軽量で抽象度の高い条件付きバリア同期とその実装方法

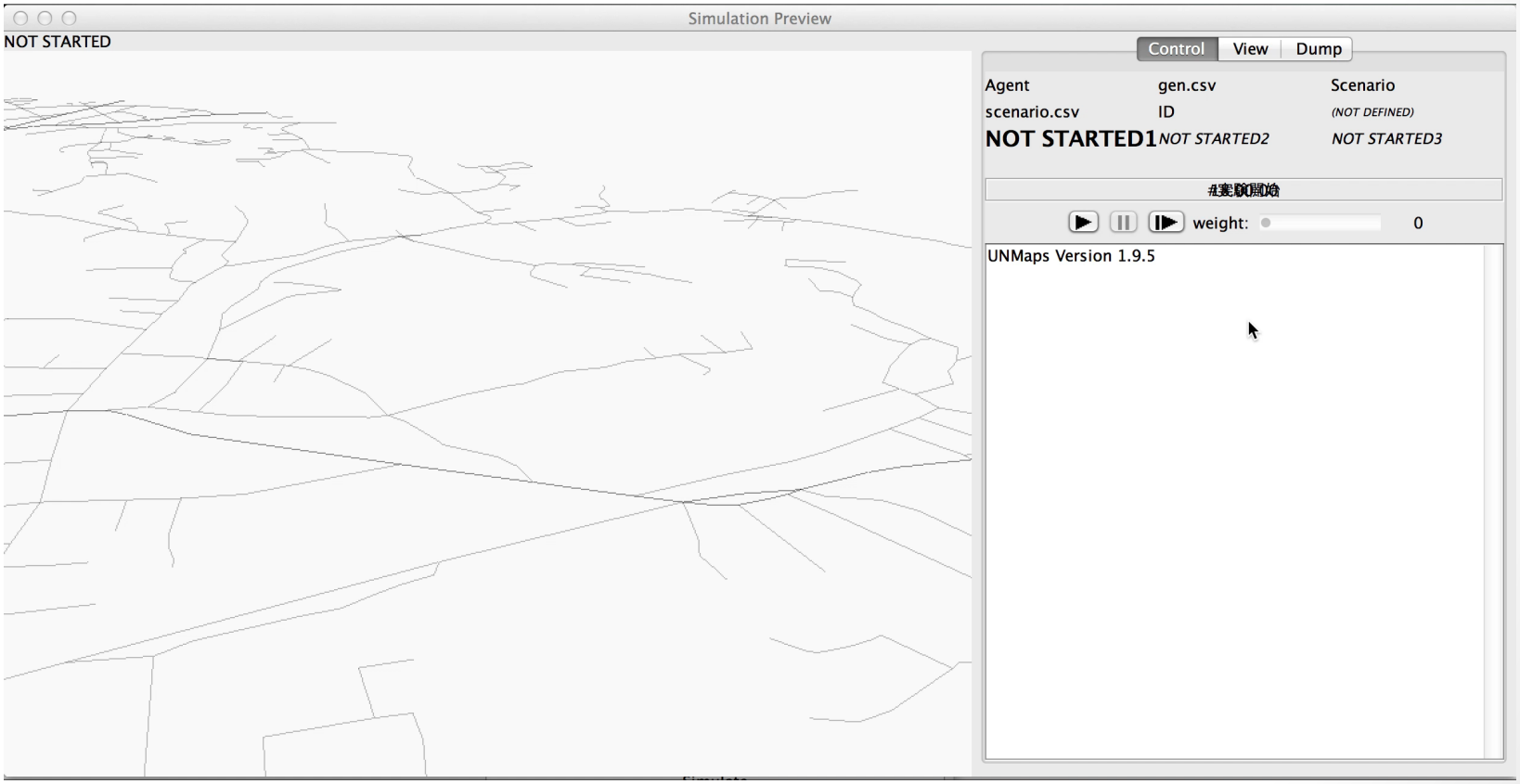
東京大学

Core Software Group

夏 澄彦, 佐藤 芳樹, 千葉 滋

CrowdWalk

- 産業総合研究所で開発されている歩行者シミュレータ
  - 大規模な屋外イベントや災害時の避難経路の最適化
- 歩行者は単位時間ごとに移動を繰り返し、ゴールに向けて進む



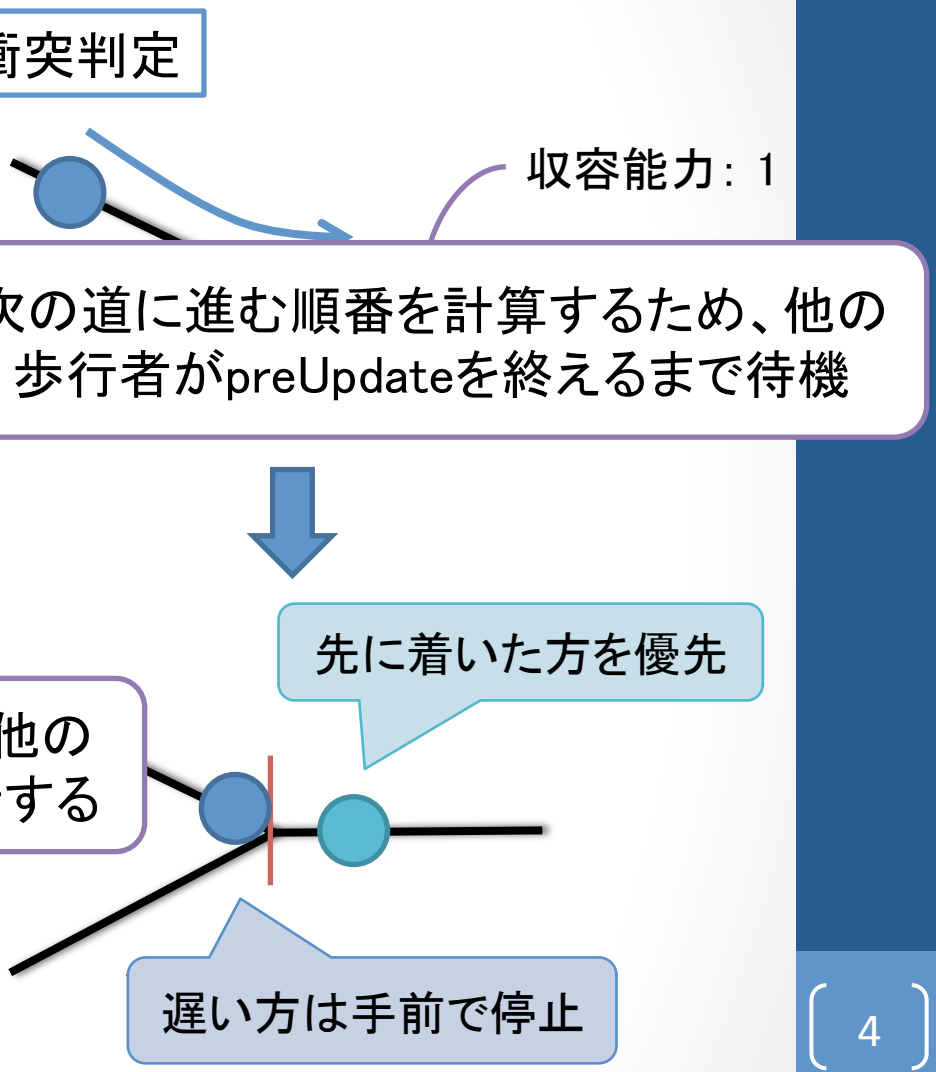
エージェントシミュレーションの並列化の難しさ

- 並列化による高速化が必要
  - 現在のCrowdWalkは逐次処理で、実時間の50倍（1,000人規模）
  - 例）北千住： 歩行者 60,000人

- 並列化を阻害する特徴
  - 社会科学者はエージェントごとにコードを書きたい
    - 現実世界の行動主体をモデル化しており、可読性や保守性を向上できるため
  - エージェント間の依存関係が複雑
    - 相互作用を及ぼしながらシステム全体の振る舞いを調べるため

CrowdWalkの歩行者の挙動

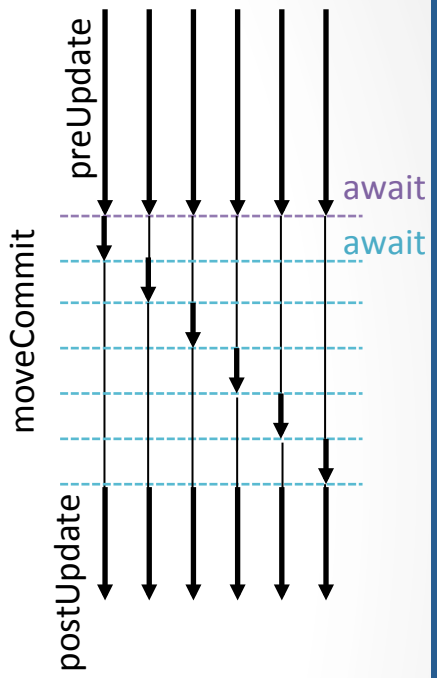
```
class Agent {
  void update() {
    // 速度や次の場所の計算
    preUpdate();

    // 位置更新(衝突判定つき)
    moveCommit();

    //
    po
  }
}
```



バリア同期を用いたCrowdWalk

```
class Agent {
  void update() {
    preUpdate();
    barrier.await();
    for(Agent agent agents.sort()) {
      if (agent == this) {
        moveCommit(); }
      barrier.await();
    }
    postUpdate();
  }
}
```

他の歩行者がpreUpdate
を実行し終えるまで待機



moveCommitを適切な順番で実行できる
ように、全体で同期しながら実行する

バリア同期を用いたCrowdWalk

```
class Agent {
  void update() {
    preUpdate();
    barrier.await();
    for(Agent agent agents.sort()) {
      if (agent == this) {
        moveCommit(); }
      barrier.await();
    }
    postUpdate();
  }
}
```

別の道に移動しない歩行者は
同期の必要がない

実際に衝突する歩行者同士で
同期すれば十分

同期時の過剰な待機スレッド増加
による性能低下

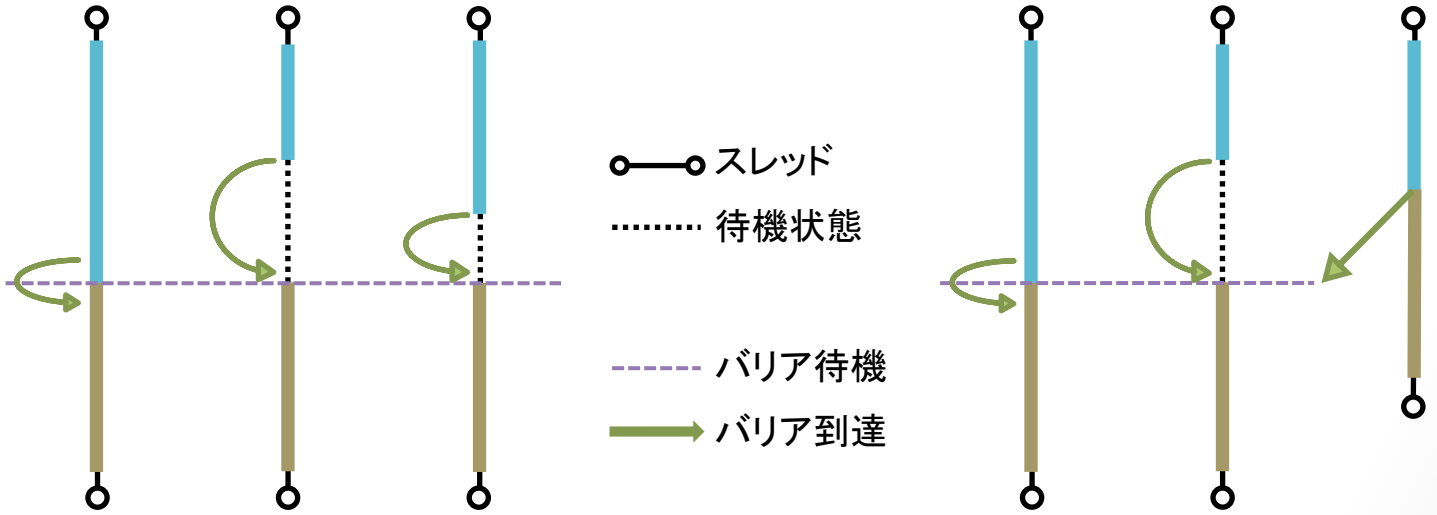
バリア同期処理の分割による性能向上

All-to-All型

- OpenMP's barrier (C, Fortran)、CyclicBarrier (Java)
- 全スレッドが同期時に待機する
- **記述が容易だが、性能が悪い**

Point-to-Point型

- CountDownLatch (Java)、Phaser (X10, Java)
- バリア同期を待機と到達に分割可能
  - 待機するかどうかをスレッド開始後に決定可能
- **記述が複雑だが、性能が良い**



Point-to-Point型を用いたCrowdWalk

```
class Agent {
  Phaser phaser = new Phaser(1);

  void update() {
    preUpdate();
    phaser.arrive();
    if (!goesOnSameStreet()) {
      neighbors().forEach(a ->
        a.phaser.awaitAdvance(0));
      conflictings().headSet(this).forEach(a ->
        a.phaser.awaitAdvance(1)); }
    moveCommit();
    phaser.arrive();
    postUpdate();
  }
}
```

別の道に移動しない場合には到達処理だけで、待機処理は行わない

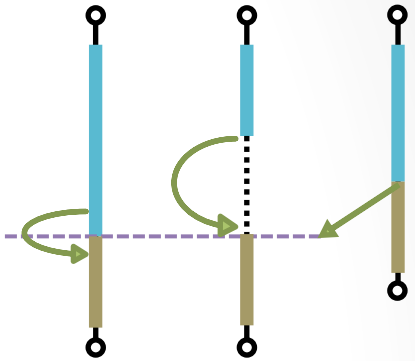
実際に衝突する歩行者を計算するため近隣の歩行者と同期

実際に衝突する歩行者とのみ同期

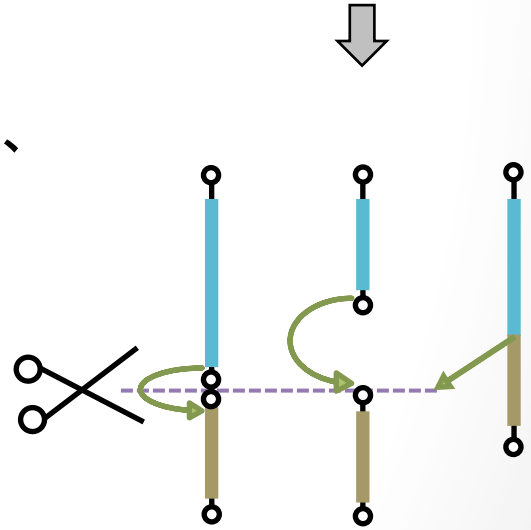
バリア到達を発行すべき箇所をプログラマが明示的に記述する必要があり、モジュラリティが低下

スレッド分割による性能向上

- 待機スレッドによる資源占有が問題
  - シミュレーションサイズが大きくなるほど重大
  - デッドロックが生じやすくなる
    - e.g. スレッドプールも使えない



- スレッド分割
  - 同期前後のコードを別スレッドに分割すれば待機スレッドを減らせる
  - 既存コードの継続渡しスタイルへの変換や、それに伴うローカル変数の退避・復元が別途必要



プログラマへの大きな負担

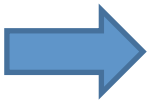
提案：バイトコード変換による軽量な条件付きバリア同期

Waitless

- 抽象度の高いAll-to-All型のバリア同期機構
  - バリア待機の条件付き
  - `barrier(agents, a -> a.nextPlaces[i] != null)`

実行時

- 条件式を元にバリア到達箇所を解析し、Point-to-Point型に変換
- バリア同期の呼び出し前後でスレッド分割



モジュラリティと実行性能を両立したバリア同期

条件付きバリア同期API

- barrier(Set<E> 同期対象, Predicate<E> 条件式)
  - 指定した全ての同期対象が条件式を満たすまで待機

    barrier(agents, a -> a.nextPlaces[i] != null)

- これを利用すると様々な同期処理を実装できる

例)逐次実行

- ordered(SortedSet<E> 同期対象, Predicate<E> 条件式,
  Runnable 逐次処理内容)

```
void ordered(SortedSet<T> targets, Predicate<T> pred, Runnable seq) {
  SortedSet<T> head = targets.headSet(item);
  if (!head.isEmpty()) barrier(head, pred, seq);
}
```

- 条件式の評価と制限
  - 同期対象オブジェクトのフィールド更新時に条件式が評価される
    - 副作用を伴う条件式や、同期対象オブジェクトのフィールド以外の変更によって評価結果が変わる条件式はバリア同期が適切に解除されない可能性がある
  - リフレクションの利用を禁止

Waitlessを用いたCrowdWalk

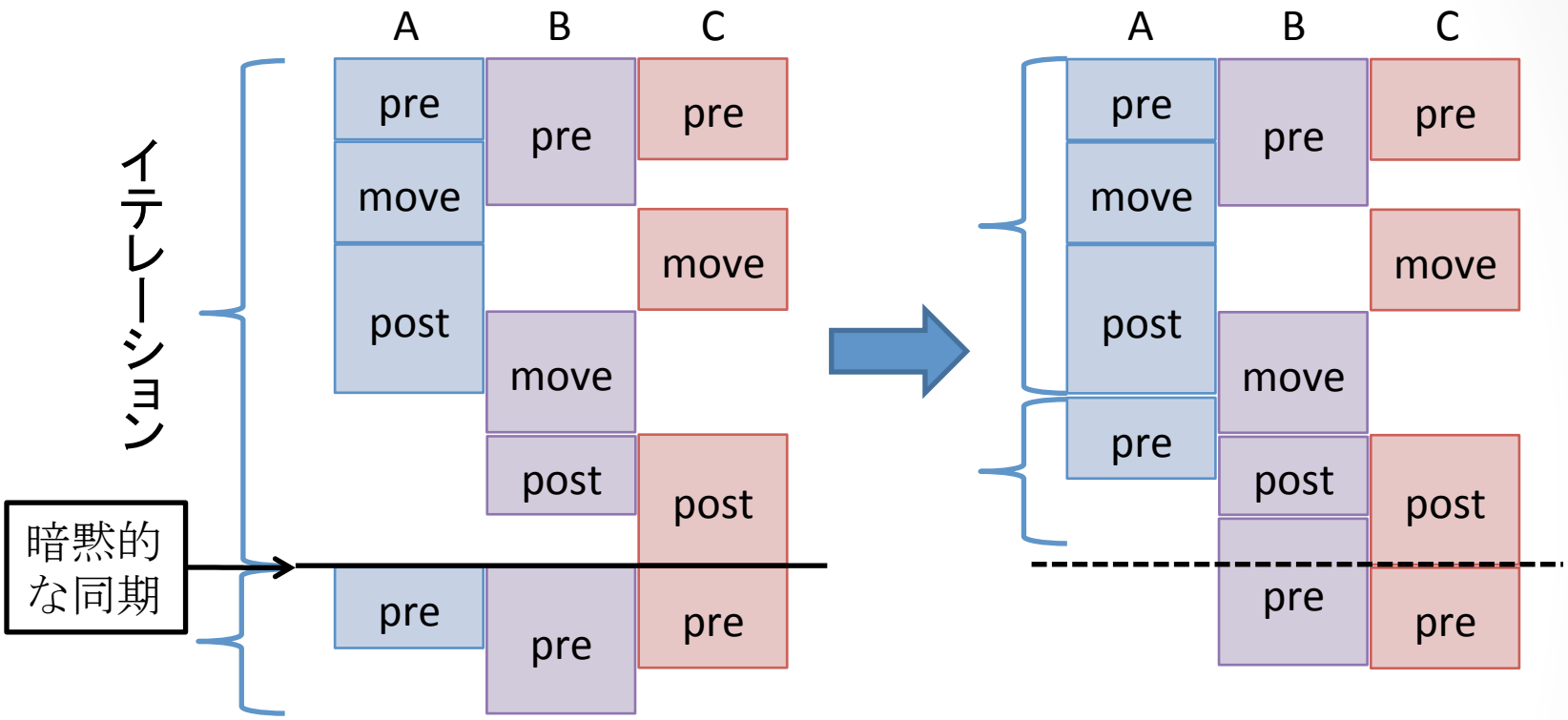
```
class Agent {
  void update(Waitless<Agent> w) {
    preUpdate();
    if (goesOnSameStreet()) {
      moveCommit();
    } else {
      w.barrier(neighbors(),
                a -> a.nextPlaces[i] != null);
      w.ordered(conflictings(),
                a -> a.places[i] != null,
                () -> moveCommit());
    }
    postUpdate();
  }
}
```

近くの歩行者が次に進む場所を計算し終えるまで待機

実際に衝突する歩行者に対して適切な順番で逐次実行を行う

抽象度の高いAll-to-All型の記述で、Point-to-Point型と同じくらい柔軟な同期制御を実現

時間ブロッキングのサポート

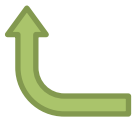


- 道を移動しない歩行者は他の歩行者を待たずに、次のイテレーションに進める
- シミュレーション全体の同期回数を削減できる

時間ブロッキングを用いたCrowdWalk

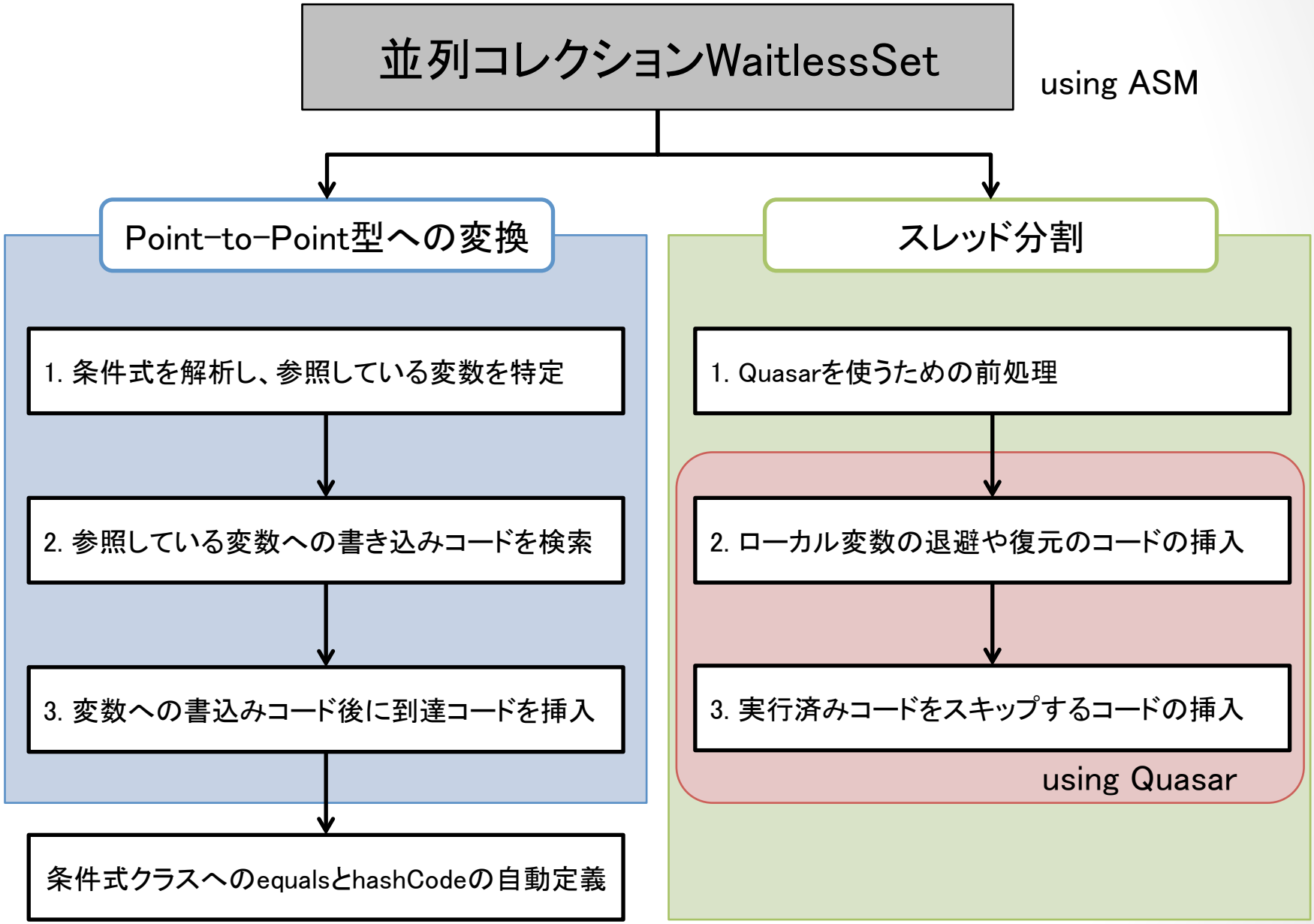
```
class Agent {
  void update() {
    preUpdate();
    if (goesOnSameStreet()) {
      moveCommit();
    } else {
      barrier(all, a -> a.i >= i);
      barrier(neighborAgents(),  a -> a.nextPlaces[i] != null);
      ordered(conflictingAgents(), () -> moveCommit());
    }
    postUpdate();
  }

  static void main() {
    new WaitlessSet(agents).parallelStream().forEach(agent -> {
      for (int i = 0; i < maxIteration; i++)
        agent.update();
    });
  }
}
```

別の道に移動する場合には、他の歩行者が自分と同じイテレーションになるまで待機する

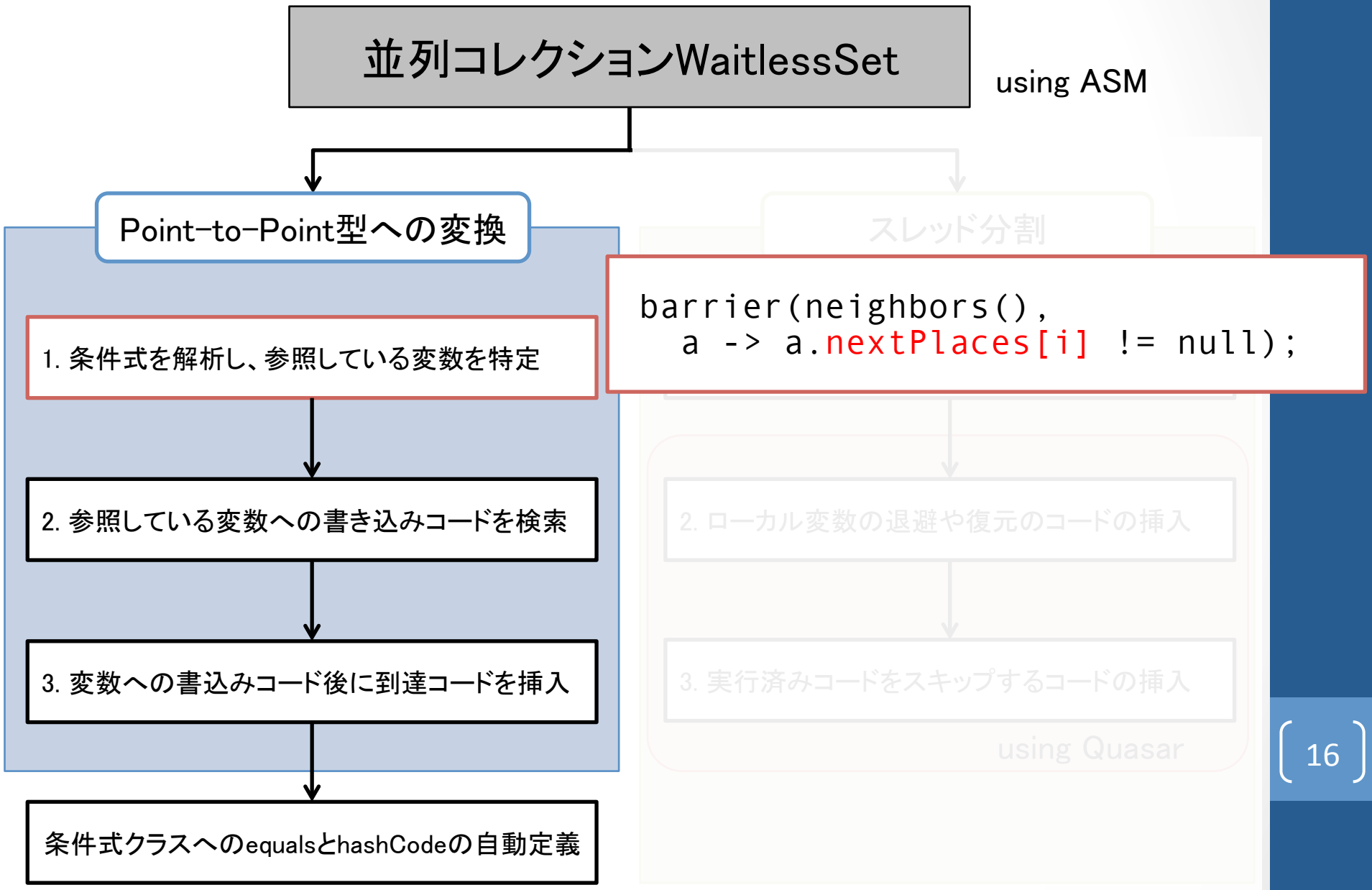


```
for (int i = 0; i < maxIteration; i++) {
  new WaitlessSet(agents).parallelStream()
   .forEach(agent -> agent.update());
}
```

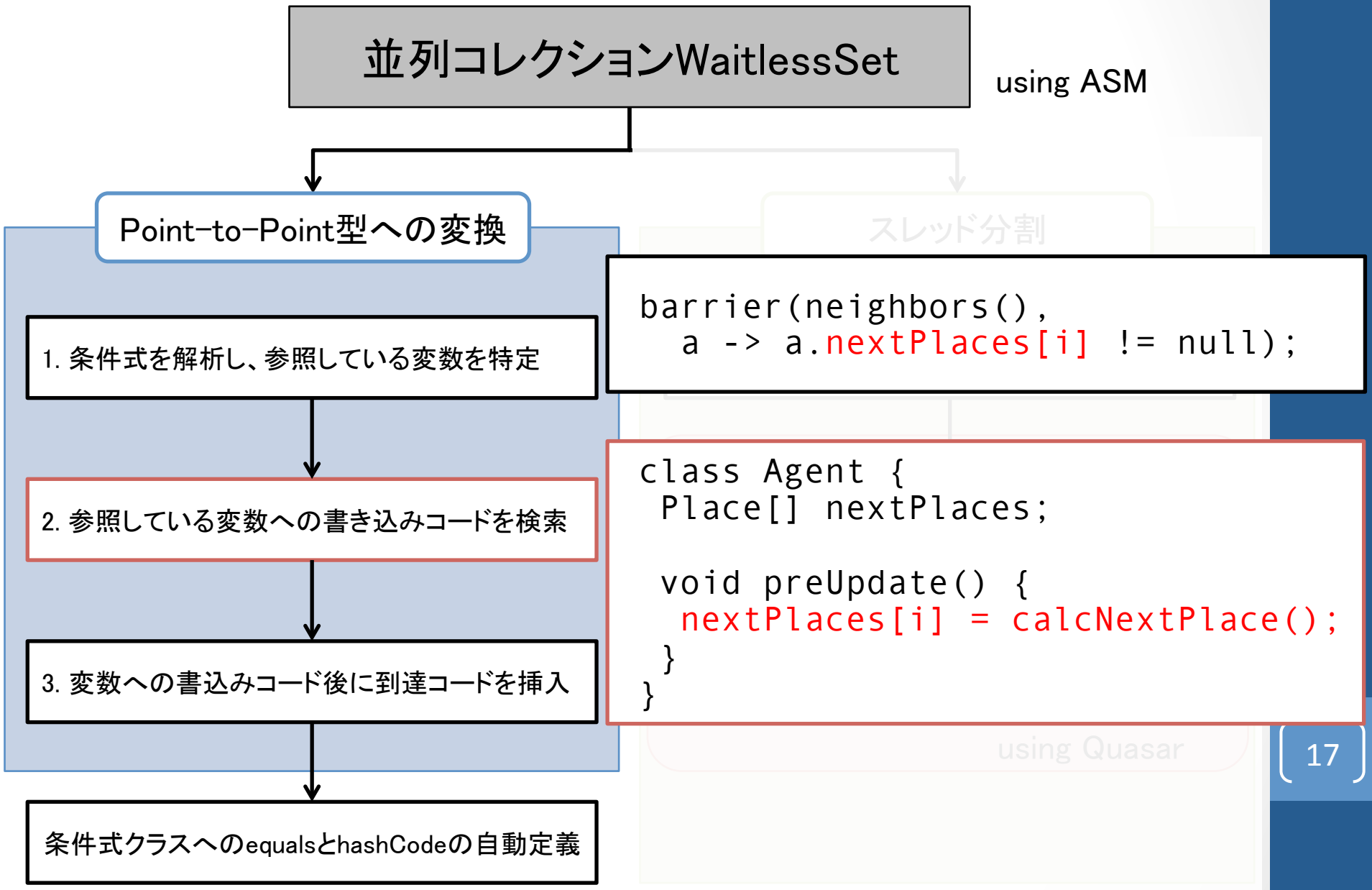
Waitlessによるバイトコード変換



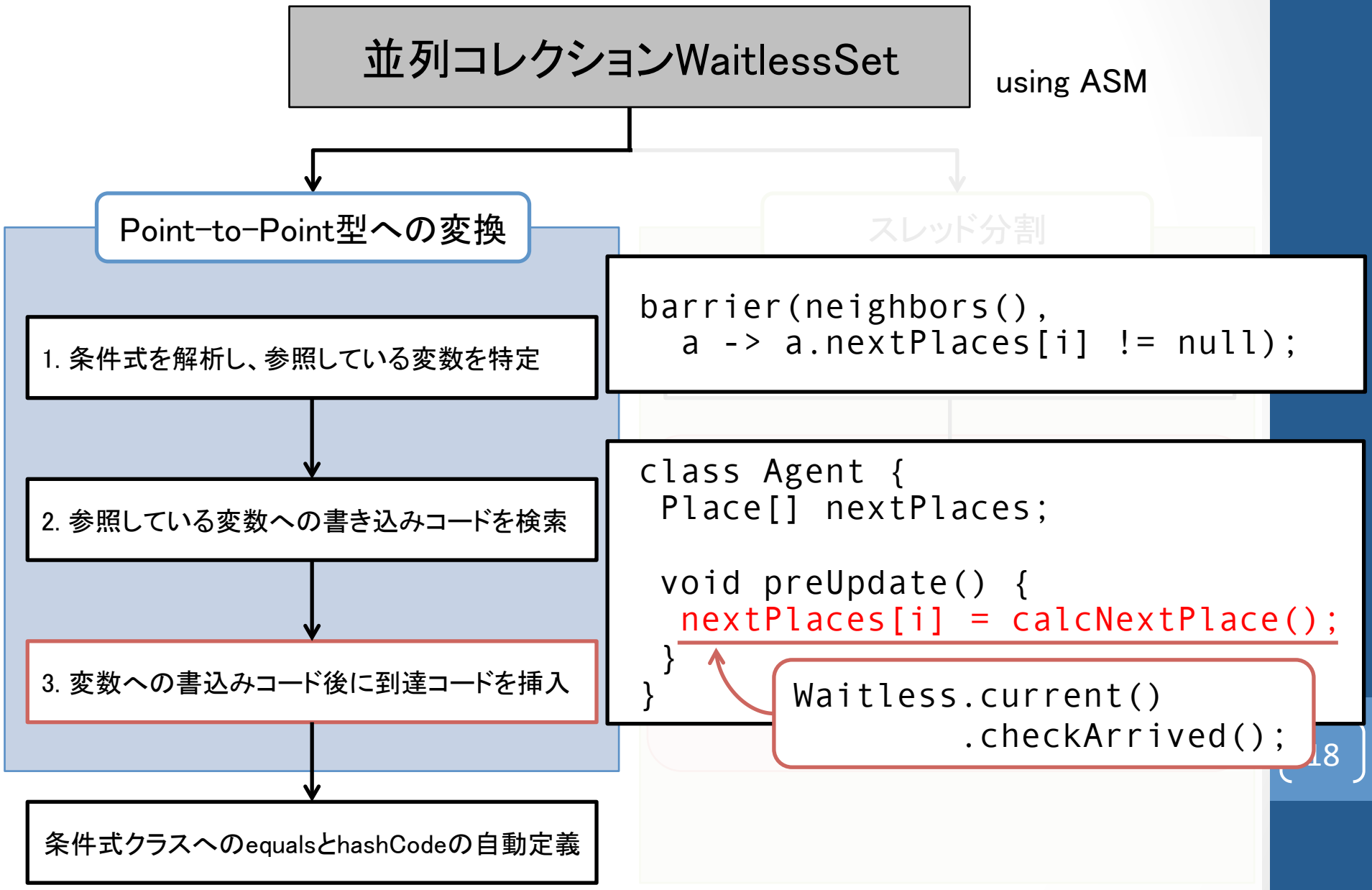
Waitlessによるバイトコード変換



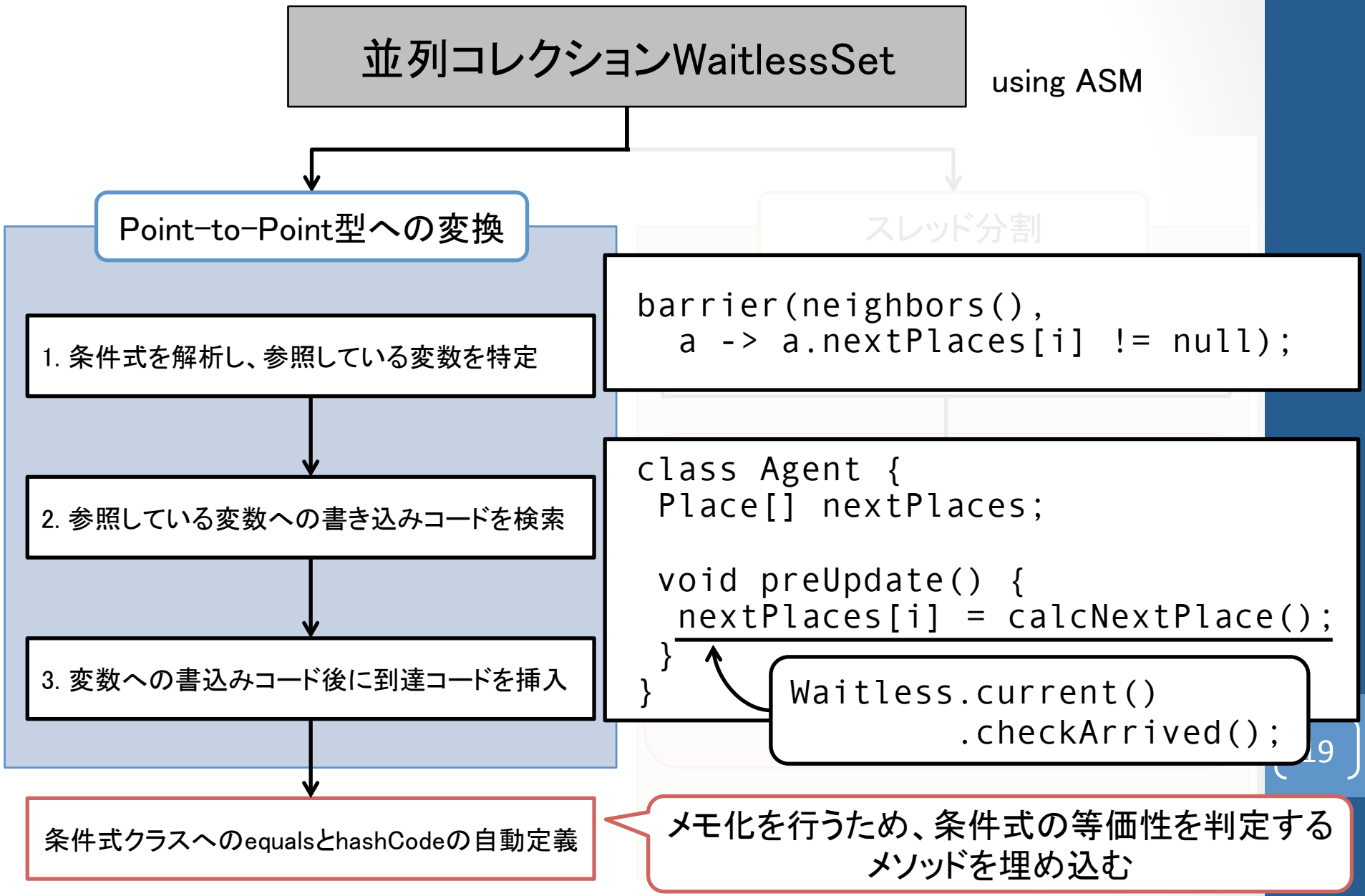
Waitlessによるバイトコード変換



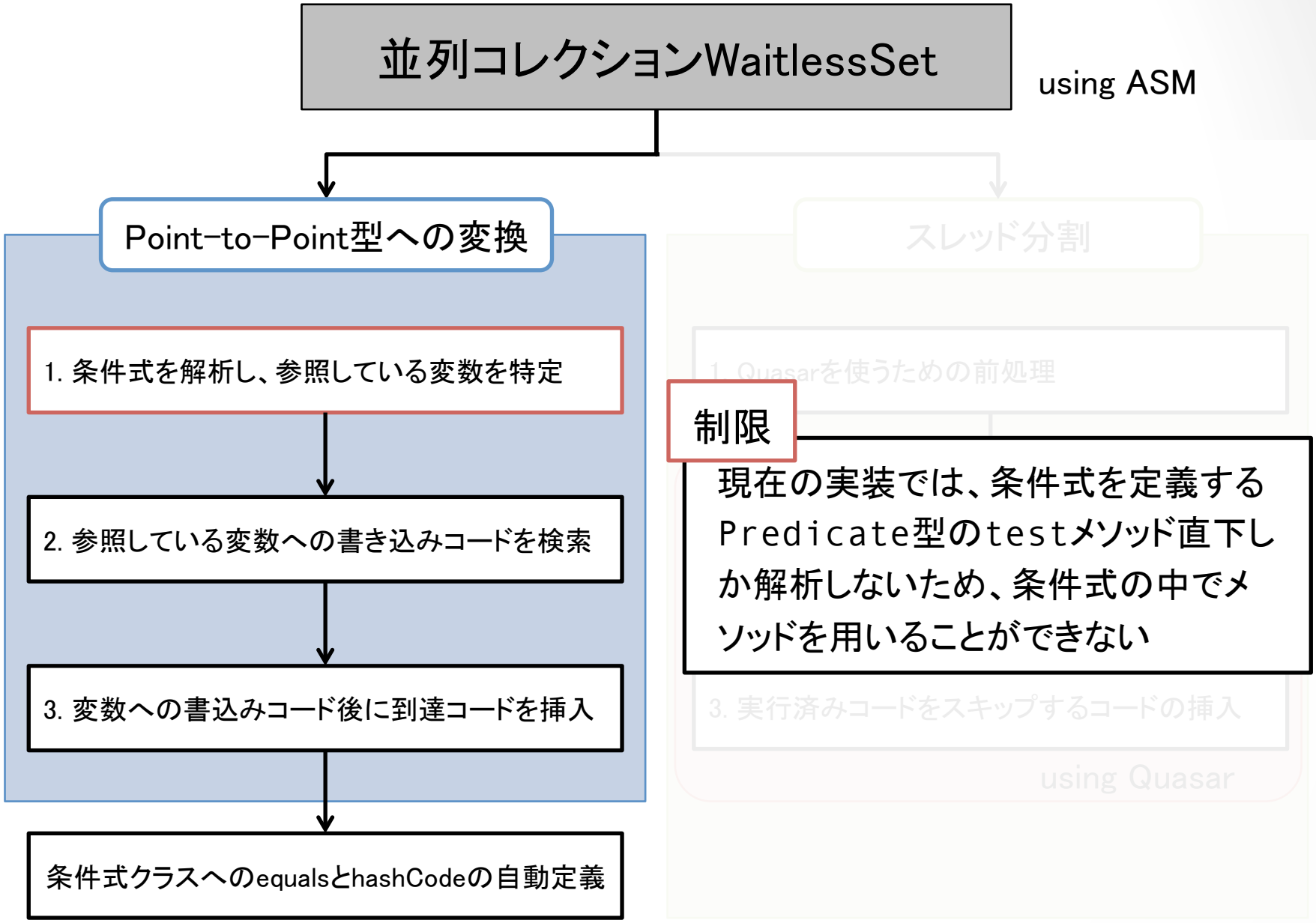
Waitlessによるバイトコード変換



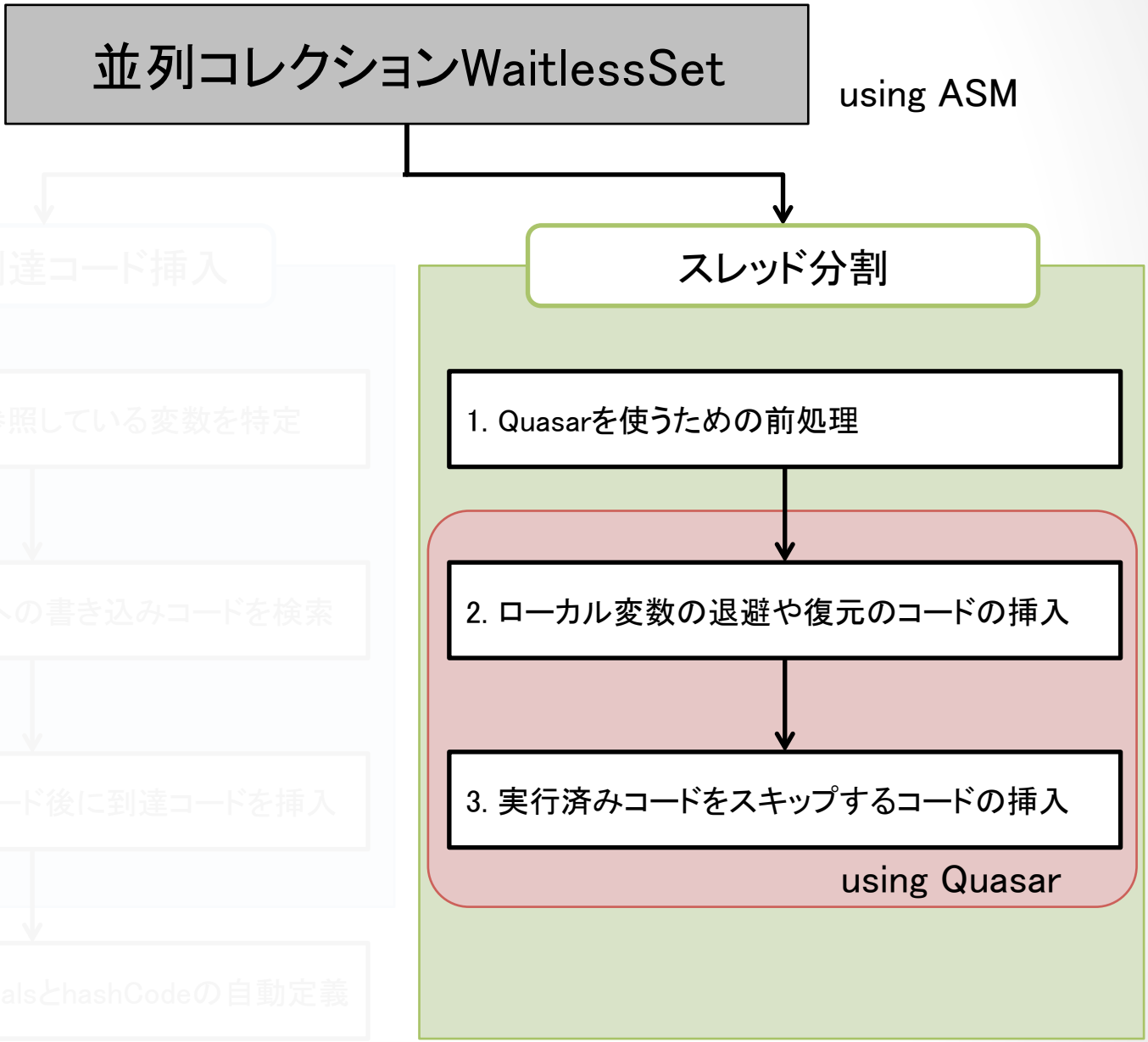
Waitlessによるバイトコード変換



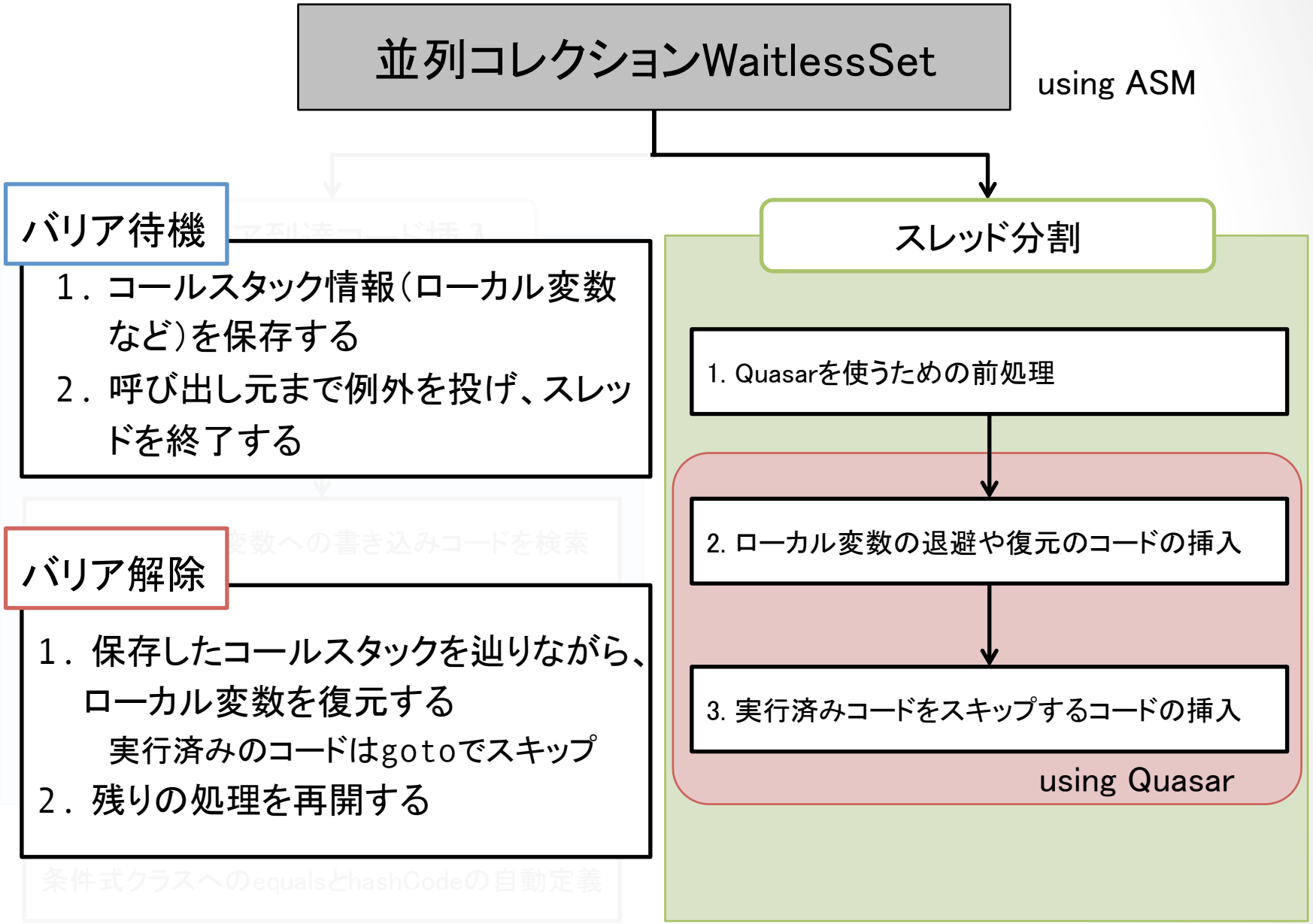
Waitlessによるバイトコード変換



Waitlessによるバイトコード変換



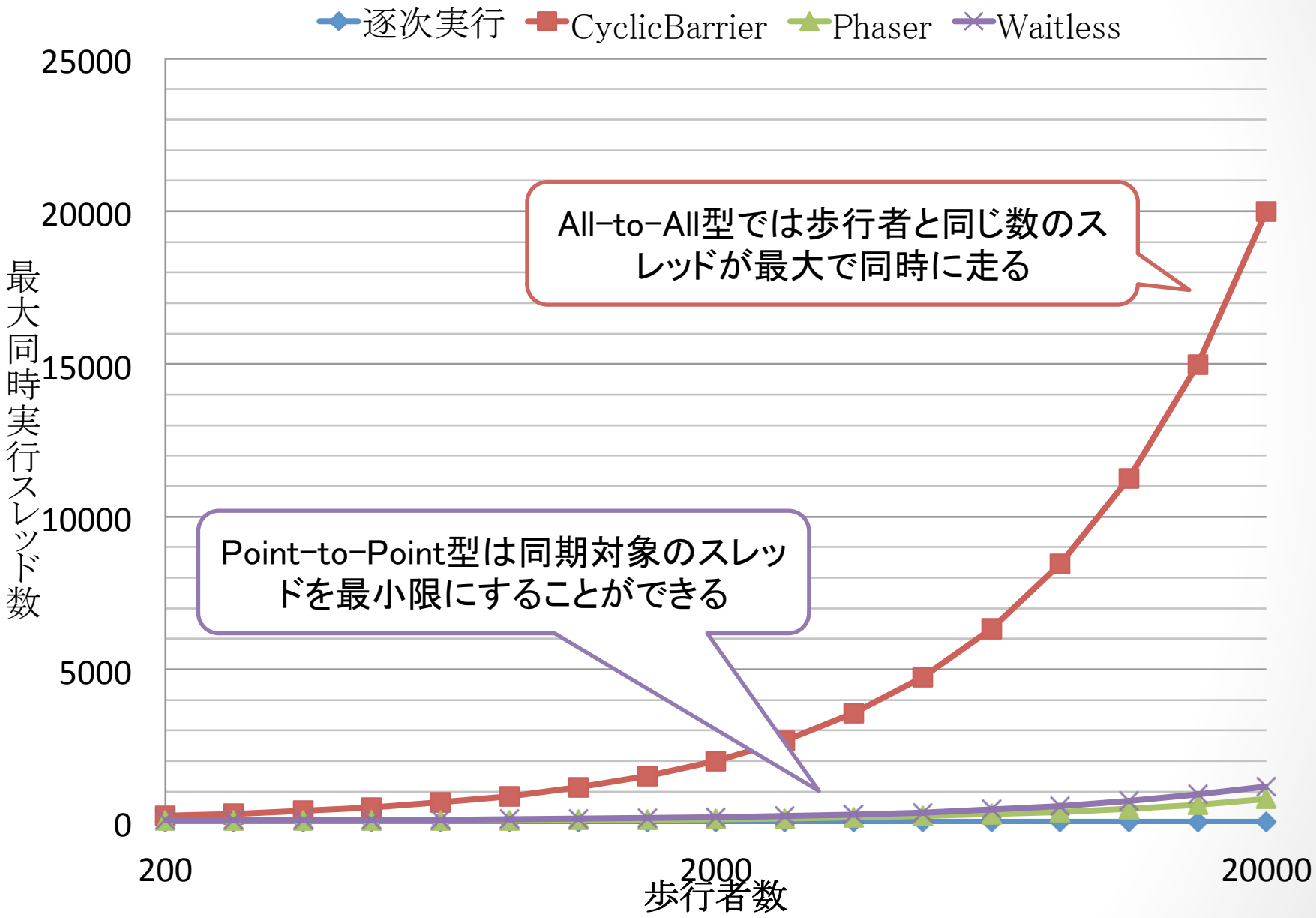
Waitlessによるバイトコード変換



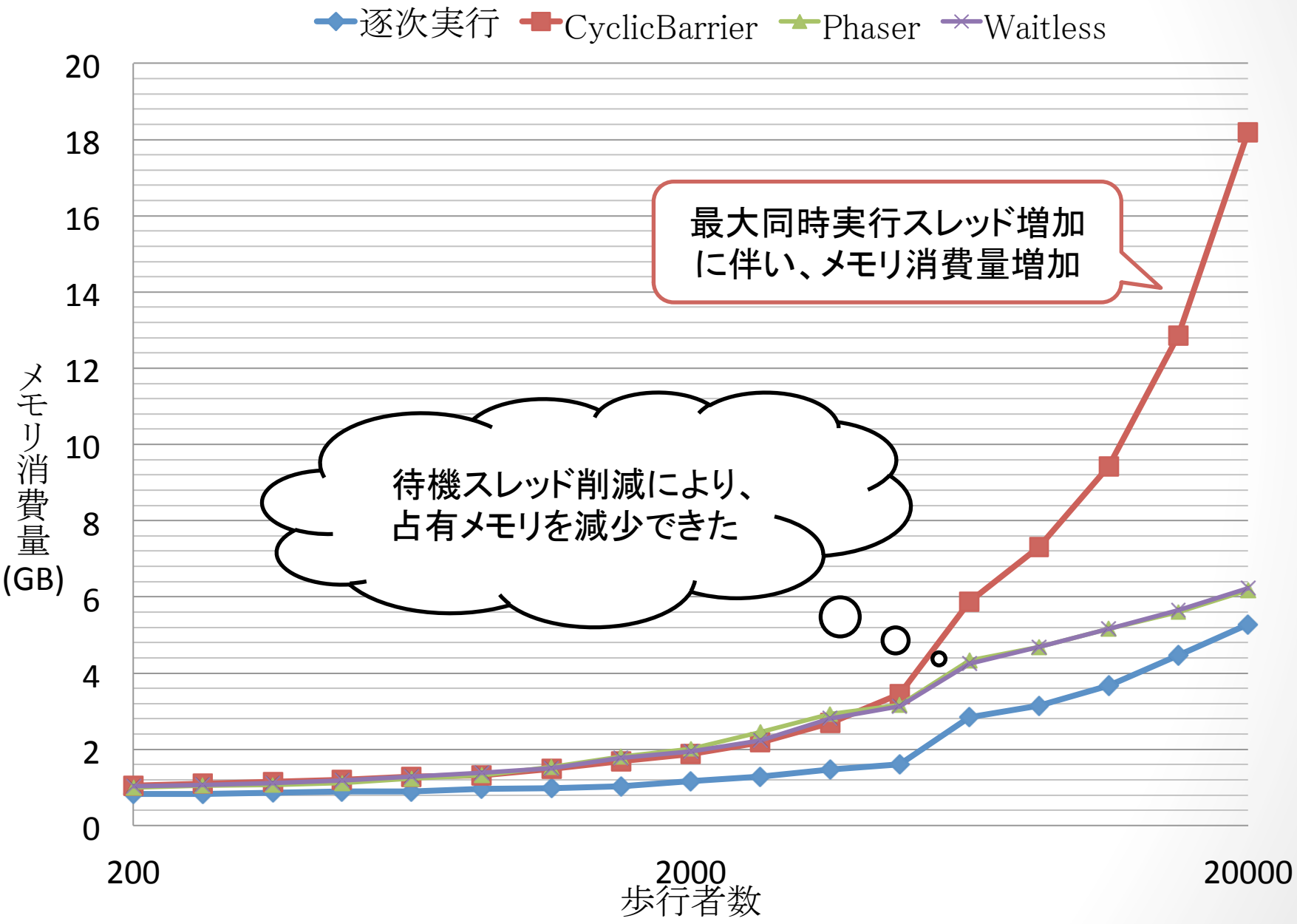
実験

- 目的
  - 開発したバリア同期による実行性能を確認するため
- 内容
  - CrowdWalkのサブセットに対して、歩行者の数を変化させた場合のスレッド数、実行時間、メモリ使用量を測定
  - Java 8で実装
- 環境
  - CPU: Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz
  - メモリ: 64GB
  - JavaVM: Java(TM) SE Runtime Environment (build 1.8.0¥_20-b26) Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
  - 実行オプション: -Xm56g -Xmx56g -Xss512m -XX:+UseG1GC -XX:InitiatingHeapOccupancyPercent=0

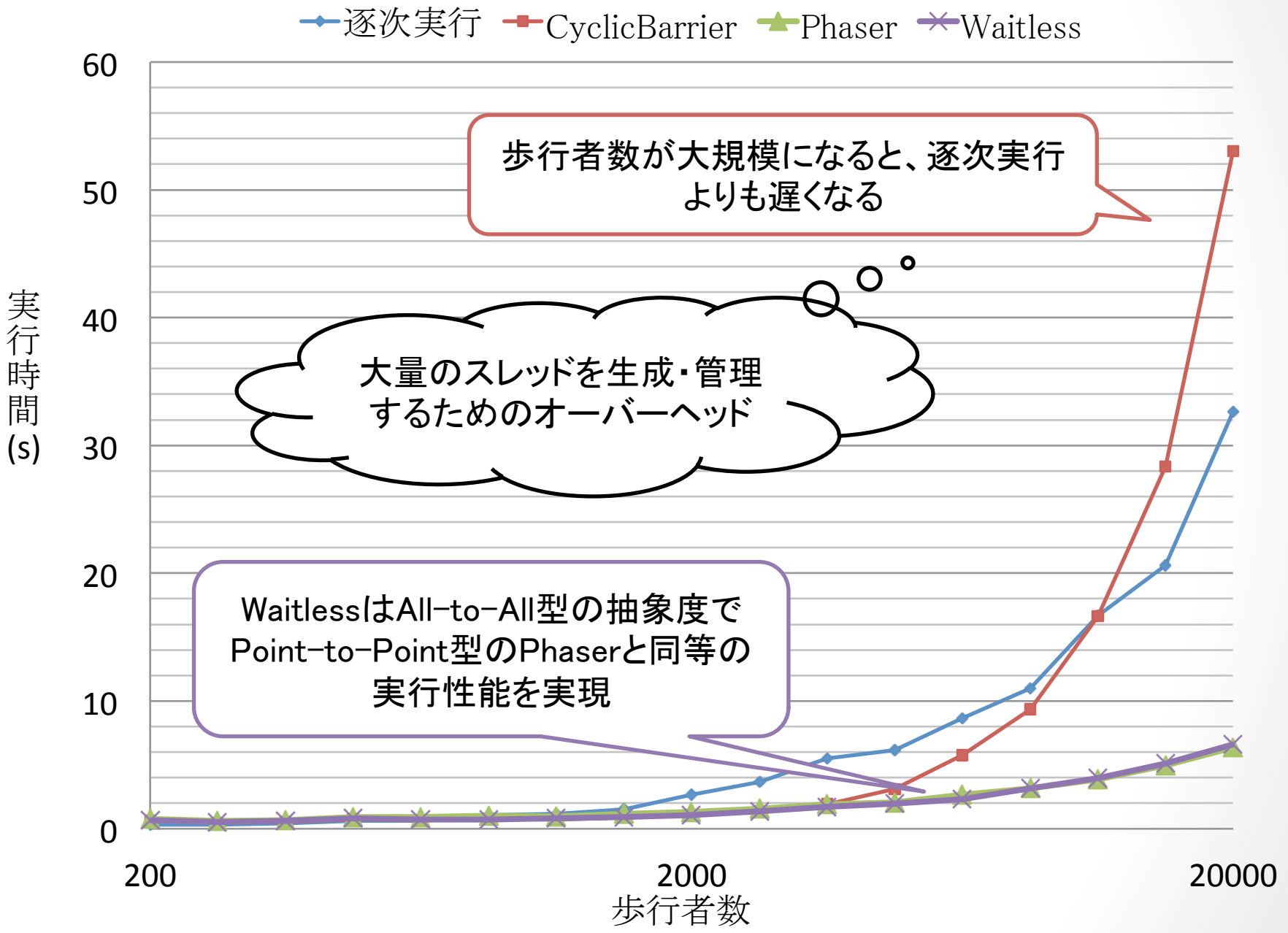
Point-to-Point型化の効果（スレッド数）



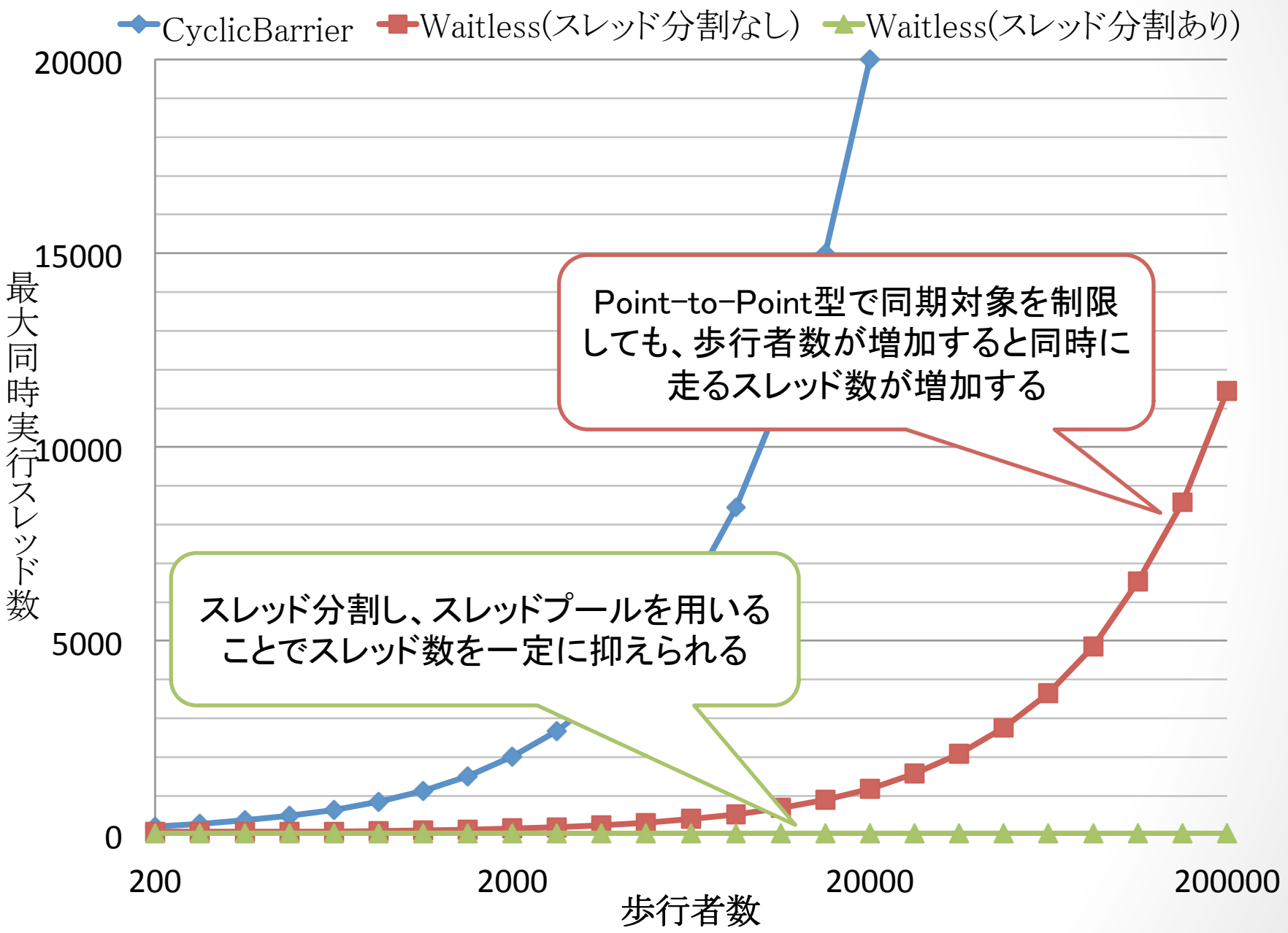
Point-to-Point型化の効果（メモリ消費量）



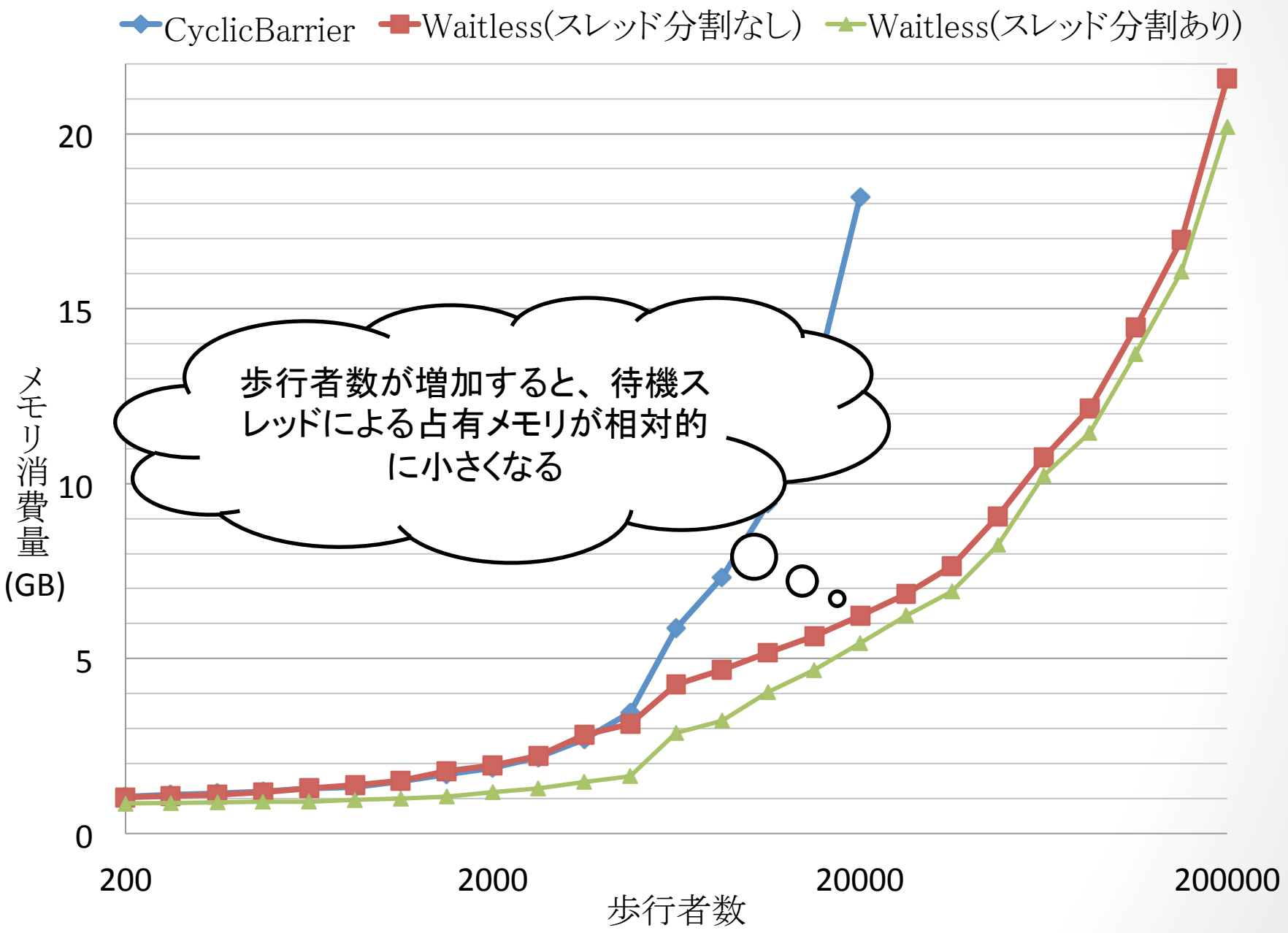
Point-to-Point型化の効果（実行時間）



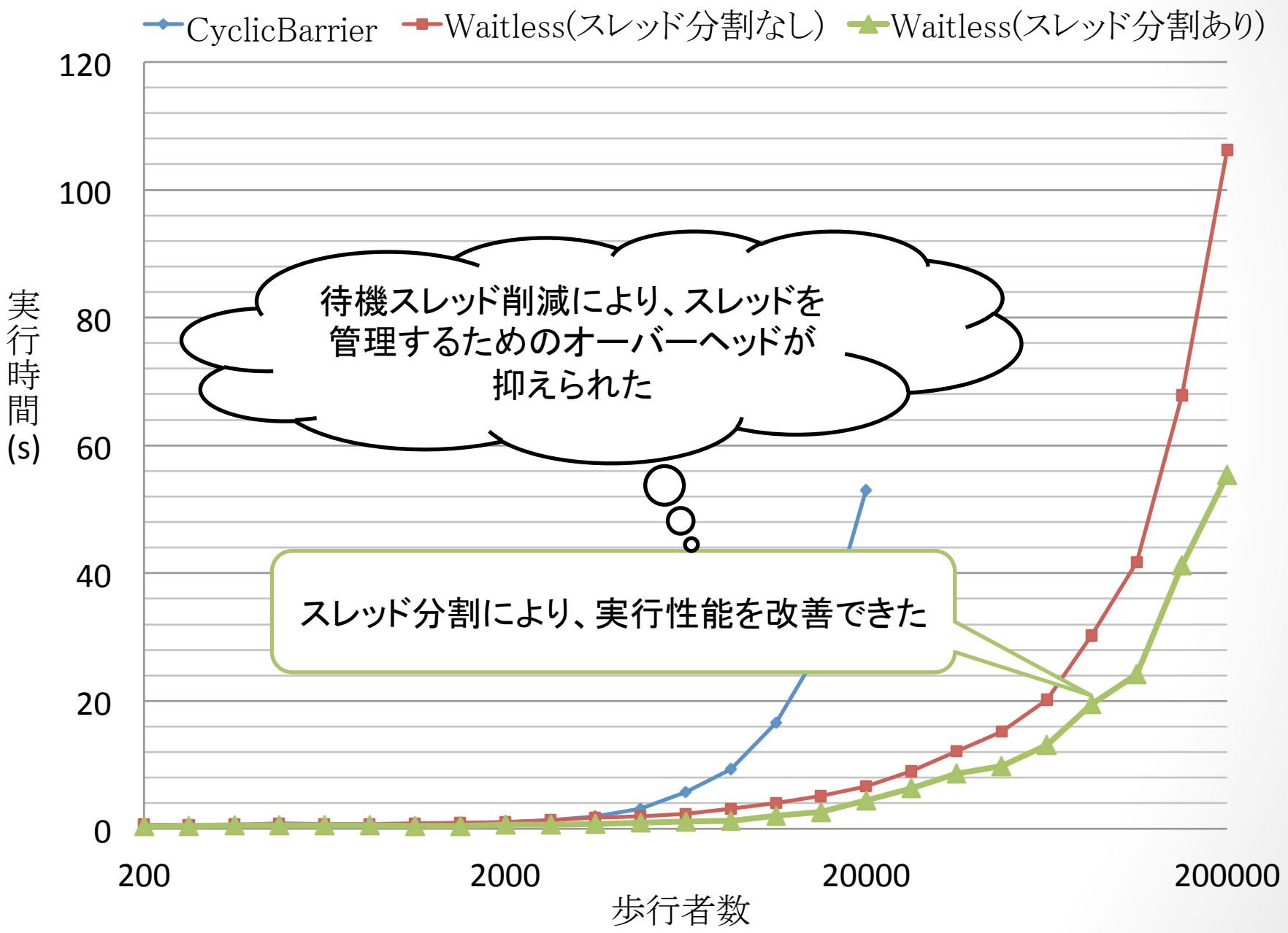
スレッド分割の効果（スレッド数）



スレッド分割の効果（メモリ消費量）



スレッド分割の効果（実行時間）



関連研究

- バリア同期
  - OpenMP
    - コメントを用いることで、容易に記述可能
    - 同期対象や逐次実行順番の動的な指定ができない
  - Phaser [Jun Shirako et al ICS ’08]
    - スレッドは動的に同期グループに参加や脱退が可能
    - 到達コードが散在し、モジュラリティが低下する
- スレッド分割による同期機構の最適化
  - Cooperative Scheduling of Parallel Tasks [Shams Imam et al ECOOP ’14]
    - ロックを用いて実装されたfutureやphaserのような同期機構をスレッド分割により、スケーラビリティを向上させる

まとめ

- 軽量な条件付きバリア同期Waitlessを開発
  - バリア待機を条件式として記述
  - ロードタイムにバイトコード変換
    - 条件式を元にバリア到達コードを自動挿入
    - スレッドの自動分割
  - モジュラリティと実行性能を両立したバリア同期を実現

- CrowdWalkのサブセットを用いて性能評価