情報処理学会論文誌　プログラミング　Vol.8　No.3　11–21 (Sep. 2015)

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

夏 澄彦[1,a)]　佐藤 芳樹[2]　千葉 滋[1]

受付日 2015年2月9日, 採録日 2015年5月25日

概要：計算に依存関係のあるプログラムの並列化には，バリア同期を利用した協調処理の記述が有用である．従来のバリア同期は過剰な同期スレッドを誘発するため，Java では CountDownLatch や Phaser のようにバリア待機処理とバリア到達処理をソースコード中の任意の位置に分けて記述できる Point-to-Point 型の同期ライブラリが用いられている．しかし，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，バリア到達処理コードの散在によるモジュラリティの低下も問題となってくる．また，大規模なシミュレーションを行う場合，同期時の待機スレッド数増加による CPU 実行効率低下や占有メモリの増加を防ぐため，ユーザが明示的にスレッドコードを分割する必要があり，並列化のための記述量が増加する．そこで，本論文では，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，条件が満たされる可能性のある箇所すべ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を自動的に分割することで，モジュラリティを維持したまま，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．

キーワード：並列分散処理，バリア同期，継続，バイトコード変換

A Lightweight and High Abstracted Conditional Barrier Synchronization

Sumihiko Natsu[1,a)]　Yoshiki Sato[2]　Shigeru Chiba[1]

Received: February 9, 2015, Accepted: May 25, 2015

Abstract: Cooperative computing with barrier synchronization is a typical approach to the parallization of a program involving dependency among its computations. Since traditional synchronization primitives often cause excessive parallelism, Java provides CyclicBarrier and Phaser, which implements point-to-point synchronization; they separate arrival at a barrier and release from a barrier. However, these primitives degrade code modularity since synchronization code is spread over the program if it deals with higher-level abstraction, for example, it is for agent simulation. Furthermore, if a program is for large-scale simulation, these primitives do not help shorten the life time of each thread so that the number of waiting threads will not be too many to cause excessive parallelism or too large memory footprint. Programmers have to manually do that and hence the code modularity is further degraded. This paper proposes a new primitive for barrier synchronization. Programmers can describe barrier synchronization with this primitive in a modular fashion. The computations at arrival at a barrier is spread over the program but they are described at one place with a condition expression on the state of objects. The proposed primitive performs bytecode translation at load time so that the code necessary at arrival at a barrier is automatically inserted according to that condition expression. Moreover, the proposed primitive splits a thread into two at barrier synchronization; the number of waiting threads is thereby decreased without sacrificing modularity. This paper also presents the result of the application of the proposed primitive to large-scale pedestrian simulation.

Keywords: parallel distributed processing, barrier synchronization, continuation, bytecode transformation

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[2] 東京大学情報基盤センター
Information Technology Center, The University of Tokyo, Bunkyo, Tokyo 113–8658, Japan
[a)] natsu@csg.ci.i.u-tokyo.ac.jp

1. 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

© 2015 Information Processing Society of Japan

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．

依存関係を持った並列プログラムの実装にはバリア同期がよく用いられる．バリア同期とは，並列タスク間の待ち合わせを記述するためのシンプルな機構である．バリア同期の対象とされた並列タスクは，すべてのタスクの実行が決められた実行地点に到達するまで続きの処理が待機させされる．

しかし，バリア同期のようなタスク制御機構を用いて，依存関係の複雑な大規模エージェントシミュレーションを効率良く実行することは難しい場合がある．OpenMP のような典型的な All-to-All 型のバリア同期では，すべ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ともない，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，その占有メモリ量も増大する．一方，Phaser [1] のようなバリア同期を待機処理と到達処理に分けて記述できる Point-to-Point 型では，それらのプリミティブを利用してユーザが直接待機スレッドを抑えるような実装が可能であるため，同期を行うスレッドグループを明示的に制限できる．しかし，待ち合わせ条件を複雑にするほど，プログラム中に到達コードが散在してしまい，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機構 Waitless 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すべ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，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．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．4 章では提案した同期機構の実装方法について述べる．5 章で提案した同期機構の実行性能を確認するために行った実験の結果と考察を示す．6 章では関連研究について論じ，7 章で本論文をまとめる．

2. 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

2.1 バリア同期

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，イテレーションごと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．たとえば，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あげられる．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．したがって，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．

一般的に，並列プログラムの待ち合わせにはバリア同期と呼ばれる同期制御が用いられ，OpenMP [2] や MPI [3] といった並列処理や分散処理向け API や，X10 や Chapel のようなプログラミング言語では標準的に提供されている．バリア同期を用いると，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．既存のバリア同期には，到達と待機をコード上の任意の箇所に記述できる Point-to-Point 型と，到達と待機が同時に行われる All-to-All 型がある．**図 2** に，歩行者シミュレータ CrowdWalk（**図 1**）[*1] をエージェントごとにモジュール化し，Java 8 で導入された並列コレクションを用いて並列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）は単位時間ごとに，グラフ構造状の道の上をたどり，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．各単位時間において，まずエージェントは，現在位置

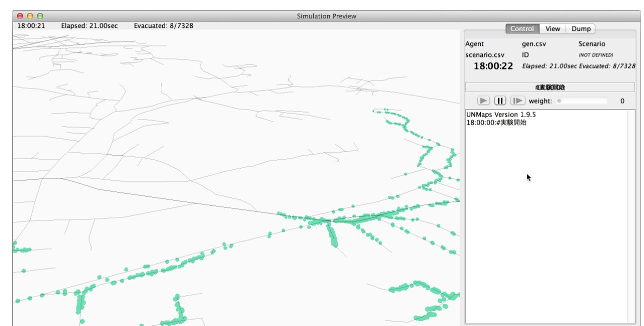

図 1　CrowdWalk
Fig. 1　CrowdWalk.

*1　現在，産業技術総合研究所で開発中．

```
class Agent {
  void update() {
    preUpdate();
    // moveCommit を適切な順番で逐次実行する必要性がある
    moveCommit();
    postUpdate();
  }

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

図 2 歩行者エージェントの処理を簡略化したプログラム

Fig. 2 Simplified program of pedestrian agent's procedure.

```
class Agent {
  // preUpdate の実行に対するバリア同期に用いる
  CountDownLatch latch1 = new CountDownLatch(1);
  // moveCommit の実行に対するバリア同期に用いる
  CountDownLatch latch2 = new CountDownLatch(1);

  void update() {
    preUpdate();
    latch1.countDown();
    if (!goOnSameStreet()) {
      neighbors().forEach(a -> a.latch1.await());
      conflictings().headSet(this)
        .forEach(a -> a.latch2.await());
    }
    moveCommit();
    latch2.countDown();
    postUpdate();
  }

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

図 3 CountDownLatch を用いた Point-to-Point 型のバリア同期例

Fig. 3 An example of Point-to-Point synchronization with CountDownLatch.

と速度から次に進もうとしている道を計算（preUpdate）する．その後，衝突判定と移動を行い（moveCommit），最後に後処理（postUpdate）を行う．preUpdate と postUpdate は各エージェントごとに独立して実行することが可能だが，moveCommit は衝突判定を行い，道の状態を排他的に更新するため，エージェントが道を移動する順番で移動処理を逐次的に行う必要がある．また，道を移動する順番は preUpdate の計算結果に依存するため，各エージェントは 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．しかし，エージェントスレッドの実行順序はスレッドスケジューラに依存するため，プログラマが明示的にスレッドの同期制御を行う必要がある．

Point-to-Point 型や All-to-All 型のバリア同期ライブラリを用いると，このような衝突判定のためのエージェントスレッド間の同期制御を明示的に実装することができる．Point-to-Point 型のバリア同期ライブラリの 1 つである Java の CountDownLatch を利用したコード例を図 **3** に示す．図 3 では，バリア到達として countDown メソッド，バリア待機として await メソッドを用い，他のスレッドと同期制御を行う．これにより，衝突する可能性のあるエージェントグループ neighbors() に対して同期が可能となる．また，10 行目のように，道の移動がなく衝突の起きないエージェントを判定し，バリア同期に参加させないような工夫も施せるようになる．さらに，12–13 行目のように，実際に衝突するエージェントグループ conflictings() に対して，自分よりも先に次の道に進入するエージェントが moveCommit を実行し終えるまで待ち，moveCommit を実行することにより，適切な順番で moveCommit を逐次実行することが可能となる．All-to-All 型のバリア同期ライブラリである Java の CyclicBarrier を用いても，同様にエージェントスレッド間の同期制御が可能である（図 **4**）．CyclicBarrier では，coundDown のようなバリア到達用のメソッドを使わずに await メソッドのみを用いて，同期に参加するすべてのスレッドの同期制御を簡潔に記述できる．All-to-All 型のバリア同期で，moveCommit を適切な順番で逐次実行するためには，9 行目や 10 行目のように，各エージェントが moveCommit を実行する前後で全体で待機する必要がある．CyclicBarrier では，バリア待機解除時に一度だけ実行されるコードを Runnable 型で渡せるため逐次実行も簡潔に実装できる．

```
class Agent {
  void update(CyclicBarrier barrier) {
    preUpdate();
    barrier.await();
    // moveCommit を適切な順番で逐次実行するため，
    // エージェントが moveCommit を実行する度に全体で同期を行う
    agents.sort().forEach(agent -> {
      if (agent == this) moveCommit();
      barrier.await(); });
    postUpdate();
  }

  public static void main() {
    CyclicBarrier barrier = new CyclicBarrier(agents.size());
    for (int i = 0; i < maxIteration; i++) {
      agents.parallelStream().forEach(a -> a.update(barrier));
    }
  }
}
```

図 4 CyclicBarrier を用いた All-to-All 型のバリア同期例

Fig. 4 An example of All-to-All synchronization with CyclicBarrier.

2.2 既存のバリア同期の問題点

しかし，既存のバリア同期機構を用いて並列プログラムの依存関係を実装すると，モジュラリティと実行性能を両立させることが難しい．Point-to-Point 型のバリア同期では，同期制御を待機と到達に分けて記述可能なため，最小限のスレッドに対してのみ同期を行うことができる．そのため，効率の良い並列化を行うことが可能だが，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

© 2015 Information Processing Society of Japan

```
1 class Agent {
2   void update(CyclicBarrier barrier) {
3     preUpdate();
4     await(barrier, 1, () -> {
5       agents.sort();
6       await(barrier, agents.headSet(this).size(), () -> {
7         moveCommit();
8         await(barrier, agents.tailSet(this).size(), () ->
9           postUpdate());
10      });
11    });
12  }
13
14  // 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
15  void await(CyclicBarrier barrier, int n, Runnable cont) {
16    if (n == 0) cont.run();
17    else {
18      new Thread(() -> {
19        // 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
20        barrier.await();
21        await(barrier, n - 1, cont);
22      }).start();
23    }
24  }
25 }
```

図 5　スレッド分割を行った並列化
Fig. 5 Parallelization with thread partitioning.

する必要がある．細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す．

一方，All-to-All 型のバリア同期を用いれば，到達と待機が同時に行われるため，バリア到達コードの散在しない，抽象度の高い記述が可能になる．しかし，同期時にすべてのスレッドが待機を行うため，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場合がある．待機状態のスレッド増加は，コンテキストスイッチなどによる CPU の実行効率の低下を招くだけでなく，占有するメモリ量の増大が大きな性能低下につながる．

さらに，バリア同期を利用した並列プログラムは，スレッドプールを利用できず，スケーラブルな並列化を実現できないという問題もある．素朴なバリア同期の実装では，同期時にスレッドプール内で待機状態となるスレッドにより，容易にデッドロックが引き起こされてしまう．このようなデッドロック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．

しかし，既存コードのスレッド分割は複雑な書き換えをともない，可読性や保守性を著しく低下させる．たとえば，**図 5** に図 4 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．バリア待機をともな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制限されているため，ローカル変数の保存・復元は手動で行う必要がある．

3.　Waitless：軽量な条件付きバリア同期

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関心事を分離し，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，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．

本章では Waitless がバリア同期プリミティブとして提供する barrier メソッドについて詳しく述べ，そのいくつかの利用例を説明する．

3.1　バリア同期プリミティブ

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 barrier メソッドを提供する（**表 1**）．Waitless を用いた並列プログラムは，WaitlessSet の保持するデータコレクションに対して，処理をラムダ式や無名内部クラスとして与えるようにして記述する．barrier メソッドは内包するコレクションに対する操作内で用いることができ，引数で与えた別のコレクション（Set 型）が条件式（Predicate 型）を満たすまで，実行中のスレッドが待機させられる．バリア対象を Java の Set 型として動的に指定できるため，実行時コンテキストに基づいて同期を行うスレッドグループを明示的に制限できる．また，バリア到達コードの挿入箇所を同期対象の要素の状態変化に対する条件式として記述できる．条件式の評価結果の変化を防ぐために，式内では副作用をともなう操作やリフレクションの利用を制限する．同様に，式を評価するタイミングはバリア対象オブジェクトのフィールド変更時に限るものとした．

図 6 は並列コレクション WaitlessSet と barrier メソッドを利用したバリア同期例である．まず，集合 all の各要素 item に対して，foo(item) を並列実行する．その後，集合 all 内の部分集合 targets の各要素 target に対して baz(target) が true を返すまで，現在実行中の処理を待機させる．targets の全要素が指定した条件式を満たした後，再度 bar(item) を並列実行する．

© 2015 Information Processing Society of Japan

表 1 Waitless の API
Table 1 Waitless API.

Class WaitlessSet⟨T⟩	
コンストラクタ	WaitlessSet(Set⟨T⟩ set) 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() 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) 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
Class Waitless⟨T⟩	
void	barrier(Set⟨T⟩ targets, Predicate⟨T⟩ predicate) 集合 targets のすべての要素が条件式 predicate を満たすまで待機する

```
1 new WaitlessSet(all).parallelStream().forEach((item, w) -> {
2   foo(item);
3   w.barrier(targets, target -> baz(target))
4   bar(item);
5 });
```

図 6 Waitless によるバリア同期の例
Fig. 6 An example of barrier synchronization with Waitless.

```
1 #pragma omp parallel for ordered
2 for (int i = 0; i < 10; i++) {
3   A(i);
4   // B(1), B(2), B(3)... の順番で呼ばれる
5   #pragma omp ordered
6   B(i);
7   C(i);
8 }
```

図 7 OpenMP の ordered ディレクティブ
Fig. 7 ordered directive of OpenMP.

```
1 // targets：指定した順番で逐次実行したい対象
2 // seq：逐次処理コード
3 // pred：seq を実行したかどうかを判定する条件式
4 void ordered(SortedSet<T> targets,
5              Predicate<T> pred, Runnable seq) {
6   SortedSet<T> head = targets.headSet(item);
7   // 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
8   // 条件式 pred を満たすまで待機する
9   if (!head.isEmpty()) barrier(head, pred);
10  seq.run();
11 }
```

図 8 barrier メソッドを用いた ordered の実装例
Fig. 8 An example of ordered implementation with barrier method.

3.2 barrier メソッドを用いた複雑なバリア同期

Waitless の barrier メソッドを用いると，バリア同期を拡張した様々な同期処理も記述することができる．たとえば，OpenMP では，並列実行されるブロック内の指定したコードを逐次実行させるための ordered ディレクティブが提供されている．orderd を用いた並列処理は**図 7** のように A(i) と C(i) が並列に実行されるが，B(i) は for ループを逐次実行した順番で呼ばれることが保証される．barrier メソッドを用いて，OpenMP の ordered を実装した例を**図 8** に示す．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すべ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．

```
1 class Agent {
2   int i; // イテレーション番号
3
4   void update(Waitless w) {
5     preUpdate();
6     if (goOnSameStreet()) {
7       // 道を移動しない場合は同期をせずに moveCommit を実行する
8       moveCommit();
9     } else {
10      w.barrier(neighbors(), a -> a.nextPlaces[this.i] != null);
11      ordered(conflictings(),
12        a -> a.places[this.i] != null, () -> moveCommit());
13    }
14    postUpdate();
15  }
16
17  public static void main() {
18    for (int i = 0; i < maxIteration; i++) {
19      agents.parallelStream()
20        .forEach((agent, w) -> agent.update(w));
21    }
22  }
23 }
```

図 9 barrier/ordered メソッドを利用した CrowdWalk の実装例
Fig. 9 An example of CrowdWalk implementation with barrier/ordered method.

Waitless の barrier および 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）は，関心事の分離を保ったまま**図 9** のように記述できる．Point-to-Point 型のコード例（図 3）と比較すると，countDown() のような到達コードを記述せず，バリア同期を近隣のエージェントや衝突するエージェントに限定した記述が可能となっている．具体的には，10 行目では，各エージェントは，近隣のエージェント neighbors() が，自分と同じイテレーション this.i の次に進む場所 nextPlaces[this.i] を計算し終えるまで待機を行っている．11 行目では，近隣のエージェント内で実際に衝突するエージェント conflictings() に対して moveCommit を逐次実行する．また，図 3 の 11–13 行目のような，他のエージェントグループに対して一括して行う待機コードを記述する必要もなくなる．

© 2015 Information Processing Society of Japan

```
1 class Agent {
2   int i;
3
4   void update(Waitless w) {
5     preUpdate();
6     if (goOnSameStreet()) {
7       moveCommit();
8     } else {
9       // 道を移動する場合には，他のエージェントが
10      // 自分と同じイテレーションに達するまで待機する
11      w.barrier(agents, a -> a.i >= this.i);
12      w.barrier(neighbors(), a -> a.nextPlaces[this.i] != null);
13      ordered(conflictings(),
14        a -> a.places[this.i] != null, () -> moveCommit());
15    }
16    postUpdate();
17  }
18
19  public static void main() {
20    agents.parallelStream()
21      // イテレーション順に update を実行する処理全体に対して
22      // 各エージェントで並列実行する
23      .forEach((agent, w) -> {
24        for (int i = 0; i < maxIteration; i++) {
25          agent.update(w); }
26      });
27  }
28 }
```

図 10　Waitless による CrowdWalk の時間ブロッキング実装例
Fig. 10　An example of CrowdWalk's time-blocking implementation with Waitless.

Waitless を用いると，さらに時間ブロッキングという手法を導入して並列化を行うことも可能である（**図 10**）．時間ブロッキングとは，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，時間方向のイテレーションを複数まとめて実行することで，周囲の格子点の処理との同期待ちを減らす手法である．時間ブロッキングを用いないコードでは，各イテレーションごとにすべてのエージェントに対して update メソッドを並列実行する（図 9 の 19–20 行目）ため，イテレーション間には暗黙的な全体での同期が存在する．時間ブロッキングの手法を適用することで，イテレーションをまたがって update メソッドが並列に実行される（図 10 の 23–24 行目）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（6 行目），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　Point-to-Point 型への変換

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．条件式内で参照しているフィールド変数を特定した

```
1 class Agent {
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    agents.parallelStream().forEach((agent, 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      agent.update(w);
21    })
22  }
23 }
```

図 11　条件式の解析と到達コードの挿入箇所の特定
Fig. 11　Analyzing conditional expression and looking up where to insert barrier-arrival codes.

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．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，条件式内で参照される変数やその変数が更新される箇所は，ロードタイムにコールフローをバイトコード解析することで特定できる．たとえば，**図 11** の 12 行目のように，Agent クラスのフィールド nextPlaces を参照する条件式では，並列コレクションに対する操作である Agent クラスの update メソッドから探索を始め，preUpdate メソッド内での nextPlaces への代入後（4 行目）に，バリア到達条件が満たされる可能性のある箇所があると分かる．バリア到達コードの挿入はロードタイムに行うが，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，条件式の真偽値の更新がネイティブメソッドの返り値に依存する場合，ロードタイムのバイトコード変換を施せないため，条件式が Java の文法的に正しくてもプログラマの意図どおりに動かない場合がある．

3.4　スレッド分割

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，

© 2015 Information Processing Society of Japan

表 2 バイトコード変換で挿入される Waitless の API
Table 2 Waitless API inserted automatically by bytecode transformation.

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

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，すべ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たどりながら，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．

4. Waitless によるコード変換

4.1 Point-to-Point 型への変換

バリア到達コードの自動挿入による Point-to-Point 型への変換を実現するため，Waitless はバイトコード操作ライブラリの ASM を用いて，ロードタイムにバイトコード変換を行う．まず Waitless を用いたプログラムでは，並列コレクション WaitlessSet の利用されている箇所が検索され，barrier メソッドの条件式が解析される．続いて，並列コレクションに対する操作からたど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が検索される．検索するバイトコードは対象フィールドに対する putfield 命令や，フィールド変数が配列の場合，aastore 命令である．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 2**）呼び出しが挿入される．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 11 のコードに対して，checkArrived を挿入すると**図 12** のようになる．ただし，現在の実装では，条件式として与えられる Predicate 型の変数の test メソッド直下の解析しか行わないため，条件式内のプログラムにはメソッドを用いることができず，変数と演算子のみ使用することができる．

さらに，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追加する．**図 13** 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もとに，barrier メソッド

```
1 // 変換前
2 void preUpdate() {
3   nextPlaces[this.i] = calcNextPlace();
4 }
5 // 変換後
6 void preUpdate() {
7   nextPlaces[this.i] = calcNextPlace();
8   Waitless.current.get().checkArrived();
9 }
```

図 12 到達コード挿入
Fig. 12 Insertion of barrier-arrival code.

```
1 // 変換前
2 new WaitlessSet(agents).parallelStream().forEach((a, w) -> {
3   w.barrier(neighbors(), _a -> _a.nextPlaces[a.i] != null);
4   // または
5   w.barrier(neighbors(), new Predicate<Agent> {
6     public boolean test(Agent _a) {
7       return _a.nextPlaces[a.i] != null; }
8   });
9 });
10
11 // 変換後
12 new WaitlessSet(agents).parallelStream().forEach((a, w) -> {
13   w.barrier(neighbors(), new CheckNextPlace(a.i));
14 });
15
16 class CheckNextPlace implements Predicate<Agent> {
17   private Integer i;
18
19   CheckNextPlace(Integer i) { this.i = i; }
20
21   public boolean test(Agent _a) {
22     return _a.nextPlaces[i] != null; }
23
24   public boolean equals(Object o) {
25     if (o instanceof CheckNextPlace) {
26       return ((CheckNextPlace) o).i.equals(i);
27     } else {
28       return false;
29     }
30   }
31
32   public int hashCode() {
33     return CheckNextPlace.class.hashCode() ^ i.hashCode();
34   }
35 }
```

図 13 バリア同期の条件式に対する equals と hashCode メソッドの挿入
Fig. 13 Insertion equals and hashCode methods to the class expressing conditional expression of barrier synchronization.

© 2015 Information Processing Society of Japan

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．

4.2　スレッド分割の実装

Waitless は Quasar [4] というライブラリを用いてスレッド分割を行う．Quasar は Java のための軽量スレッド Fiber を提供し，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，3.4 節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．barrier メソッドによりバリア待機が行われると，Waitless は 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停止する．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実行をスキップし，ローカル変数を復元し，処理の続きを行う．

5.　性能評価

Waitless によるバリア同期の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 10 m で，収容能力が 5 人の道からなる格子状のマップを与え，300 m 間隔でゴールを設置し，移動速度が 1–10 m/s のエージェントを一様に展開し，初期位置に最も近いゴールを目指して進ませた．イテレーションを 10 回繰り返す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 @ 3.10 GHz（論理コア数：16）
- Memory：64 GB
- 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

実験では，まず逐次実行を行った場合と，All-to-All 型の CyclicBarrier で全体で同期する場合，Point-to-Point 型の Phaser・Waitless で同期数を制限した場合を性能評価した．CyclicBarrier を利用したコードは図 4 を用い，Phaser と Waitless は同じ Point-to-Point 型の図 3 をもとにしたコードである．**図 14**，**図 15**，**図 16** は，それぞれエージェントの数を 200 から 20,000 まで変化させたときの最大スレッド数，メモリ使用量，実行時間である．図 14 か

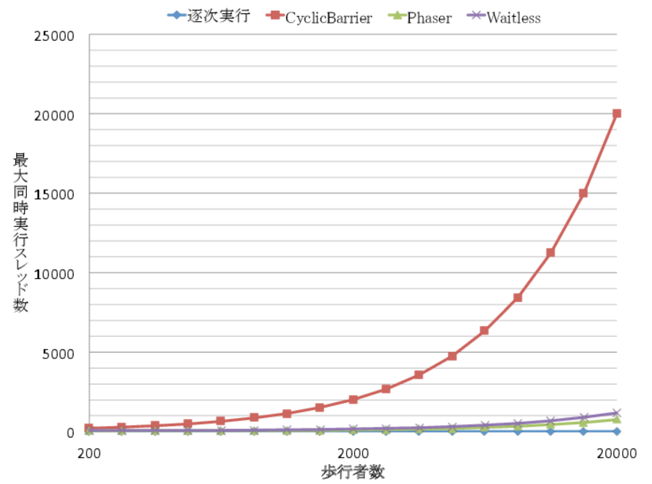

図 14　グループバリア同期した場合の最大同時実行スレッド数
Fig. 14　The maximum number of concurrent threads when using group barrier synchronization.

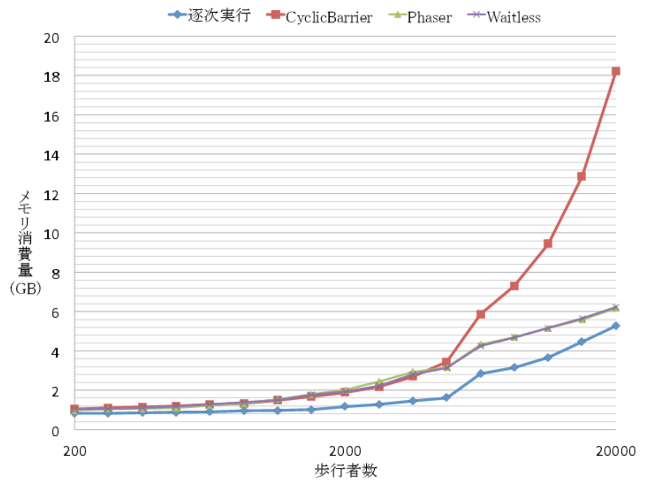

図 15　グループバリア同期した場合のメモリ使用量
Fig. 15　The memory usage when using group barrier synchronization.

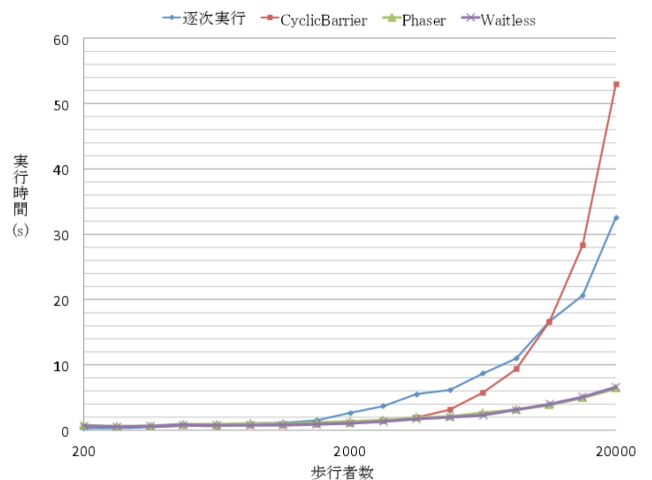

図 16　グループバリア同期した場合の実行時間
Fig. 16　The execution time when using group barrier synchronization.

© 2015 Information Processing Society of Japan

```java
class Agent {
  public static void main() {
    for (int i = 0; i < maxIteration; i++) {
      agents.parallelStream().forEach(a -> a.preUpdate());
      agents.forEach(a -> a.moveUpdate());
      agents.parallelStream().forEach(a -> a.postUpdate());
    }
  }
}
```

図 17　update メソッドを手動分割し，Java 標準の並列ストリームを用いた並列化

Fig. 17　Partitioning update method manually and parallelizing with Java standard parallel stream.

ら，All-to-All 型のバリア同期を用いて並列化したとき，スレッド数が最大で歩行者とほぼ同じ数だけ同時に走っていることが分かる．それにともない，図 16 から，歩行者の数が小規模の場合には，どのバリア同期を用いても，並列実行によって逐次実行よりも実行時間を短縮できたが，歩行者数が 11,246 を超えたところから，All-to-All 型のバリア同期を用いた並列化の実行時間は急激に悪化した．これに対して，Waitless を用いて同期するエージェントを制限すると，エージェント数 20,000 のとき，最大同時実行スレッド数を 20,002 から 1,170 に削減でき，それにより，メモリ使用量も 34%に削減され，さらに実行時間を 34%に短縮することができた．また，Waitless を用いた場合でも Phaser と同等の性能で並列化できることが確認できた．

次に，スレッド分割によるスレッド数削減の効果を測定するため，**図 17** のようにエージェント内の処理コードを別々に分割して，preUpdate メソッドと postUpdate メソッドを Java 標準の並列ストリームを用いてそれぞれ並列化した場合と，図 9 のように Waitless を用いてバリア同期対象の制限を利用したうえで，バイトコード変換によるスレッド分割の有無を変えた場合で実験を行った．スレッド分割なしの Waitless は，比較実験のために用意したものであり，バリア同期は Phaser を利用した．**図 18**，**図 19**，**図 20** は大規模歩行者シミュレーションに対する並列化の性能評価を行うため，歩行者の数を 2,000 から 200,000 まで変化させた場合の最大同時実行スレッド数，メモリ使用量，実行時間である．図 14 から，スレッド分割を行い，スレッドプールを用いることで，歩行者数を 200,000 まで大規模化でき，最大同時実行スレッド数を 11,456 から 29 に削減することができた．歩行者数が大規模になっているため，スレッド数増加によるメモリ圧迫のオーバヘッドは相対的に小さくなっているものの，実行時間は 46%に短縮され，シミュレーションの大規模化にあたっても性能を低下させず並列化を行えたことが分かる．また，Waitless を用いることで，実際に衝突するエージェントのグループ間で並列に moveCommit を逐次実行できるようになるため，moveCommit が全エージェントで逐次実行される図 17 に対して実行時間を 88%に削減することができた．

次に，スケーラビリティの評価を行うため，プロセスが

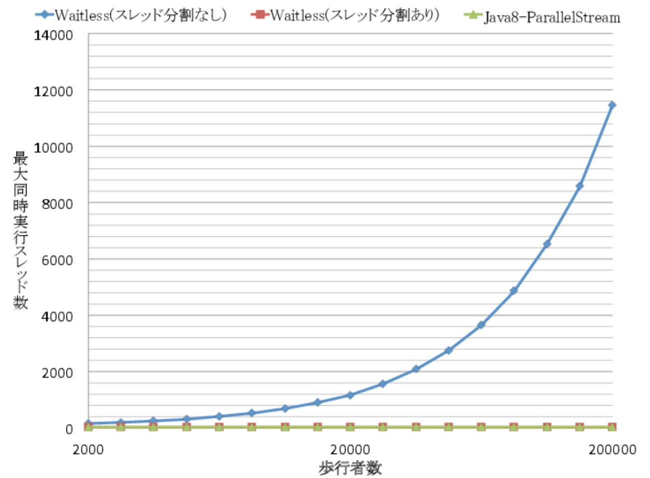

図 18　スレッド分割した場合の最大スレッド数

Fig. 18　The maximum number of concurrent threads when partioning thread codes.

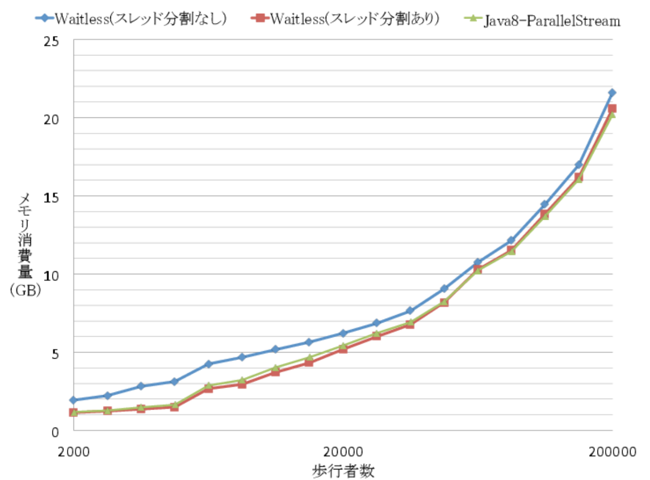

図 19　スレッド分割した場合のメモリ使用量

Fig. 19　The memory usage when partioning thread codes.

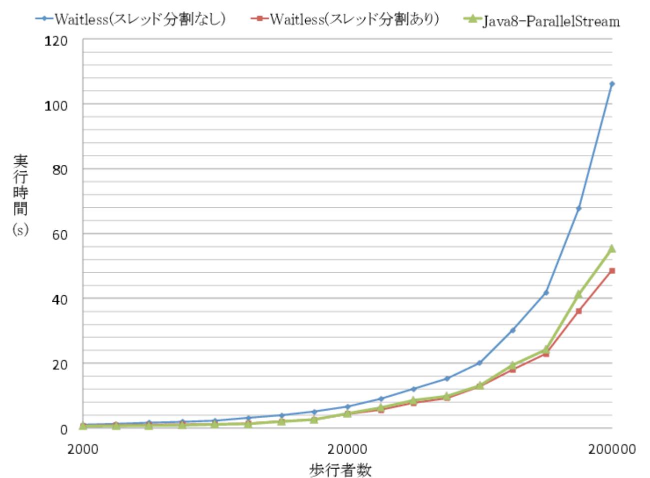

図 20　スレッド分割した場合の実行時間

Fig. 20　The execution time when partioning thread codes.

使用する CPU のコア数を taskset コマンドによって変化させて実験を行った．**表 3** は歩行者の数を 10,000 としたと

© 2015 Information Processing Society of Japan

表 3　コア数を変化させたときの実行時間（秒）
Table 3　The elapsed time of the simulation while changing the number of cores (s).

コア数	CyclicBarrier	Java8-ParallelStream	Waitless（スレッド分割あり）
1	1,887	108.5	119.3
2	401.0	59.66	60.32
4	414.3	31.87	31.28
8	411.2	18.67	16.46
16	324.3	13.6	10.00

表 4　バリア同期のみを実行するシミュレーションの実行時間
Table 4　The elapsed time of the simulation only performing barrier sychronization(s) changing the number of cores (s).

コア数	CyclicBarrier	Java8-ParallelStream	Waitless（スレッド分割あり）
1	3,069.5	0.2189	7.278
2	214.3	0.2301	4.072
4	233.9	0.1045	2.087
8	231.6	0.1023	1.158
16	262.4	0.1121	0.8601

```
1 // CyclicBarrier
2 void update(CyclicBarrier barrier) {
3   i++;
4   barrier.await();
5 }
6
7 // Java8-ParallelStream
8 void update() {
9   i++;
10 }
11
12 // Waitless
13 void update(Waitless w) {
14   i++;
15   w.barrier(agents, a -> a.i >= this.i);
16 }
```

図 21　バリア同期自体のコストを測定するためのコード
Fig. 21　The update methods to measure each barrier synchronization cost.

きに，Java プロセスが使用する CPU コア数を 1，2，3，4，8，16 と変化させた場合の実行時間である．CyclicBarrier を用いた場合，スレッド生成のオーバヘッドにより，コア数を増加させた場合でも十分な台数効果を得られなかった．これに対して，Waitless を用いた場合，コア数を 16 個まで増加させることで，コア数が 1 個の場合に比べて実行時間を 8.4%に短縮することができた．

次に，バリア同期自体の純粋なコストを測定するため，各イテレーションの処理がバリア同期のみになるように update メソッドから preUpdate と moveCommit，postUpdate の呼び出しを削除し，update メソッド内では，イテレーション番号のインクリメントと同期を行う処理のみのコード（**図 21**）で実験を行った．**表 4** は，歩行者の数を 10,000 としたときに，Java プロセスが使用する CPU コア数を 1，2，3，4，8，16 と変化させた場合の実行時間である．手動でスレッド分割を施した Java8-ParalllelStream と比べて，Waitless を用いた場合ではバリア同期処理を行うための計算（図 21 の 16 行目）を必要とするため，実行時間は増加する．しかし，衝突するエージェントグループ間での並列実行の効果で，コア数を 16 まで増加させればバリア同期のコストがスケールし，コア数 1 の場合に比べて実行時間を 11.8%に短縮することができた．

6.　関連研究

バリア同期を実現するためアプローチは数多く研究されてきており，様々な並列分散用のライブラリやプログラミング言語で実現されている．共有メモリ型並列計算の標準規格の 1 つである OpenMP [2] は barrier や ordered といった同期構文をサポートしており，コメントやディレクティブを用いて宣言的に記述することで，短いコード量でバリア同期を行う並列タスクを記述できる．しかし，OpenMP の barrier は同時に実行されているすべてのスレッドに対して同期をとるため，同期スレッドを任意のグループに抑えることが難しい．分散メモリ型並列計算の標準規格の 1 つである MPI [3] でも，同様に MPI_barrier というバリア同期機構を備えており，コミュニケータを介して指定したプロセスグループに対する同期を行うことが可能である．言語レベルでバリア同期をサポートしている X10 [5] も Clocks を用いてバリア同期が可能であり，X10 の並列タスクである Activity の処理の中で同じ Clock オブジェクトが指定されている他の Activity と同期することが可能である．これらのバリア同期は同期対象のスレッド集合を動的に指定できるため，待機スレッドを削減することが可能だが，同期に関わる並列タスクはすべて相互に同期を行うため，非対称な同期が不可能である．Clocks や 2 章で紹介した 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 [1] という機能が提案されており，Java

© 2015 Information Processing Society of Japan

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 を用いると Clocks や CyclicBarrier では不可能であった非対称な同期が可能なうえ，実行中の並列タスクから同期グループへの動的な登録や解除が可能である．しかし，Phaser では並列タスクを実行する前に同期グループが作成されていることを想定しており，本研究のように並列タスク内から動的に同期対象のタスクグループを作成することは困難である．

HJ [6] や X10，Chapel といった言語では，fork-join 操作が用いられている場合，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，効率の良い並列化が可能となる．しかし，future [7] や barrier，Phaser のように同期時にスレッドがブロックされる同期機構は，スレッドプールを用いることができないため，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コストが発生するため，効率良く並列化することが難しい．このような同期機構はスレッド分割を行うことで，待機スレッドの削減を行い，効率良く並列実行を行うことが可能である [8]．RIFE [9] や Javaflow [10]，Kilim [11] のように，Java プログラムの継続をサポートするフレームワークも数多く実装されている．

7. まとめ

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 を提案した．Waitless は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，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．開発したバリア同期ライブラリ Waitless を歩行者シミュレーションに対して適用し，シミュレーションサイズが増加しても，実行性能を低下させることなく，並列化できたことを確認した．

参考文献

[1] Shirako, J., Peixotto, D.M., Sarkar, V. and Scherer, W.N.: Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization, *Proc. 22nd Annual International Conference on Supercomputing, ICS '08*, pp.277–288, ACM (online), DOI: 10.1145/1375527.1375568 (2008).

[2] OpenMP Application Program Interface Version 3.0 (2008), available from ⟨http://www.openmp.org/mp-documents/spec30.pdf⟩.

[3] MPI A Message-Passing Interface Standard 3.0 (2012), available from ⟨http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf⟩.

[4] Parallel Universe Quasar, available from ⟨http://docs.paralleluniverse.co/quasar/⟩.

[5] X10 Language Specification version 2.4 (2012), available from ⟨http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf⟩.

[6] Cavé, V., Zhao, J., Shirako, J. and Sarkar, V.: Habanero-Java: The New Adventures of Old X10, *Proc. 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11*, pp.51–61, ACM (online), DOI: 10.1145/2093157.2093165 (2011).

[7] Halstead, Jr., R.H.: MULTILISP: A Language for Concurrent Symbolic Computation, *ACM Trans. Program. Lang. Syst.*, Vol.7, No.4, pp.501–538 (online), DOI: 10.1145/4472.4478 (1985).

[8] Imam, S. and Sarkar, V.: Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns, *ECOOP 2014–Object-Oriented Programming*, pp.618–643, Springer (2014).

[9] RIFE Java web application framework, available from ⟨http://rifers.org/⟩.

[10] Apache Commons: Javaflow, available from ⟨http://commons.apache.org/sandbox/javaflow/⟩.

[11] Srinivasan, S. and Mycroft, A.: Kilim: Isolation-Typed Actors for Java, *Proc. 22nd European Conference on Object-Oriented Programming, ECOOP '08*, pp.104–128 (2008).

夏 澄彦

1991 年生．2013 年横浜国立大学工学部電子情報工学科卒業．2015 年東京大学大学院情報理工学系研究科創造情報学専攻修了．プログラミング言語の研究に従事．

佐藤 芳樹　（正会員）

1977 年生．2000 年東北大学工学部情報工学科卒業．2002 年同大学大学院情報科学研究科情報基礎科学専攻修士課程修了．2005 年東京工業大学大学院情報理工学研究科数理・計算科学専攻博士（理学）取得．（株）三菱総合研究所，（株）日立製作所，日本オラクル株式会社を経て，現在東京大学情報基盤センター特任講師．言語処理系，HPC の研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年同大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師，東京工業大学大学院情報理工学研究科教授を経て，2012 年より東京大学大学院情報理工学系研究科教授．博士（理学）．プログラミング言語，システムソフトウェアの研究に従事．

© 2015 Information Processing Society of Japan