と引数が格納される。RemoteChannel への書き込みもリモート関数呼び出しで実現されているので、同様に `CallMsg` で表現される。次に、`send_msg_`でこの構造体をバイト列にシリアライズし、バイト列ストリームに書き込む。通常の分散環境ではこのバイト列ストリームとしてソケットが用いられる。受け取り側では、`message_handler_loop` が非同期にこのストリームを監視しており、データを読み出してデシリアライズして構造体を再構成し、これを引数として `handle_message` を呼び出し、関数の実行を行う。

2.1.4 MPI.jl

MPI.jl[8] は Julia で MPI を利用するためのラッパライ

```
// 言語のC MPI_Send
int MPI_Send(const void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

# Julia の MPI.Send
MPI.Send(buf, comm::Comm; dest::Integer,
         tag::Integer=0)
```

図 3 C と Julia の MPI

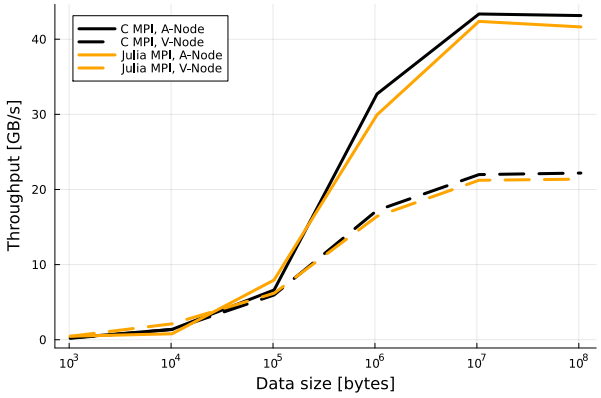

図 4 ABCI 上での MPI のネイティブ性能

ブラリである。提供する API は C や Fortran の MPI とほぼ同じであるが、Julia では配列のデータ型やサイズが取得できるため、明示的に指定する必要はない。図 3 に、それぞれの送信関数のシグネチャを示す。

図 4 に MPI による相互通信速度を示す。実験環境としては 4.1 節で説明する ABCI の A-node, V-node を用いた。横軸がデータサイズ、縦軸がスループットである。横軸はログスケールである。いずれの場合も十分大きいデータに対しては、理論性能にかなり近い性能が出ていることがわかる。

2.1.5 MPIClusterManagers.jl

MPIClusterManagers.jl[9] は、MPI を通信レイヤとして Distributed.jl の機能を使用することを可能にする。MPIClusterManagers.jl は、いくつかの使用モードを提供しているが、本稿ではマスタとすべてのワーカを MPI で管理するモードを使用する。このモードでは、mpirun を用いて Julia プロセスを各ノードで実行し、rank 0 がマスタとなり、rank 1 以降がワーカとして動作する。

図 5 に MPIClusterManagers における RPC の様子を示す。MPIClusterManagers は、Distributed.jl をそのまま用い、最下部のバイトストリームを MPI で置換している。送信側では、シリアライズされたデータは、メモリ上のバイトストリームに書き込まれる。このバイトストリームを監視するコルーチンが、データをバイト列として読み出して、これを `MPI_Isend` を用いて送信する。受信側では、`MPI_Irecv` を用いてこのバイト列を読み出して、受信用のバイトストリームに書き込む。

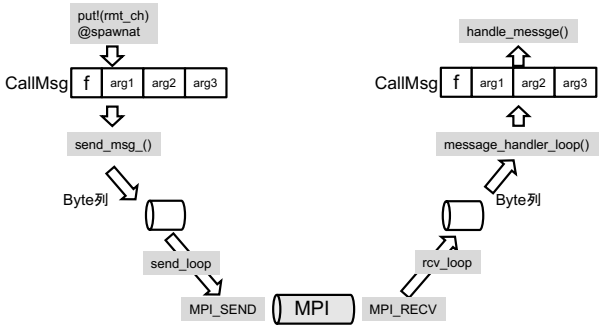

図 5 MPIClusterManagers.jl の実装

2.1.6 Julia での既存パッケージの改変

Julia 言語は十分に高速であるため多くのパッケージの大部分が Julia 言語そのもので記述されている。これは Python のパッケージの多くが C/C++で書かれたライブラリに依存したものとなっているのと対象的である。また、Julia 言語では既存パッケージの一部を非常に容易に改変することができる。具体的にはソースコードでパッケージ関数をインポートした上で同名の関数を定義するだけで、その関数を置き換えることができる。

本稿では、Julia 標準の Distributed.jl モジュールの関数をこの方法で置き換えることで MPI 環境下での最適化を行った。

2.2 Actor

Actor は、1970 年代に人工知能の計算モデルとして Hewitt らによって提唱され [10]、その後分散並列オブジェクトモデルとして Agha によって定式化された計算モデル [11] である。1980 年代から 90 年代にかけて、ABCL[12], [13] など多くの言語で基本的な並列計算言語機構として採用された。現在でも Erlang[14]/Elixir[15] や Akka[16] などで、並行実行の単位として用いられる。

通常のスレッドモデルでは、データを抽象化した構造体やオブジェクトなどのデータ構造と、並列実行の主体であるスレッドが直交している。すなわち、複数のスレッドがデータ構造の内部データに対して同時にアクセスすることが可能である。これに対して Actor モデルではデータ構造と実行主体が一体として扱われるのが特徴である。Actor モデルは外部からの処理命令 (メッセージ) を 1 つづつ受け取り、1 つづつ処理して、内部状態を更新する。したがって、内部状態を更新する際に排他制御を明示的に行う必要がない。

Actor 間の通信は一般に非同期のメッセージパッシングで行われる。つまり、メッセージは一方通行で、送信者はメッセージに対する返答を待たない。各 Actor はメッセージキューを持ち、そこにメッセージが到着順にキューイングされる。Actor は、メッセージキューから順にメッセージを取り出し、1 つづつ処理する。メッセージを受信した

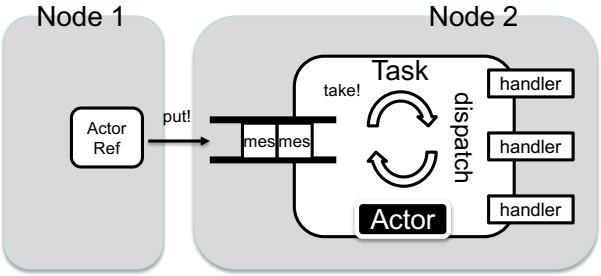

図 6　Actor の実装

Actor は、メッセージに応じて、状態の更新、他 Actor の生成、他 Actor へのメッセージ送信、を行い、それが完了したら次のメッセージの受信を試みる。もし次のメッセージが到着していなければ、そこでブロックする。

2.3　Julia における Actor 実装

われわれは、[1] において、Julia 言語への Actor の導入を提案している。本機構の概要を図 6 にしめす。本機構はリモートチャンネルへの参照を Actor への参照として用い、メッセージパッシングをリモートチャンネルへのメッセージ構造体の書き込みとして表現する。Actor はリモートチャンネルに対して非同期に読み出しを行うコルーチンと、内部状態を表す構造体として実現される。このコルーチンは、リモートチャンネルからメッセージ構造体を読みだし、内部状態を表す構造体とメッセージ構造体とをメッセージハンドラに渡すことで処理を行う。Actor からの返り値は、Julia の通常の RPC と同様に Future として返される。

3.　高速化手法の実装

分散強化学習などのアプリケーションでは、環境の状態を交換する事が必要となる。環境は一般に数値の配列として表現されるため、数値配列を高速に転送することが必要となる。本稿では配列の転送スループットに着目して測定と改良を行う。

3.1　MPIClusterManagers のスループット

MPIClusterManagers を用いた RPC と Actor のスループットを、ABCI 上で測定した。結果を図 7 に示す。いずれのケースも 1GB/s に満たず、十分な性能が得られていない。これは MPIClusterManagers の実装ではすべてのデータ通信が、図 5 の最下部でのバイトストリームへの書き込み/読み出しで実現されるため、この際のコピーのコストが大きいためであると考えられる。

3.2　高速化手法の概要

性能低下の原因は、メッセージ構造体をシリアライズした上でバイトストリームにコピーしていることであると考

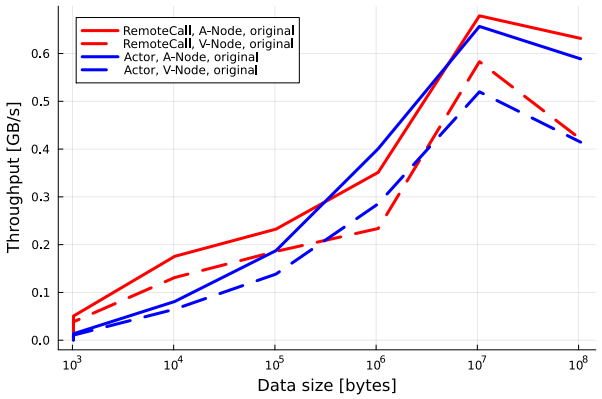

図 7　スループット

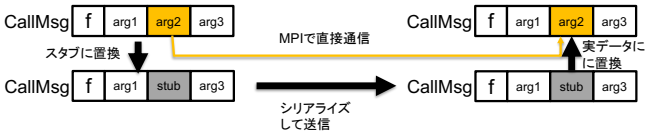

図 8　高速化手法

えられる。引数に現れるデータ配列は、メッセージ構造体の一部としてシリアライスされ、コピーされ、バイト列として MPI 転送される。しかしデータ配列は MPI が直接サポートしているので、これだけ別経路で転送することができれば、シリアライズやバイトストリームへのコピーのコストを回避することができる。

この発想に基づき高速化手法を考案した。まず、メッセージ構造体の引数列のなかからデータ配列を抜き出し、スタブとなる構造体と置き換え、それを転送する。データ配列は独立した MPI 通信で別途転送する。受信側では、メッセージ構造体を受領後、処理を行う前にメッセージ構造体内のスタブを別経路で受信したデータ配列で置き換える。この様子を図 8 に示す。

3.3　スタブ構造体

スタブ構造体の定義を図 10 に示す。この構造体には、データの ID と送信元の rank が格納されている。データの ID は受信後にデータを識別するために用いられる。8 ビットしか無いのは、後述するデータヘッダ内で使用したビット数が 8 ビットであるためである。この ID はローテーションして再利用されるため実用上問題はない。

3.4　データ配列の転送

データ配列を送信する際には、データ配列の型とサイズについて事前に合意しておく必要がある。またデータの ID も同時に共有することが望ましい。これらの情報を 64 ビットの符号なし整数にエンコードし、データ配列本体に先立ってヘッダとして送信する。64 ビットの内訳は以下の通りである。

- データ・タイプ:3 ビット

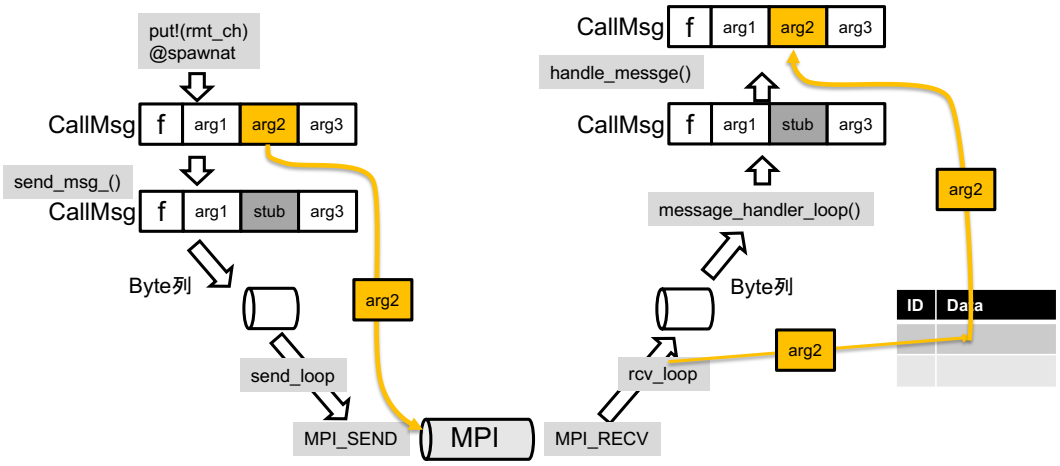

図 9 改良版の実装

```
struct DataStub
  id::UInt8
  rank::UInt32
end
```

図 10 スタブ構造体

- ID:8 ビット
- 配列サイズ:53 ビット

3.5 データ配列の受信

データ配列の受信は MPIClusterManagers の受信ループで行う。MPIClusterManagers ではすべての通信をタグ 0 で行っているので、それ以外のタグをつかうことで、通信を区別することができる。ヘッダ転送にはタグ 1 を、本体のデータ配列転送にはタグ 2 を用いた。

タグ 1 のデータを受信したら、ヘッダであると判断し、ヘッダを解析し、その情報を用いて後続するデータ配列本体を受信する。受信したデータは、送信元のランクと ID をキーにしてデータ格納バッファに登録する。

3.6 メッセージ構造体の回復

メッセージ構造体を処理する際に、引数にスタブ構造体があれば、別経路でデータ配列が送信されていると判断し、スタブ構造体内に格納されている ID 情報を用いてデータ格納バッファからデータ配列を取得し、メッセージ構造体を再構成する。この再構成したメッセージ構造体を用いてその後の処理を行う。この様子を図 9 に示す。

4. 高速化手法の評価

提案高速化手法の評価を行った。

4.1 評価環境: ABCI

評価は、産総研の保有する大規模 AI 向けクラスタシステム ABCI(AI 橋渡しクラウド)[17] で行った。ABCI は、

表 1 ABCI の構成

A-node	
ノード数	120
メモリ	512 GB
CPU	Intel 8360Y × 2
GPU	NVIDIA A100 × 8
ネットワーク	Infiniband HDR × 4
理論バンド幅	25 GB/s × 4
V-node	
ノード数	1088
メモリ	384 GB
CPU	Intel 6148 × 2
GPU	NVIDIA V100 × 4
ネットワーク	Infiniband EDR × 2
理論バンド幅	12.5 GB/s × 2

大別して V-node と A-node の 2 つのシステムで構成される。主要な構成を表 1 に示す。詳細な構成は文献 [18] を参照されたい。A-node, V-node ともに Infiniband による高速な通信が提供されている。

本稿では MPI の実装として Open MPI[19] 4.1.3 を用いた。

4.2 評価結果

評価結果を図 11 に示す。A-node に関しては 4GB/s を超える性能となり、大幅な性能の向上が見られた。

一方で、MPI 本来の性能と比較すると大幅に劣る結果となっている (図 12)。

4.3 議論

データ転送部分を別経路にすることでバイトストリームに対するコピーのオーバヘッドは大幅に低減されたにもかかわらず十分な性能が得られていない。この原因としては、MPI の受信部が、ビジーループとなっている点が考えられる。

前述のように Julia の通信ライブラリは libuv によって非

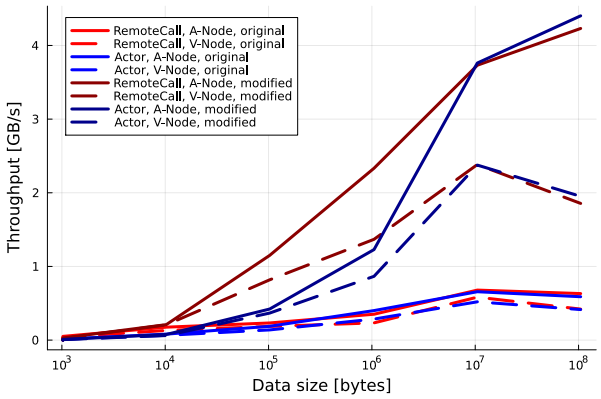

図 11 高速化手法の適用結果

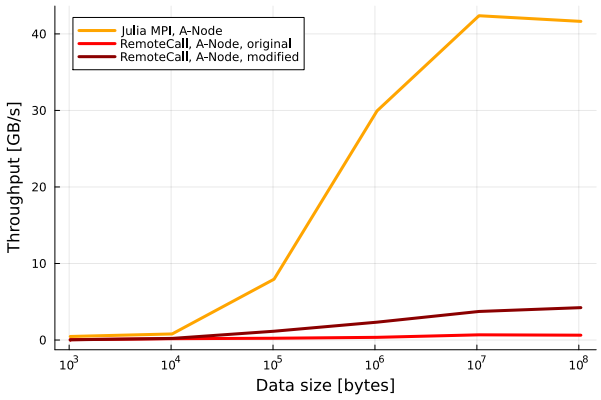

図 12 高速化手法の適用結果と MPI ネイティブ性能の比較

同期化されており、一つのコルーチンがストリーム読み込みでブロックすると自動的に他のコルーチンにスケジュールが移行するように設計されている。しかし、MPI 通信は libuv の管理化にないため同様の処理を行うことはできない。このため、`MPI_Iprobe` でデータの到着を監視し、データがとうちゃくしていなければ `yield` で他のスレッドへ制御を移す構造となっている。MPI 受信専用スレッドを立て、ブロックすることもできるはずだが、現状はそのようになっていない。

5. Actor 記法の再検討

5.1 従来の記法

文献 [1] で提案した Actor 記法は、メッセージ構造体を定義し、それを処理するジェネリック関数 `handle` を定義するものであった。例としてカウンタを実装したものを図 13 に示す。Actor へのメッセージ送信時には、`callOn` 関数に、Actor への参照とメッセージオブジェクトを渡す。

この記法は、Actor のモデルをそのまま反映したもので直感的ではあるが、やや冗長である。また、Julia の通常のプログラム記述法との親和性も低い。ローカルに作成した構造体をアクター化するためには、すべての関数に相当する構造体を定義し、さらに `handle` メソッドで置き換える必要がある。これは理想的とはいえない。

```
# Actor の状態を表す構造体
mutable struct Counter <: Actor
  v::Int64
end

# メッセージを表す構造体
struct Add <: Message
  v::Int64
end
struct Sub <: Message
  v::Int64
end

# それぞれのメッセージに対するハンドラ
function MyActor2.handle(a::Counter, mes::Add)
  a.v += mes.v
end

function MyActor2.handle(a::Counter, mes::Sub)
  a.v -= mes.v
end

# Actor をノード 2 に作成
counter = @startat 2 Counter(0)

# Add を 呼び出して結果を取得
fetch(callOn(counter, Add(10)))
```

図 13 従来の記法によるカウンタ

```
# Actor の状態を表す構造体
mutable struct Counter <: Actor
  v::Int64
end

# @remote マクロを用いて通常の関数としてハンドラを定義
@remote function add(c::Counter, v::Int64)
  c.v += v
end

@remote function sub(c::Counter, v::Int64)
  c.v -= v
end

# Actor をノード 2 に作成
counter = @startat 2 Counter(0)

# 通常の関数呼び出しで Actor のメッセージハンドラを実行
fetch(add(counter, 10))
```

図 14 新しい記法によるカウンタ

5.2 新しい記法

この問題を解決するため、新たな記法を導入した。新たな記法では、通常の関数定義にマクロ`@remote` を付加しただけの記法でハンドラ関数を定義することができる。また、メッセージ送信時は、通常の関数呼び出しで行う事ができる。この記法を用いて記述した例を図 14 に示す。記述量が少ないこと、通常の関数と親和性の高い方法で記述できる事がわかる。

5.3 新たな記法の実装

新たな記法は、`@remote` マクロで実現されている。この

```
function add(c::Counter, v::Int64)
  c.v += v
end

function add(c::ActorRef, v::Int64)
  callOn(c, (add, (v,)))
end
```

図 15 展開後の関数

マクロは後続する関数定義を、2 つの関数定義に展開する。1 つは入力とまったく同じもの、もう 1 つは第一引数を、アクタへの参照 `ActorRef` とし、ボディ部では `callOn` 関数を呼び出すものだ。例えば図 14 の `add` 関数は、図 15 に示す 2 つの関数定義に展開される。

メッセージとして明示的に構造体を作成するのではなく、関数を含むタプルで代用している。呼び出し側では、`ActorRef` を引数とするバージョンが呼び出され、Actor 側では `Counter` を引数とするバージョンが用いられる。新しい記法を用いると、既存のプログラムを容易に Actor 化することができる。

6. おわりに

本稿では、Julia 言語の分散並列実行機能の高性能並列環境での性能測定を行い、通信スループットの改善を試みた。性能改善の結果 10 倍程度の向上は見られたが、本来のスループットには遠く及ばなかった。

また、Actor の記述方法を再検討し、メッセージ構造体を用いない記法を提案した。記述のコストを大幅に低減することができた。

今後の課題としては、以下が挙げられる。

- スループットの低下原因を解析し、スループットのさらなる向上を試みる。
- 応用プログラムを Actor を用いて記述し、記述手法の妥当性を検証する。

謝辞 本研究は JSPS 科研費 JP19K11994 の助成を受けたものです。

参考文献

[1] 中田秀基: Julia 言語への Actor の導入, 第 135 回 情報処理学会プログラミング研究会 (2021).
[2] : The Julia Language - A fresh aapproach to technical computing, `https://www.julialang.org/`. Accessed: 2020-06-17.
[3] Lattner, C. and Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation, *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, pp. 75–86 (2004).
[4] DeMichiel, L. G. and Gabriel, R. P.: The Common Lisp Object System: An Overview, *ECOOP' 87 European Conference on Object-Oriented Programming* (Bézivin, J., Hullot, J.-M., Cointe, P. and Lieberman, H., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 151–170 (1987).
[5] Ana L ´ uciaMoura, Ierusalimschy, R.: Revisiting Coroutines, *ACM Transactions on Programming Languages and Systems*, Vol. 31, No. 2 (2009).
[6] : libuv, `https://libuv.org/`. Accessed: 2023-03-1.
[7] : Julia Standard Library / Distributed Computing, https://docs.julialang.org/en/v1/stdlib/Distributed/. Accessed: 2020-06-17.
[8] : MPI interface for the Julia language, `https://github.com/JuliaParallel/MPI.jl`. Accessed: 2020-06-17.
[9] : MPIClusterManagers.jl, `https://github.com/JuliaParallel/MPIClusterManagers.jl`. Accessed: 2020-06-17.
[10] Hewitt, C., Bishop, P. and Steiger, R.: A universal modular ACTOR formalism for artificial intelligence, *Proc. International Joint Conference on Artificial Intelligence*, pp. 235–245 (1973).
[11] Agha, G.: *Actors: a model of concurrent computation in distributed systems*, MIT Press (1986).
[12] Yonezawa, A., Briot, J.-P. and Shibayama, E.: Object-Oriented Concurrent Programming in ABCL/1, *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, New York, NY, USA, Association for Computing Machinery, p. 258–268 (online), DOI: 10.1145/28697.28722 (1986).
[13] Yonezawa, A., Briot, J.-P. and Shibayama, E.: Object-Oriented Concurrent Programming in ABCL/1, *SIGPLAN Not.*, Vol. 21, No. 11, p. 258–268 (online), DOI: 10.1145/960112.28722 (1986).
[14] : Erlang Programming Language, `https://erlang.org/`. Accessed: 2021-05-12.
[15] : The Elixir programming language, `https://elixir-lang.org/`. Accessed: 2021-05-12.
[16] : akka, `https://akka.io`. Accessed: 2021-07-12.
[17] : ABCI AI Bridge Infrastructure: `https://abci.ai/`. Accessed: 2019-02-01.
[18] Takizawa, S., Tanimura, Y., Nakada, H., Takano, R. and Ogawa, H.: ABCI 2.0: Advances in Open AI Computing Infrastructure at AIST, 情報処理学会研究報告 2021-HPC-180 (2021).
[19] : Open MPI: Open Source High Performance Computing, `https://www.open-mpi.org/`.