Actor Frameworks for the JVM Platform: A Comparative Analysis

秋山 茂樹

全体ミーティング (2010/10/13)

Actor Frameworks for the JVM Platform: Comparative Analysis

- 各種 Java Actor ライブラリについて、プログラマビリティと性能のトレードオフを分析した論文
  - 最終的に（共有メモリマシン上で）

    Actor が保証すべき各種性質を満たした、
    十分性能のよい Actor ライブラリを実装できる

    ことを示している

Actor Frameworks for the JVM Platform: Comparative Analysis

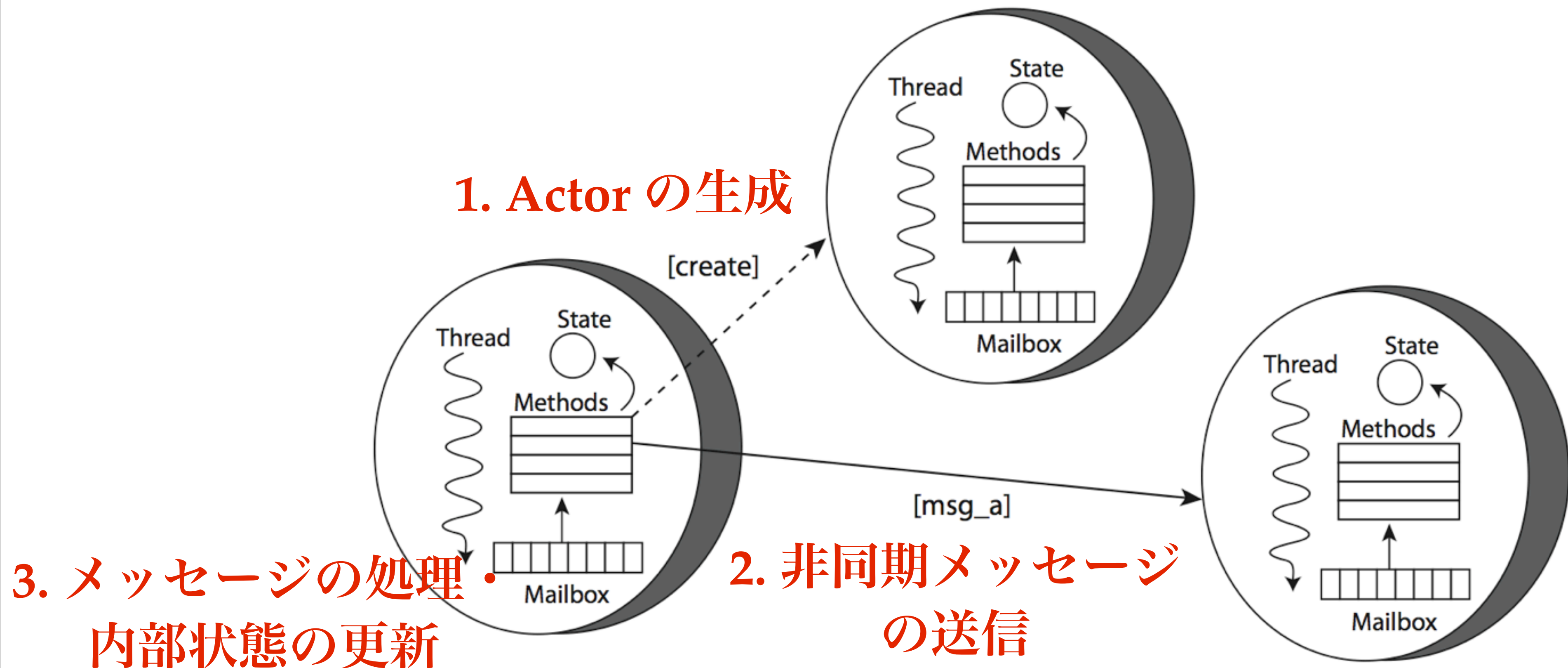
- 著者
  - Rajesh K. Karmani, Amin Shali, Gul Agha (University of Illinois at Urbana-Champaign)
- 発表場所
  - International Conference on the Principles and Practice of Programming in Java (PPPJ), 2009

発表の流れ

1. Actor モデルの概要
2. Actor が満たすべき性質
3. 実装戦略と評価
4. Future work

Actor モデル

- 並行計算のモデル
  Actor: **並行**に動作する計算実体



Actor が満たすべき性質

- カプセル化 (encapsulation)
- スケジューリングの公平性 (fair scheduling)
- 位置透過性 (location transparency)
- 参照の局所性 (locality of reference)
- 透過的な actor の移動 (transparent migration)

2. Actorが満たすべき性質

カプセル化

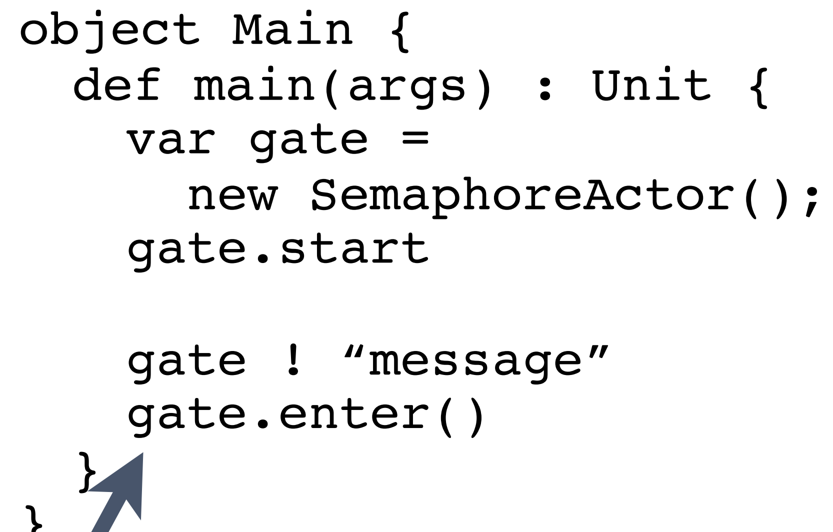
- 他の Actor の内部状態にアクセスできないこと
  ➡ (外部からの) Actor への変更が安全
  ➡ 競合状態にならない

- 次の二つに分けて考える
  1. State encapsulation
  2. Safe messaging

State Encapsulation

- 他の Actor の内部状態に直接アクセスさせない
  - メッセージを介してのみ他の Actor の状態を知ることができるようにする

```
class SemaphoreActor
           extends Actor {
   ...(snip)...
  def enter() {
    if (num < MAX) {
      // critical section
      num = num + 1;
    }
  }
}
```



```
object Main {
  def main(args) : Unit {
    var gate =
      new SemaphoreActor();
    gate.start

    gate ! “message”
    gate.enter()
  }
}
```

Scala Actor だと public メソッドを直接呼び出せてしまう！

2. Actor が満たすべき性質

Safe Messaging

- メッセージはコピーして送らないといけない (共有メモリ環境)
    - Deep copy messaging が遅いからといって、mutable なデータの参照をそのまま送ると状態が漏れる

2. Actor が満たすべき性質

スケジューリングの公平性

- メッセージは必ず相手に到着し、処理される
  - どの Actor も (永続的な) 飢餓状態にならない

2. Actor が満たすべき性質

スケジューリングの公平性: 例

```
class FairActor extends Actor {
  ...(snip)...
  def act() {
    loop { react {
      case (v : Int) => {
        data = v
      }
      case “wait” => {
        if (data > 0) println(data)
        else self ! “wait”
      }
      case “start” => {
        calc ! (“add”, 4, 5)
        self ! “wait”
      }
    } }
  }
}
```

2. wait 状態で calc から結果を受信すると、それを出力

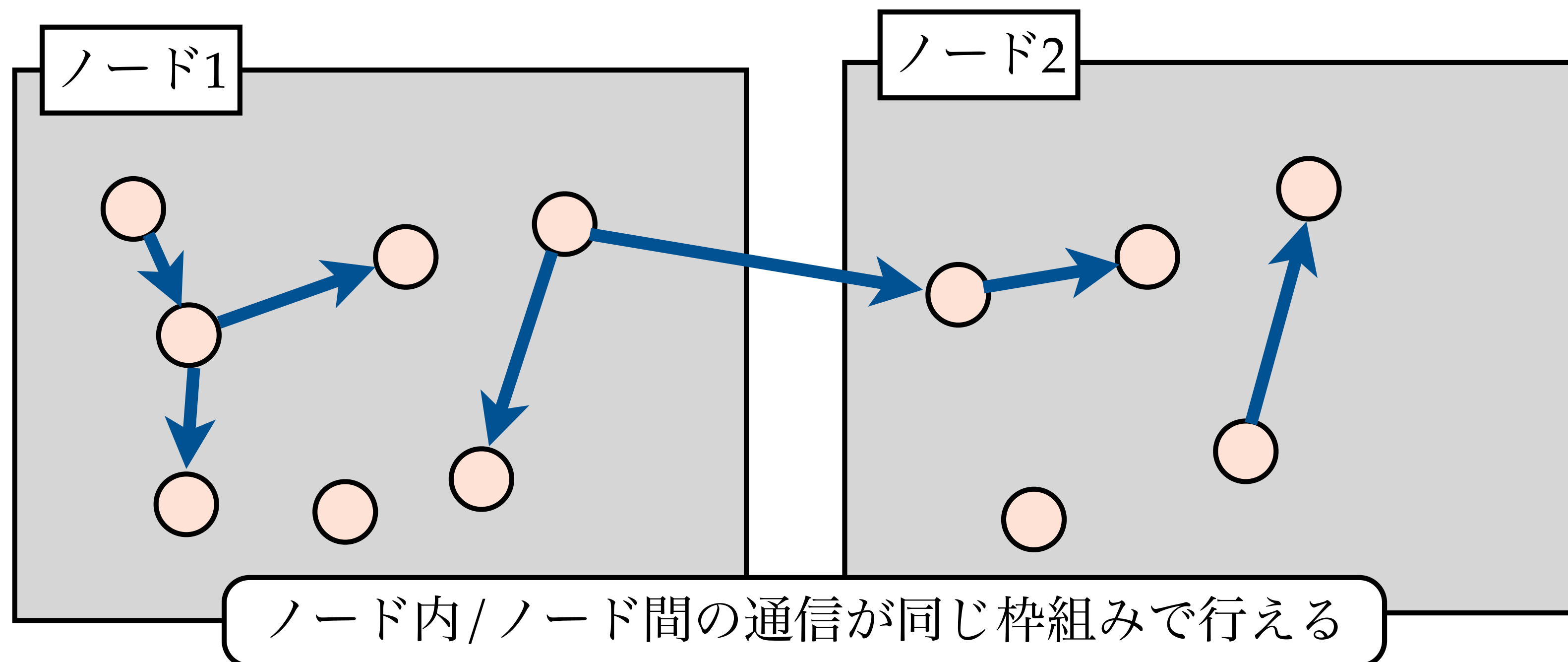
1. calc へ msg を送信したのち、wait 状態になる

公平でないスケジューリングだと、
calc に制御が移らず永遠に wait し続ける可能性がある

2. Actor が満たすべき性質

位置透過性

- Actor が実行されている場所の意識が不要
  - Actor を "名前" でのみ識別



2. Actor が満たすべき性質

移動性

- 異なるノード間で Actor を移動できること
  - 強い移動性 (strong mobility)
    - コードと状態 (execution context) の移動
    - メッセージを処理している Actor を移動可能
  - 弱い移動性 (weak mobility)
    - コードの移動 (+ 初期状態)
    - Mailbox が空の状態の Actor なら移動可能

2. Actor が満たすべき性質

移動性導入のメリット

- Load balancing
  - 並列性がノード間で偏っている場合に、
    Actor を暇なノードに移動させることができる
- Fault-tolerance

問題: カプセル化保証のコスト

- Safe messaging
  - Deep copy は重い
  - By-reference semantics を採用 (Kilim, Scala, etc.)
    - Encapsulation をプログラマが保証する必要がある

2. Actor が満たすべき性質

問題: 公平性の実現

- プリエンプションのサポート
  - Java thread と一対一対応させる
    - 遅い
  - フレームワークで実装

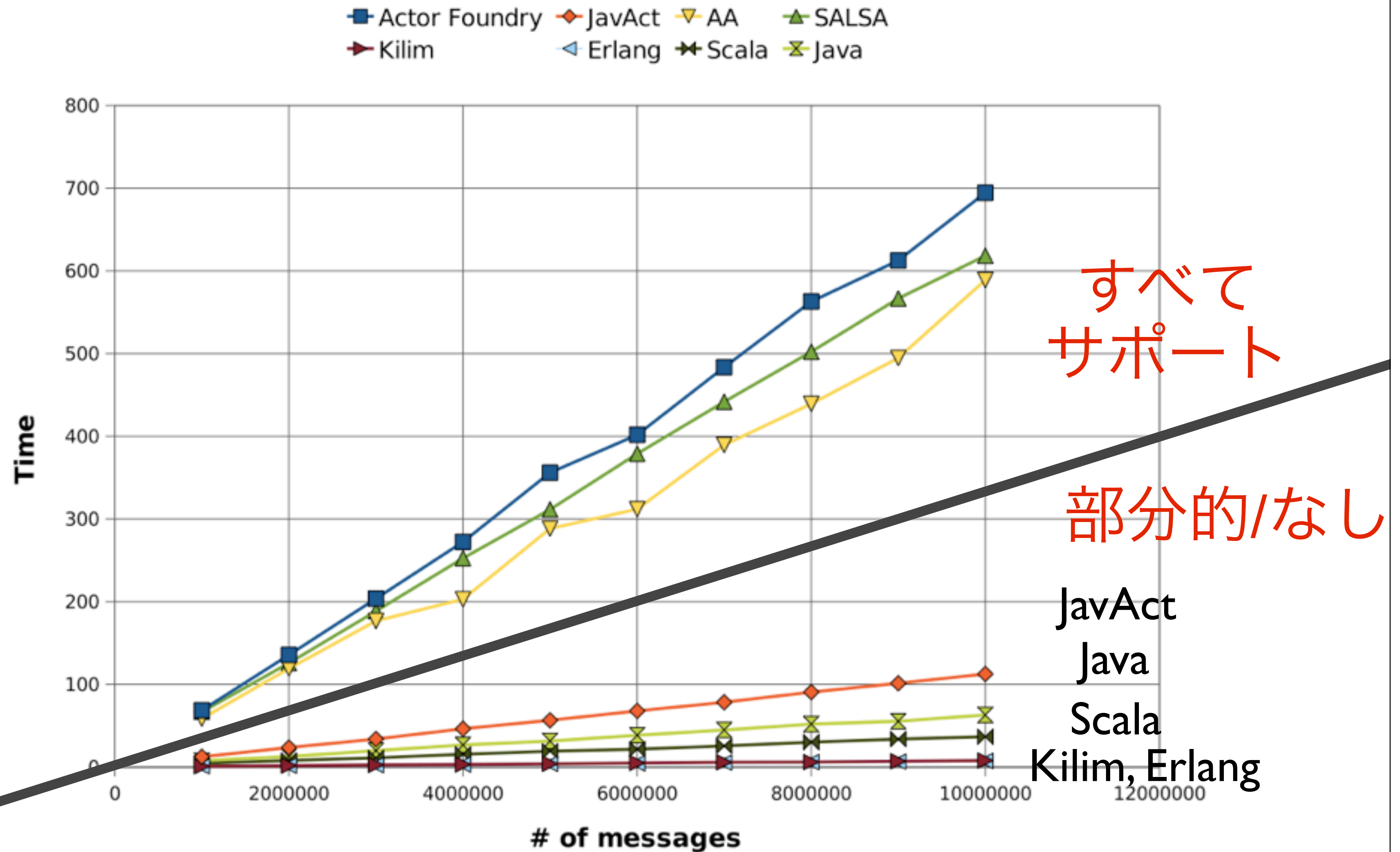
3. 実装戦略と評価

各種ライブラリの機能

	SALSA	Scala Actors	Kilim	Actor Architecture	ActorFoundry
State Encapsulation	○	×	×	○	○
Safe Messaging	○	×	×	○	○
Fair Scheduling	○	○	×	○	○
Location Transparency	○	×	×	○	○
Mobility	○	×	×	○	○

3. 実装戦略と評価

各種ライブラリの性能 (thread-ring)



Actor Foundry

- Java で実装された Actor フレームワーク
  - 基本的なAPI
    - create(node, behavior, args)
    - send(actor, msg, args)
    - call(actor, msg, args)

3. 実装戦略と評価

Actor Foundry v0.1.14 の特徴

- **Actor は Java スレッドと一対一対応**
- メッセージの引数は serializable なオブジェクト
  - **send 時に deep copy**

- メッセージはメソッド名と対応
- 位置透過性の保証
- 弱い移動性のサポート

Actor はスレッドと一対一対応

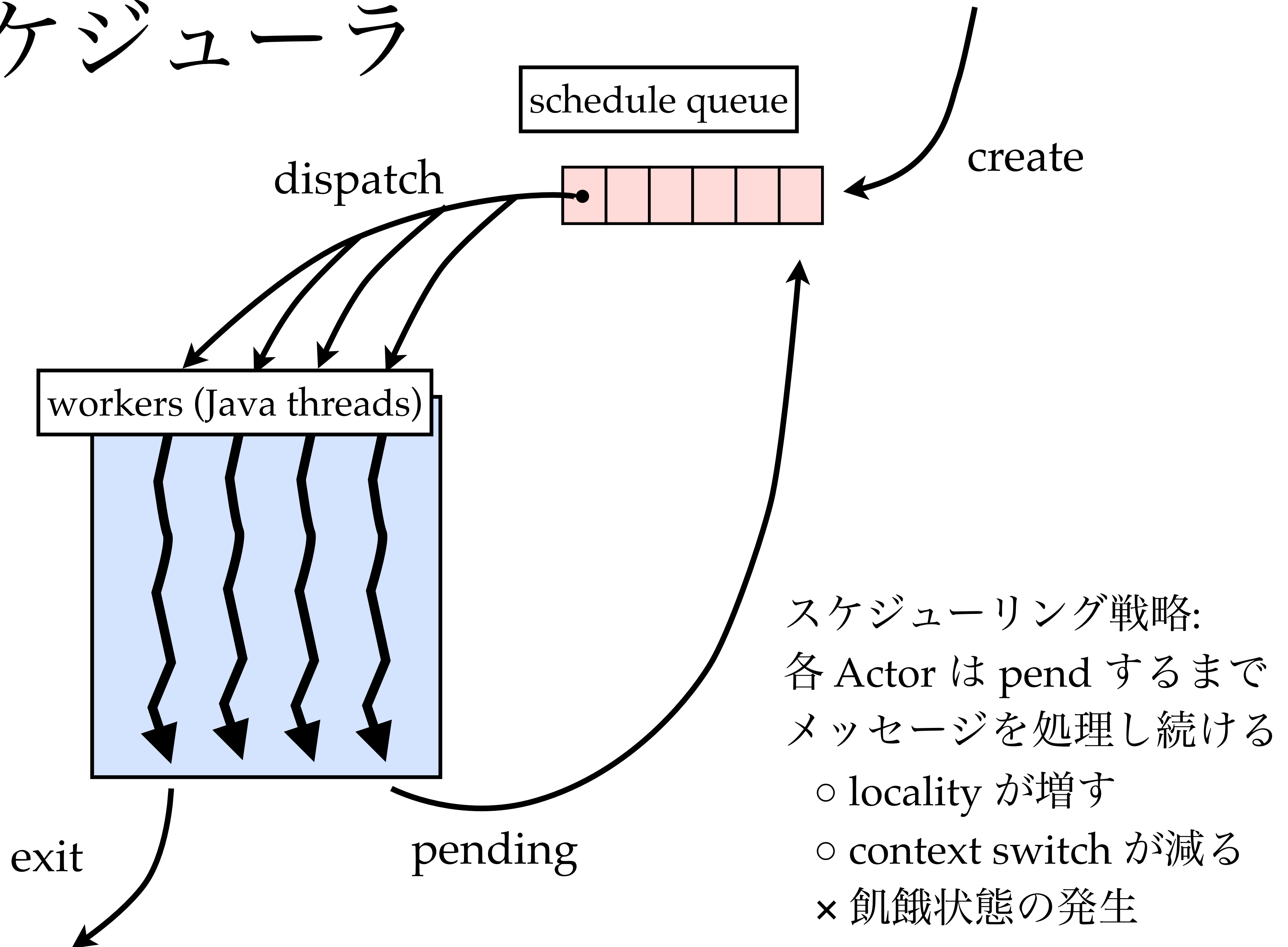
- Actor の生成/コンテキストスイッチのコストが Java スレッドと同等
    - **遅い**

Continuation based actors [Kim97]

- Actor 生成、コンテキストスイッチを高速化
  - Kilim の実装[Srinivasan06] を採用
    - スレッドが軽量
    - バイトコードポストプロセッサで CPS 変換
  - ただし次の変更を施している
    - Reflection によるメッセージの dispatch に対応
    - **Actor 用スケジューラの導入**

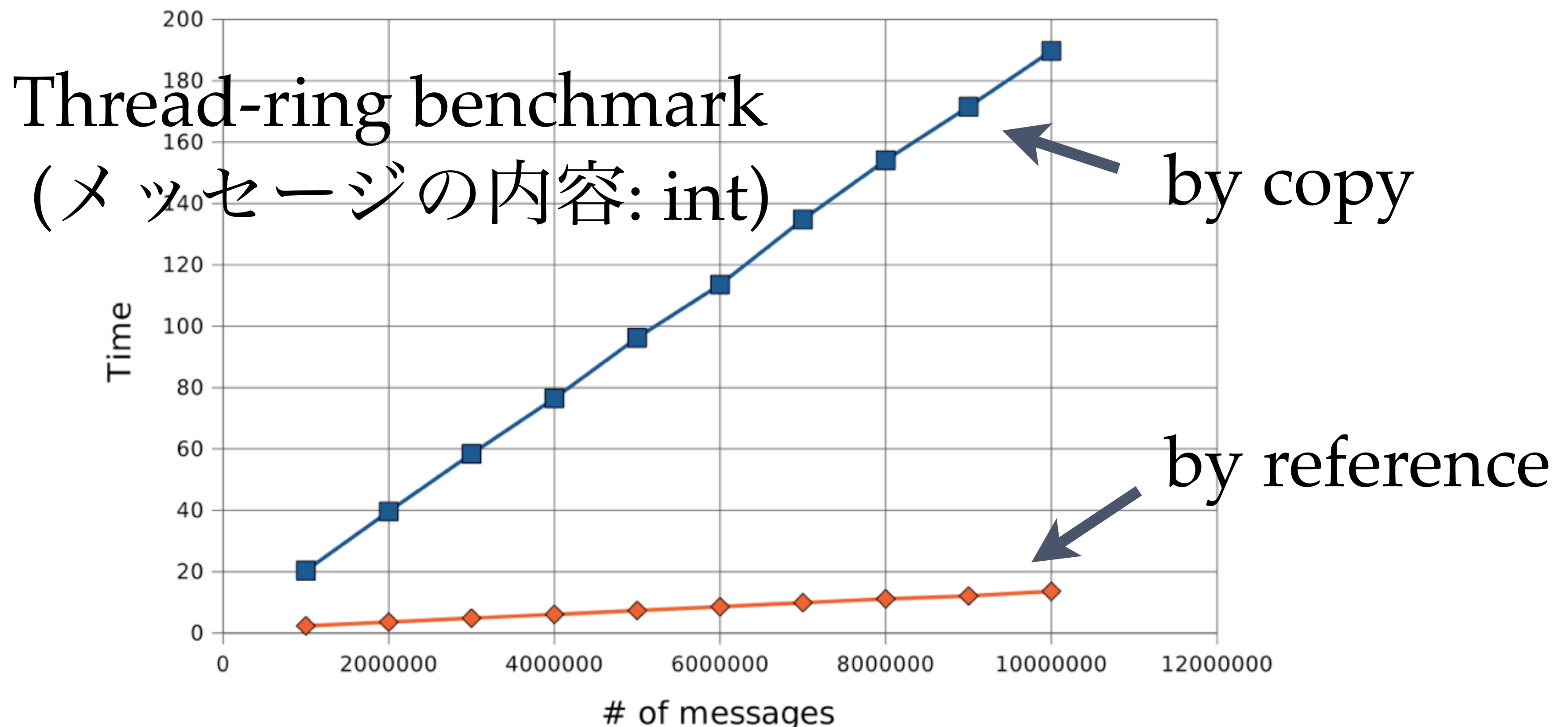
3. 実装戦略と評価

スケジューラ



Zero-copy messaging (on SMP)

- Immutable であることが明らかなオブジェクトの送信時に参照を渡すように変更



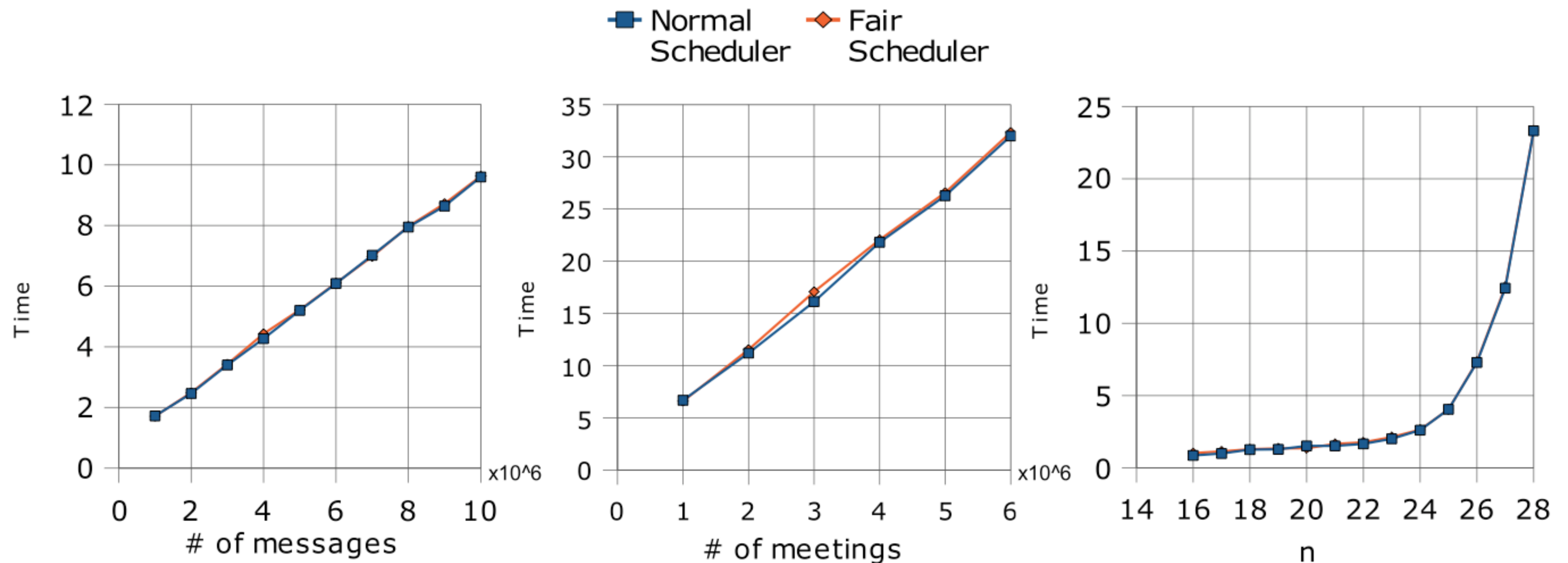
公平なスケジューリングの実装

- Monitoring thread の導入
    - Actor が dispatch されているか定期的に調べ、されていなければ新しい worker thread を生成する
    - トレードオフ:
        - 判定の間隔が短く、処理が粗粒度である場合: 誤って worker を生成する可能性, 判定のコスト増
        - 判定の間隔が長い場合: アプリのレスポンスが悪くなる

3. 実装戦略と評価

公平なスケジューリングのコスト

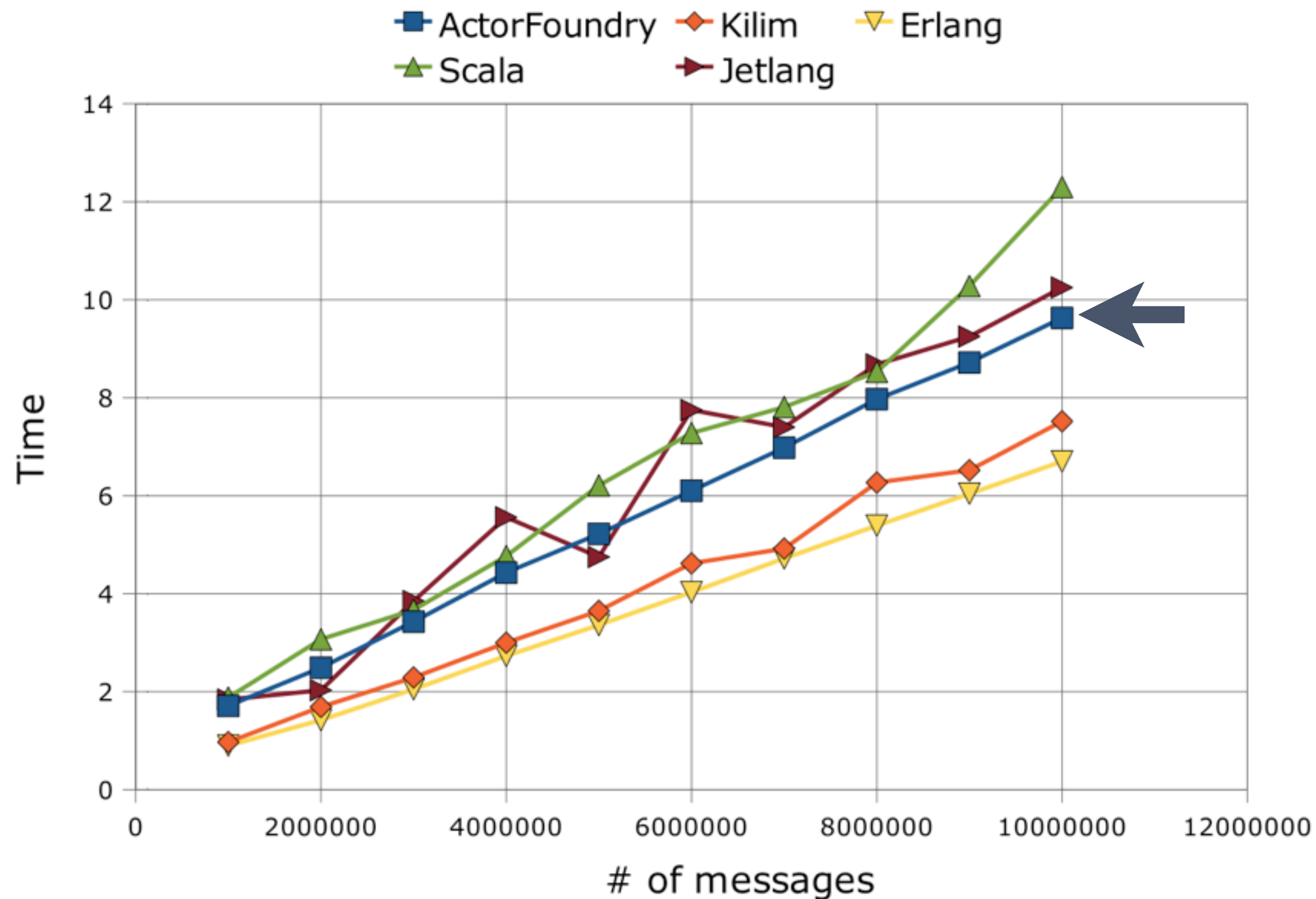
- オーバーヘッドは無視できる程度



(a) thread-ring (b) Chameneos-redux (c) Naive fibonacci

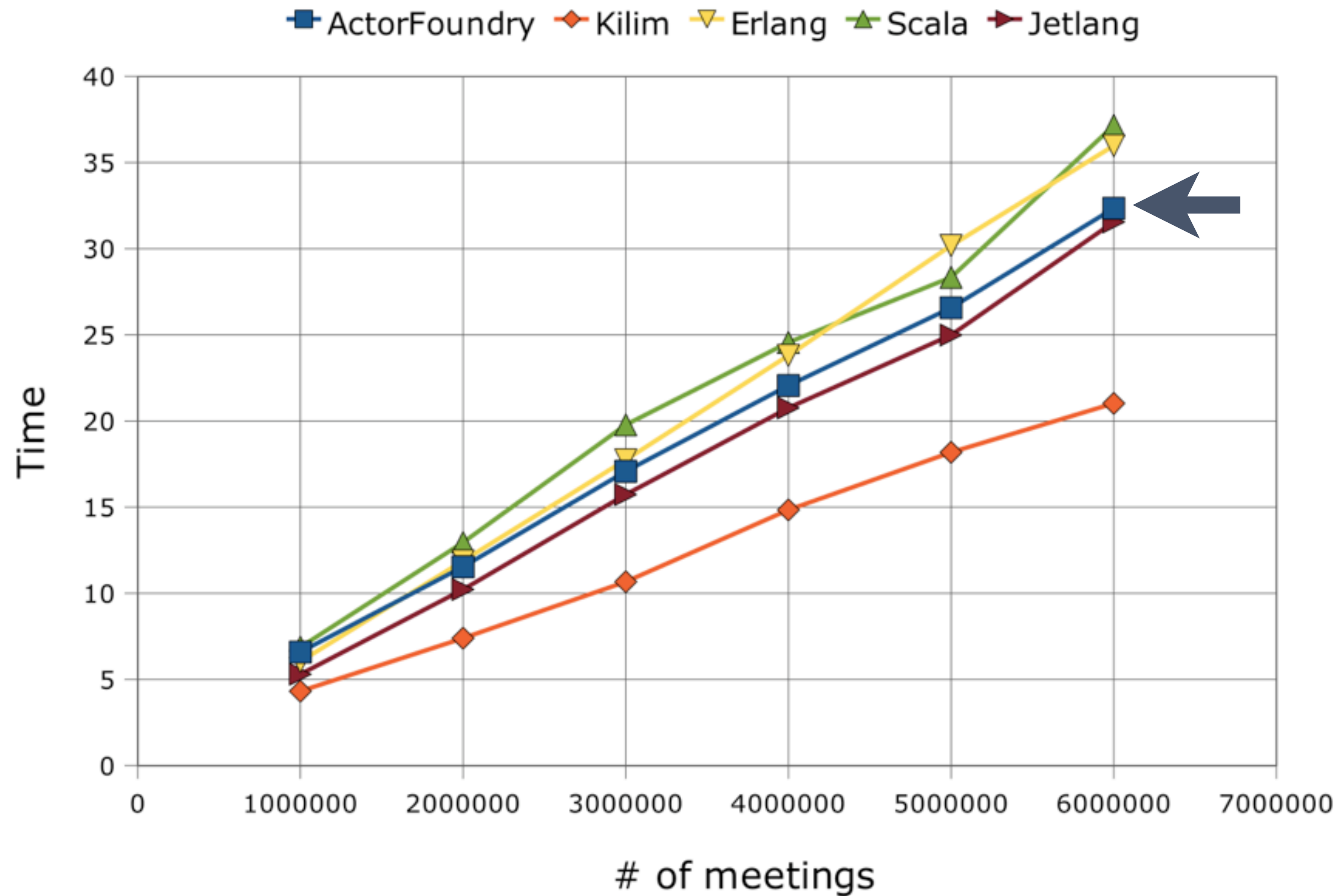
3. 実装戦略と評価

性能比較 (thread-ring)



3. 実装戦略と評価

性能比較 (Chameneos-redux)



Future work

- 位置透過性と移動性によるコストの解析
- より効率のよい messaging
  - 静的なプログラム解析
- 同じ場所に存在する Actor 間の通信と同期を改善できないか