Julia 言語を用いた高性能並列実行環境の構築

中田秀基

産業技術総合研究所

{デジタルアーキテクチャ,人工知能}研究センター

本研究はJSPS科研費JP19K11994の助成を受けたものです

背景

- 強化学習には膨大な計算が必要
  - 並列計算が不可欠
  - 複雑な同期制御
  - 並列計算機の利用が不可欠
- Python向け分散フレームワークRay
  - 柔軟だが通信が低速
- Julia言語
  - 高速
  - 機械学習分野で利用が広がる
- Julia 向けに分散フレームワークがほしい

関連研究: Ray

- Berkeley riselab のPython用並列分散フレームワーク
  - 強化学習を対象
- Actorとして記述
  - remote をつけたクラスをリモートでActorとして生成
- 同期はfuture

```
import ray
ray.init()

@ray.remote
class Counter(object):
    def __init__(self):
        self.n = 0
    def increment(self):
        self.n += 1
        return self.n

c = Counter.remote()
future = c.increment.remote()
print(ray.get(future))
```

先行研究: Julia Actor [’21 中田]

- Julia のActorフレームワークを提案
  - Juliaの既存分散実行フレームワークDistributed.jlの機能を利用して軽量に実装
- 高性能計算機環境での性能評価は行われていない
- 記法がナイーブでJuliaの通常の記法と整合しない

本発表の目的と成果

- 目的
  - Juliaの分散並列実行機構を高性能計算機環境で評価
  - Actor記法の再検討

- 成果
  - 高性能計算環境での性能を改善
  - マクロの導入によるJuliaと親和性の高い記法の導入

本発表の構成

- 背景
  - Julia言語
  - Julia言語の分散並列機構
    - Distributed.jl, MPI.jl, MPIClusterManagers.jl
  - Actor
  - JuliaでのActorの実装
- 高性能計算環境での性能評価と改善
- Actor記法の改善
- おわりに

Julia言語

- 高速なスクリプト「風」言語
  - LLVMを用いたJITコンパイル
  - 実行時に実際に呼び出された型に特化したコードを動的に生成
  - 型を静的に指定することも可能
  - Cに匹敵する速度
- コルーチン(Task)とチャンネルをnativeにサポート
  - libuvを用いたI/O
  - スレッドもサポート

Julia言語(2)

- CLOS的なOOP – struct と Generic function で構成
  - クラスに属する「メソッド」がない

Python

```
class Shape:
  pass
class Circle(Shape):
  def __init__(self, radius):
    self.radius = radius
  def area(self):
    return self.radius * self.radius\
           * 3.14

class Rectangle(Shape):
  def __init__(self, width, height):
    self.width = width
    self.height = height
  def area(self):
    return self.width * self.height

print(Circle(10).area())
print(Rectangle(10,20).area())
```

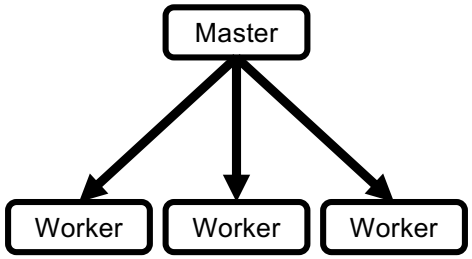
Julia

```
abstract type Shape end

struct Circle <: Shape
    radius::Real
end

struct Rectangle <: Shape
    height::Real
    width:: Real
end

area(c::Circle) = c.radius * c.radius * pi
area(r::Rectangle) = r.height * r.width

area(Circle(5.0))
area(Rectangle(10,20))
```

Distributed.jl

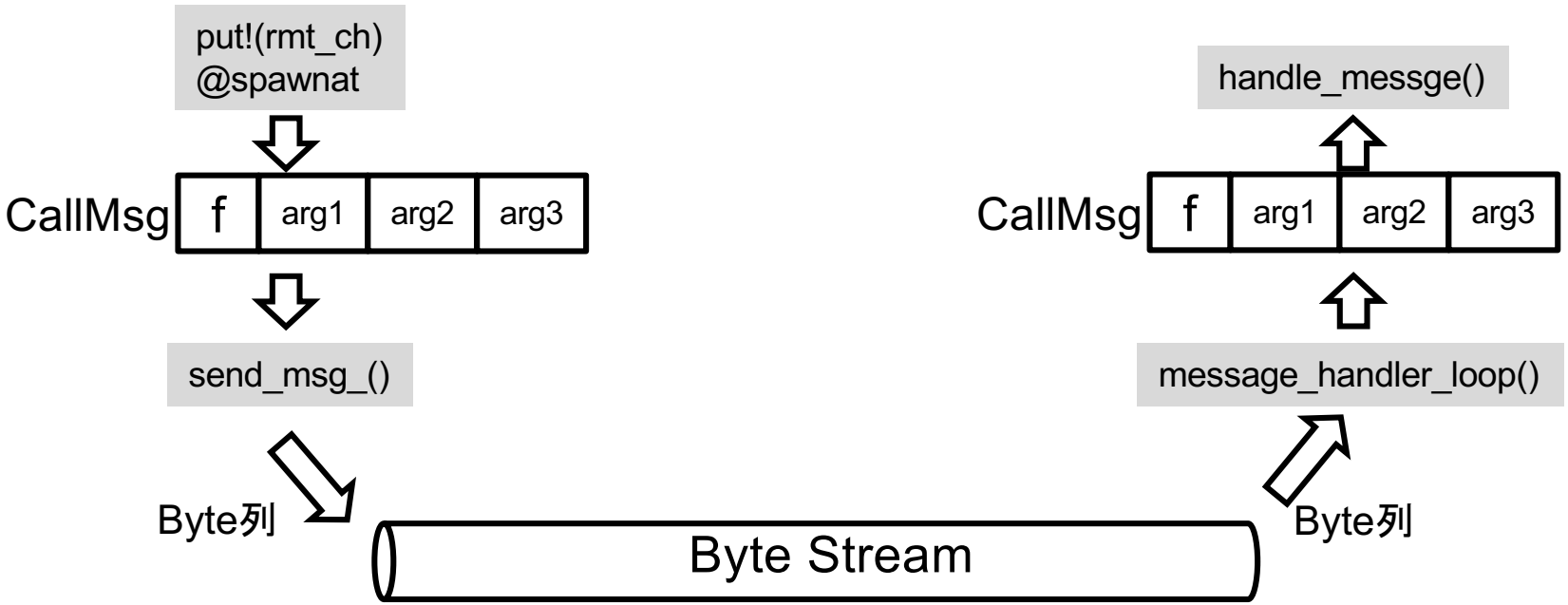
- RPC – 関数の実行ノードを指定
  - 引数は自動的に転送される
  - 呼び出しは即時にリターン
  - 返り値はfuture – 同期機構
  - futureの値をfetchしようとした時点で、まだ計算が終了していなければそこでブロック

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



- リモートチャンネル
  - リモートノード上のチャンネルのグローバルな参照
  - 直接書き込み可能
- 問題点
  - リモートノード上の状態を管理する方法がない
  - グローバル変数に書き込むことは可能

Distributed.jl の実装



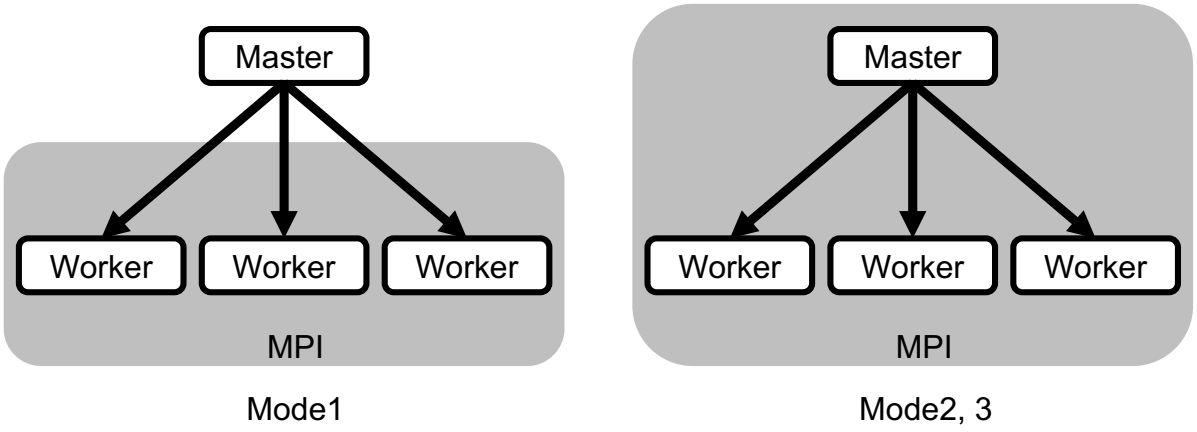
MPI.jl

- 任意のMPI実装を利用可能
- プログラミングモデルはMPIと同じ
  - 複数の全く同一のプログラムがすべてのノードで動く

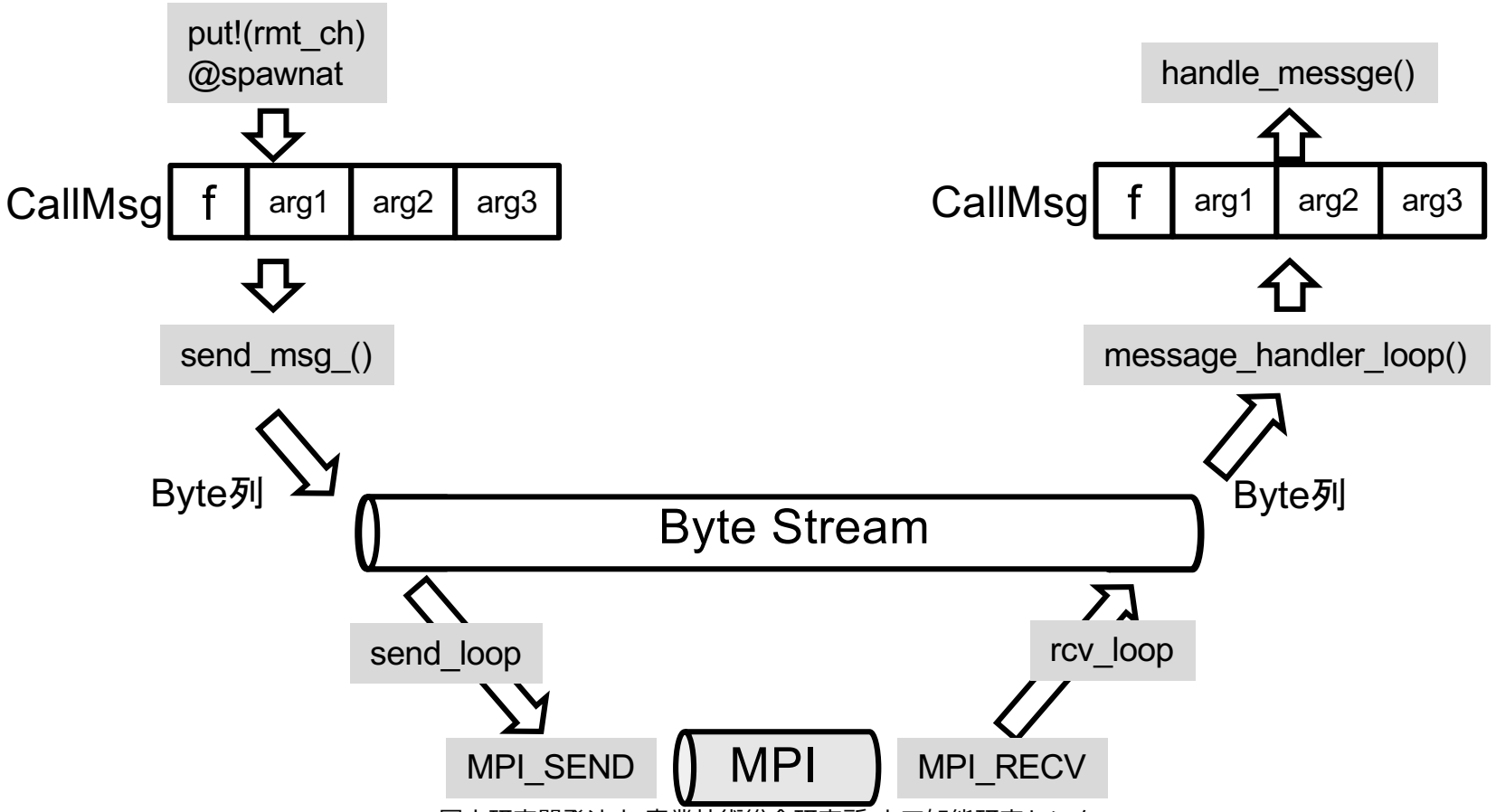
```
using MPI
MPI.Init()
comm = MPI.COMM_WORLD
# 送信
a = rand(dtypes[dindex], datasize)
MPI.Send(a, target, tag, comm)
...
# 受信
MPI.Recv!(a, target, tag, comm)
```

MPIClusterManagers.jl

- ワーカを起動するためのプラグインの一つ
- 3つのモードを持つ
  - モード1 - ワーカのみがMPIで起動される
  - モード2,3 – すべてのノードがMPIで起動
    - モード2 – TCP通信、モード3 - MPI通信

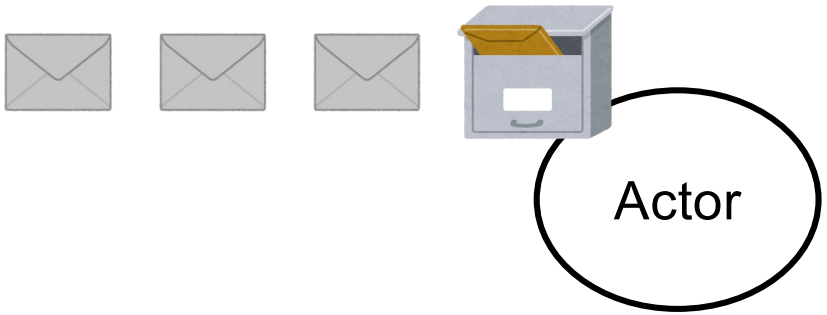


MPIClusterManagersの実装



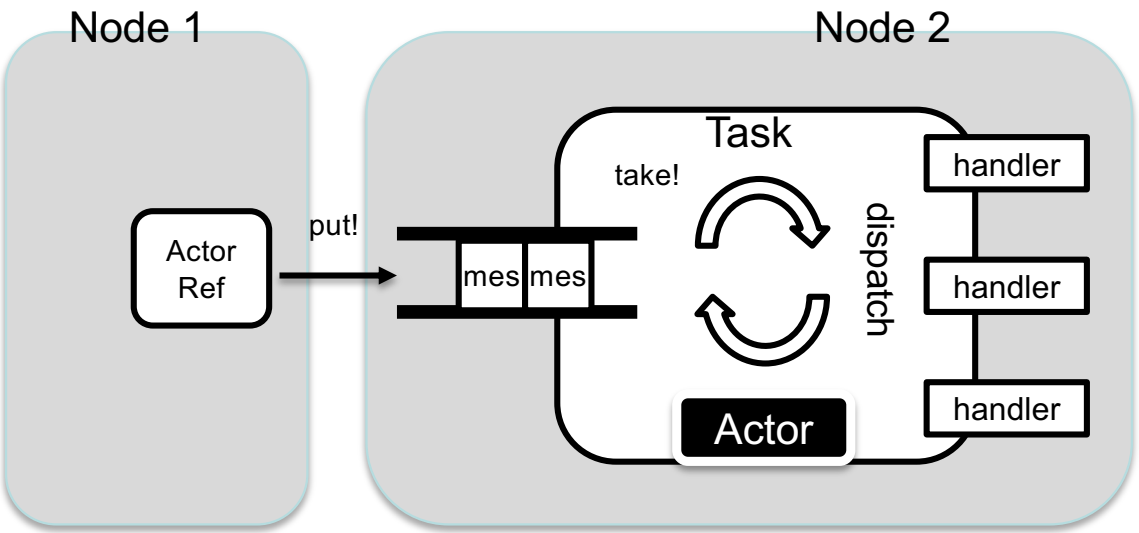
Actorとは

- 状態を持つ実行主体がメッセージを受信して処理
  - 状態と実行主体が1対1に対応
    - Object ＋ Threadとは本質的に異なる
  - メッセージの処理は一つずつ
  - Actor内の状態更新には排他制御が不要
    - メッセージキューの時点で逐次化されているため



JuliaのActor実装

- リモートチャンネルで、Actorのメッセージキューを表現
- リモートチャンネルから読み出してハンドラを読み出すループで Actorを表現




```
function taskLoop(cn::AbstractChannel,
                  target::Actor)
  while true
    msg = take!(cn)
    handle(target, msg)
  end
end
```

Actorの記述例

- メッセージを構造体として記述
- メッセージハンドラをジェネリック関数として記述
  - メッセージの型を用いてディスパッチ

```
# 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
# 現在の値を返り値として返している
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 Counterをノード2で起動

counter = @startat 2 Counter(0)

# counter に対してメッセージ Add を送信
# 返答をFuture f に格納

f = callOn(counter, Add(10))

# Future f から値を取得
# 実行中であれば、ここでブロック

fetch(f)
```

本発表の構成

- 背景
  - Julia言語
  - Julia言語の分散並列機構
    - Distributed.jl, MPI.jl, MPIClusterManagers.jl
  - Actor
  - JuliaでのActorの実装
- 高性能計算環境での性能評価と改善
- Actor記法の改善
- おわりに

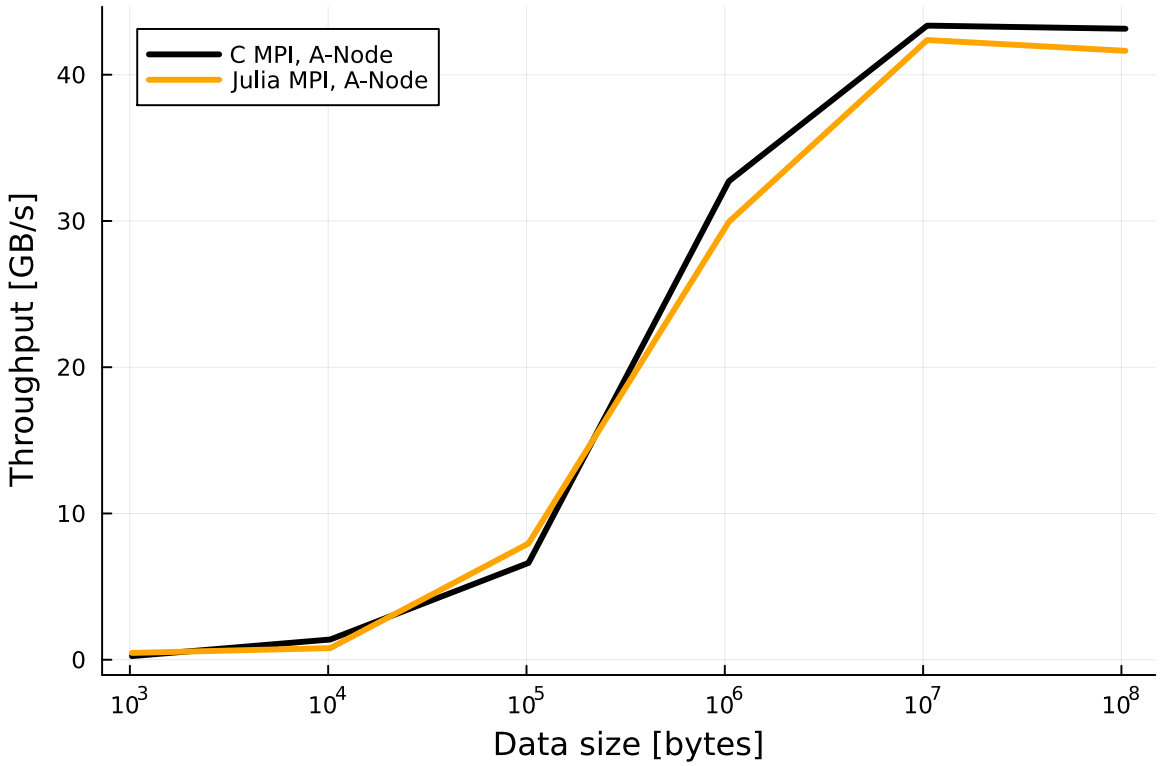
高性能計算環境でのスループット測定

- 高性能計算環境
  - ABCI (AI橋渡しクラウド)を使用
  - A-nodeの結果のみを示す
- データサイズを変更しつつpingpong スループットを測定
- 比較
  - C言語のMPI
  - JuliaのMPI
  - Distributed.jl の RPC
  - Actor

	A-node	V-node
GPU	A100 x 8	V100 x 4
ネットワーク	Infiniband HDR x 4	Infiniband EDRx2
理論ピーク帯域	25GB/s x 4 2ノード間では最大50GB/s	12.5GB/s x 2

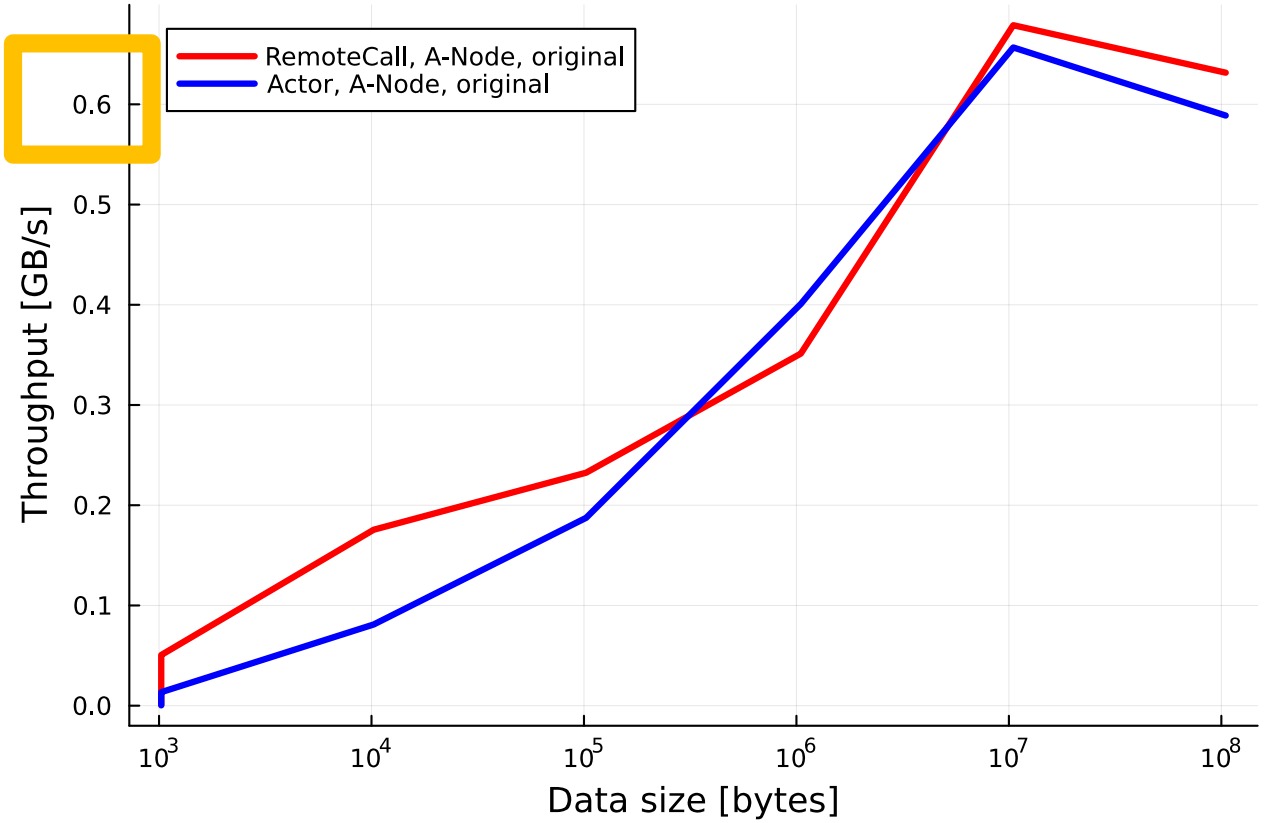
C-MPIとJulia MPI

- いずれも50GB/s近い性能
- JuliaのMPIはCのMPIとほぼ同等



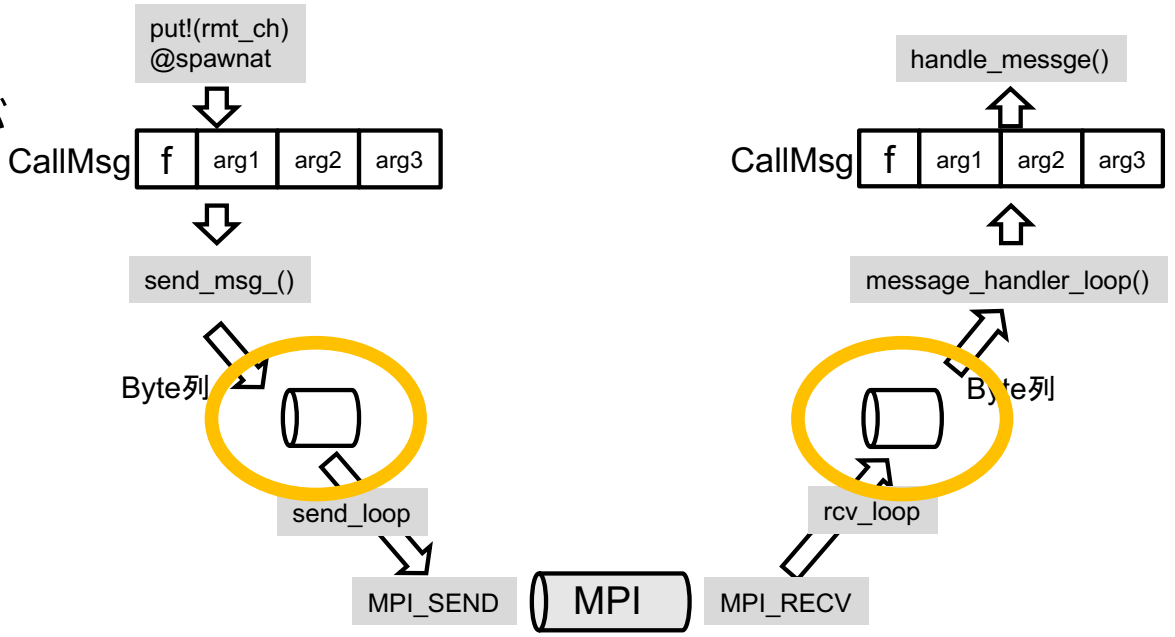
RPCとActorの性能

- 本来の1%程度の性能
- RPCとActorはほぼ同じ
  - RPC上に実装されているので当然



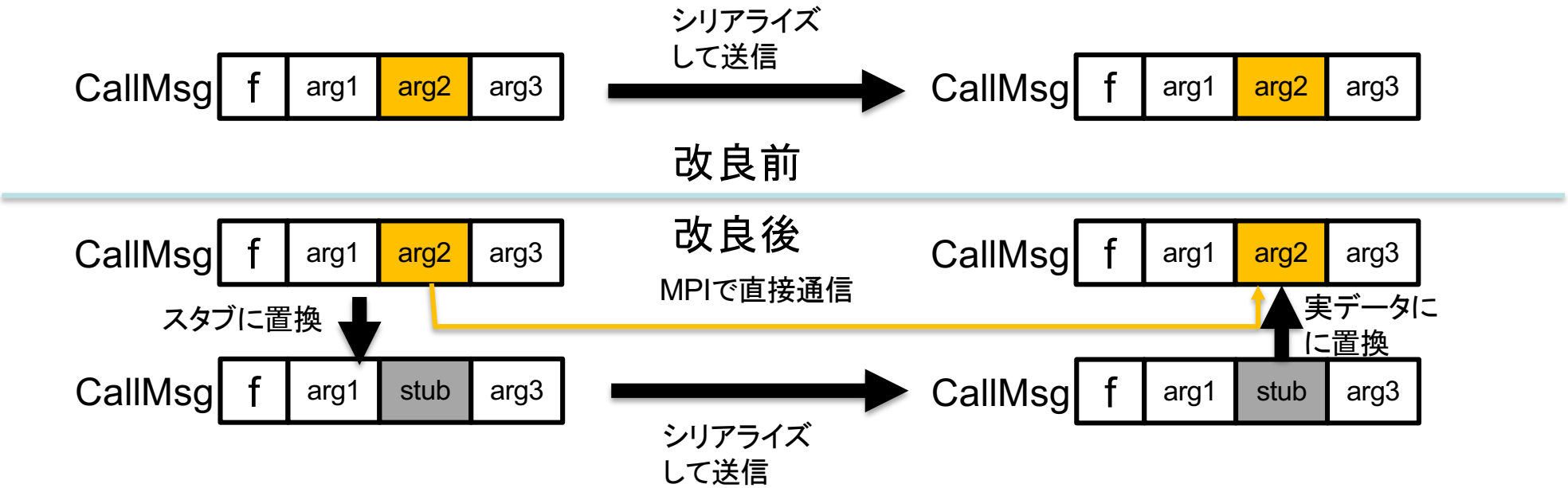
スループット低下原因の考察

- オンメモリのバイトストリームに
  データを書き出す
- バイトストリームから読み出した
  バイト列をMPIで転送
- バイトストリームに対するコピーが
  送信側、受信側でそれぞれ発生



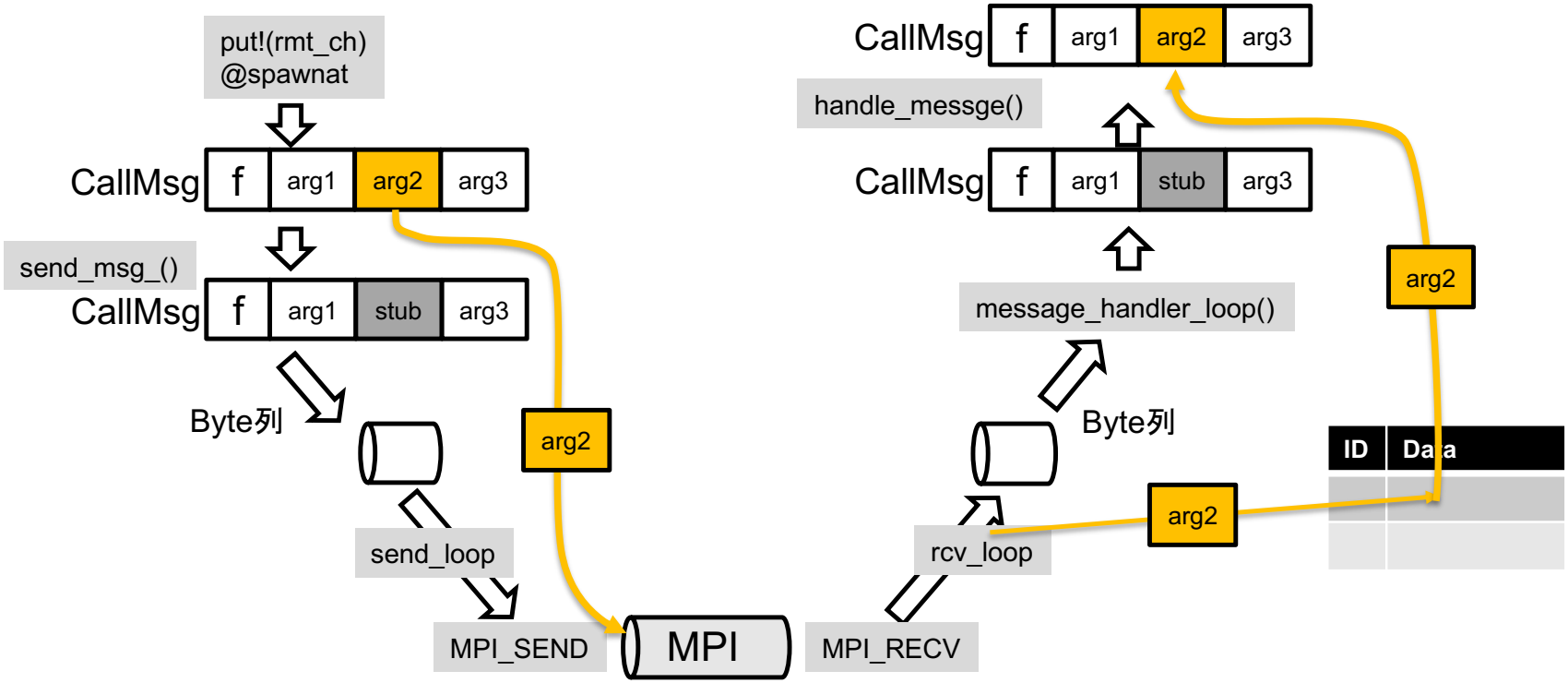
スループット低下原因に対する対策

- コピーが大きな問題になるのは、大規模な配列だけ
- 配列だけはバイトストリームを介さずに直接MPIで送信すれば、コピーによる性能低下を回避できる
- 配列データをスタブデータに置き換えて送信



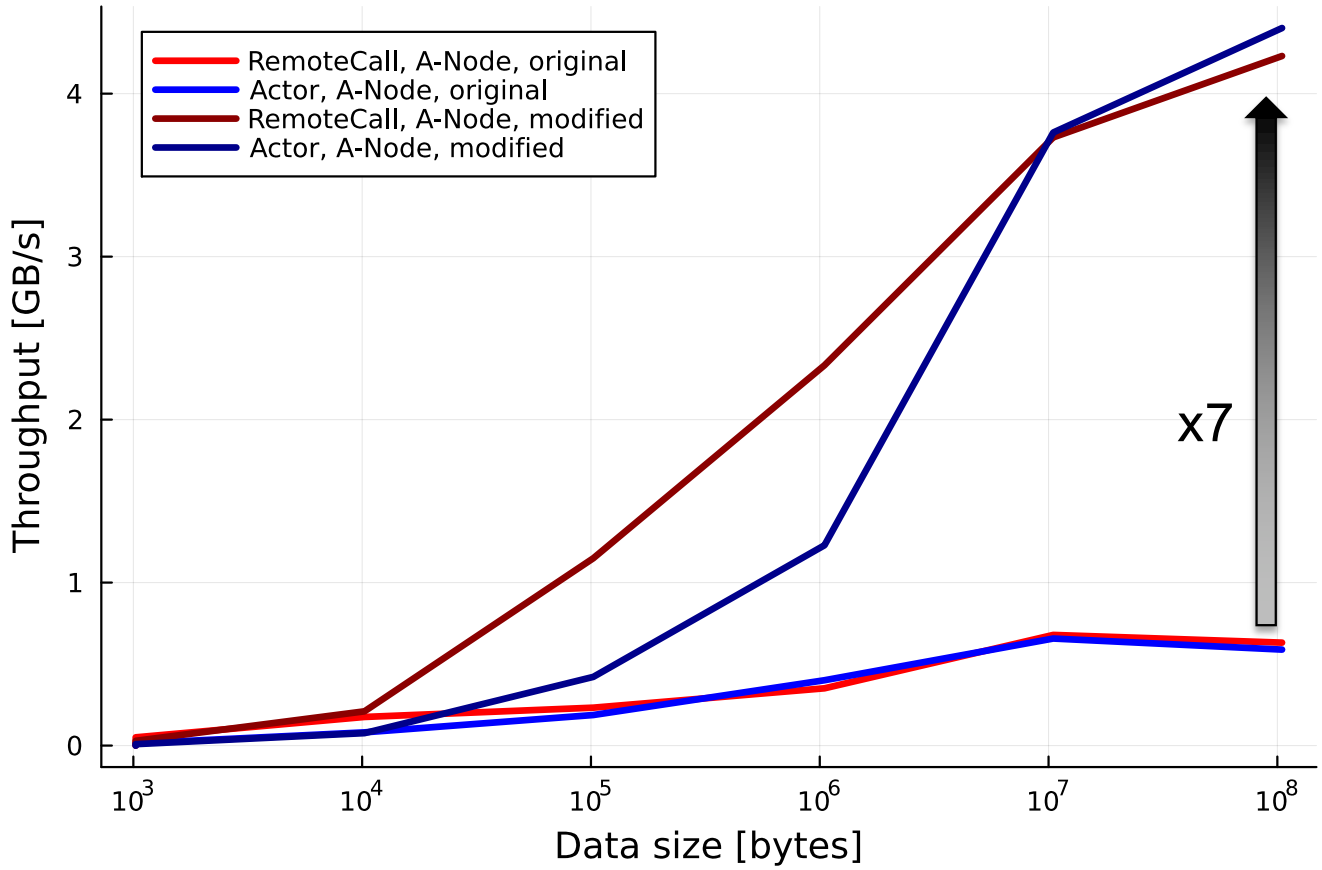
改善手法の実装

- MPIの受信ループは共有
- 受信したデータをテーブルに保持
  - スタブを置換



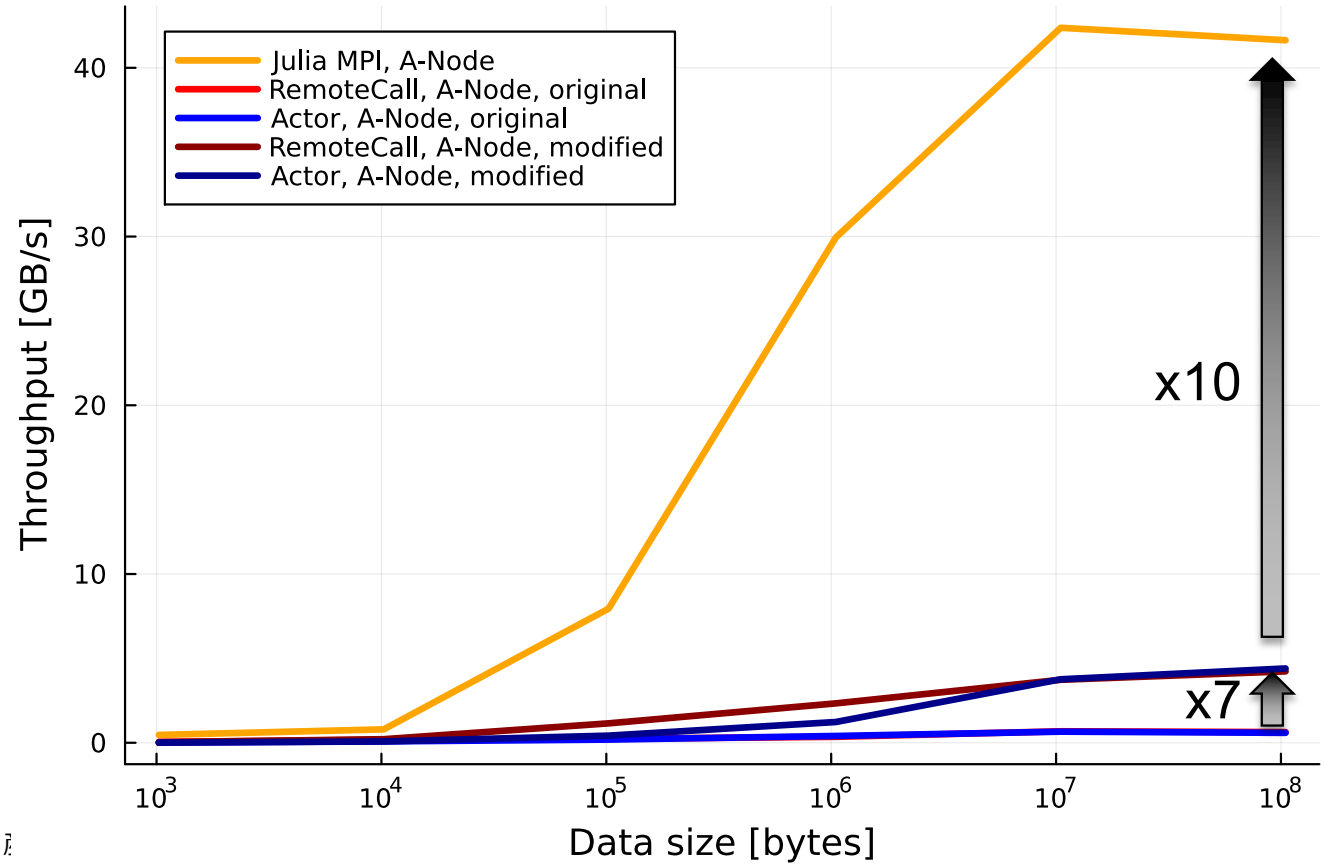
改善手法の効果

- 大幅なスループット向上
  - およそ7倍程度



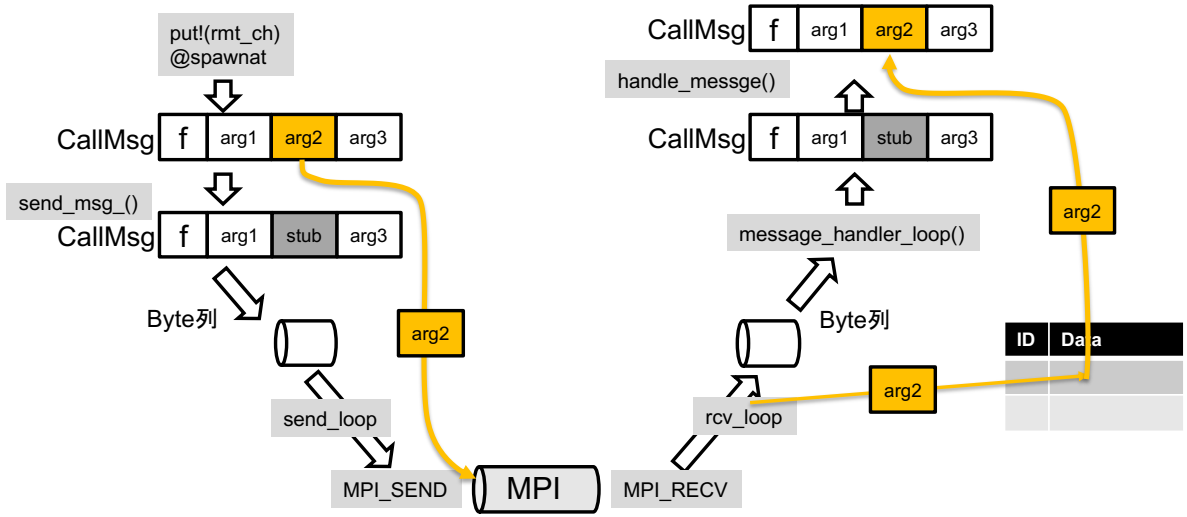
しかし…

- 本来のMPIのスループットには遠く及ばない



議論

- MPIとコルーチンの相性
  - MPIはlibuvの管理下にない
  - 受信部分がビジーループになっている
- Threadを使えばよいはず



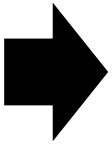
本発表の構成

- 背景
  - Julia言語
  - Julia言語の分散並列機構
    - Distributed.jl, MPI.jl, MPIClusterManagers.jl
  - Actor
  - JuliaでのActorの実装
- 高性能計算環境での性能評価と改善
- Actor記法の改善
- おわりに

Actor記法の改善

- 先行研究記法の問題点
  - メッセージを構造体として定義する必要がある
    - Actorモデルのナイーブな反映
    - 通常の関数と記述方法が乖離
  - 呼び出し時に専用の関数を使用する必要がある

```
mutable struct Counter
     v::Int64
End

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



```
mutable struct Counter <: Actor
     v::Int64
End

struct Add <: Message
    v::Int64
End

function MyActor2.handle(a::Counter, mes::Add)
  a.v += mes.v
end
```

改善方法

- メッセージパッシングをナイーブに表現することをやめ通常の関数と同じように定義させる
- 呼び出し時に用いるメッセージ送信用のスタブ関数をマクロを用いて自動生成
- @remote マクロ
  - メッセージハンドラの定義からスタブ関数を自動生成

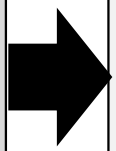
```
@remote function f(a::ActorA, x, y)
  ...
end
```

```
function f(a::ActorA, x, y)
  ...
end
```

```
function f(a::ActorRef, x, y)
  callOn(a, (f, (x, y))
end
```

Actorの記述の比較

```
mutable struct Counter <: Actor
     v::Int64
End

struct Add <: Message
    v::Int64
End

struct Sub <: Message
    v::Int64
End

function MyActor2.handle(a::Counter, mes::Add)
  a.v += mes.v
end

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

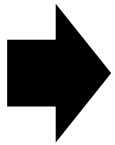


```
mutable struct Counter <: Actor
  v::Int64
end

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

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

Actor使用方法の比較

- メッセージ構造体が不要
- callOn関数も不要

```
counter = @startat 2 Counter(0)

f = callOn(counter, Add(10))

fetch(f)
```



```
counter = @startat 2 Counter(0)

f = add(counter, 10)

fetch(f)
```

本発表の構成

- 背景
  - Julia言語
  - Julia言語の分散並列機構
    - Distributed.jl, MPI.jl, MPIClusterManagers.jl
  - Actor
  - JuliaでのActorの実装
- 高性能計算環境での性能評価と改善
- Actor記法の改善
- おわりに

おわりに

- まとめ
  - JuliaのRemote Procedure Callを高性能計算機環境で測定
  - スループットの改善を試みる
    - 一定の高速化を確認
    - しかし本来の性能は出ず
  - Actorの記法を再検討
    - Julia本来の構文に近い書き方を実現
- 今後の課題
  - 一層のスループット改善
  - 分散アプリケーションの記述を通じて検証