**東京大学**
**情報理工学系研究科 創造情報学専攻**
**修士論文**

# 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

Lightweight conditional barrier synchronization for parallelizing agent simulation programs

**夏 澄彦**
Sumihiko Natsu

**指導教員　千葉 滋 教授**

2015 年 1 月

# 概要

複雑な依存関係を持つエージェントシミュレーションプログラムの並列化には、バリア同期を利用した協調処理の記述が有用である。しかし、既存のバリア同期機構を大規模なエージェントシミュレーションに適用すると、待機スレッドの増加が、CPU 効率を低下させるばかりでなく、その専有メモリの影響も無視できない。本研究では、バリア同期を待ち合わせるスレッドのグループと到達条件を実行時に与えられる、軽量な条件付きバリア同期とそのコード変換器を開発した。これを用いると、バリア到達条件をオブジェクトの状態変化に対する条件式として直感的に記述できる。さらに、コード変換によって、待機スレッドの発生を抑えるためのスレッドコードの分割と、バリア到達条件の監視コードの自動挿入を行えるようにした。また、開発したバリア同期機構を、大規模な歩行者シミュレーションアプリケーションへ適用して実行性能を評価した。

# Abstract

Barrier synchronization is useful to parallelize programs that have dependencies in their calculations. However, if a programmer naively applies existing barrier synchronization to a large agent simulation, performance decrement and memory-usage increment become a big problem because the number of waiting threads largely increases while synchronizing. To overcome this program, we propose Waitless, which supports writing barrier synchronization for a target-thread set with a condition using runtime information so that programmers can write barrier arrival condition like an observer of object-state changes. Waitless divides thread code and inserts barrier-arrival signal code by converting bytecode at load-time. We performed experiments with a subset of an existing pedestrian simulators to show the performance of Waitless.

# 目次

# 第 1 章

# はじめに

マルチエージェントシミュレーションのような現実世界の自律的な行動主体をモデリングするソフトウェアは、他の多くの科学技術計算とは異なり、オブジェクト指向のような抽象度の高い設計と親和性が高い。また、エージェントの動作を実装したプログラムは、計算粒度が大きく、他のエージェントとの協調動作が必要となり、依存関係が複雑になる傾向がある。このようなエージェント間の処理の依存関係は、大規模シミュレーションのためのソフトウェアの並列分散化の障壁となっている。

依存関係が単純な一般的な科学技術計算を並列化する際にはバリア同期がよく用いられている。バリア同期とは、依存関係のある並列タスクを記述するためのシンプルな機構である。バリア制御の対象とされた並列タスクは、全てのタスクの実行が終了するまで続きの処理が待機させされる。しかし、大規模エージェントシミュレーションを科学技術計算プログラムでよく利用されるバリア同期のようなタスク制御機構を用いて効率良く実行することは難しい。なぜなら、OpenMP のような典型的な All-to-All 型のバリア同期では、全てのスレッドが同期を待ち合わせるため、シミュレーションサイズの増加に伴い、バリア同期時に発生する待機スレッド数が増えると、CPU 実行効率が低下するばかりでなく、その専有メモリ量も増大する。また、Phaser のような Point-to-Point 型のバリア同期を用いることで、バリア同期処理を、待機処理と到達処理に分けて記述できるため、バリア同期で待機するスレッド数を最小化できることが可能となる。しかし、バリア到達処理コードが散在するため、エージェントシミュレーションのような抽象度の高いソフトウェアでは、モジュラリティの低下も問題となってくる。

そこで、本研究では、ロードタイムにバイトコード変換を行うことで、散在するバリア到達処理をオブジェクトの状態に基づく直感的な条件式として記述可能なバリア同期を提案する。与えられた条件式に基づきクラスファイルを解析し、条件が満たされる可能性のある箇所全てにバリア条件のチェックコードと、到達処理コードが自動挿入される。さらに、バリア同期処理の前後でスレッドが実行するコードの分割を行い、待機スレッドを削減できるようにした。本研究で開発したバリア同期機構は、大規模な歩行者シミュレーションアプリケーションへ適用して実行性能を評価した。

以下、第 2 章では既存のバリア同期について紹介し、それらを用いてエージェントシミュレーションを並列化した場合の問題点を述べる。第 3 章では、エージェントシミュレーション

のような抽象度が高く、計算の依存関係が複雑なプログラムの並列化に有用な条件付きバリア同期機構を提案する。第 4 章では提案した同期機構の実装方法について述べる。第 5 章で提案した同期機構の有用性を確認するために行った実験の結果と考察を示す。そして最後に、第 6 章でまとめと今後の課題について述べる。

# 第 2 章

# バリア同期によるエージェントシミュレーションの並列化

エージェントシミュレーションのような依存関係が複雑で、抽象度の高いプログラムの並列化にはバリア同期が有用である。しかし、大規模なエージェントシミュレーションに対して既存のバリア同期を適応した場合、モジュラリティと実行性能を両立させたスレッド制御は困難である。本章では、実際に開発されているエージェントシミュレーションのコードを参考に、バリア同期による並列化と、その問題点について記す。

## 2.1 エージェントシミュレーション

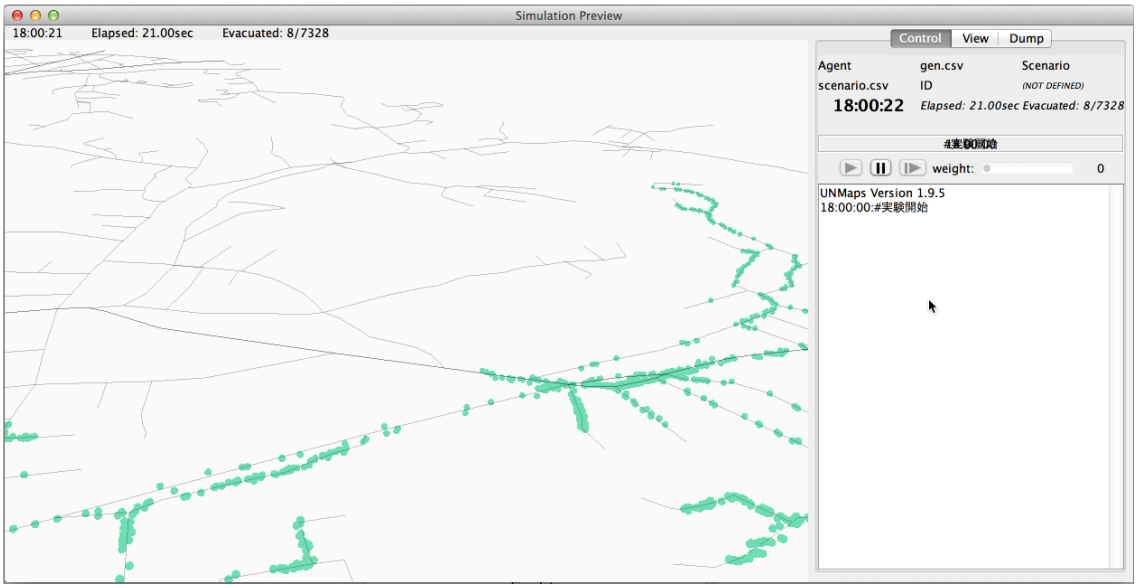
近年、社会システムの高度化やビックデータの利活用のため、計算科学の視点から社会システムを研究する必要性が迫られている。その手法の一つとして、エージェントシミュレーションが存在する。エージェントシミュレーションとは、一定のルールに基いて行動する様々な自律的な行動主体（エージェント）を相互作用させることにより、システム全体の挙動を表現するシミュレーションである。各エージェントのミクロな振る舞いが、エージェント間の相互作用により、環境全体のマクロな振る舞いを変化させるため、現実世界のように、個々のエージェントの動きの解析だけでは想定できないようなマクロの現象をボトムアップ的に解析するのに効果的であり、生態系や経済市場、避難行動など様々な分野での解析・予測に用いられている。

エージェントシミュレーションのような依存関係が複雑で抽象度の高いソフトウェアの並列化にはバリア同期が有用である。多くのエージェントシミュレーションは時間軸上の繰り返し処理でエージェントの内部状態や外部環境を更新していくため、イテレーション毎の処理はエージェント単位での実装が可読性や保守性に優れる。例えば、挙動を拡張する際に変更箇所を特定しやすくなるといった利点が挙げられる。しかし、多くのエージェントシミュレーションは、エージェントを相互作用させることでシステム全体の振る舞いを調べるため、エージェント間に計算の依存関係が存在する。従って、エージェント単位のモジュール化を維持したまま、全体のプログラムを正しく動かすためには、エージェントごとに並列化し、計算の依存関

係に合わせて同期制御する必要がある。

図 2.2 に、現在，産業技術総合研究所で開発中の歩行者シミュレータ CrowdWalk（図 2.1）の歩行者エージェントの処理を簡略化したプログラム例を示す。このシミュレーションでは、各エージェント（歩行者）が単位時間ごとに、グラフ構造状の道の上を辿り、ゴールに進む。道にはそれぞれ限界容量があるため、進入できるエージェントの上限が決められており、衝突判定を行う必要がある。単位時間毎の処理は各イテレーションで実行される。まず各エージェントは、速度を更新し、現在位置と速度から次に進もうとしている道をシーケンシャルに計算（preUpdate）する。その後、衝突判定を行うために、道を移動する順番で全てのエージェントを並び替えてから、一つずつ移動を行い（moveCommit）、最後に後処理（postUpdate）を行う。

**図** 2.1. CrowdWalk




衝突判定のようなエージェント間の依存関係を考慮しなければ、これらの処理を図 2.3 のようにエージェント毎にまとめてスレッド化し、並列実行可能である。しかし、実際には moveCommit のような共有オブジェクト（道）への書き込みや衝突判定など、他のエージェントとの依存関係を考慮すべき処理の順序が、Java のスレッドスケジューラに依存するという問題が発生する。エージェント処理のモジュラリティを維持したまま、適切な実行順序でスレッドを制御するためには、各エージェントが互いに同期しながら処理を進めていくようにコードを改変しなければならない。具体的には、moveCommit が呼ばれる順番は、preUpdate の計算結果である次の道へ移動する順番に依存するため、エージェントは自分の preUpdate が終わった後、moveCommit を実行する前に、他のエージェントが preUpdate が実行し終えるまで待たなければならない。

**図** 2.2. 歩行者エージェント全体の処理を簡略化したプログラム

```
class Pedestrian {
  // 速度を再計算し、そこから次に進むべき場所を計算する
  void preUpdate() {
    speed = calcSpeed();
    nextPlaces[i] = calcNextPlace();
  }

  // 衝突判定と実際の移動を行う
  void moveCommit() {
    places[i] = calcPlace();
  }

  // 後処理を行う
  void postUpdate() {
    ...
  }

  static void main(String args[]) {
    Collection<Pedestrian> all = ...;
    for (int i = 0; i < maxIteration; i++) {
      for (Pedestrian pedestrian : all) {
        pedestrian.preUpdate();
      }
      // 次の道に入る順番で歩行者を並び替える
      all.sort();
      for (Pedestrian pedestrian : all) {
        pedestrian.moveCommit();
      }
      for (Pedestrian pedestrian : all) {
        pedestrian.postUpdate();
      }
    }
  }
}
```

## 2.2 関連研究及び既存のバリア同期による並列化

このような、計算に依存関係のあるプログラムの並列化には、バリア同期を用いることで簡潔に記述できる。バリア同期とは、並列に動作するスレッドの実行を待ち合わせるために用いられ、一般的な科学技術計算プログラムのように、簡単な依存関係のある計算を行うプログラムに対して、最小限のコードで並列化を行うことが可能である。通常は、MPI[1] や OpenMP[2] といった並列分散向けライブラリで提供されているが、X10[3] や Chapel[4, 5] のように専用のプリミティブとして標準提供されている言語も数多く登場している。バリア同期

**図 2.3.** 依存関係を無視した並列化

```
class Pedestrian {
  void update() {
    preUpdate();
    // moveCommit は適切な順番で呼ばれる必要がある
    moveCommit();
    postUpdate();
  }

  public static void main(String args[]) {
    for (int i = 0; i < maxIteration; i++) {
      for (Pedestrian pedestrian : pedestrians) {
        new Thread(() -> pedestrian.update()).start();
      }
    }
  }
}
```

を用いると、スレッドがコード上の指定した箇所に到達するまで、同期に参加する他のスレッドを待機させることができる。

既存のバリア同期は大別すると OpenMP に代表される All-to-All 型と、Phaser や CountDownLatch に代表される Point-to-Point 型の 2 種類に分類することができる。

### 2.2.1　All-to-All 型

全てのスレッドが同期を行う、典型的なバリア同期であり、atomic なカウンターとロックオブジェクトを用いることで容易に実装出来る [6]。All-to-All 型のバリア同期は宣言的に記述することが可能なため、並列化に詳しくない開発者でも容易に既存のプログラムを並列化することが可能である。

例えば、OpenMP では、スレッドが互いに待ち合わせする位置に同期のための指示文をコメントとして記述することが可能である (図 2.4)。この例では、それぞれのスレッドで、並列に A() が実行された後、お互いに他のスレッドを待ち合わせ、全てのスレッドが barrier 指示文に到達してから、スレッドが再開され、並列に B() が実行される。

**図 2.4.** OpenMP の barrier 指示文

```
#pragma omp parallel
{
  A();
  // 他のスレッドがこの箇所に到達するまで､実行中のスレッドを待機させる
  #pragma omp barrier
  B();
}
```

また、MPI にも同様の同期機構 MPI_Barrier(comm) が存在し、指定したコミュニケータ comm 内の全プロセスで同期を取ることが可能となる（図 2.5）。この例では、それぞれのプロセスで、並列に A() が実行された後、全プロセスが MPI_Barrier を呼ぶまでプロセスをブロックし、その後、並列に B() が実行される。

**図** 2.5. MPI_Barrier

```
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  A();
  // MPI_COMM_WORLD コミュニケータに属する他のプロセスがこの箇所に到達するまで、
  // 実行中のプロセスを待機させる
  MPI_Barrier(MPI_COMM_WORLD);
  B();
  MPI_Finalize();
  return 0;
}
```

言語レベルでバリア同期をサポートしている X10[7, 3] も Clocks を用いてバリア同期が可能である（図 2.6）。X10 の並列タスクである Activity は生成時に、clocked に Clocks オブジェクトを渡すことで、共通のバリア同期に参加することが可能であり、Clock.advanceAll() を用いることで、同じ Clock オブジェクトに参加している他の Activity と同期を行う。これにより、A1() と B1() が並列に実行された後、バリア同期が行われ、A2() と B2() が並列に実行される。

**図** 2.6. X10 の Clocks

```
public class ClockExample {
  public static def main(args: Rail[String]) {
    finish async {
      val cl = Clock.make();

      async clocked(cl) { // Activity A
        A1();
        Clock.advanceAll();
        A2();
      }

      async clocked(cl) { // Activity B
        B1();
        Clock.advanceAll();
        B2();
      }
    }
  }
}
```

OpenMP では逐次実行をサポートする ordered という同期構文も存在する（図 2.7）。これは for ループを並列化する際、コードの一部を for で指定した順番でシーケンシャルに実行することが可能である。この例では、for ループ内の A() が並列実行された後、B() が逐次に実行され、最後に C() が並列に実行される。

**図** 2.7. OpenMP の ordered 指示文

```
#pragma omp parallel
{
  #pragma omp parallel for ordered
  for (int i = 0; i < NUM; ++i) {
    A();
    #pragma omp ordered
    B();
    C();
  }
}
```

前節で紹介した図 2.2 を Java 6 の java.util.concurrent で導入された CyclicBarrier を用いて並列化を行うと，図 2.3 は図 2.8 のように記述することが可能となる。エージェントは各イテレーション（update）内で、preUpdate が終わると、スレッドを待機させ、全てのエージェントに対して同期を行っている。他のエージェントが preUpdate を実行し終わると、まず、CyclicBarrier の第 2 引数で渡した Runnable が実行され、正しい順番で moveCommit がシーケンシャル実行された後、スレッドが再開され、postUpdate が並列実行される。

しかし、All-to-All 型のバリア同期の場合、全てのスレッドが同期に参加するため、全スレッドが最も処理時間の長いスレッドの計算が終わるまで待機することになり、不必要な同期待ちが発生する。このため、スレッド数の増加に伴い、待機状態のスレッド数も増加し、結果的に、CPU の実行効率の低下を招くだけでなく、占有するメモリ量を増大させる。例えば、CrowdWalk の場合、ゴールまでの距離や周囲の状況によって、各エージェントの preUpdate にかかる時間は不均一である。したがって、全てのエージェントに対して同期を行うと、最も preUpdate の計算時間が長いエージェントの実行が完了するまで、他のエージェントは待機することになる。それにより、大量の待機スレッドが発生する。実際の交通シミュレーションや避難シミュレーションでは、エージェント数は数十万から数百万にのぼるため、OS のスレッド上限を超える可能性も出てくる。

### 2.2.2　Point-to-Point 型

Point-to-Point 型のバリア同期は All-to-All 型のバリア同期とは異なり、同期のための待機コードと到達コードを任意の場所に記述することが可能である。これにより、待機を行うかどうかを動的に変更できる上、同期を行うタイミングを柔軟に指定できるため、待機スレッドを最小限にすることが可能となる。

図 2.8. CyclicBarrier を用いた並列化

```
1 class Pedestrian {
2   void update(CyclicBarrier barrier) {
3     preUpdate();
4     // 他の歩行者が preUpdate を実行し終えるまで待機する
5     barrier.await();
6     // CyclicBarrier のコンストラクタの第 2 引数で渡した Runnable が一度だけ実行される
7     //（sort や全歩行者への moveCommit が実行される）
8     postUpdate();
9   }
10
11  public static void main(String args[]) {
12    for (int i = 0; i < maxIteration; i++) {
13      CyclicBarrier barrier = new CyclicBarrier(all.size(), -> {
14        all.sort();
15        for (Pedestrian pedestrian : all) {
16          pedestrian.moveCommit();
17        }
18      });
19      for (Pedestrian pedestrian : pedestrians) {
20        new Thread(() -> pedestrian.update(barrier)).start();
21      }
22    }
23  }
24 }
```

X10 の Clocks や Java の CyclicBarrier に対して、より柔軟なバリア同期を行える機構として、Phaser[8, 9] が提案されており、Java 7 の java.util.concurrent で Java の標準ライブラリとして導入されている。Phaser は同期のためのメソッドが、バリア到着のための arrive メソッドと待機のための awaitAdvance メソッドに分割されている。arrive メソッドを実行した後、awaitAdvance メソッドを実行しない場合、他のスレッドの到着を待機せずに処理を続けることが可能である。

Phaser を用いて図 2.2 の並列化を行うと、図 2.9 のように記述できる。All-to-All 型のバリア同期では、非対象な同期が不可能なため、動的に決定されるスレッドグループに対して同期を行うことが出来ない。しかし、Phaser のような Point-to-Point 型のバリア同期の場合、同期対象の各スレッドが Phaser オブジェクトを持つことにより、動的に計算される近隣の歩行者のようなスレッドグループに対しても、バリア到達のシグナルを受信することで、必要最低限の同期を行うことが可能となる（19〜21 行目）。さらに、道を移動しない歩行者はそもそも衝突が起こらないため、バリア到達のシグナルを送るだけで、他の歩行者を待たずに処理を進めることが可能となる。

しかし、Point-to-Point 型のバリア同期は、効率のよい並列化を行うことが可能な反面、バリア到達を発行すべき箇所をプログラマが特定し、明示的に記述する必要がある。従って、細

**図** 2.9. Phaser を用いた並列化

```
class Pedestrian {
  Phaser phaser;

  void preUpdate() {
    speed = calcSpeed();
    nextPlaces[i] = calcNextPlace();
    // preUpdate を実行し終えたことを通知する
    phaser.arrive();
  }

  void moveCommit() {
    places[i] = calcPlace();
    // postUpdate を実行し終えたことを通知する
    phaser.arrive();
  }

  void update() {
    preUpdate();
    if (goOnSameStreet()) {
      moveCommit();
    } else {
      // 近隣の歩行者が preUpdate を実行し終えるまで待機する
      neighborPedestrians().forEach(p -> {
        p.phaser.awaitAdvance(0);
      });
      // 実際に衝突する歩行者のうち、自分よりも先に次の道に進む歩行者が
      // moveCommit を実行し終えるまで待機する
      conflictingPedestrians().headSet(this).forEach(p -> {
        p.phaser.awaitAdvance(1);
      });
      moveCommit();
    }
    postUpdate();
  }

  public static void main(String args[]) {
    for (int i = 0; i < maxIteration; i++) {
      for (Pedestrian pedestrian : pedestrians) {
        pedestrian.phaser = new Phaser(1);
      }
      for (Pedestrian pedestrian : pedestrians) {
        new Thread(() -> pedestrian.update()).start();
      }
    }
  }
}
```

粒度の同期制御を実装する場合、バリア到達コードはプログラム中に散在し、モジュラリティの低下を引き起こし、可読性や保守性が低下する。

### 2.2.3 スレッド分割

HJ[9] や X10、Chapel といった言語では、fork-join モデルを用いてスレッド間の強調処理を行っている場合、スレッドをブロックすることなく同期することが可能なため、効率のよい並列化が可能となる。しかし、future[10] や barrier、Phaser のような一般的な同期機構はロックを用いて実装されているため、同期時にスレッドが待機状態となる。そのため、スレッドプールを用いた場合、同期時に待機状態となったスレッドにより資源が専有され，容易にデッドロックが引き起こされてしまう。スレッドプールが用いることができない場合、スレッド作成や専有メモリ、コンテキストスイッチングなどのオーバーヘッドが発生するため、効率よく並列化することが難しい。

このような問題を回避するプログラミングテクニックとして、バリア同期前後の処理コードを別々のスレッドに分割し、処理依存を排除する方法が知られている。スレッド間の依存関係を排除すれば、スレッドプールを用いることが可能となるため、スレッドの同時実行数を一定に抑えられ、メモリ圧迫や実行性能低下を回避できる [11]。

しかし，既存コードのスレッド分割は複雑な書き換えを伴い，可読性や保守性を著しく低下させる。例えば、図 2.10 に図 2.8 を手動で継続渡しスタイルに変更し、スレッド分割したコード例を示す。バリア待機を伴うコードを実行するためには、19 行目のように待機を含む残りの処理である継続をラムダ式を用いて、新しいスレッドに割り当てて実行する必要がある。このように、スレッド間の処理依存を手動で排除するためには，既存のコードにスレッド分割のためのコードを追加する必要がある。また、Java において、ラムダ式や無名内部クラスを用いた継続渡しでは、ローカル変数に対する代入が不可能なため、ローカル変数の保存・復元は手動で行う必要がある。

これに対して、RIFE[12] や Javaflow[13]、Kilim[14] のように、バイトコード変換により Java プログラムの継続をサポートするフレームワークを利用することで、スレッド分割を自動化することが可能となる。例えば、Kilim の場合、Pausable のアノテーションが指定されたメソッドに対して、ローカル変数の退避や復元、実行済みのコードをスキップするためのコードをバイトコード変換により挿入するため、図 2.12 のようなコードは図 2.12 のように変換され、スレッド分割が行われる。Fiber.pause() が呼ばれると、ローカル変数などのフレームの情報をスタックとして Fiber インスタンスに格納し、メソッドの実行を停止し、呼び出し元に戻る。再度 Fiber.pause() が呼ばれると、Fiber インスタンスからフレーム情報を取り出し、呼び出し階層の復元を行う。これを用いることで、待機状態になる箇所で Fiber.pause() を呼び、待機解除時に再度 Fiber.pause() を呼ぶことで、スレッドコードの手動分割をすることなく、スレッドプールを利用することができる。

**図** 2.10. 手動でスレッド分割を行った並列化

```
class Pedestrian {
  void update(CyclicBarrier barrier) {
    preUpdate();
    await(barrier, 1, () -> {
      pedestrians.sort();
      await(barrier, pedestrians.headSet(this).size(), () -> {
        moveCommit();
        await(barrier, pedestrians.tailSet(this).size(), () ->
          postUpdate());
      });
    });
  }

  // n 回新しいスレッドでバリア同期を行った後、cont を実行する
  void await(CyclicBarrier barrier, int n, Runnable cont) {
    if (n == 0) cont.run();
    else {
      new Thread(() -> {
        // バリア待機を伴うコードの実行は新しいスレッド内で行う
        barrier.await();
        await(barrier, n - 1, cont);
      }).start();
    }
  }
}
```

**図** 2.11. Kilim のバイトコード変換によるスレッド分割前

```
@Pausable
void foo() {
  x = ...;
  bar();
  System.out.println(x);
}

@Pausable
void bar() {
  ...
}
```

**図** 2.12. Kilim のバイトコード変換によるスレッド分割後

```
1  void foo(Fiber f) {
2    switch (f.pc) {
3    // 初めて実行した場合
4    case 0: goto START;
5    // 一度実行され、bar メソッドの中で停止した
6    case 1: goto CALL_BAR;
7    }
8  START:
9    x = ...;
10 CALL_BAR:
11   f.down();
12   bar(f);
13   f.up();
14   switch (f.status) {
15   // bar メソッドの中で停止することなく、実行完了した
16   case NOT_PAUSING__NO_STATE:
17     goto RESUME;
18   // ２回目の実行後、残りの処理を行うため、ローカルの復元を行う
19   case NOT_PAUSING__HAS_STATE:
20     restore state;
21     goto RESUME;
22   // bar メソッドの中で停止したため、ローカル変数を退避させ、呼び出しに戻る
23   case PAUSING__NO_STATE :
24     capture state;
25     return;
26   }
27 RESUME:
28   System.out.println(x);
29 }
```

# 第 3 章

# 条件付きバリア同期機構

本研究では抽象度の高い All-to-All 型の記述で、Point-to-Point 型に近い柔軟な同期制御が可能な Java 向けバリア同期ライブラリ Waitless を提案する。Waitless は実行時情報を用いた、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させることで、散在する到達コードを条件式の形に集約できる。条件式は、ロードタイムに解析され、必要な到達コードがバイトコード変換で自動挿入される。さらにバリア同期の呼び出し前後で、スレッドコードの自動分割を行うことで、同期待ちによる待機スレッドの発生を防ぎ、モジュラリティを維持したまま、スケーラブルな並列化を実現する。

上記のような目的を満たすため、Waitless はバリア同期のためのメソッドとして barrier（図 3.1）を導入する。これは、同期対象の集合（Set 型）と同期到達の条件式（Predicate 型）の 2 つを受け取り、指定された全ての同期対象が、条件式を満たすまで、現在実行中のスレッドを待機させる。

```
1   barrier(Set<E> 同期対象集合, Predicate<E> 同期到達条件式)
```

**図 3.1.** barrier メソッド

本研究は、前身研究 [15] に対して、実行時情報を用いたバリア条件の記述とスレッド分割の自動化を行った。本章では、まず、図 3.1 のバリア条件の記述方法について述べ、続いて、到達シグナルコードの自動挿入方法とスレッド分割方法を説明し、最後に利用方法について説明する。

## 3.1 実行時情報に基づいたバリア条件の記述

### 3.1.1 並列コレクションと同期対象の指定

Waitless では抽象度の高い記述で、柔軟性の高い同期制御を実現するための API として、並列コレクション WaitlessSet と、そこから生成される並列ストリーム WaitlessStream を提供し、WaitlessStream の各要素に対する処理の中で barrier メソッドを用いた条件付きバリア

同期を利用することができる（表 3.1）。Waitless では、実行中のコンテキストに応じてバリア同期対象を並列コレクションの部分集合（Set 型）から動的に指定できる。All-to-All 型のバリア同期とは異なり、同期待ちのタスクグループを限定するため、待機スレッド数を最小限に留めることが可能である。さらに、実行時にタスクグループを指定できるため、静的には分からないタスク間の依存関係があるアプリケーションに対しても有効である。

**表** 3.1. Waitless の並列コレクション

| Class WaitlessSet⟨T⟩ | |
|---|---|
| コンストラクタ | WaitlessSet(Set⟨T⟩ set)<br>set の各要素を入力として実行可能な並列コレクション WaitlessSet を作成する |
| WatilessStream | parallelStream()<br>並列ストリーム処理を行うための WaitlessStream を返す |
| **Class WaitlessStream⟨T⟩** | |
| void | forEach(BiConsumer⟨T, Waitless⟨T⟩⟩ consumer)<br>barrier によるバリア同期を呼び出し可能なコード consumer を並列に実行する |

例えば、歩行者シミュレーション（図 2.8）の場合、エージェント同士の依存関係が動的に決定可能であるため、実行時にどのエージェントと同期すべきかが決定できる。各エージェントは道を移動する際に、衝突する可能性のあるエージェントが近傍のみであり、動作している全てのエージェントと同期する必要がない。従って、図 3.2 の 3 行目のように barrier メソッドを呼び出すことにより、全ての歩行者 allPedestrians の部分集合である近隣の歩行者 neighborPedestriansOf(pedestrian) に対してのみ同期を行うことが出来る。

**図** 3.2. Waitless の利用例

```
1 new WaitlessSet(allPedestrians).parallelStream().forEach((pedestrian, w) -> {
2   foo(pedestrian);
3   w.barrier(neighborPedestriansOf(pedestrian), neighbor -> baz(neighbor))
4   bar(pedestrian);
5 });
```

### 3.1.2 同期の到達条件の指定

Waitless では、また、同期到達の条件をオブジェクトの状態変化に対する条件式として直感的に記述可能である。条件式は Predicate 型で定義され、引数が 1 つで返り値が boolean のラムダ式として記述できる。これにより、引数として受け取った並列コレクション WaitlessSet の各要素に対して、ラムダ式の評価を行い、条件式の返り値が true となるまで、現在実行中

のスレッドを待機する。但し、条件式の評価が結果の変化につながることを防ぐため、条件式内では変数の代入のような副作用を伴う操作はできないものとする。

例えば、歩行者シミュレーション（図 2.8）の場合、他の歩行者が次に進もうとしている場所の計算を終えるまで待つ必要がある。このような場合には、図 3.3 のように記述することで、他の歩行者の nextPlaces[i] が null ではなくなった時に、バリア到達シグナルコードが実行される。

```
1    (Predicate<Pedestrian>) pedestrian -> pedestrian.nextPlaces[i] != null
```

**図** 3.3. 同期の到達条件の指定

## 3.2　利用例

### 3.2.1　逐次実行

Waitless が提供する barrier メソッドを用いると、OpenMP の ordered ディレクティブのように、スレッド内の一部の処理を逐次実行をするためのコードを容易に実装することが可能である。

```
1  ordered(SortedSet<E> スレッド集合, Predicate<E> 条件式, Runnable 逐次処理)
```

**図** 3.4. ordered メソッド

OpenMP の ordered は全てのスレッドが逐次実行を完了するまで、各スレッドは次の処理に進めない。このタイプの逐次実行（asymmetric ordered）は Waitless の barrier を用いて、図 3.5 のように実装できる。逐次実行を行うグループを順序付き集合 targets で与え、その中で自分より前の全ての要素が逐次処理コード seq を実行し終えるまで待機してから、自分の seq を実行することで、グループ全体で指定した順番で逐次処理を実現することができる。

これに対して、各スレッドが逐次処理内容を行った後に他のスレッドが逐次実行を完了するのを待たずに次の処理を開始するタイプの逐次実行（symmetric ordered）も Waitless の barrier を用いて実装することができる（図 3.6）。

また、OpenMP の ordered では、全てのスレッドが指定した処理を逐次実行するため、一つのコアしか使えない。しかし、アプリケーションによっては全てのスレッドが逐次実行をする必要性が無く、特定のスレッドグループ内での逐次実行で十分な場合がある。Waitless は動的に同期を行うスレッドグループができるため、このようなケースに対応することができ、逐次実行を行うスレッドグループごとに並列実行を行うことで、実行性能を改善することが可能である。

**図** 3.5. barrier を用いた asymmetric ordered の実装

```
// targets：指定した順番で逐次実行したい対象
// seq：逐次処理コード
// predicate：seq を実行したかどうかを判定する条件式
void ordered(SortedSet<T> targets, Predicate<T> predicate, Runnable seq) {
  SortedSet<T> head = targets.headSet(item);
  // 自分が先頭でない場合､自分より前の同期対象オブジェクトが
  // 条件式 pred を満たすまで待機する
  if (!head.isEmpty()) {
    barrier(head, predicate);
  }
  seq.run();
  barrier(targets, predicate, () -> {})
}
```

**図** 3.6. barrier を用いた symmetric ordered の実装

```
void ordered(SortedSet<T> targets, Predicate<T> predicate, Runnable seq) {
  SortedSet<T> head = targets.headSet(item);
  if (!head.isEmpty()) {
    barrier(head, predicate);
  }
  seq.run();
  // 同期グループの他の要素が逐次実行を完了する前に呼び出しに戻る
}
```

### 3.2.2 CrowdWalk の並列化

Waitless の barrier 及び ordered メソッドを用いると、エージェント単位にモジュール化した CrowdWalk（図 2.3）は、関心事の分離を保ったまま図 3.7 のように記述することができる、各エージェントの preUpdate 処理の後、同じ道へ移動するエージェントを判別し (goOnSameStreet)、それら一部のスレッド集合のみを対象にバリア同期を行っている。その後、実際に衝突するエージェントを計算し、その中で moveCommit をシーケンシャル実行する。これにより、同期対象のエージェントを最小化し、また、道を移動しないような場合には、他のエージェントと同期しないことが可能となる。

さらに、Waitless を用いることで、図 3.9 のように、時間ブロッキングを用いたシミュレーションも表現することが可能である。時間ブロッキングという手法は、主にステンシル計算を行う際に用いられ、各格子点の計算を行う際、複数のイテレーションをまとめて実行することで、周囲の格子点との同期回数を減らす手法である。図 3.7 のようなシミュレーションでは、歩行者は各イテレーションごとに他の全ての歩行者と暗黙的な同期を行っていた（図 3.8）。これに対して、時間ブロッキングの手法を導入すると、update メソッドをイテレーション順に

図 3.7. Waitless を用いた並列化

```
class Pedestrian {
  void preUpdate() {
    speed = calcSpeed();
    nextPlaces[i] = calcNextPlace();
  }

  void moveCommit() {
    places[i] = calcPlace();
  }

  void update(Waitless<Pedestrian> w) {
    preUpdate();
    if (goOnSameStreet()) {
     moveCommit();
    } else {
      // 近隣の歩行者を計算し、その歩行者が次に進むべき場所を計算し終えるまで待つ
      w.barrier(calcNeighborPedestrians(), a -> a.nextPlaces[i] != null);
      // 実際に衝突する歩行者を計算し、その歩行者グループに対して
      // 次の道に入る順番で moveCommit を逐次実行する
      w.ordered(calcConflictingPedestrians(), a -> a.places[i] != null, () -> {
        moveCommit();
      });
    }
    postUpdate();
  }

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

実行する処理全体が各エージェントで並列実行されるため、全体でのバリア同期は、衝突が発生する可能性のあるエージェントに限定することができ、別の道に移動しないエージェントは同期をせずに次のイテレーションに進むことが可能となる。

## 3.3　バリア条件到達コードの自動挿入

到達コード自動挿入による Point-to-Ponint 型への変換は、barrier メソッドの条件式に基づくソースコード解析によって行う。barrier メソッドに与えるバリア到達条件では、内包メソッドのスコープが継承される。そのため、条件式の真偽値が更新される箇所は、条件式でアクセ

図 3.8. 時間ブロッキング

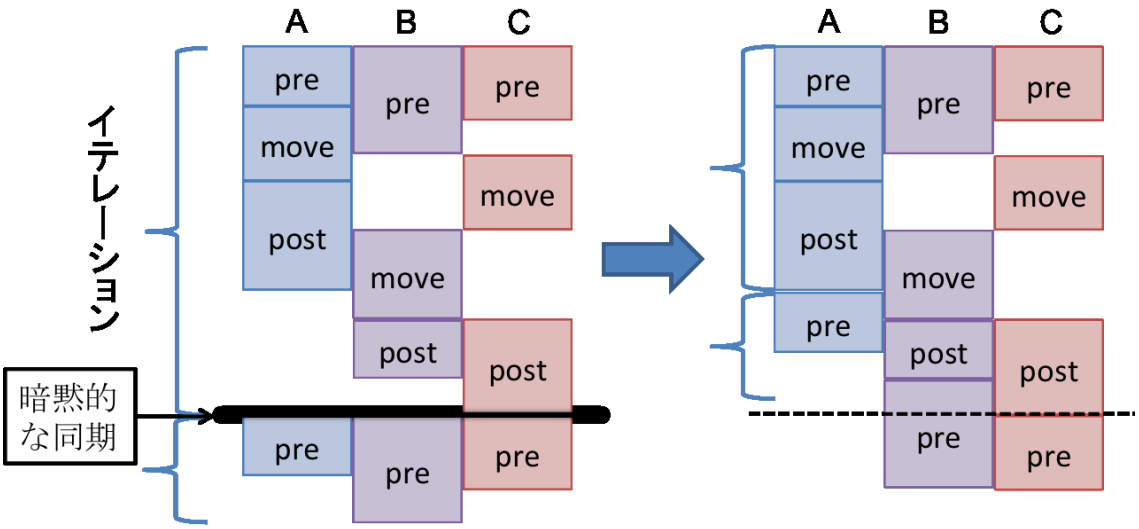



図 3.9. 時間ブロッキングを用いた並列化

```
class Pedestrian {
  // イテレーション番号
  int i;

  void update(Waitless<Pedestrian> w) {
    preUpdate();
    if (goOnSameStreet()) {
      moveCommit();
    } else {
      // 道を移動する場合には、他のエージェントが自分と同じイテレーションに達するまで待機する
      w.barrier(pedestrians, a -> a.i >= i);
      w.barrier(calcNeighborPedestrians(), a -> a.nextPlaces[i] != null);
      w. ordered(calcConflictingPedestrians(), a -> a.places[i] != null, () -> {
        moveCommit();
      });
    }
    postUpdate();
  }

  static void main() {
    // イテレーション順に update を実行する処理全体に対して各エージェントで並列実行する
    new WaitlessSet(pedestrians).parallelStream().forEach(pedestrian -> {
      for (int i = 0; i < maxIteration; i++) {
        pedestrian.update();
      }
    });
  }
}
```

スする変数が更新される場合に限られる。ただし、ローカル変数は、別スレッドから更新されることがないため、対象とする変数は条件式内で参照しているフィールド変数のみである。全クラスのフィールドやメソッドアクセスが静的に確定している場合、実行直前に制定木条件式内のコールフローを解析することで、参照される全ての変数を特定できる。条件式内で参照しているフィールド変数を特定した後、その変数に対して書き込みを行っているコードを、並列コレクション対する処理から検索することで、バリア到達コードを挿入すべき箇所を特定できる。例えば、図 3.10 の 8 行目のように、Pedestrian クラスのフィールド nextPlaces を参照する条件式が指定された場合、並列コレクションに対する処理として Pedestrian クラスの update メソッドが呼ばれており、そこから preUpdate を実行し、nextPlaces に対して代入を行っているので、バリア到達条件が満たされる可能性のある箇所は 3 行目の後だということがわかる。バリア到達コードの挿入はロードタイムに行われるため、フィールド変数が標準ライブラリによって更新される場合や、条件式の真偽値が更新がネイティブメソッドの返り値に依存する場合はロードタイムのバイトコード変換が不可能なため、条件式が Java の文法的に正しくてもプログラマの意図通りに動かない場合がある。

**図** 3.10. 条件式の解析と到達コードの挿入箇所の特定

```
1  class Pedestrian {
2    void preUpdate() {
3      ...;
4      nextPlaces[this.i] = calcNextPlace();
5      // ここにバリア到達コードを挿入する
6      ...;
7    }
8
9    void update(Waitless w) {
10     preUpdate();
11     ...;
12     w.barrier(neighbors(), a -> a.nextPlaces[this.i] != null);
13     ...;
14   }
15
16   public static void main() {
17     pedestrians.parallelStream().forEach((pedestrian, w) -> {
18       // ここから呼ばれるコールフローの中で
19       // バリア到達コードを挿入すべき箇所を検索する
20       pedestrian.update(w);
21     })
22   }
23 }
```

## 3.4 スレッドコードの自動分割

さらに、Waitless では、バリア同期を目印にしてスレッドコードを自動で分割することにより、モジュラリティを維持したまま、効率的な並列化を実現する。同期コードを目印にしてスレッドコードを分割することで、スレッド間の処理依存を排除することが可能となる。これにより、バリア同期を用いたスレッドコードを実行する場合でも、スレッドプールを利用することが可能となり、スレッドの同時実行数を一定に抑えられ、メモリ圧迫や実行性能低下を回避することができる。

Waitless では、スレッドコードの分割を実現するため、以下に示す処理を行うためのバイトコードをロードタイムに挿入する。まず、バリア待機時にコールスタックと全ローカル変数を保存し、スレッドを終了させる。その後、全ての同期対象が条件式を満たした後、バリア後の処理を再開するために終了したスレッドを再度実行する。この際、保存したコールスタックを辿りながら、ローカル変数を復元し、実行済みのコードは goto でスキップする。

# 第 4 章

# 実装

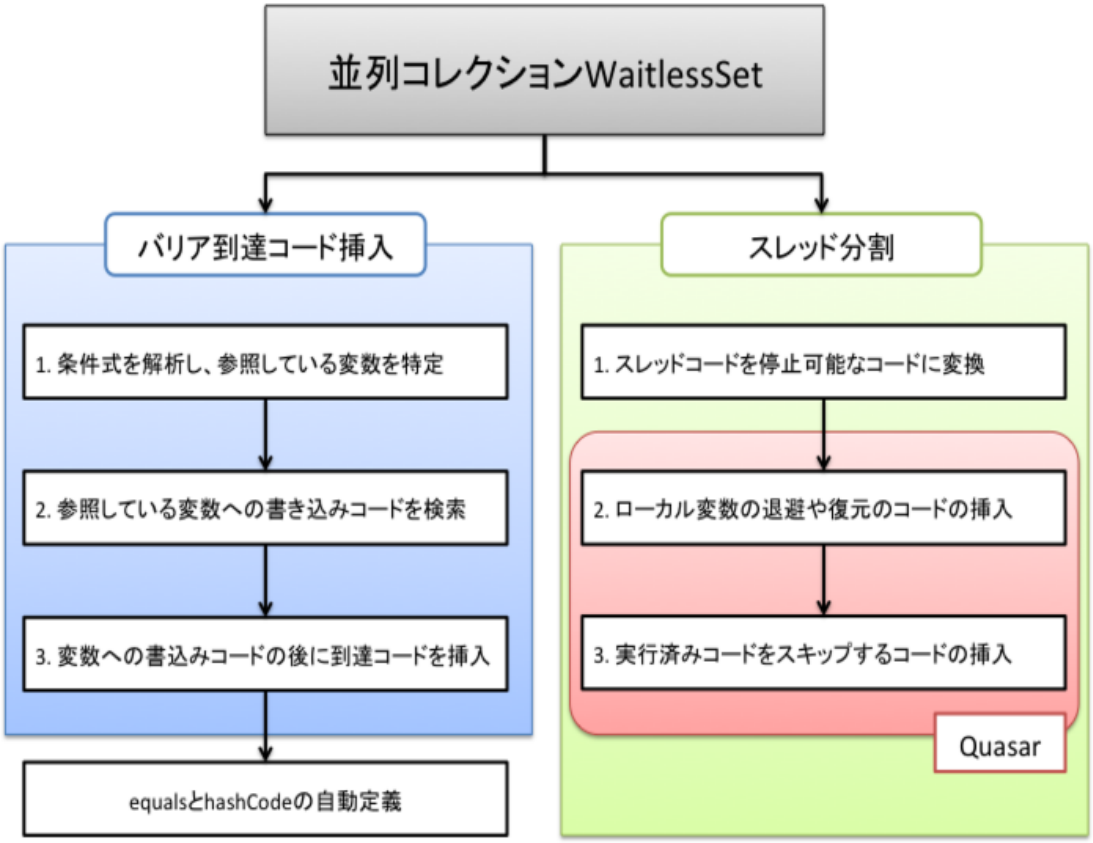
本研究では、ロードタイムにバイトコード変換を行うことにより、軽量な条件付きバリア同期 Waitless を実現する。ロードタイムにコード変換を行うことにより、利用者は特別な Java コンパイラを利用する必要がなく、-javaagent という Java 標準の実行オプションに対して、Waitless のバイトコード変換クラスである com.hottestseason.waitless.instrument.JavaAgent を指定することで、バイトコード変換を行うことが出来る。Waitless の JAR ファイルのマニフェストの属性 Premain-Class には waitless.instrument.JavaAgent が代入されており、Java 仮想マシンの初期化後、実際のアプリケーションの main メソッドが呼ばれる前に、このクラスの premain メソッドが実行される。waitless.instrument.JavaAgent クラスは、バイトコード操作ライブラリの ASM[16] を用いて、ロードされたクラスのバイトコードを解析や変換を行う。

## 4.1 ASM によるバイトコード変換

ASM は OW2 Consortium のプロジェクトの一つで Java のバイトコードを操作するためのフレームワークである。ASM には org.objectweb.asm という Visitor パターンによりバイトコードを操作する API と、org.objectweb.asm.tree というバイトコードをツリー構造として扱える API がある。前者は、現在参照しているオブジェクト以外をメモリ上にロードしないため、高速で、かつ、省メモリにバイトコード変換を行える。一方、後者は、一度クラスのバイトコード全てを解析し、そこからツリー構造を作成するため、余分なオーバーヘッドが発生するものの、より複雑なバイトコードの解析を行うことが可能である。本研究ではプロトタイプであることや、複雑な解析・変換が求められることから、実装の容易な後者の API を用いてバイトコード変換を行った。

Waitless は、ロードタイム時に main メソッドから辿れる全てのクラス・メソッドを探索し、並列コレクション WaitelsssSet を検索し、そこから図 4.1 に示すようなバイトコード変換を行う。

**図** 4.1. Waitless によるバイトコード変換




## 4.2 バリア到達コードの自動挿入

まず Waitless は、並列コレクション WaitlessSet 内で利用されている barrier メソッドを検索し、第 2 引数で渡された Predicate 型の条件式を特定する。続いて、並列コレクションに対する操作から辿れるコールフロー内で、条件式で参照しているフィールド変数に対して書き込みを行っているバイトコードを検索する。検索するバイトコードは主に対象フィールドに対する putfield 命令だが、フィールド変数が配列の場合、aastore などによって特定の要素のみが更新される場合もあるため、それらについても検索を行う。例えば、図 3.7 の 15 行目のような条件式が与えられた場合、同期対象のオブジェクトに対して、参照を行っているインスタンス変数は nextPlaces である。nextPlaces が変更されている場所は、preUpdate メソッドの AASTORE 命令（図 4.2 の 6 行目）となる。

最後に、発見した書き込みコードの直後に Waitless クラスの checkArrived メソッド (表 4.1) 呼び出しの挿入を行う。checkArrived メソッドは実行中の並列コレクションの要素に対してバ

**図** 4.2. preUpdate メソッドの Java バイトコード

```
    void preUpdate();
    descriptor: ()V
    Code:
      aload_0
      getfield #65 // Field nextPlaces:[Lcom/hottestseason/hokolator/map/Map$Street$Place;
      aload_0
      getfield #60 // Field iteration:I
      aload_1
      aastore
      return
```

リア待機を行っている要素を検索し、バリア同期の条件式を評価し、真の場合にはバリア到達を呼び出す。現在実行中の並列ストリームの Waitless インスタンスは Waitless.current というスレッドローカルなスタティック変数に保存しているため、checkArrived は current 変数経由で実行する。図 3.10 のコードに対して、checkArrived を挿入すると図 4.2 のようになる。

但し、現在の実装では、条件式に直接記述された変数への書き込みコードのみ検索し、メソッド呼び出し先で参照されている変数に対する書き込みコードの検索は行わない。また、リフレクションなどにより実行時に参照される変数が変化する場合への対応も行わない。

**表** 4.1. Waitless

| Class Waitless⟨T⟩ | |
|---|---|
| static ThreadLocal⟨Waitless⟩ | current<br>現在参照中の Waitless インスタンスを保存するための<br>スレッドローカルなスタティック変数 |
| void | checkArrived()<br>自分に対して待機を行っているバリア同期を検索し、<br>条件式の評価と到達呼び出しを行う |

```
// 変換前
void preUpdate() {
  speed = calcSpeed();
  nextPlaces[this.i] = calcNextPlace();
}
// 変換後
void preUpdate() {
  speed = calcSpeed();
  nextPlaces[this.i] = calcNextPlace();
  Waitless.current.get().checkArrived();
}
```

## 4.3 equals メソッドと hashCode メソッドの自動定義

更に、Waitless はバリア同期の条件式の評価結果のメモ化を行うため、equals と hashCode メソッドが定義されていない条件式クラスに対して、自動的にメソッド定義を行う。図 4.3 のように、並列コレクションに対する操作の中で、ラムダ式 (3 行目) や無名内部クラス (5-8 行目) を用いて条件式を与える場合、並列コレクションの各要素ごとに別の条件式オブジェクトが作成される。そのため、クラスとフィールドが同じ条件式オブジェクトに対して、同じ入力で評価した場合でも、結果を再利用することができず、無駄な計算が発生する。Waitless は、これを防ぐため、クラスやフィールドの情報を元に、barrier メソッドの条件式クラスに対して、オブジェクトの同値性の評価に用いられる equals メソッド (24 行目) と hashCode メソッド (32 行目) を自動的に定義する。

## 4.4 スレッドコードの分割

最後に、Waitless は Quasar[17] というライブラリを用いてスレッドコードの分割を行う。Quasar は Parallel Universe で開発されているライブラリで、Java のための軽量スレッド Fiber を提供しており、ロードタイムに co.paralleluniverse.fibers.instrument.JavaAgent クラスの premain を実行することで、3.3 で述べたようなローカル変数の退避や復元などの処理をバイトコード変換により挿入する。並列ストリーム WaitlessStream は forEach の引数として受け取る BiConsumer 型の処理コードを、Fiber で用いられる停止可能な処理コードに変換するため、SuspendableBiConsumer 型に変換する（図 4.4)。処理コード SuspendableBiConsumer 内で barrier メソッドによりバリア待機が行われると、Waitless は現在実行中の軽量スレッド Fiber.currentFiber() に対して Fiber.park メソッドを呼び出し、ローカル変数を退避し、SuspendExecution という例外を投げることで、現在実行中のスレッドが終了する。その後、checkArrived メソッドによりバリア到達が行われると、Waitless は Fiber.unpark メソッドを用いて、前回 Fiber.park メソッドが呼ばれた場所まで goto を用いてスキップし、ローカル変数を復元し、処理の続きを行う。

**図 4.3.** バリア同期の条件式に対する equals と hashCode メソッドの挿入例

```
// 変換前
new WaitlessSet(pedestrians).parallelStream().forEach((p, w) -> {
  w.barrier(neighbors(), _p -> _p.nextPlaces[p.i] != null);
  // または
  w.barrier(neighbors(), new Predicate<Pedestrian> {
    public boolean test(Pedestrian _p) {
      return _p.nextPlaces[p.i] != null;
    }
  });
});

// 変換後
new WaitlessSet(pedestrians).parallelStream().forEach((p, w) -> {
  w.barrier(neighbors(), new CheckNextPlace(p.i));
});

class CheckNextPlace implements Predicate<Pedestrian> {
  private Integer i;

  CheckNextPlace(Integer i) { this.i = i; }

  public boolean test(Pedestrian _p) {
    return _p.nextPlaces[i] != null;
  }

  public boolean equals(Object o) {
    if (o instanceof CheckNextPlace) {
      return ((CheckNextPlace) o).i.equals(i);
    } else {
      return false;
    }
  }

  public int hashCode() {
    return CheckNextPlace.class.hashCode() ^ i.hashCode();
  }
}
```

**表 4.2.** Quasar の軽量スレッド Fiber

| Class Fiber⟨V⟩ | |
|---|---|
| コンストラクタ | Fiber(SuspendableRunnable target)<br>SuspendableRunnable を実行する新しい軽量スレッド Fiber を作成する |
| static Fiber | currentFiber()<br>現在実行中の Fiber を返す |
| void | park()<br>ローカル変数や実行済みのコードを保存し、Fiber を停止する |
| void | unpark()<br>停止中の Fiber を再開する |
| void | join()<br>Fiber が終了するまで現在実行中のスレッドを待機させる |

**図 4.4.** SuspendableBiConsumer への変換

```
// 変換前
new WaitlessSet(pedestrians).parallelStream().forEach(new PedestrianUpdation());

class PedestrianUpdation implements BiConsumer<Pedestrian, Waitless<Pedestrian>> {
  public void accept(Pedestrian pedestrian, Waitless<Pedestrian> w) {
    pedestrian.update(w);
  }
}

// 変換後
new WaitlessSet(pedestrians).parallelStream().forEach(new PedestrianUpdation());

// 実装するインターフェースを BiConsumer を SuspendableBiConsumer に変換する
class PedestrianUpdation
        implements SuspendableBiConsumer<Pedestrian, Waitless<Pedestrian>> {
  // 例外 SuspendableExecution を投げるように変換する
  public void accept(Pedestrian pedestrian, Waitless<Pedestrian> w)
                  throws SuspendableExecution {
    pedestrian.update(w);
  }
}
```

# 第 5 章

# 実験

Waitless による実行性能を確認するために，CrowdWalk のソースコードを参考に最低限の歩行者シミューレションの機能を実装したサブセットに対して，Waitless のバリア同期を実装し実験した．入力としては，長さが 10m で，収容能力が 5 人の道からなる格子状のマップを与え，移動速度が 1-10m/s の歩行者を一様に配置した．ゴールをマップに対して 300m 間隔で設置し，エージェントは最も近いゴールを目指して進み，10 回イテレーション繰り返したシミュレーションを 10 回実行し，最大と最小を除いた 8 回の平均の最大同時実行スレッド数，メモリ使用量，実行時間を測定した．

実験は以下の環境で行った。

- CPU: Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz
- Memory: 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)
- VM Option: -Xm56g -Xmx56g -XX:+UseG1GC -XX:InitiatingHeapOccupancyPercent=0

## 5.1 バリア同期対象の制限による効果

バリア同期対象の制限による効果を調べるため、エージェントの数を 1 から 10,000 まで変化させた時の All-to-All 型のバリア同期を行った場合と、近隣のエージェントや実際に衝突するエージェントのみに対してグループバリア同期を行った場合の実験を行った。図 5.1 から図 5.4 がその時の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間を測定した結果であり、横軸のエージェント数が対数スケールの片対数グラフである。表 5.1 はエージェント数が 10,000 の時の All-to-All 型とグループバリア同期の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間である。

図 5.1 より、All-to-All 型のバリア同期を用いて、全エージェントを同期させる場合、エージェント数の増加とともに最大同時実行スレッド数が単調増加し、エージェント数 10,000 の時には 10,021 となっている。それに伴い、図 5.2 からエージェント数が増加するにつれメモ

リ使用量が増加していることがわかる。これは、待機スレッドによって専有されるメモリ使用量が増加したためだと推測される。さらに、図 5.4 より、All-to-All 型のバリア同期を用いて並列化を行った場合、エージェント数が大規模になると実行時間が急激に増加していることがわかる。これは、スレッド生成やコンテキストスイッチなど、エージェント数と同数のスレッドを管理するオーバーヘッドによるものと考えられる。一方、グループバリア同期を用いて並列化を行った場合、エージェント数を 10,000 まで増加させても最大同時実行スレッド数は 63 に抑えられ、メモリ使用量は 46.99% に削減され、さらに、実行時間は 9.2% に短縮することができ、大規模なシミュレーションサイズになっても急激な性能低下は見られないことが確認できた。

**図 5.1.** バリア同期対象の制限による最大同時実行スレッド数の比較

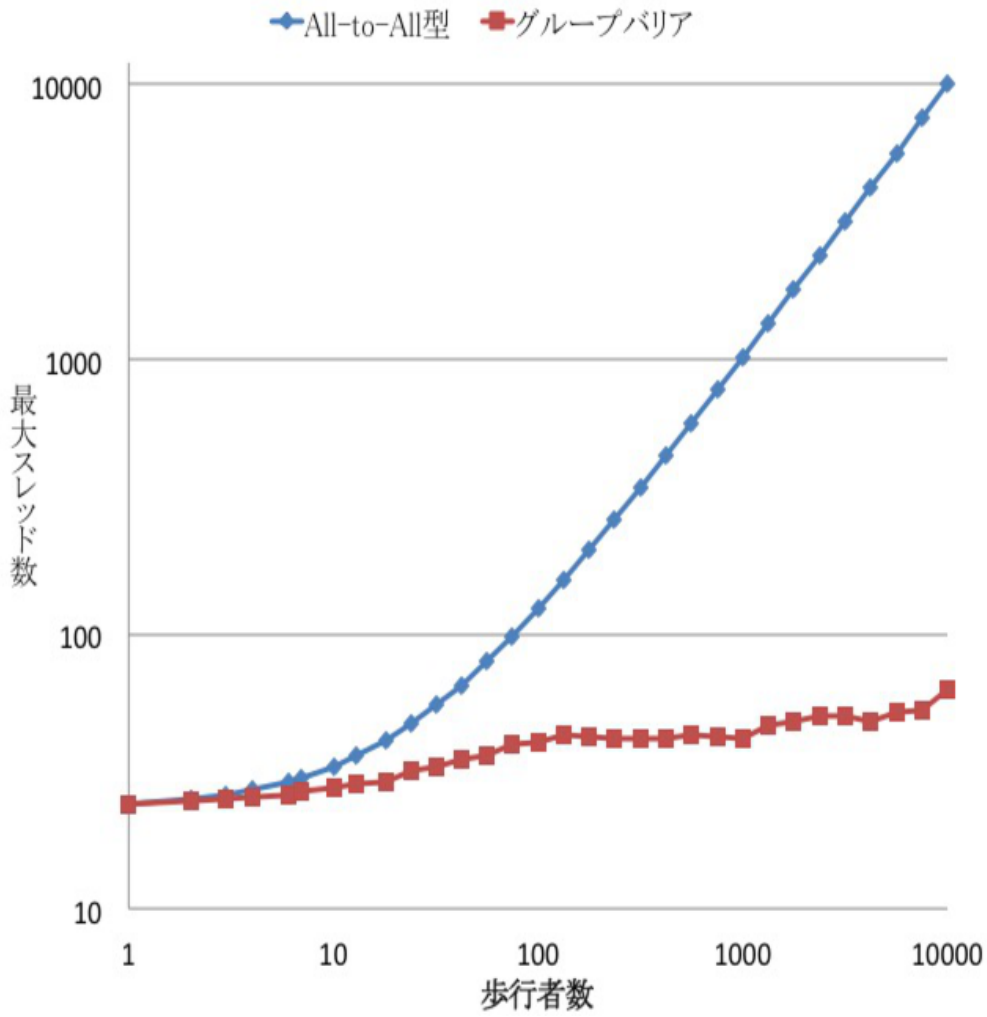


**図 5.2.** バリア同期対象の制限によるメモリ使用量の比較

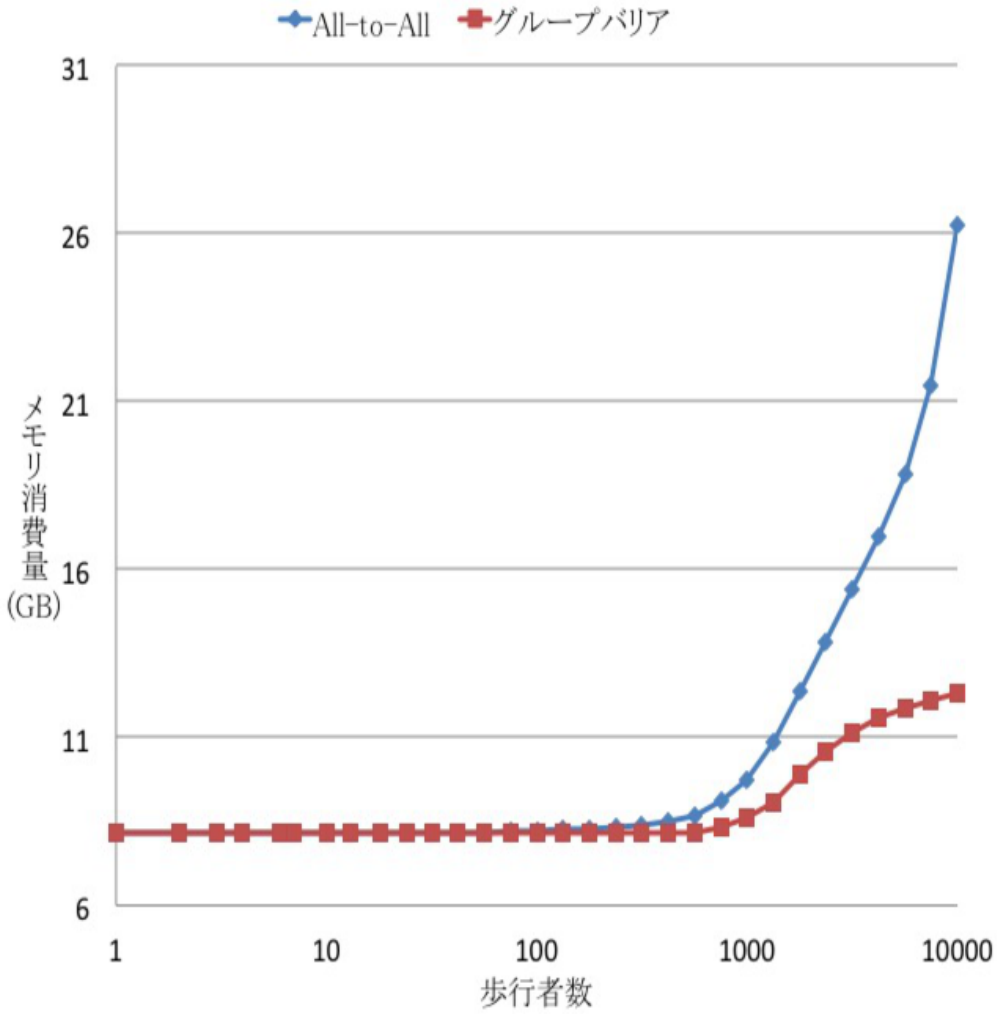
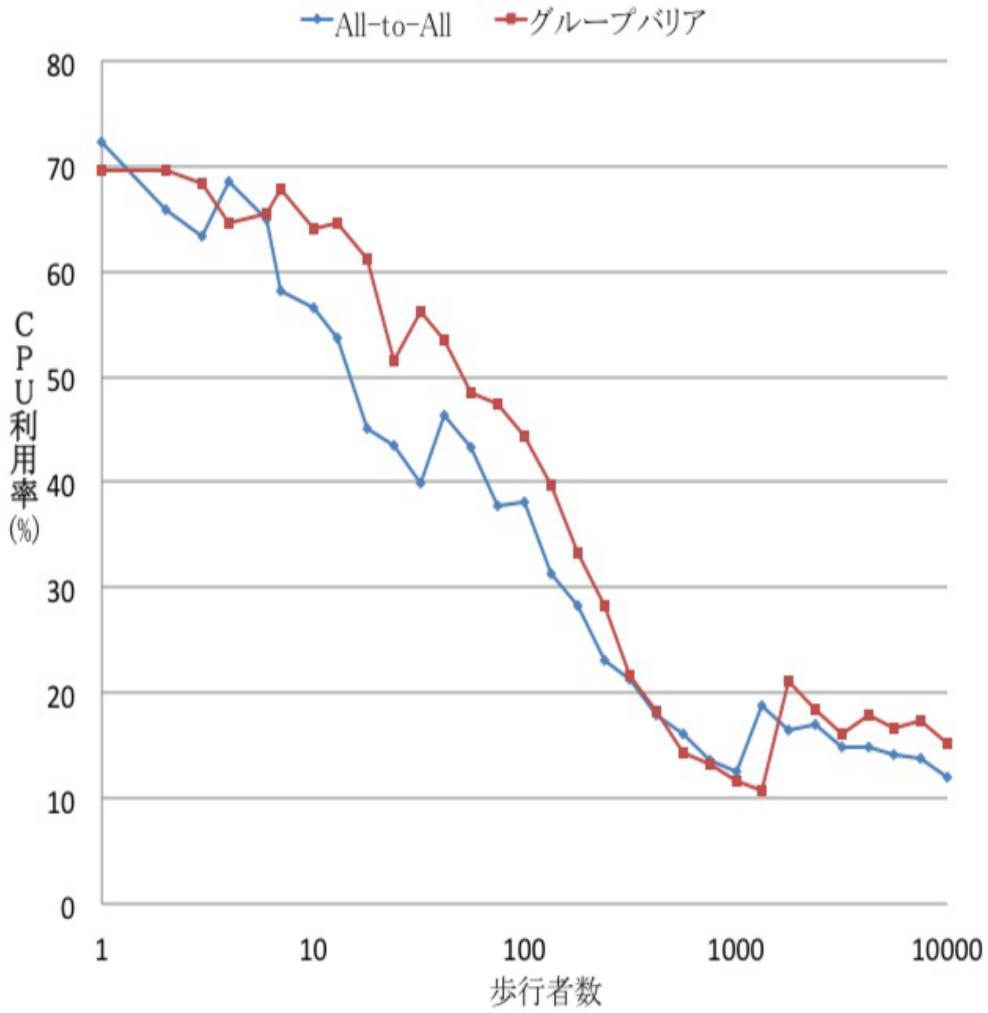



**表 5.1.** バリア同期対象の制限によるエージェント数 10,000 の時の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間

| | All-to-All 型 | グループバリア |
|---|---|---|
| 最大同時実行スレッド数 | 10,021 | 63 |
| メモリ使用量 (GB) | 26.24 | 12.33 |
| CPU 利用率 (%) | 11.99 | 15.22 |
| 実行時間 (s) | 521.6 | 47.93 |

**図 5.3.** バリア同期対象の制限による CPU 利用率の比較

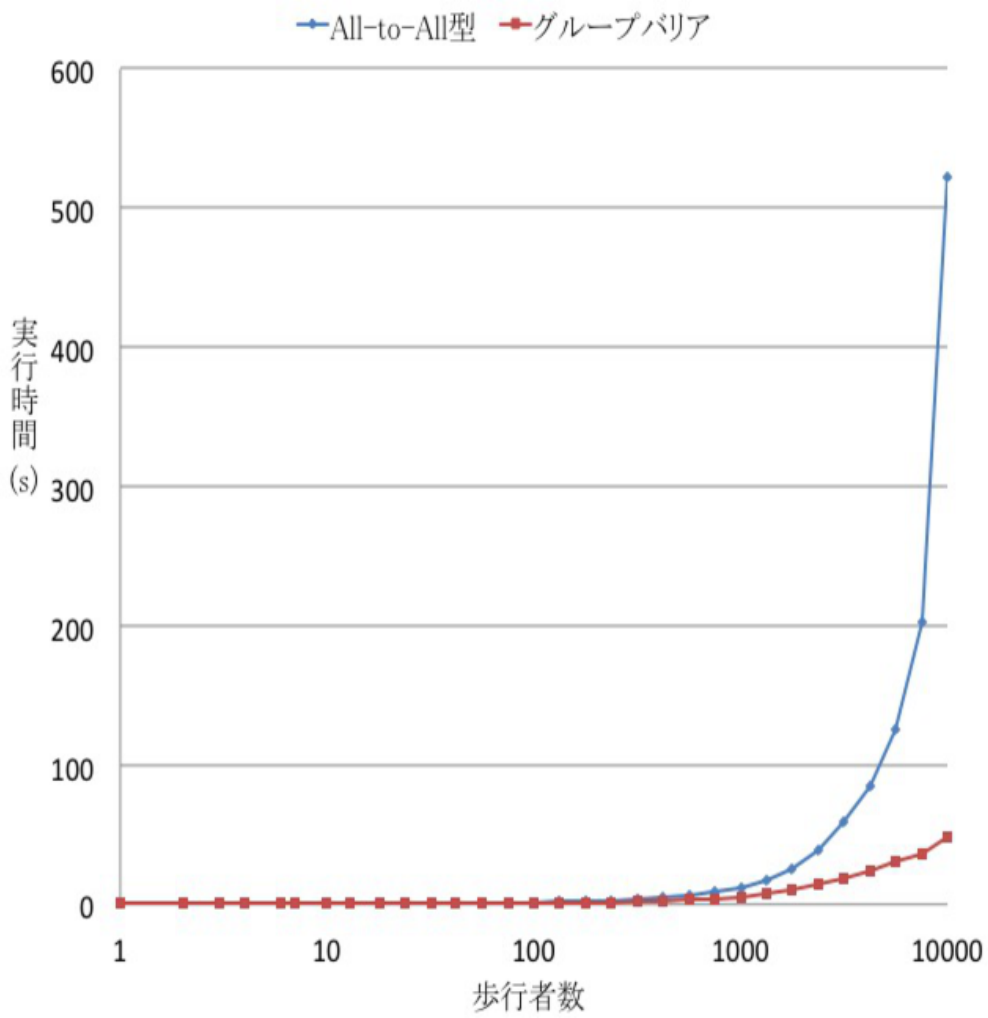



## 5.2 スレッド分割による効果

次に、スレッド分割を行い、スレッドプールを用いた場合のスレッド数削減の効果を測定するための実験を行った。図 5.5 から図 5.8 は Watiless を用いてグループバリア同期を行った上で、ロードタイムのスレッド分割の有無を変えて比較した場合で、エージェント数を 1 から 237,137 まで変化させた時の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間を測定した結果であり、こちらも横軸のエージェント数が対数スケールの片対数グラフである。表 5.2 はエージェント数が 237,137 の時の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間である。

図 5.5 より、グループバリア同期を行った場合でも、スレッド分割をしない場合、エージェント数が 10,000 を超えた辺りから最大同時実行スレッド数が急激に上昇しており、エージェ

**図** 5.4. バリア同期対象の制限による実行時間の比較




ント数 237,137 の時には 10,104 となっている。これは、エージェント数が上昇したため、同期する対象を近隣のエージェントや実際に衝突するエージェントに限定したとしても、同期時に大量の待機スレッドが発生したためだと思われる。これに対して、Waitless を用いてロードタイムにスレッドコードを自動分割し、スレッドプールを用いることで、エージェント数が 237,137 になってもスレッド数を 23 に抑えることが可能となった。これに伴い、図 5.6 より待機スレッドによって専有されるメモリもエージェント数が 237,137 の時、87.07% に削減できたことが確認できた。また、図 5.7 より、スレッド分割を行った場合、CPU 利用率が 6.60 倍に向上し、さらに、図 5.8 より、実行時間が 1.93% に短縮できたことが確認できる。これは、スレッド生成やコンテキストスイッチなどの大量のスレッドを管理するオーバーヘッドが抑えられたためだと推測され、スレッド分割により、シミュレーションサイズが大規模になっても、効率的な並列化を行うことができたことが示された。

**図 5.5.** スレッド分割の有無による最大同時実行スレッド数の比較

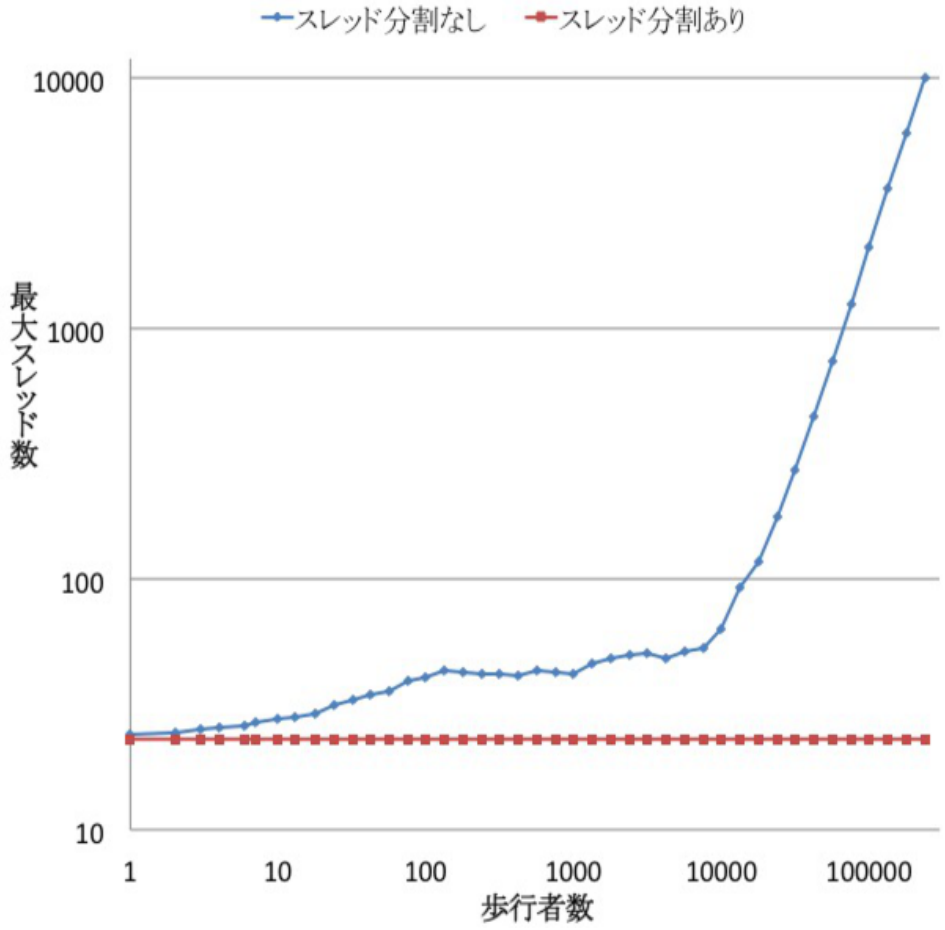


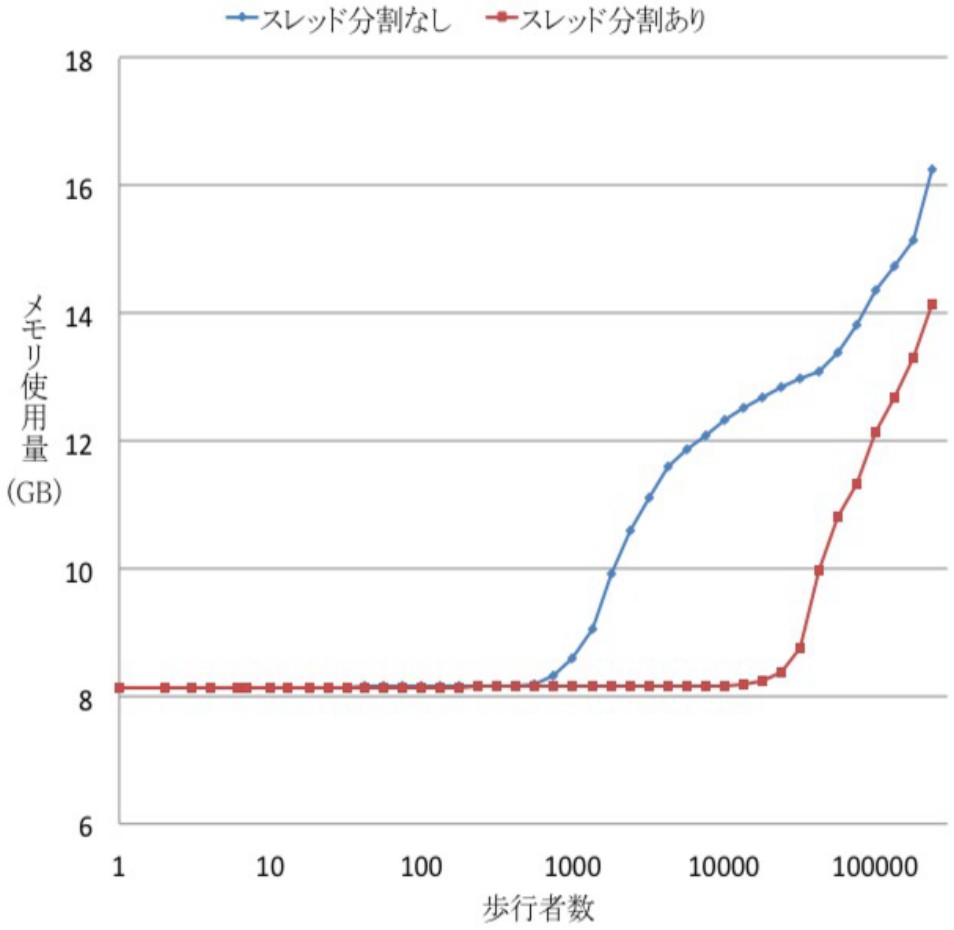

**表 5.2.** スレッド分割の有無によるエージェント数 237,137 の時の最大同時実行スレッド数、メモリ使用量、CPU 利用率、実行時間

| | スレッド分割なし | スレッド分割あり |
|---|---|---|
| 最大同時実行スレッド数 | 10,104 | 23 |
| メモリ使用量 (GB) | 16.24 | 14.14 |
| CPU 利用率 (%) | 13.48 | 89.02 |
| 実行時間 (s) | 1387 | 26.85 |

## 5.3 時間ブロッキングによる効果

さらに、時間ブロッキングによる効果を調べるため、エージェントの数を 1 から 1,000,000 まで変化させた時の、Waitless を用いてスレッド分割された後のプログラムと、そこに、さらに図 3.9 のように時間ブロッキングを導入した場合の実験を行った。図 5.9 と図 5.10 は、そ

**図** 5.6. スレッド分割の有無によるメモリ使用量の比較




の時の CPU 利用率、実行時間を測定した結果であり、横軸のエージェント数が対数スケールの片対数グラフである。表 5.3 はエージェント数が 1,000,000 の時の CPU 利用率、実行時間である。

図 5.9 から、時間ブロッキングという手法を用いると、CPU 利用率が向上し、エージェント数 1,000,000 の時には 104.2% に増加しており、さらに、図 5.10 より実行時間が 91.47% に短縮できることが確認できた。これは、時間ブロッキングにより、イテレーションごとに全体で行われていた暗黙的な同期がなくなり、別の道に移動せず、他のエージェントと衝突が起きないエージェントは一切同期せずにが先のイテレーションに進めることできたため、シミュレーション全体の同期数を削減できたことが原因だと考えられる。

**図 5.7.** スレッド分割の有無による CPU 利用率の比較

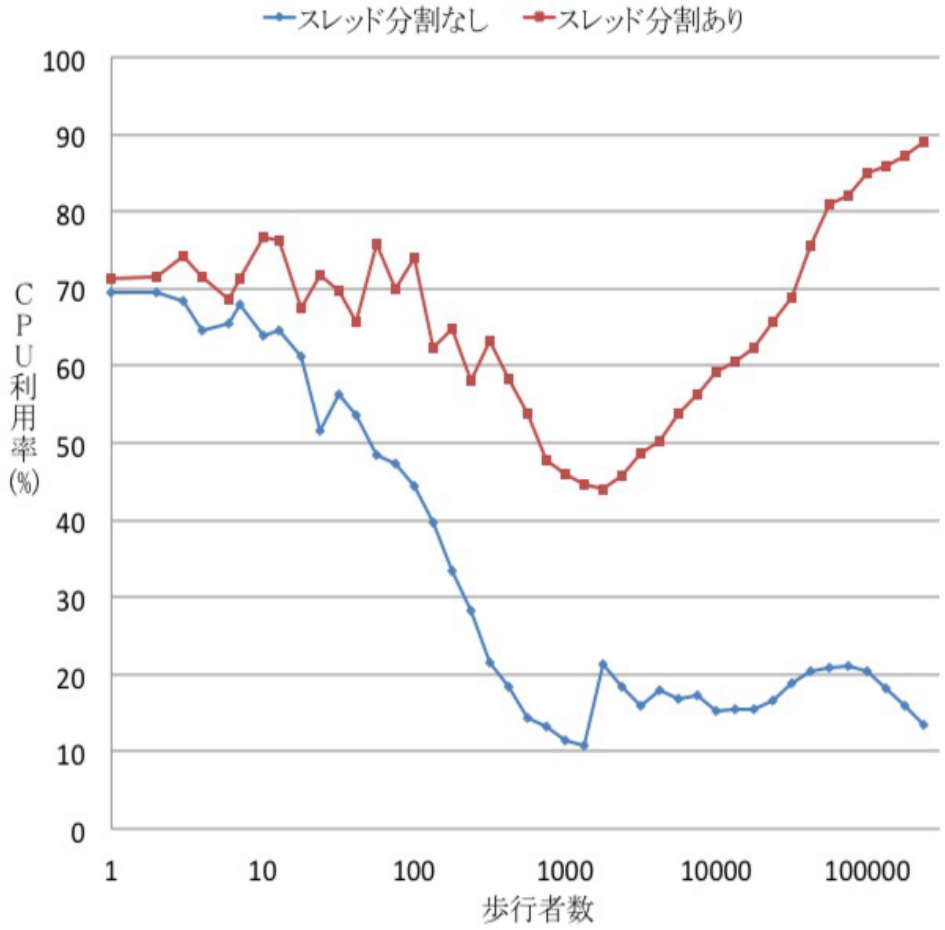



**表 5.3.** 時間ブロッキングの有無によるエージェント数 1,000,000 の時の CPU 利用率、実行時間

| | 時間ブロッキングなし | 時間ブロッキングあり |
|---|---|---|
| CPU 利用率 (%) | 86.25 | 89.85 |
| 実行時間 (s) | 98.25 | 89.87 |

**図** 5.8. スレッド分割の有無による実行時間の比較

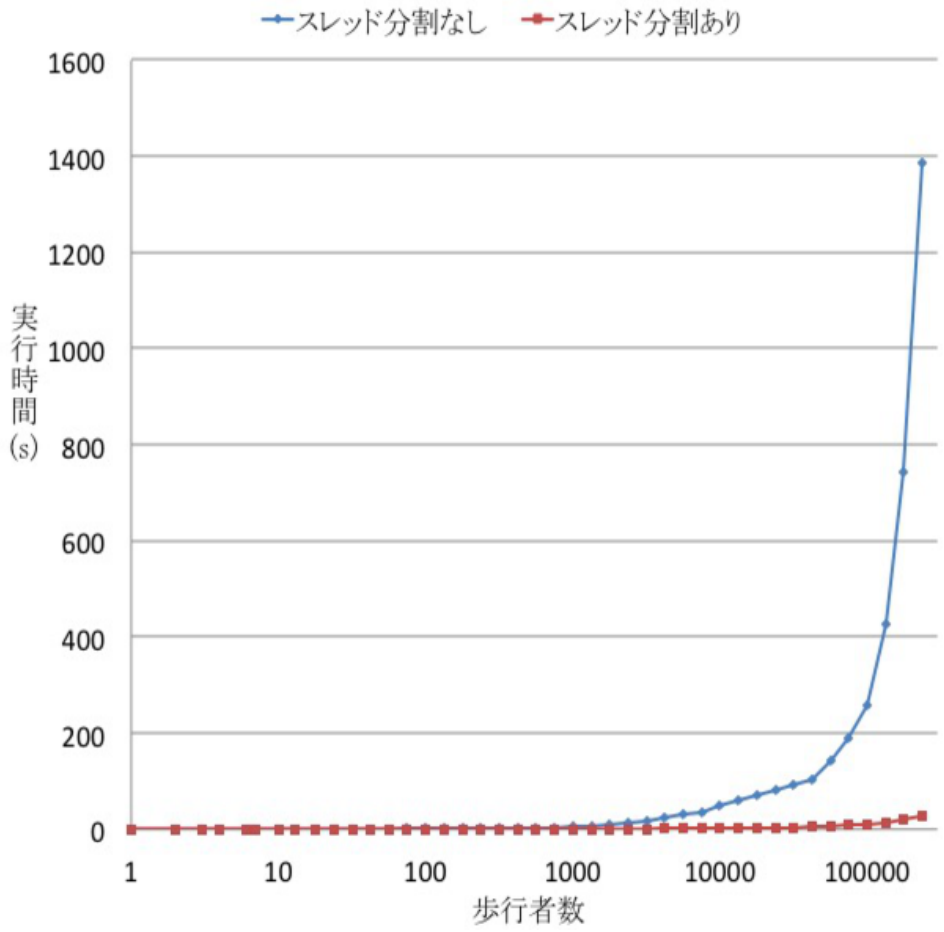


**図** 5.9. 時間ブロッキングの有無による CPU 利用率の比較

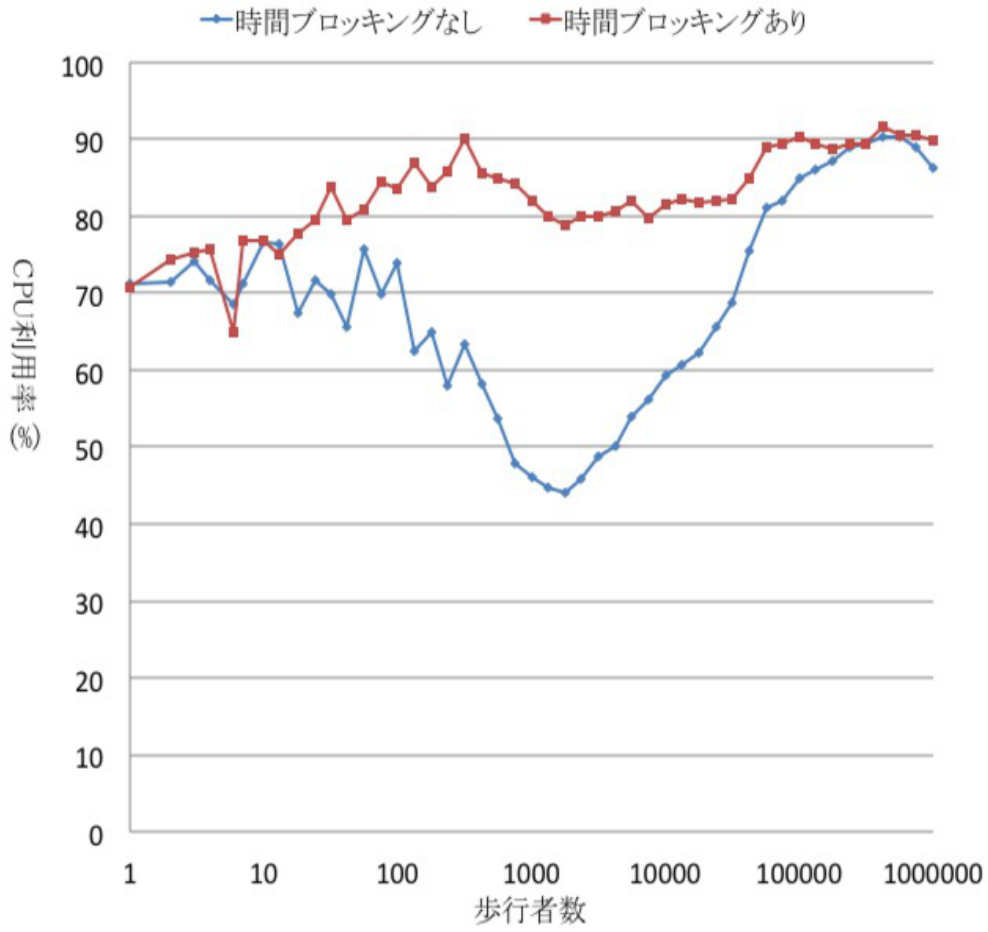
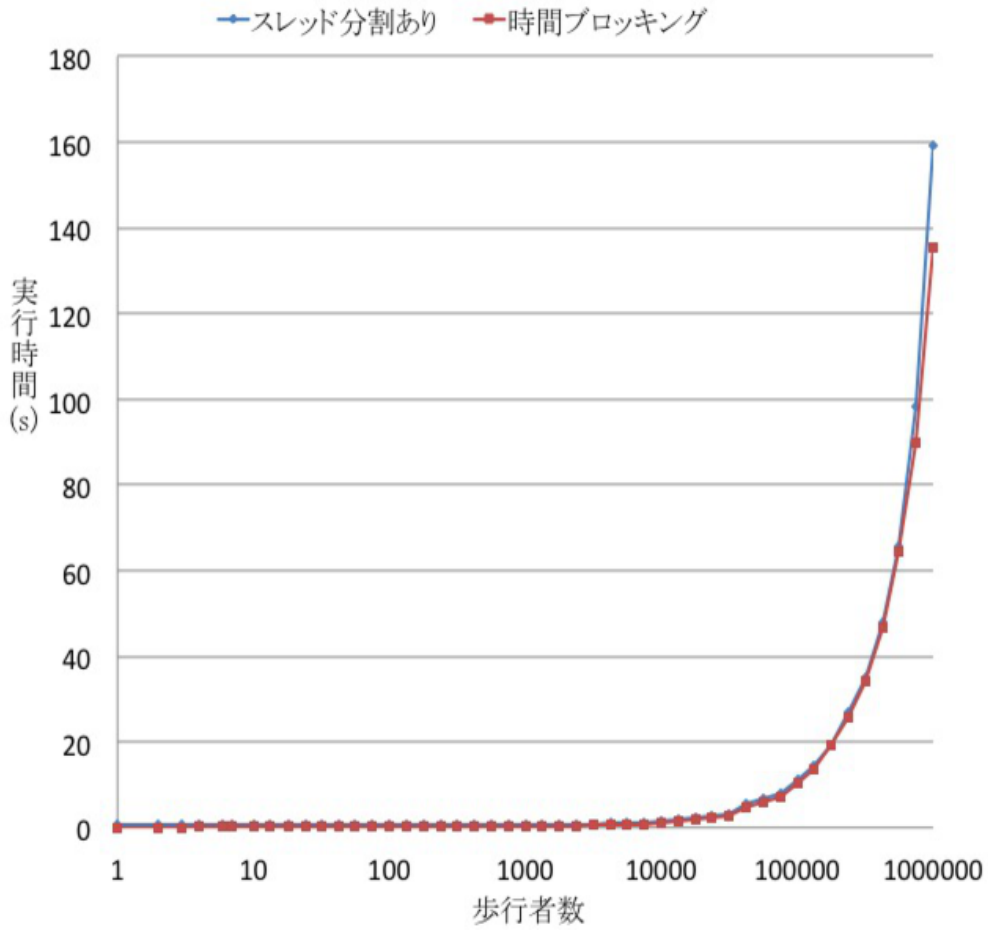


**図** 5.10. 時間ブロッキングの有無による実行時間の比較



# 第 6 章

# まとめと今後の課題

## 6.1 まとめ

本研究では、ロードタイムにバイトコード変換を行うことで、実行時情報に基づいてバリア同期の条件式が記述可能な同期機構 Waitless を提案した。既存のバリア同期を用いる場合、All-to-All 型では、レガシーコードを容易に並列化できる反面、過剰な同期スレッドを誘発していた。また、Point-to-Point 型のバリア同期は、複雑な同期処理が記述可能で、それにより待機スレッドを削減することが可能だが、到達処理コードが散財するため、モジュラリティが低下してしまう。それらに対し、Waitless は散在するバリア到達処理をオブジェクトの状態に基づく直感的な条件式として記述可能なため、Point-to-Point 型のような複雑な同期を All-to-All 型のようなアブストラクションで記述することが可能となり、エージェントシミュレーションのような抽象度の高いソフトウェアでも、モジュラリティを低下させることなく並列化可能となる。また、既存のバリア同期では、シミュレーションサイズの増加により、待機スレッド数が増加した場合、CPU 実行効率が低下するばかりでなく、その専有メモリ量も増加する。そこで Waitless ではバリア待機コードを目印にスレッド分割を行うことで、待機スレッド数の削減を行った。

Waitless は ASM という Java のバイトコード操作ライブラリ用いて、バリア条件式の解析やバリア到達処理コードの自動挿入を行った。また、スレッド分割には Quasar という軽量スレッドを提供するライブラリを用いて実現した。

Waitless の有用性を示すため、CrowdWalk のソースコードを参考に最低限の機能を実装したサブセットに対して実験を行い、既存のバリア同期と比較して、スレッド数が削減され、メモリ使用量の減少やパフォーマンスの改善につながることを確認した。

## 6.2 今後の課題

今後はさらに多くのアプリケーションに適用することで、他の並列計算のためのライブラリや言語に対する有用性を示す必要がある。また、現在の Waitless では、バリア条件式内に直接インスタンス変数を参照するコードを記述しなければ、変数の変更を追跡できないという問

題点ある。今後は、条件式内でメソッドを記述可能にするため、条件式のより詳細な解析を行うことで、変数の変更を追跡する必要がある。

# 発表文献と研究活動

(1) 夏澄彦, 佐藤芳樹, 千葉滋.　依存関係をもった並列タスクのための動的グループバリア同期とその効率的な実装.　日本ソフトウェア科学会第 31 回大会, 2014.9.7-10.

# 参考文献


[1] Message Passing Interface Forum. MPI a message-passing interface standard 3.0, 2012.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[3] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.4, May 2012. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, Vol. 21, No. 3, pp. 291–312, August 2007.

[5] The chapel parallel programming language. `http://chapel.cray.com/`.

[6] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, Vol. 9, No. 1, pp. 21–65, February 1991.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 519–538, October 2005.

[8] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pp. 277–288, New York, NY, USA, 2008. ACM.

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pp. 51–61, New York, NY, USA, 2011. ACM.

[10] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, Vol. 7, No. 4, pp. 501–538, October 1985.

[11] Shams Imam and Vivek Sarkar. Cooperative scheduling of parallel tasks with general

synchronization patterns. In *ECOOP 2014–Object-Oriented Programming*, pp. 618–643. Springer, 2014.

[12] RIFE java web application framework. `http://rifers.org/`.

[13] Apache commons. javaflow. `http://commons.apache.org/sandbox/javaflow/`.

[14] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pp. 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] 千葉 滋夏 澄彦. 依存関係をもった並列タスクのための動的グループバリア同期とその効率的な実装. 日本ソフトウェア科学会第 31 回大会, 2014.

[16] Objectweb asm. `http://asm.ow2.org/`.

[17] Quasar. `http://docs.paralleluniverse.co/quasar/`.

# 謝辞

本研究を進めるにあたって、素晴らしい研究環境やご指導・ご助言を与えていただいた指導教員の千葉滋教授に心より感謝致します。また、東京大学情報基盤センターの佐藤芳樹特任講師には、研究の詳細や論文の執筆に関して多くのアドバイスを頂きました。暑くお礼を申し上げます。

さらに、産業総合研究所の野田五十樹氏、山下倫央氏には、産総研で開発している CrowdWalk のソースコードの提供を始め、様々な情報や助言をいただきました。深く感謝いたします。

最後に、様々な助言を頂くとともに、気分転換に付き合って頂いた千葉研究室や I-ref 棟の皆様に深く深く感謝致します。