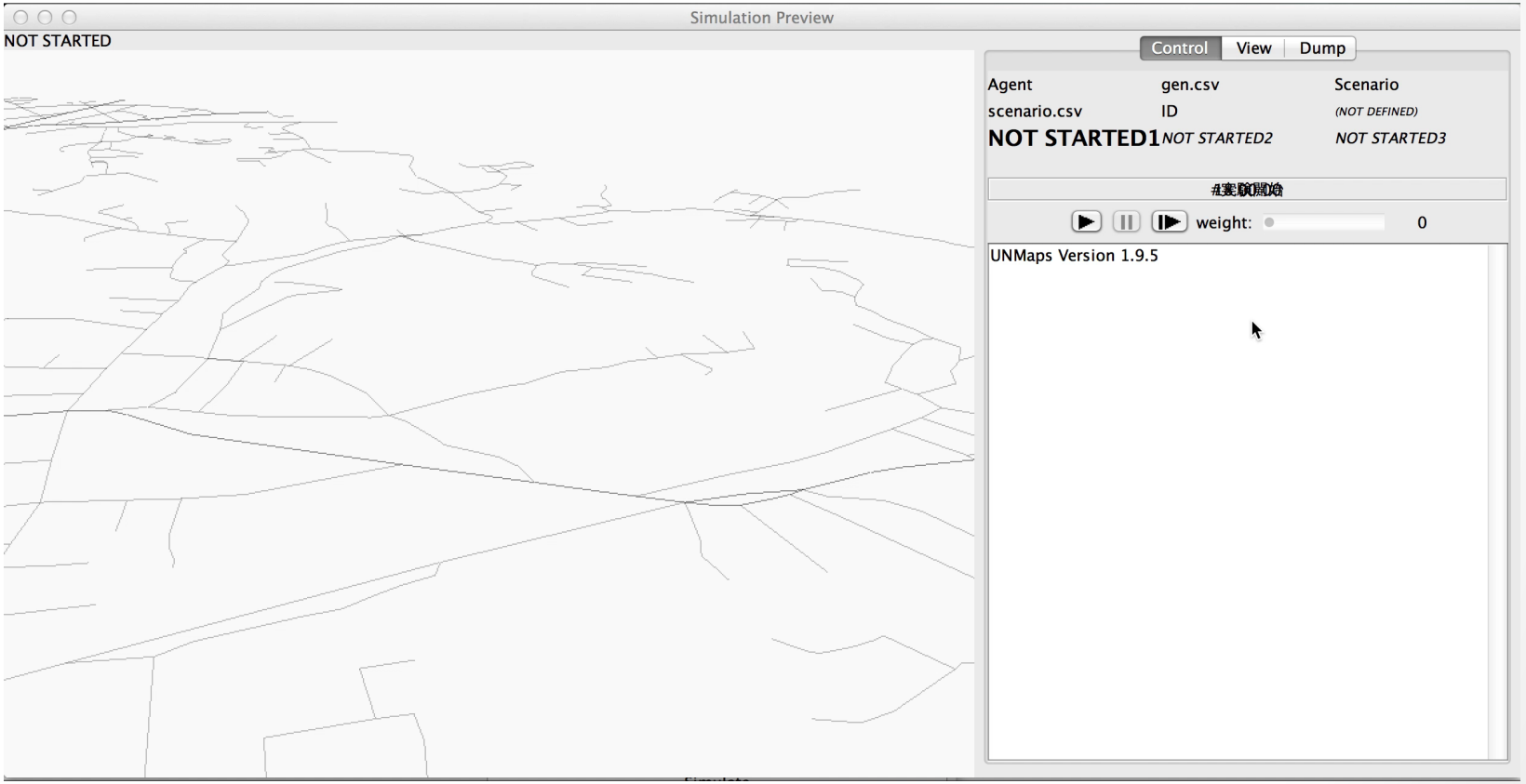
エージェントシミュレーションの並列化のための軽量な条件付きバリア同期の開発

東京大学
Core Software Group
夏 澄彦

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計算を進めるため

1イテレーションの歩行者の処理

```
class Pedestrian {
  void preUpdate() {
    ...
  }
  void moveCommit() {
    ...
  }
  void postUpdate() {
    ...
  }
}
```

独立実行

1. 速度や次の場所の計算

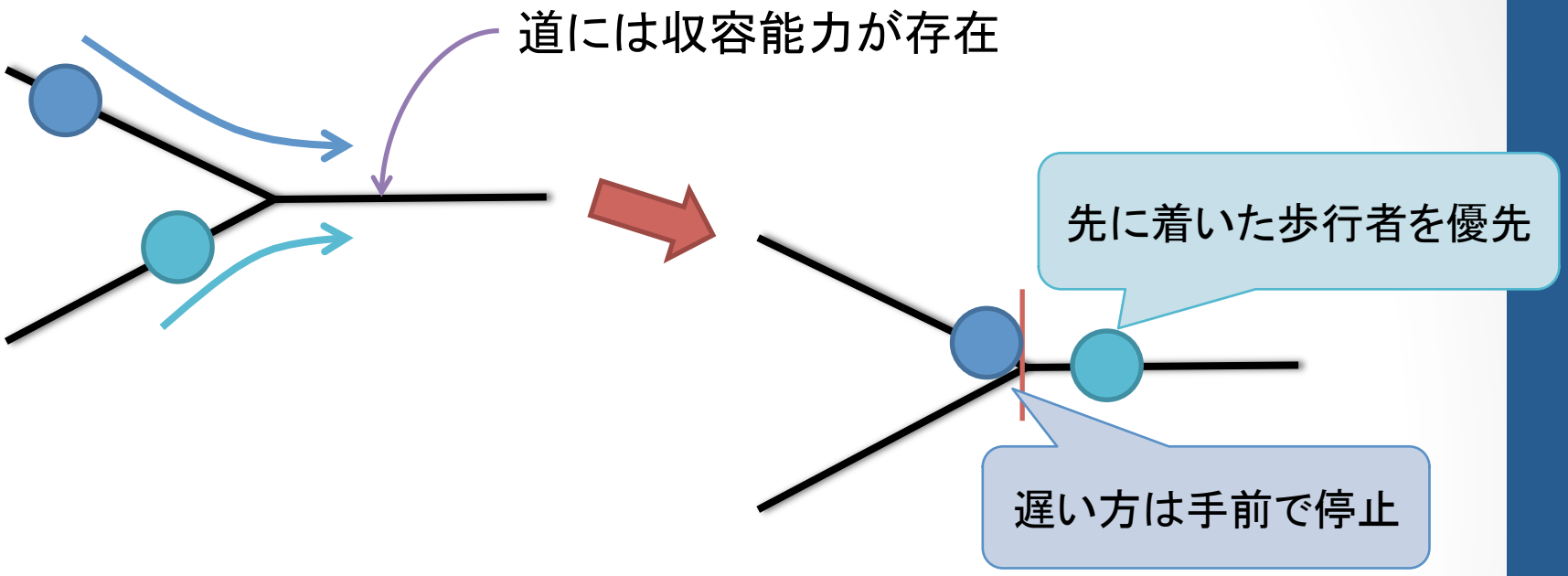
協調実行

2. 衝突判定＆位置更新

独立実行

3. 後処理

CrowdWalkの衝突判定



- 道に入る順番でmoveCommitを実行する
- moveCommitを実行するには他の歩行者が速度を計算(preUpdate)し終えている必要がある

現状のCrowdWalkの実装

歩行者間の処理依存を考慮すると、歩行者のコードが分割され、モジュラリティが低下する

```
class Agent {
  static void main() {
    for (int i = 0; i < maxIteration; i++) {
      for (Agent agent : agents)
        agent.preUpdate();
      sort(agents);
      for (Agent agent : agents)
        agent.moveCommit();
      for (Agent agent : agents)
        agent.postUpdate();
    }
  }
}
```

1イテレーション

バリア同期＋α
＝ モジュラリティと実行性能の両立

- 並列プログラムにおけるモジュラリティと実行性能の両立には、抽象度の高い柔軟なバリア同期が求められる

- バリア同期は並列に動作するプログラムの待ち合わせに利用
  - All-to-All型：OpenMP (C/C++, Fortran)、CyclicBarrier (Java)
    - 全ての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けて記述可能
    - **記述が複雑だが、性能が良い**

Point-to-Point型を用いたCrowdWalk

```
class Agent {
  Phaser(仮) phaser;

  void preUpdate() { ...; phaser.arrive(); }
  void moveCommit() { ...; phaser.arrive(); }

  void update() {
    preUpdate();
    if (goOnSameStreet()) {
      moveCommit();
    } else {
      neighbors().forEach(a -> a.phaser.await());
      headConflictings().forEach(a -> a.phaser.await());
      moveCommit();
    }
    postUpdat
  }
}
```

○複雑な依存関係や、その同期制御を柔軟に記述できるため効率的な並列化が可能

別の道に移動しない場合には同期しない

近隣の歩行者や実際に衝突する歩行者と同期

×バリア到達コードが散在し、モジュラリティが低下

All-to-All型を用いたCrowdWalk

```
class Agent {
  static CyclicBarrier(仮) barrier

  void update() {
    preUpdate();
    barrier.await();
    barrier.ordered(() -> {
      moveCommit()
    });
    postUpdate();
  }
}
```

言語やライブラリによってはスレッド内の一部を逐次実行するための構文が存在
（OpenMPのordered）

○抽象度の高いコードで計算の依存関係が記述可能

×依存関係のある処理すべてで全スレッドが待機するため、過剰な同期が誘引され、待機スレッド増加
=>メモリ圧迫や実行性能低下

提案：Waitless
バイトコード変換による軽量な条件付きバリア同期

目的

抽象度の高いAll-to-All型の記述で、Point-to-Point型と同等の柔軟な同期制御

手法

- オブジェクトの状態変更に対する条件式としてバリア同期を記述
  - 条件式を元にバリア到達箇所を解析
  - バリア到達コードを自動挿入
  - 軽量化のためにスレッドコードを自動分割

実装

バイトコード変換によるJava向けバリア同期ライブラリWaitless

条件付きバリア同期のための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 処理内容)
  - 順序付けした同期対象に対して与えた処理を逐次実行

```
ordered(agents, a -> places[i] != null,
                a -> a.moveCommit())
```

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

Waitlessを用いたCrowdWalk

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

  static void main() {
    for (
      Wai
        a
    }
  }
}
```

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とのみ逐次実行を行う

抽象度の高いAll-to-All型の記述で、Point-to-Point型と同等の柔軟な同期制御を実現

バイトコード変換による到達コードの自動挿入

- ASMを用いて、バイトコード解析・変換を行う
- 条件式を解析し、同期対象のフィールドの書き込みコードを検索
  - putfield etc.
- 書き込みコードの後にバリア条件のチェックと到達コードを挿入

```
class Agent {
  Place[] nextPlaces;

  void preUpdate() {
    nextPlaces[i] = calcNextPlace();
  }

  void update() {
    preUpdate();
    barrier(neighbors(),  a -> a.nextPlaces[i] != null);
    ...
  }
}
```

```
checkArrived();
```

スレッド分割

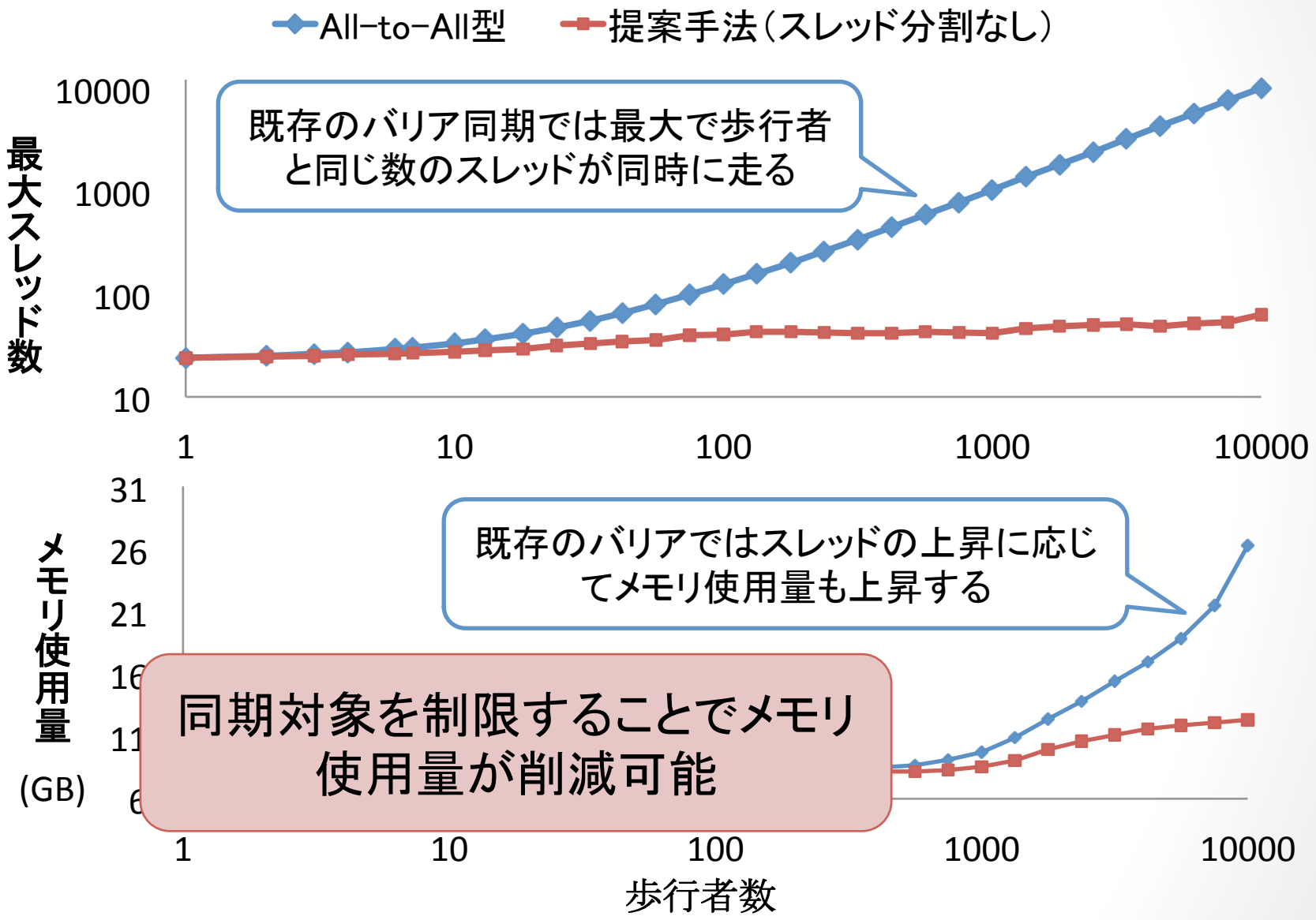
- スレッド分割ってなに？
  - 待機スレッドを開放し、残りの処理を任意の時点で別スレッドに割り当てることで、待機スレッド増加によるメモリ圧迫や実行性能低下を防ぐ仕組み
  - 例）イベントドリブンモデル、継続渡しスタイル

- Waitlessでの実装詳細
  - バリア待機
    1. コールフローと全ローカル変数を保存する
    2. 呼び出し元まで例外を投げ、スレッドを終了する
  - バリア再開
    1. スレッドを再度開始する
    2. 保存したコールフローを辿りながら、ローカル変数を復元する
       - 実行済みのコードはgotoでスキップ
    3. 残りの処理を再開する
  - Quasarというライブラリが提供する軽量スレッドFiberを用いて実装

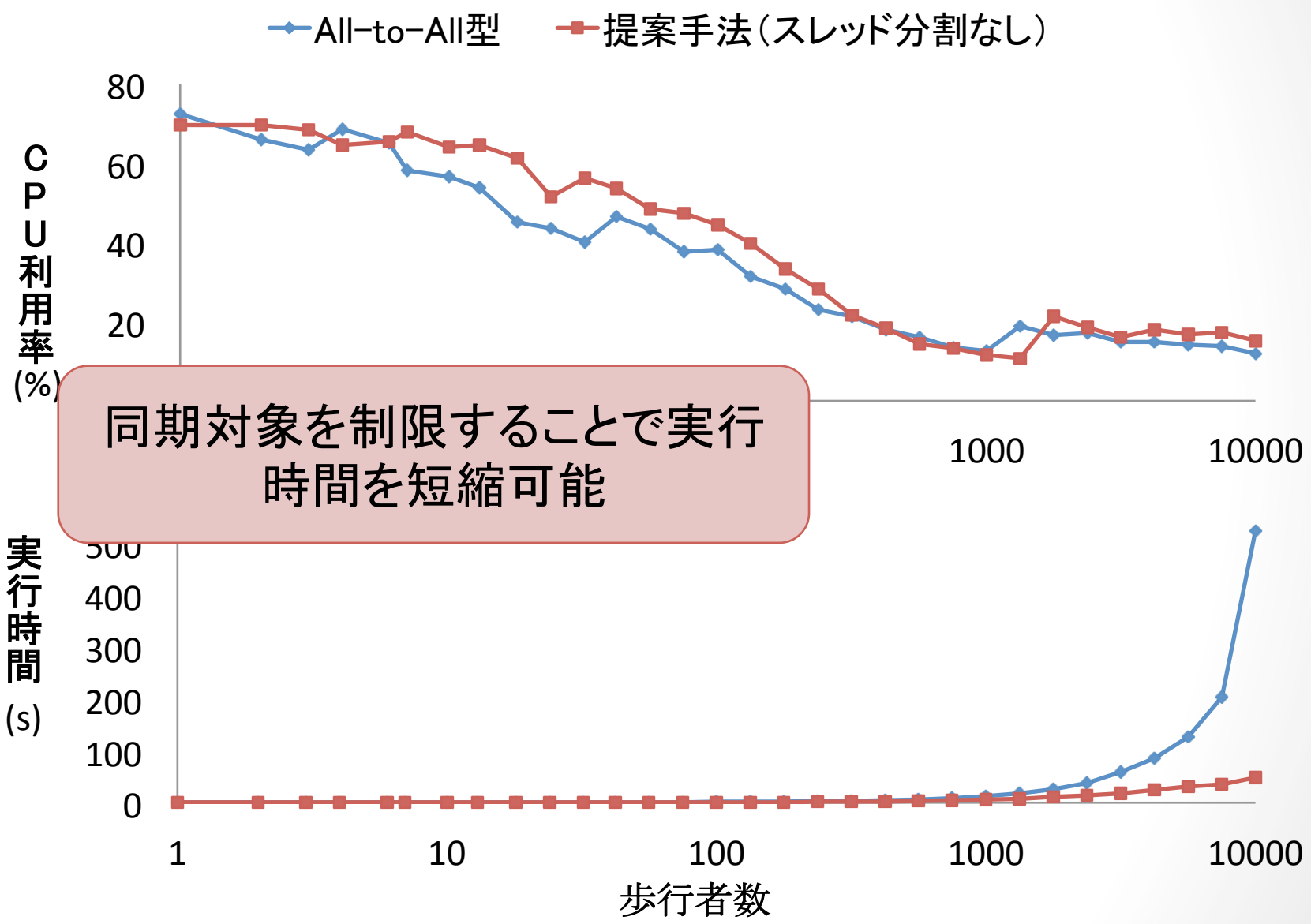
実験

- 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、CPU利用率、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: OpenJDK Runtime Environment (rhel-2.3.10.4.el6¥_4-x86¥_64) OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)
  - 実行オプション: -Xm32g -Xmx32g -Xss512m -XX:+UseG1GC -XX:InitiatingHeapOccupancyPercent=0

All-to-All型のバリア同期との比較

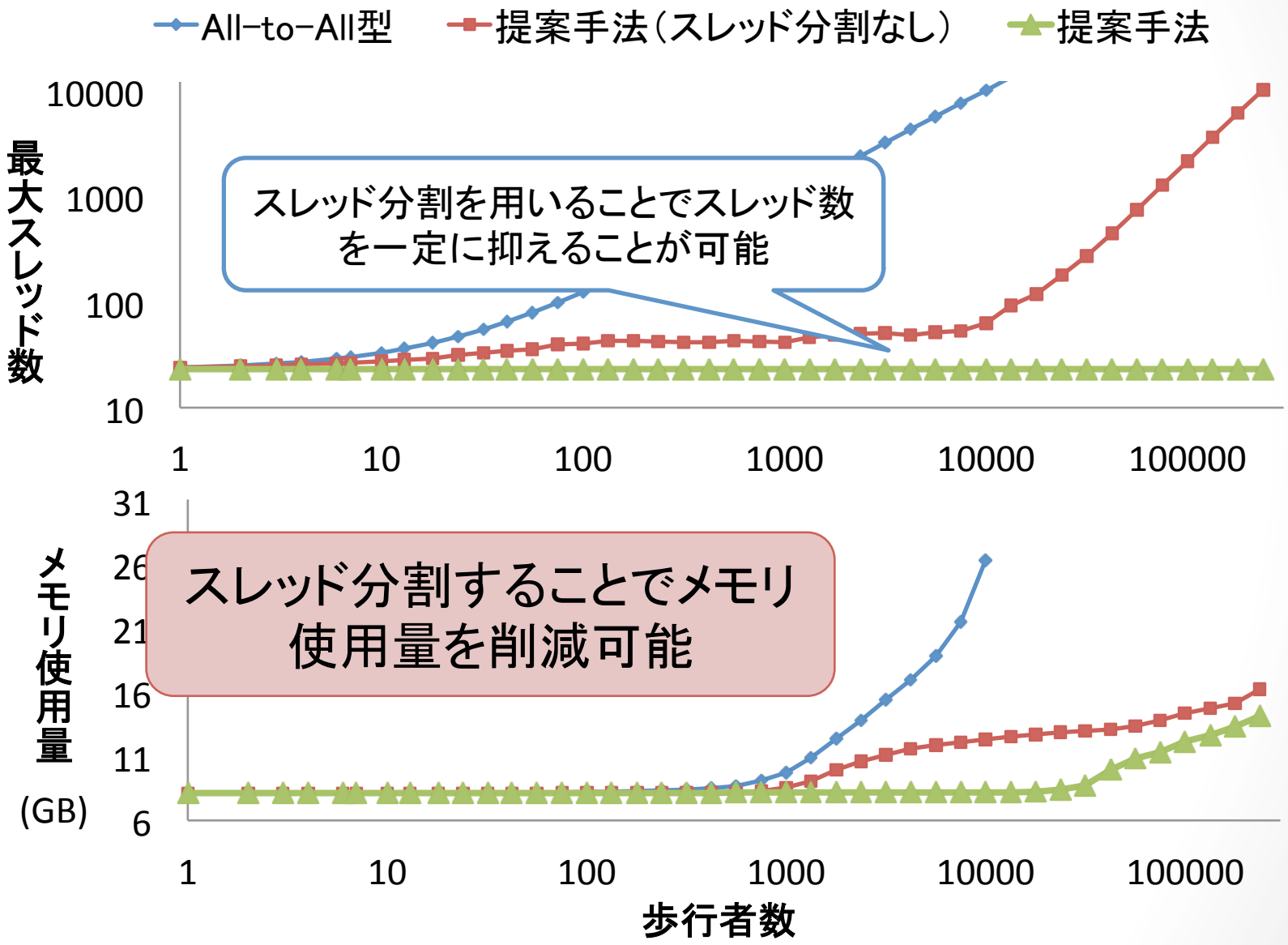


All-to-All型のバリア同期との比較

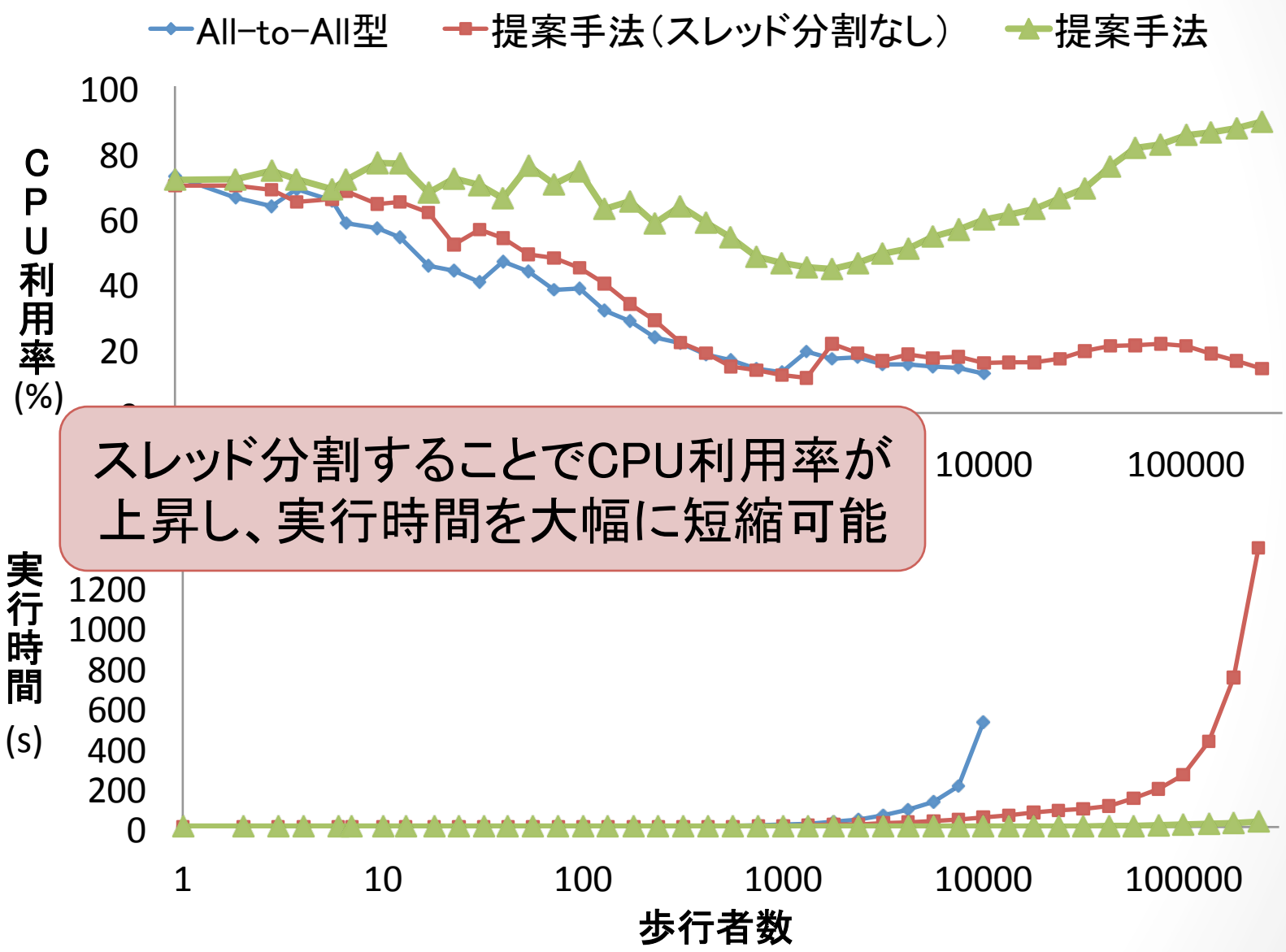


同期対象を制限することで実行時間を短縮可能

スレッド分割の効果


All-to-All型
提案手法（スレッド分割なし）
提案手法
最大スレッド数
10000
1000
100
10
スレッド分割を用いることでスレッド数を一定に抑えることが可能
1
10
100
1000
10000
100000
メモリ使用量
(GB)
31
26
21
16
11
6
スレッド分割することでメモリ使用量を削減可能
1
10
100
1000
10000
100000
歩行者数

スレッド分割の効果



スレッド分割することでCPU利用率が上昇し、実行時間を大幅に短縮可能

関連研究

- OpenMP
  - barrier と ordered をサポート
  - 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]
  - JavaのCyclicBarrierやX10のClocksを拡張
  - X10のライブラリとして提案され、Java 7で標準ライブラリに導入
  - 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

Waitlessの制限事項

- バイトコード変換ができないケース
  - 動的ロードしたクラス内でWaitlessを利用
- 条件式が実行時に変更されるケース
  - 条件式中で呼び出すメソッドのオーバーライド

まとめ

- 軽量な条件付きバリア同期を開発
  - オブジェクトの状態変更に対する条件式としてバリア同期を記述可能
  - バイトコード変換によりモジュラリティを維持したまま、待機スレッドを削減
    - 条件式を解析し、バリア到達コードを自動挿入
    - スレッドの自動分割
- CrowdWalkのサブセットを用いて性能評価
- 発表活動
  - 第16回プログラミングおよびプログラミング言語ワークショップ
  - 日本ソフトウェア科学会第31回大会

Waitlessを用いたCrowdWalk

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

  static void main() {
    for (int i = 0; i < max; i++) {
      Waitless.parallel(agents)
              .forEach(agent ->
        agent.update());
    }
  }
}
```

```
class Agent {
  static void main() {
    for (int i = 0; i < max; i++) {
      for (Agent agent : agents)
        agent.preUpdate();
      sort(agents);
      for (Agent agent : agents)
        agent.moveCommit();
      for (Agent agent : agents)
        agent.postUpdate();
    }
  }
}
```

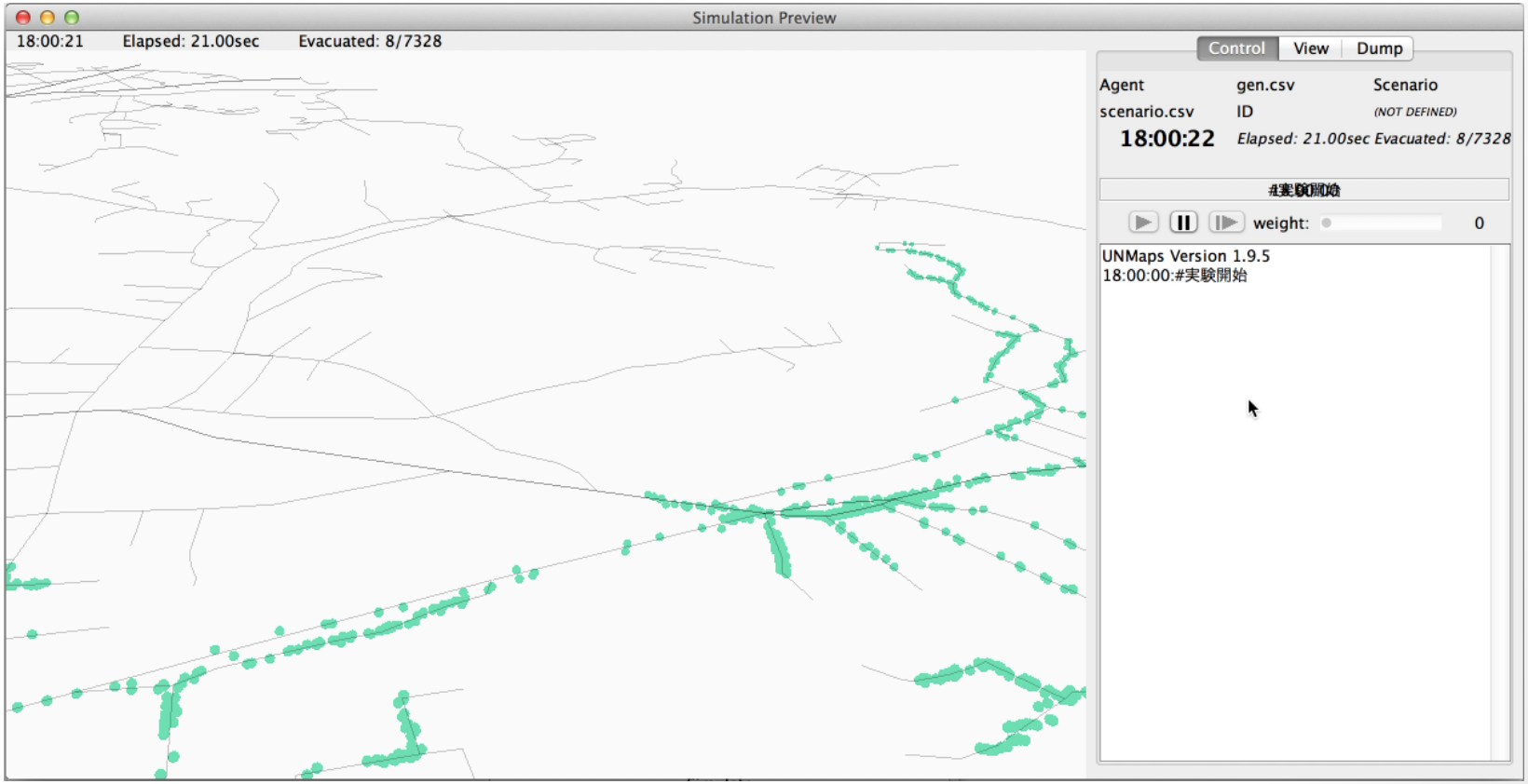
Waitlessを用いたCrowdWalk

```
class Agent {


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

  static void main() {
    for (int i = 0; i < max; i++) {
      Waitless.parallel(agents)
              .forEach(agent ->
        agent.update());
    }
  }
}
```

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

  void update() {
    preUpdate();
    if (goOnSameStreet()) {
      moveCommit();
    } else {
      neighbors()
        .forEach(a -> a.phaser.await());
      headConflictings()
        .forEach(a -> a.phaser.await());
      moveCommit();
    }
    postUpdate();
  }

  void preUpdate() { ...; phaser.arrive(); }
  void moveCommit() { ...; phaser.arrive(); }

  static void main() {
    for (int i = 0; i < max; i++) {
      for (Agent agent : agents)
        new Thread(() -> a.update());
    }
  }
}
```

CrowdWalk

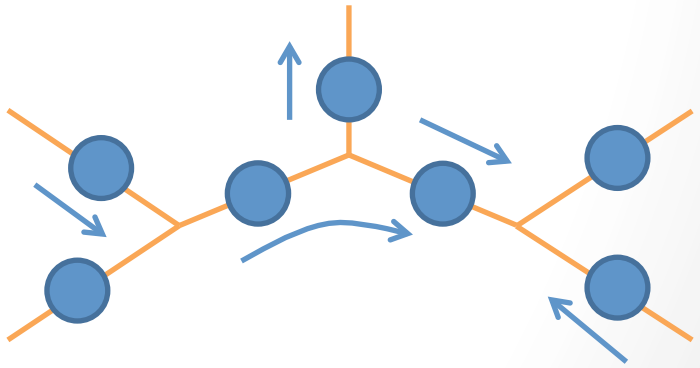
- 産業総合研究所で開発されている歩行者シミュレータ
  - 大規模な屋外イベントや災害時の避難経路の最適化
- マルチプラットフォーム
  - Java 6 (LOC: 24431)



CrowdWalkの並列化

- 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
  - 歩行者数：最大100万人
  - 例）北千住
    - 交差点：17,151
    - 道路数：25,414
    - 歩行者：60,000

- 計算速度：実時間の50倍（1,000人規模）



モジュラリティと実行性能

- 抽象度の高いプログラムではモジュラリティと実行性能を意識した並列化が重要

- バリア同期
  - 依存関係のあるプログラムの記述に有用
  - 並列に動作するプログラムの待ち合わせに利用される

  - All-to-All型：OpenMP (C/C++, Fortran)、CyclicBarrier (Java)
    - 全ての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けて記述可能
    - **記述が複雑だが、性能が良い**