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

夏 澄彦　佐藤 芳樹　千葉 滋

依存関係をもった並列タスクを記述するにはバリア同期がよく用いられる．バリア同期とは，並列に動作するプログラムの実行を待ち合わせるために用いられ，プログラミング言語や並列分散ライブラリによって提供される．しかし，エージェントシミュレーションのような不均一な並列タスクを持つアプリケーションでは，タスク数の増加に伴い待機状態のスレッドが増加し，CPU の実行効率が低下するばかりでなく，待機スレッドの占有するメモリ量も増大する．そのため，スケーラブルな並列プログラムを実装するには，バリア同期による待ち時間を最小化するよう細心の注意が必要となる．我々はバリア対象のスレッドを実行時に指定し，さらにスレッドプログラムを分割することにより，軽量なバリア同期機構を開発した．本論文では，開発した動的グループバリア同期機構と，歩行者シミュレーションのアプリケーションに適用した結果について報告する．

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ようなソフトウェアは，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．一方，一般にエージェントを実装したプログラムは，計算粒度が大きく，他のエージェントや外部システムとの協調が必要となり，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．

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．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．素朴なバリア同期の実装では，全てのタスクが同期を待ち合わせるため，並列タスク数の増加に伴い待機スレッド数も増加し，その結果，スレッドによって占有されるメモリ量が増大する．また，タスク間の実行時間が不均一なアプリケーションの場合，全てのスレッドで同期を行うと，CPU の実行効率が低下し，プログラムのパフォーマンスも悪化してしまう．

我々は，エージェントプログラムのモジュラリティを保ったまま，バリア同期対象のタスクを動的に指定できる機構を開発した．プログラマによってバリア対象のタスクを実行時に指定することで，同期に参加するスレッドを最小限に抑え，待機スレッドの増加を抑制することが可能となる．さらに，待機スレッド数を削減するために，エージェントを実装するスレッドコードを分割し，ノンブロッキングな形式の同期を行うことが可能なプリミティブも提供する．

以下，1 章では不均一な並列タスクに対してバリア同期を用いた場合の問題点を述べる．2 章で開発した実行時のグループバリア同期機構について説明し，3 章では更にスレッド分割を行った場合の同期機構について説明する．4 章では，開発したバリア同期機構を歩行者シミュレーションに対して適用した実験を報告

Sumihiko Natsu,　Shigeru Chiba, 東京大学大学院情報理工学系研究科創造情報学専攻, Dept. of Creative Informatics Graduate School of Information Science and Technology,　The University of Tokyo.
Yoshiki Sato, 東京大学情報基盤センター, Information Technology Center,　The University of Tokyo.

図 1　OpenMP の barrier 指示文

```
#pragma omp parallel
{
  A();
  #pragma omp barrier
  B();
}
```

する．5 章で関連研究について論じ，6 章で本論文をまとめて，今後の課題を述べる．

2　不均一な並列タスクのバリア同期

計算に依存関係のある並列タスクを記述するにはバリア同期が有用である．バリア同期とは，並列に動作するプログラムの実行を待ち合わせるために用いられ，通常は，MPI や OpenMP といった並列分散向け機能を提供する専用ライブラリによって提供されており，X10 や Chapel のように専用の言語プリミティブを持ったプログラミング言語も数多く登場してきている．OpenMP では，並列化したタスク内で互いに待ち合わせする位置に明示的に同期のための指示文を記述する (図 1)．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，全てのスレッドがこの 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．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．

互いに協調しながら動作するようなエージェントシミュレーションも，バリア同期を必要とする並列タスクの一例に挙げられる．図 3 に，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．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いるため，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，まずエージェントの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，道を移動する順に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．

依存関係を考慮しなければ，これらの処理を図 4 のようにエージェント毎にまとめてスレッド化できる．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，意図した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

図 2　CrowdWalk

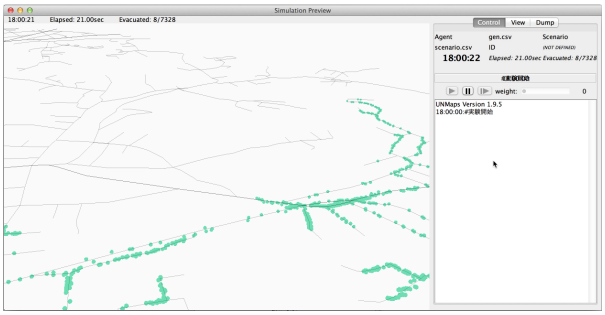


図 3　歩行者エージェント全体の処理を簡略化したプログラム

```
class Pedestrian implements Comparable<Pedestrian> {
  static void main(String args[]) {
    Collection<Pedestrian> all = ...;
    for (int i = 0; i < iteration; i++) {
      for (Pedestrian pedestrian : all) {
        pedestrian.preUpdate();
      }
      all.sort();
      for (Pedestrian pedestrian : all) {
        pedestrian.moveCommit();
      }
      for (Pedestrian pedestrian : all) {
        pedestrian.postUpdate();
      }
    }
  }
}
```

†1　現在，産業技術総合研究所で開発中

図 4　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 (Pedestrian pedestrian : pedestrians) {
      new Thread(-> pedestrian.update()).start();
    }
  }
}
```

図 5　バリア同期を用いた並列化

```
class Pedestrian {
  static CyclicBarrier barrier;

  void update() {
    preUpdate();
    barrier.await();
    // 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
    //（sort や全歩行者への moveCommit が実行される）
    postUpdate();
  }

  public static void main(String args[]) {
    barrier = new CyclicBarrier(all.size(), -> {
      all.sort();
      for (Pedestrian pedestrian : all) {
        pedestrian.moveCommit();
      }
    });
    for (Pedestrian pedestrian : pedestrians) {
      new Thread(-> pedestrian.update()).start();
    }
  }
}
```

ない．具体的には，moveCommit が呼ばれる順番が，preUpdate の計算結果に依存するため，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．

このような並列タスク間の依存関係は，バリア同期を用いることで簡潔に記述できる．例えば，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 4 は図 5 のように記述できる．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，各タスクの処理内容が不均一である場合，大規模シミュレーションにおいて，既存のバリア同期ではスケールしなくなるという問題が顕著に出てくる．関係する全スレッドを同期させるような素朴なバリア同期の実装では，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．待機状態のタスクの増加は，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きてしまう．

3　実行時のグループバリア同期機構

そこで本研究では，実行中のコンテキストに応じてバリア同期対象のタスクを指定可能なグループバリア動機機構を API として開発した．既存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止められる．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．

例えば，歩行者シミュレーション（図 5）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．さらに，道を移動しないエージェントはそもそも衝突しないため，他のエージェントと同期する必要がない．

本研究では，プログラマが柔軟にグループバリア同期を利用できるように，同期待ち (barrier メソッド) と完了通知 (finished メソッド) のプリミティブを API として提供する（表 1）．barrier メソッドには，バリア同期の対象となるスレッドの集合を引数として与えられる．加えて，各スレッド内では，finished メソッドを呼び出すことで，スレッド実行の完了を通知し，バリア同期を明示的に解除できる．

さらに，指定した複数の並列タスクをシーケンシャル実行するためのプリミティブ (ordered メソッド) を提供する．ordered メソッドは内部的には barrier メソッドを使って実装されており，同期待ち対象のスレッド集合に対して，同期解除されたタイミングで行う処理と，その実行順序が引数として与えられる．

これらのグループバリア同期機構を用いると図 5 のプログラムは図 6 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．

図 6　グループバリアを用いた並列化

```
1  class Pedestrian {
2    void update(Scheduler<Pedestrian> s) {
3      preUpdate();
4      s.finished(1);
5      if (goOnSameStreet()) {
6       moveCommit();
7      } else {
8        s.barrier(1, calcNeighborPedestrians());
9        s.ordered(2, calcConflictingPedestrians(), -> {
10         moveCommit();
11       });
12     }
13     postUpdate();
14   }
15
16   static void main() {
17     new ParallelSet<Pedestrian>(all)
18       .map((Pedestrian pedestrian,
19             Scheduler<Pedestrian> s) -> {
20         pedestrian.update(scheduler);
21         return pedestrian;
22       });
23   }
24 }
```

4　スレッド分割によるバリア同期の軽量化

開発したグループバリア同期機構は，同期待ちによる待機スレッド数をさらに削減するためにスレッドプログラムの分割を行う．通常，並列タスク間の依存関係を取り除けない場合，待機スレッドが占有するリソース消費の影響は避けられない．そのような依存したタスクが増加すれば，並列処理による実行効率のスケーラビリティは向上できなくなってしまう．そこで，本機構では，スレッドプログラムを複数のノンブロッキングなプログラムに分割し，そのシーケンシャルな実行に置き換える．ノンブロッキング化によって，スレッドプログラム中の任意の処理は独立したスレッドとして扱えるようになる．それによって，同期待ちスレッドの前処理と後処理は別々のスレッドに分割されるため，依存関係を持ったタスクの前処理を待つために，スレッドを中断する必要がなくなる．

スレッド分割に対応した同期待ち (ibarrier メソッド) とシーケンシャル実行 (iordered メソッド）のための API を表 2 に示す．プログラマは，barrier や ordered メソッドの呼び出し後に記述していた後処理を，Runnable クラスのメソッドに切り出し，ibarrier, iordered メソッドに渡すことでスレッドを分割できるようになる．これにより，並列タスクは同期待ちをする必要が無くなり，同期後の処理も単独でスレッド化されキューに作成される．したがって，待機スレッドによる不要な CPU やメモリリソースの消費は避けられる．歩行者シミュレーション（図 5）に適用させると図 7 のようになる．barrier や ordered をノンブロッキングな API である ibarrier や iordered に置き換えるため，それらのメソッドの後に実行される処理を Runnable でラップし，ibarrier や iordered に渡している．現時点では，ノンブロッキングのグループバリア同期 API は，同期前後でのローカル変数の保存をサポートしていないため，プログラマが手動で保存と復

表 1　グループバリア同期 API

Scheduler⟨T⟩ API	説明
void barrier(int tag, Set⟨T⟩ set)	**set** の全ての要素が **tag** に到達するまでスレッドを停止させる
void finished(int tag)	自身が **tag** に到達したことを通知し，依存する待機スレッドを再開させる
void ordered(int tag, Set⟨T⟩ set, Runnable seq)	**set** に対して自然順序付けで **seq** をシーケンシャル実行する
void ordered(int tag, Set⟨T⟩ set, Comparator⟨T⟩ comp, Runnable seq)	**set** に対して **comp** 順で **seq** をシーケンシャル実行する

表 2　ノンブロッキングのグループバリア同期 API

Scheduler⟨T⟩ API	説明
void ibarrier(int tag, Set⟨T⟩ set, Runnable cont)	set の全ての要素が tag に到達した後に cont を実行する
void iordered(int tag, Set⟨T⟩ set, Runnable seq, Runnable cont)	set に対して自然順序付けで seq をシーケンシャル実行すし，その後 cont を並列実行する
void iordered(int tag, Set⟨T⟩ set, Comparator⟨T⟩ c, Runnable seq, Runnable cont)	set に対して comparator 順で seq をシーケンシャル実行し，その後 cont を並列実行する

図 7　ノンブロッキングなグループバリアを用いた並列化

```
1  class Pedestrian {
2    void update(Scheduler<Pedestrian> s) {
3      preUpdate();
4      s.finished(1);
5      if (goOnSameStreet()) {
6        moveCommit();
7        postUpdate();
8      } else {
9        s.ibarrier(1, calcNeighborPedestrians(), -> {
10         s.iordered(
11           2, calcConflictingPedestrians(), -> {
12             moveCommit();
13           }, -> {
14             postUpdate();
15           });
16       });
17     }
18   }
19 }
```

元を行う必要がある．

歩行者シミュレーション（図 6）をスレッド分割し、ノンブロッキングなバリア同期を用いる（図 8）と図 7 のようになる．図 8 では，まず ordered の実行後に行われる postUpdate() を Runnable でラップし、iordered の引数として与えることで，ordered をノンブロッキングな iordered に変換している。続いて、barrier の実行後に実行される iordered(...) を Runnable でラップし、ibarrier に引数として与えることで barrier をノンブロッキングな ibarrier に変換する。この一連の操作によりプログラム全体をノンブロッキング化することが可能となる。

5　予備実験

開発したグループバリア同期による実行性能の向上を確認するために予備的な実験を行った．実験では，最低限の歩行者シミューレションの機能を実装したアプリケーションに対して，歩行者数を変化させ，スレッド数，メモリ使用量，CPU 使用率，実行時間の変化を計測した．入力としては，長さが 10m で，収容容量が 2 人の道からなる 500x500 の格子状のマップ与え，移動速度 1m/s の歩行者をランダムに配置した．ゴール数は 25 で，歩行者は最も近いゴールを目指して進み，20 回イテレーション繰り返した結果を測定した．実験環境は Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz，メモリ 64GB である．Java VM は OpenJDK Runtime Environment (rhel-2.3.10.4.el6_4-x86_64) OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode) を使い、オプションは -Xms8g -Xmx8g -Xss512m -XX:+UseG1GC で実行し，シミュレーション実行時の最大スレッド数，平均メモリ使用量，平均 CPU 使用率，実行時間について測定を行った．

図 9 は，歩行者の数を 1,000 から 10,000 まで変化させた時の全エージェントに対してバリア同期を行った場合と近傍のエージェントに対してグループバリア同期を行った場合の結果である．全エージェントを同期させる場合，歩行者の増加とともに最大スレッド数は単調増加し，それに伴ってメモリ使用量，CPU 使

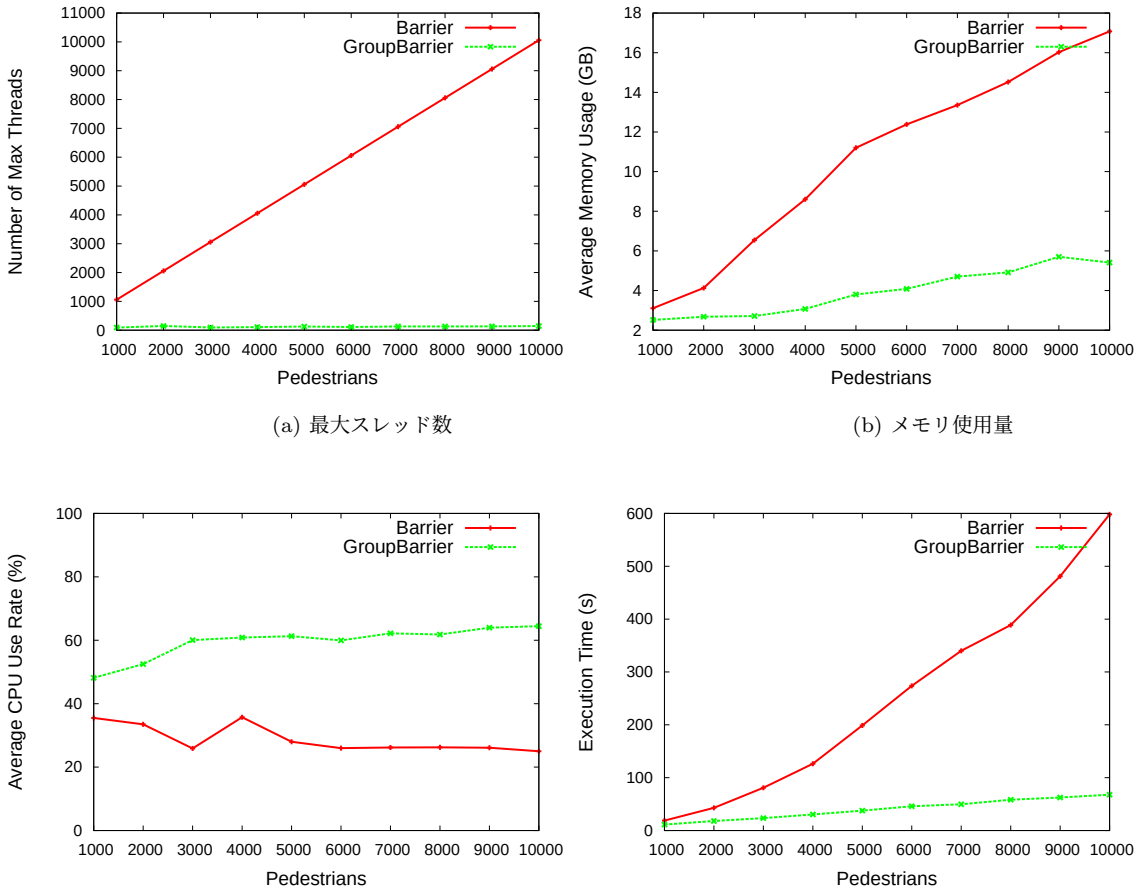
図 8　非同期なグループバリアを用いるためのスレッド分割

用率，実行時間のすべてが急激に増加していることがわかる．一方，グループバリア同期では，歩行者を増加させても最大スレッド数は 100 前後に抑えられるため，急激な性能低下は見られなかった．

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の実験を行った．図 10 はグループバリア同期を利用した上で，スレッド分割の有無を変えて比較した結果である．分割により，同時実行スレッド数が大幅に増やせるようになったため，歩行者の数を 10,000 から 160,000 まで変化させた．まず，スレッド分割を施すことで，通常のグループバリア同期に比べても，さらに最大スレッド数が抑えられていることがわかる．それに伴い，メモリ使用量，CPU 使用率，実行時間もシミュレーションの大規模化にあたっても十分に抑えられていることがわかる．

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一つである OpenMP [4] 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全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一つである 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



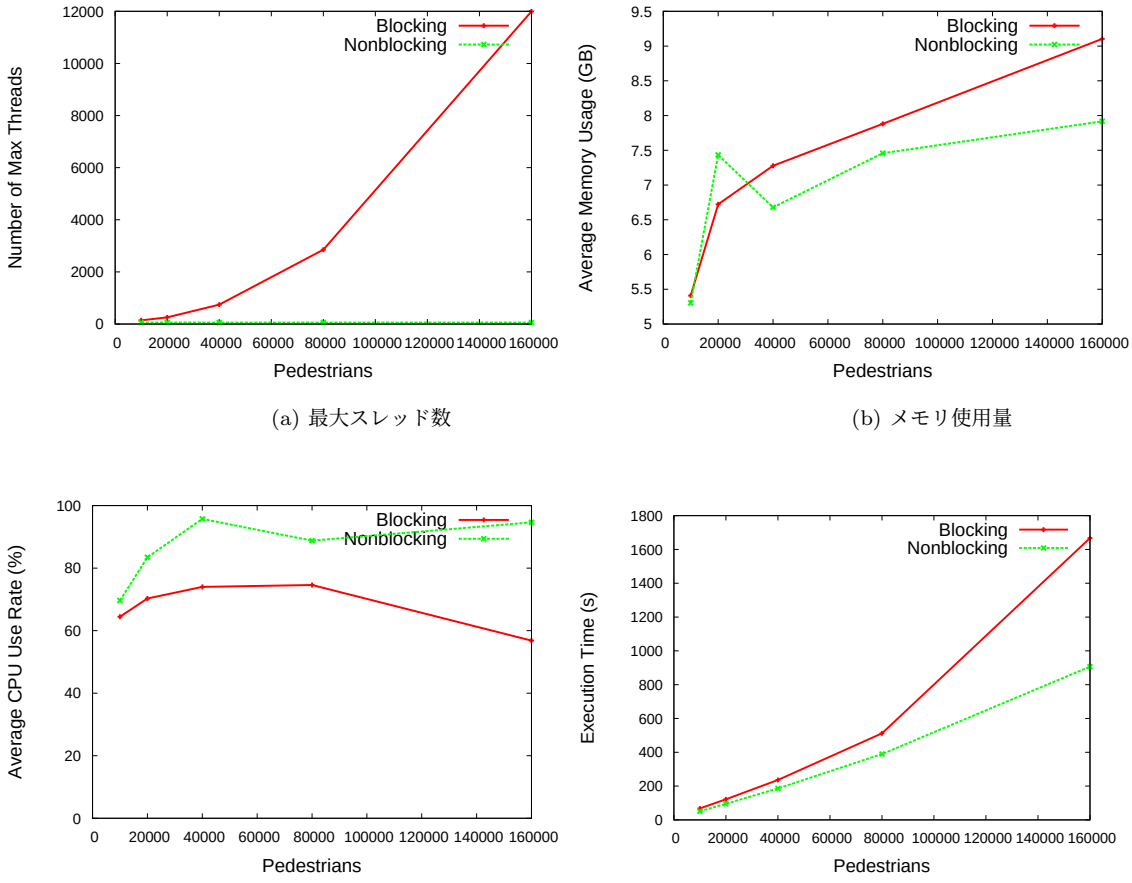
(a) 最大スレッド数　(b) メモリ使用量

(c) CPU 使用率　(d) 実行時間

図 9　グループバリア同期によるスレッド数削減の効果

している x10 [6] 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全て相互に同期を行うため，一方的な同期は不可能である．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 [5] という機能が提案されており，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 を用いると Clocks や CyclicBarrier では不可能であった一方的な同期が可能な上，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．

3 章で述べたようなスレッド分割によって変換されたプログラムは一般的に継続渡しスタイル（CPS: Continuation-Passing-Style）と呼ばれ，これまでに 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 RIFE [2] や Commons Javaflow [1]．これらの実装技術を用いてスレッド分割を施せば，現在本研究でサポートしていない制御文



(a) 最大スレッド数　(b) メモリ使用量

(c) CPU 使用率　(d) 実行時間

図 10　スレッド分割によるスレッド数削減の効果

内でのバリア同期や，ローカル変数の保存なども可能になる．

7　まとめ

本研究では，エージェントシミュレーションのような不均一で大規模な並列タスクを持つアプリケーションに有効な軽量なバリア同期機構を開発した．並列タスクが不均一の場合，一般的な全てのスレッドに対するバリア同期を用いると，タスク数が増加した際，大量の待機スレッドが発生する．本研究では待機スレッドの数を削減するため，実行時に対象のタスクを指定可能なグループバリア同期機構を開発し，また，スレッド分割に対応した API も用意した．これらを組み合わせることでスレッド数が削減され，メモリ使用量の現象やパフォーマンスの改善につながることを，実際の歩行者シミュレーションのサブセットに適用して確認した．

今後は同期機構をさらに拡張し，finished 構文の自動挿入やスレッド分割の自動化などを検討している．スレッド分割を自動化を行うことで一般的なバリア同期と同じプログラムを並列タスク数が増加しても効率よく実行することが可能となる．

謝辞 本研究を進めるにあたり，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。

また本研究は JST CREST の支援を受けています。

参 考 文 献

[1] Apache Commons. Javaflow. http://commons.apache.org/sandbox/javaflow/.

[2] RIFE Java web application framework. http://rifers.org/.

[3] MPI A Message-Passing Interface Standard 3.0, 2012.

[4] OpenMP Application Program Interface Version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[5] Shirako, J., Peixotto, D. M., Sarkar, V., and Scherer, W. N.: Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization, *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, New York, NY, USA, ACM, 2008, pp. 277–288.

[6] X10 Language Specification version 2.4, May 2012.