情報処理学会論文誌　プログラミング　Vol.8　No.3　22–32 (Sep. 2015)

並列分散処理向けプログラミング言語X10向けの対話的にフィルタリング可能な挙動可視化ツールの開発

板橋 晟星[1,a)]　佐藤 芳樹[2,b)]　千葉 滋[1,c)]

受付日 2015年2月9日, 採録日 2015年6月2日

概要：本研究では，並列分散処理向けプログラミング言語 X10 の暗黙的な挙動を解析するためのプロファイラおよびビジュアライザを開発した．X10 では非同期処理や分散処理が抽象化されるため，スレッド生成や同期，移動，分散ノード間通信のような低レベルな操作が X10 処理系によって自動的に行われる．そのような暗黙的なプログラムの動作は可読性や保守性を高める一方で，チューニングやデバッグを困難にさせる可能性が出てくる．しかしながら，今後メニーコアやアクセラレータ上で大量の並列スレッドを用いたソフトウェア開発が求められるなか，すべての暗黙的な動作の実行履歴をユーザに開示することは現実的ではない．本研究で開発したプロファイラは，X10 固有の非同期・分散処理構文に対してコンパイル時に追跡コードを挿入し，スレッド移動，同期，通信のような動作イベントを収集する．さらに，コンパイル時の静的解析によって分散ノード間で暗黙的に送受信されるデータ量を見積もれるようにした．収集したイベント情報を可視化するビジュアライザには，可視化範囲を制限するための Stream API ライクなフィルタリング DSL 機能を実装した．ユーザは実行時情報に対するフィルタ条件をソースコードに紐付けて指定し，大量のイベントやアクティビティを対話的に絞ることが可能となる．

キーワード：並列分散処理，コンパイラ，プロファイラ，DSL，X10

An Interactive Profiler/Visualizer with Event Filters for Parallel Distributed Language X10

Seisei Itahashi[1,a)]　Yoshiki Sato[2,b)]　Shigeru Chiba[1,c)]

Received: February 9, 2015, Accepted: June 2, 2015

Abstract: We developed a profiler and visualizer to analyze the implicit behavior of X10, which is a programming language for parallel distributed processing. Since X10 abstracts asynchronous and distributed processing, low-level operations such as thread creation, synchronization, thread movement and data communication among distributed nodes are performed automatically by X10 runtime. Although such implicit operations of programs improve the readability and maintainability, on the other hand, it would make tuning and debugging more difficult. However, software development with large number of parallel threads on many cores and accelerators is demanded. Thus, it is not realistic to show all of the operation logs to users. Our profiler inserts tracing codes toward asynchronous and distributed processing constructs that are original to X10, and collects the information of action events such as thread movement, synchronization and data communication. Moreover, our profiler can estimate the size of implicitly transferred data among distributed nodes by static analysis. On the other hand, Stream API-like filtering DSL function that restricts the range of visualization is implemented to our visualizer that visualizes the result of profiling. It is possible for users to restrict the massive event and activity information by designating a filtering condition toward runtime information, which is linked to source code.

Keywords: parallel distributed processing, compiler, profiler, DSL, X10

1. はじめに

MPI や X10 をはじめとする並列分散処理向けプログラミング言語やライブラリでは，複数のプロセッサや分散ノードを利用した低レベルな処理を抽象化して，ソフトウェアの開発生産性を向上させる．X10 では特に抽象化されたメモリやスレッドモデルを提供し，分散クラスタやスーパーコンピュータ向けのソフトウェアを開発しやすくなる．その一方，低レベルの処理は隠蔽されてユーザに見えづらくなるため，意図しない挙動が起きたり，挙動を把握するのが困難であるという問題がある．

本論文では，X10 の暗黙的な挙動の可視化ツール X-Eye について述べる．X-Eye は X10 の実行時情報を取得するプロファイラとそれを可視化するビジュアライザから構成される．X-Eye プロファイラは，コンパイル時に X10 プログラムを解析し，X10 固有の並列分散構文に対して追跡コードを挿入することで，実行時にそれらの構文イベントの情報を取得する．たとえば，async によるアクティビティ（スレッド）の生成，at によるアクティビティのプレース（メモリ空間）間の移動，finish によるバリア同期が構文イベントである．また，X-Eye ビジュアライザは，イベント情報を可視化するだけでなく，大規模並列分散処理で大量のイベントが発生した場合でも挙動を把握できるよう，可視化するイベントの範囲を専用 DSL によって適切に絞り込むことができるイベントフィルタリング機能を備える．

X-Eye の実用性を検討するために，性能解析の際の X-Eye プロファイラおよびビジュアライザのオーバヘッドや描画速度を測定した．性能解析では，X10 ベンチマークの一部に，同一プレースへのアクティビティの不必要な移動や無駄なバリア同期待ちのようなオーバヘッドの要因となる暗黙的な挙動を確認することができた．X-Eye プロファイラの挿入する追跡コードによるオーバヘッドを測定した結果，アクティビティの移動が頻繁に発生する場合はやや大きくなるが，頻繁ではない場合は十分に抑えられることが分かった．頻繁な移動でオーバヘッドが大きくなるのは，アクティビティの移動によって追跡コードの一部が送受信されて発生するオーバヘッドに起因すると考えられる．また，専用 DSL によるイベントフィルタリングの有無による描画速度を測定した結果，フィルタリングをすることで描画速度が改善された．フィルタリングによって可視化するイベント数や，グラフの点ノードや線ノードの生成回数が大幅に減少したためだと考えられる．

以下，2 章では X10 の抽象化された構文とメモリモデルを説明するとともに，その問題点を具体的なソースコードを交えて示し，3 章では我々の開発した X-Eye の解析手法と実装について述べる．4 章で X-Eye の実用性を調べるためのサンプルプログラムへの適用結果とプロファイラの性能測定をした結果を提示する．5 章では関連研究について論じ，6 章で本論文のまとめを述べる．

2. 並列分散言語 X10 の暗黙的な挙動

2.1 X10 の抽象化されたメモリモデルと並列分散構文

X10 は並列分散処理向けに抽象化されたメモリモデルと専用の言語構文を有するプログラミング言語である．X10 では，メモリ空間をプレースと呼ばれる区分化されたメモリ空間に分割しており，ユーザが物理メモリを意識し局所性を考慮したプログラムを書くことができる．**図 1** に示すように，X10 における遠隔処理や遠隔参照は，プレース間でのアクティビティ（スレッド）の移動によって実現される．また，プレース間にまたがる分散配列が提供され，複数プレースを利用した分散処理も容易に記述できる．プレース数はユーザが任意に設定し，単一ノード上のスレッド並列と複数ノードを用いたノード並列を柔軟に設定することができる．

X10 では，抽象化した分散メモリを前提として，並列分散処理向けの様々な専用構文が用意されている．たとえば，プレースを移動するための at，アクティビティを生成するための async，バリア同期用の finish を利用したコード例を**図 2** に示す．図 2 は X10 の並列分散構文を使用したコード例である．1 行目の async 構文によってアクティビティが生成され，非同期処理が開始される．アクティビティは，移動先プレースで参照するローカル変数のみを，暗黙的にコピーして移動する．たとえば，3 行目の at ブロック内の処理 S1 は移動先で実行され，4 行目でローカル変数 a が参照されているため，a の値は at ブロックの先頭でコピーして送受信される．at ブロック終了時には，元のプレースへ再度移動し，処理 S2 が実行される．複数アクティビティによるバリア同期は finish ブロックで async ブロックを包含して指定する．たとえば，10–13 行目は，新しく作られたアクティビティが S3 と S4 をそれぞれ実行して終了する

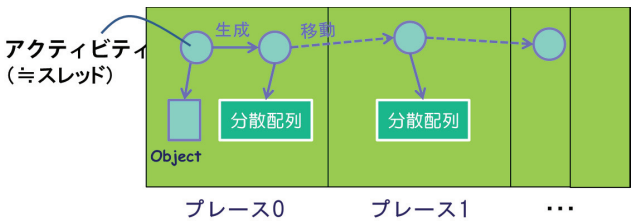

図 1　X10 のアクティビティとプレース
Fig. 1　Activity and place in X10.

[1] 東京大学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) seisei@csg.ci.i.u-tokyo.ac.jp
b) yoshiki@cc.u-tokyo.ac.jp
c) chiba@csg.ci.i.u-tokyo.ac.jp

```
1  async { //アクティビティ（スレッド）生成
2    val a:long = 1;
3    at (here.next()) { //隣のプレースへ移動
4      Console.OUT.println(a);
5      S1;
6    }
7    S2;
8  }
9
10 finish { //バリア同期
11   async { S3; }
12   async { S4; }
13 }
```

図 2　X10 の並列分散構文のソースコードの例
Fig. 2　An example code of X10's parallel/distributed constructs.

```
1  //変数 a は 10 要素の配列で，すべて 1 で初期化
2  val a = new Rail[Long](10, 1);
3  val b:long = 1; //long 型の変数 b は 1 で初期化
4
5  at (here.next()) { //プレース 1 へ移動
6    at (here.next()) { //プレース 2 へ移動
7      doSomeTasks();
8      //変数 a の配列の最初の要素のみが参照される
9      Console.OUT.println(a(0));
10   }
11 }
```

図 3　暗黙的な送受信が起きる例
Fig. 3　An example code of implicit data communication.

まで，元のアクティビティは finish の実行を終了しない．

2.2　X10 の暗黙的な挙動による問題

X10 の抽象化された並列分散構文は生産性の向上に寄与するが，データ通信やスレッド制御のような暗黙的な挙動を把握した低レベルなチューニングが難しくなる問題がある．X10 は低レベルな処理を抽象化することで，データ送受信，アクティビティ生成そして同期制御のプログラミングを容易にする一方，意図しないデータ送受信やバリア同期のブロッキングが性能を低下させる場合がある．これらは，X10 処理系によって暗黙的に処理されるためユーザが気づきにくく，原因も特定しにくい．たとえば，at 構文では，プレース間移動や遠隔参照の際にデータ送受信を暗黙的に行うためにユーザの意図しない送受信を引き起こす．これは，移動元プレースで宣言されているローカル変数のうち，移動先プレースで参照されている変数が暗黙的に送受信されるためである．たとえば図 3 の 2 行目では 10 個の要素を持つ配列 a がローカル変数として宣言され，すべての要素が 1 で初期化されている．3 行目では long 型のローカル変数 b が 1 で初期化されている．そして 5，6 行目でそれぞれプレース 1，プレース 2 へ移動し，9 行目で配列 a の最初の要素のみが参照されている．X10 処理系の実装では，配列 a のすべての要素が，プレース 1 へ送信され次にプレース 2 へ送られる．通信オーバヘッドを削減するためには，送受信データサイズや送信経路を把握したチューニングを行う必要がある．たとえば，ローカル変数の宣言箇所やアクセス箇所の削減によって間接的に送信サイズを減少させたり，不要な多重の at ブロックの回避による中継プレースでの通信処理の削減が考えられる．しかし，送受信サイズや経路は X10 処理系によって決められるため，プログラマは，まず送受信される変数の特定（変数 a および b）と個々の変数のサイズ（配列 a の要素数）を解析する必要がある．その上で，送受信経路を把握するためには，移動経路の追跡と各移動先プレースで参照されている変数を解析する必要も出てくる．これらは，移動先プレースでのより多くの変数の参照，配列サイズの動的変化やオブジェクト型変数の送受信の発生，at 構文の多重入れ子など，コードの複雑化にともなってより困難となる．

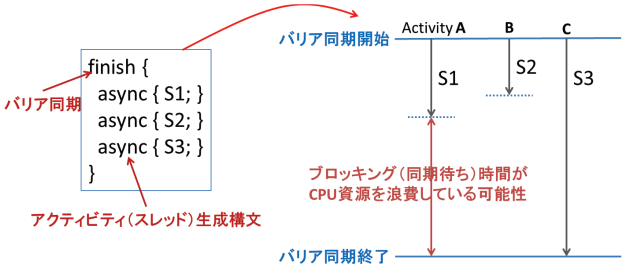

図 4　不要な同期待ち
Fig. 4　Unnecessary barrier synchronization.

```
1  abstract Scheduler {
2    abstract def run(taskSet: Set[Task], num:
         long);
3
4    def doTask(taskSet: Set[Task], num: long) {
5      finish { //同期制御の開始
6        run(taskSet, num);
7      }
8    }
9  }
```

図 5　Scheduler クラス
Fig. 5　The Scheduler class.

さらに finish，async のような抽象度の高いバリア同期構文は，不要な待ち時間を誘発する可能性がある．これらの直観的な構文によってバリア同期やアクティビティ生成を容易に記述できるため，たとえば図 4 のアクティビティ A，B，C のような不均一な処理時間によって，同期待ち時間の長いプログラムを実装しがちとなってしまう．

さらに，finish と async はプログラム中の離れた場所に別々の構文で記述可能であるため，バリア同期に参加するアクティビティを静的に把握することは難しい．図 5 の抽象クラス Scheduler の doTask メソッドのバリア同期内の処理は，抽象メソッド run の実装クラスに依存する．たとえば，図 6 に示す実装クラス OptimizedParallel の場合，run メソッド内の async ブロックを囲むループの繰返し条件を

```
1 class OptimizedParallel extends Scheduler {
2   def run(taskSet: Set[Task], num: long) {
3     for(var i:long=0; i<taskSet.length; i+=num)
4       async {
5         for (var j:long=i; j<i+num; j++)
              taskSet.get(j).run();
6       }
7   }
8 }
```

図 6　OptimizedParallel クラスの run メソッド
Fig. 6　The run method of the OptimizedParallel class.

```
1 class Sequential extends Scheduler {
2   def run(taskSet: Set[Task], num: long) {
3     //すべてのタスクを1つのアクティビティで処理
4     for (task in taskSet) task.run();
5   }
6 }
```

図 7　Sequential クラスの run メソッド
Fig. 7　The run method of the Sequential class.

```
1 class Parallel extends Scheduler {
2   def run(taskSet: Set[Task], num: long) {
3     //1タスクにつき1アクティビティを生成
4     for (task in taskSet) async { task.run(); }
5   }
6 }
```

図 8　Parallel クラスの run メソッド
Fig. 8　The run method of the Parallel class.

決める変数 taskSet や num を確認しなければ，バリア同期に参加するアクティビティを特定できない．この場合，同期制御のコード内で生成されるアクティビティの個数は，変数 taskSet と変数 num の値に依存するため，これらの変数の値を追跡する必要がある．

それに加えて，複数のサブクラスがバリア同期内のメソッド呼び出しを上書きする場合，さらにアクティビティの特定が困難となる．Schedular クラスの run メソッドが，OptimizedParallel クラスに加えて，2 つのサブクラス Sequential クラス（図 7）と Parallel クラス（図 8）で上書きされる場合を考える．Sequential クラスの run メソッドは，1 つのアクティビティによる逐次処理で，Parallel クラスの run メソッドは 1 つのタスクにつき 1 つのアクティビティを生成し並列処理するメソッドである．Scheduler オブジェクトに対する doTask の呼び出しで，コンテキストに応じて実装クラスやパラメータが変化する場合，実行がどの実装クラスにディスパッチされるかは静的には容易に特定できない．したがって，いくつのアクティビティが同期に参加するのかも静的には特定できなくなる．

3.　X10 の暗黙的な挙動可視化ツール X-Eye

本研究では，X10 向けの暗黙的な挙動可視化ツール X-Eye を開発した．X-Eye は，プログラムの挙動を解析するプロファイラと，解析結果を可視化するビジュアライザから構成されている．プロファイラは，コンパイル時に X10 プログラムに追跡コードを埋め込み，実行時に X10 固有のイベントの実行時情報を取得する．ビジュアライザは，プロファイラの解析結果に基づいて X10 の暗黙的な挙動を可視化する．また，大規模処理で大量のイベントが発生した場合でも可視化された情報をユーザが適切に把握できるようにするために，可視化範囲を適切に絞り込むことができるイベントフィルタリング機能と専用 DSL を提供する．

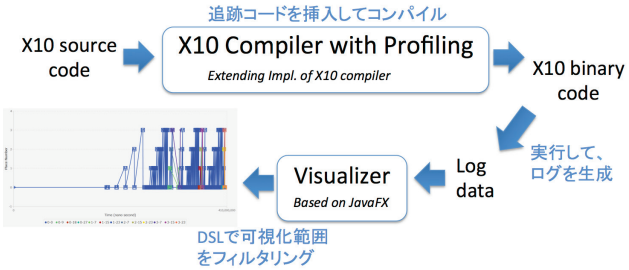

図 9　X-Eye の挙動可視化の流れ
Fig. 9　An overview of profiling and visualizing of activities in X-Eye.

X-Eye プロファイラは，Java 向けの Managed X10 と C++向け Native X10 のどちらにも対応できるよう X10 コンパイラ拡張によるソースコード変換で実現した．X-Eye による可視化の過程を図 9 に示す．まず，X-Eye プロファイラは，X10 ソースコードに対してコンパイル時に，挙動を追跡するコードを X10 固有の並列分散構文を対象に挿入する．コンパイル後に追跡コードが反映された X10 バイナリファイルが出力され実行されると，X10 固有の抽象化構文によるイベントの実行時情報に関するログデータが出力される．そして Java の GUI ライブラリである JavaFX で開発したビジュアライザが，ログデータを解析してグラフとして可視化する．

3.1　プロファイラが収集する実行時情報

X-Eye プロファイラは X10 ランタイムを修正せずに，暗黙的な実行時情報を取得するために，構文だけでなく型やスレッドに関する情報を収集する．取得する実行情報を以下に示す．

- X10 固有の並列分散構文によるイベント
- 暗黙的なデータ送受信
- アクティビティ識別子

まず，プロファイラはコンパイル時に X10 固有の並列分散構文（at，async，finish，atomic などの構文および関連 API の呼び出し）の前後に対して追跡コードを挿入し，それらのイベントの実行時情報を取得する．情報は各プレースごとに保持してプログラムの実行終了前にプレース 0 へ転送し，プレース 0 においてすべての情報を JSON 形式に整形してログファイルとして出力する．

暗黙的なデータ送受信サイズについては，コンパイル時にプログラムの静的解析により送受信される変数を推定し，見積もる．具体的には，at ブロック内で参照されている変数がブロック外で宣言される場合は送受信が行われるものと見なす．送受信サイズはプリミティブ型変数については，Int と Float が 4 バイト，Long と Double は 8 バイトとして計算する．変数が配列である場合は，配列要素数を求め，オブジェクト型である場合は，フィールドサイズを再帰的に求める．ただし，配列要素数が動的に変化する場合は正確に算出することができない．推定した送受信サイズは，コンパイル時に変数名と対にして追跡コードに埋め込み，実行時に送受信プレース情報とまとめて記録する．

アクティビティ識別子は，アクティビティどうしを区別し，イベントとアクティビティを紐付けるために，X-Eye が付与する．標準的な X10 では，アクティビティを特定する機能が提供されていないため，追跡コードがアクティビティ識別子を引数として受け取るようにコードを生成する．我々は，既存の X10 処理系を改造せずにアクティビティと識別子を紐付けるために，コンパイル時にプロファイラが既存メソッドに新たな引数を追加し，その引数で識別子を受け渡すようにした．これにより，追跡する並列分散構文に対応するイベントをアクティビティ識別子を割り当てて記録することができる．

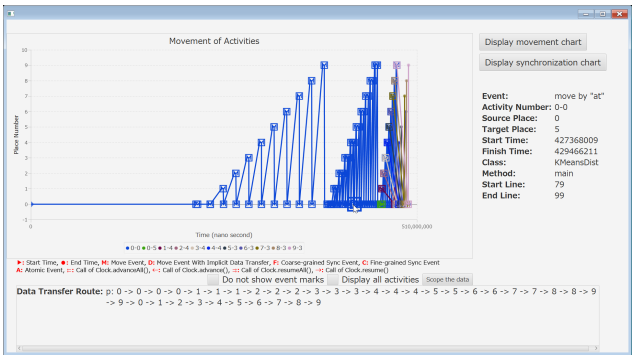

図 10　Movement Chart
Fig. 10　Movement Chart.

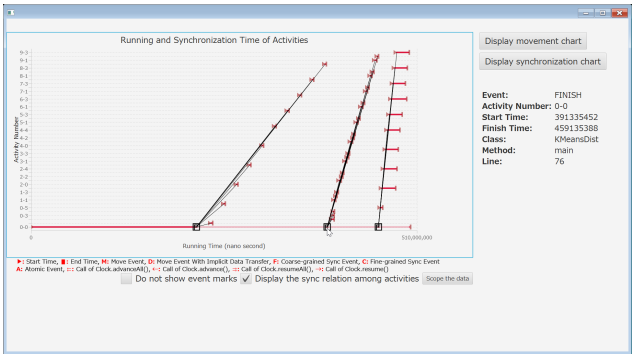

図 11　Synchronization Chart
Fig. 11　Synchronization Chart.

3.2　DSL によるイベントフィルタリング機能が実装されたビジュアライザ

X-Eye ビジュアライザには，収集したイベント情報を可視化するだけでなく，イベントが大量に発生した場合でも適切に可視化する範囲を絞り込むことができる専用 DSL によるイベントフィルタリング機能を実装した．X10 が利用されるような大規模並列分散処理では，大量のイベントが発生しやすい．そのため，それらすべてを可視化すると膨大な情報量やグラフの複雑化によって，プログラムの実行速度低下の原因などをユーザが把握することが困難になる．

ビジュアライザは，Movement Chart と Synchronization Chart の 2 種類のグラフを提供し，それぞれアクティビティの移動と同期による挙動の詳細を可視化する．グラフにおける x 軸（横軸）は両グラフともにプログラム開始からの経過時間を表し，Movement Chart では**図 10** のように，y 軸（縦軸）がプレース番号に対応し，Synchronization Chart では**図 11** のように y 軸がアクティビティ番号に対応している．アクティビティ番号は，プレース番号とプレース内でのアクティビティ番号をハイフンで繋いだ文字列として表示する．X10 固有の構文によるイベントはグラフ中に M（at），D（データ送受信をともなう at），F（finish）のマークとして表示し，クリックすることで右側にイベントの実行時の詳細情報，データ送受信をともなう at イベントの場合はさらに下側に送受信された変数名と送受信経路を表示する．

イベントフィルタリング用 DSL は，Java 8 の Stream API ライクに設計しラムダ式でフィルタ条件を記述できるようにした．ユーザは収集したイベント情報を保持している Stream 型変数 eventStream に対し，filter メソッドを使ってフィルタリングすることができる．filter メソッドは引数に，ラムダ式の形で記述されたフィルタ条件をとり，ストリーム中のデータを 1 つ 1 つフィルタ条件に適合するかどうか照合することでデータを絞り込む．eventStream が保持しているデータの型は，我々で定義したイベント情報を表すクラス Event.Elem 型となっており，**表 1** で示した public 修飾子のついているフィールドを使ってフィルタ条件を記述する．例として，**図 12** に，プレース 0 で発生したイベントのうちアクティビティ 0–0（main 関数を実行したメインアクティビティ）によって実行されたイベントのみ可視化する DSL の記述例を示す．filter メソッドの引数となっている Event.Elem 型の e を通して，イベントを実行したプレースを表すフィールド execPlace に対する条件を記述し，その値が 0 であるイベントのみを指定している．

また，ユーザが適切なフィルタ条件を見つけやすくするために，X-Eye ビジュアライザは対話的なフィルタリングも可能としている．特に，オーバヘッドの原因を分析する場合，ユーザは複数のフィルタ条件を少しずつ適用しなが

表 1　フィルタ条件の引数としてとる Event.Elem クラスのフィールド

Table 1　The fields of the Event.Elem class to be used for filtering.

フィールド名	型	説明
activityNumber	String	イベントを実行したアクティビティ番号
startTime	long	イベントの開始時間
finishTime	long	イベントの終了時間
className	String	イベントを実行したクラス
methodName	String	イベントを実行したメソッド
sourcePlace	int	（移動イベント限定）移動先プレース番号
targetPlace	int	（移動イベント限定）移動元プレース番号
execPlace	int	（移動イベント以外）イベントの実行されたプレース番号
startPosition	long	コード中でのイベントの開始行
endPosition	long	コード中でのイベントの終了行
type	EventType（Enum）	イベントの種類（MOVE, EXEC, FINISH, CLOCK, ATOMIC を保持）

```
1 eventStream = eventStream.filter(
2         e -> e.execPlace == 0)
3                         .filter(
4         e -> e.activityNumber.equals(``0-0''));
```

図 12　プレース 0 で発生したイベントのうちアクティビティ 0–0 によって実行されたイベントを表示する DSL の記述例

Fig. 12　An example code to display the events executed by the activity 0–0 and created at the place 0.

ら原因を絞り込むことが多い．そのため，フィルタ条件を記述している途中でもその途中の条件による可視化結果を見ながら対話的にフィルタリングできるようにした．具体的には**図 13** のような，DSL を記述するためのシェル上で DSL を記述し，filter メソッドの記述後にセミコロン（;）もしくはドット（.）を入力することで DSL で記述したフィルタ条件が適用される．たとえば図 12 は，プレース 0 で実行したイベントのみの可視化結果を表示した後，アクティビティ 0–0 が実行したイベントに絞って表示する．

3.3　実装

X-Eye プロファイラは X10 コンパイラを拡張し，コンパイル時に X10 の AST（抽象構文木）に対して追跡コードを挿入するようにした．X10 コンパイラは Java コンパイラ拡張フレームワークである Polyglot [1] の拡張実装であるため，X-Eye は Polyglot 用にビジタークラスを新たに追加することで実装した．Polyglot では，コンパイル時に AST を巡回する様々な処理をビジターを実装することで追加できる．**図 14** に示すように，ObjectRecorder クラスはオブジェクトの参照関係をたどり，各フィールドの

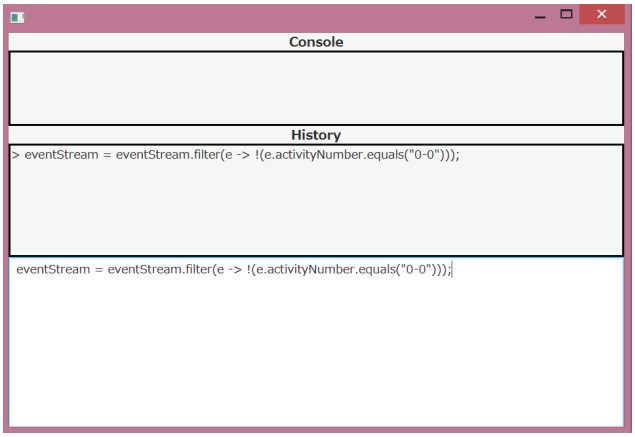

図 13　イベントフィルタリング用のシェル．Console 部は標準出力，History 部は入力した DSL コードの実行履歴を表示

Fig. 13　A shell for event filtering. The Console part displays the standard output. The History part displays the history of input code.

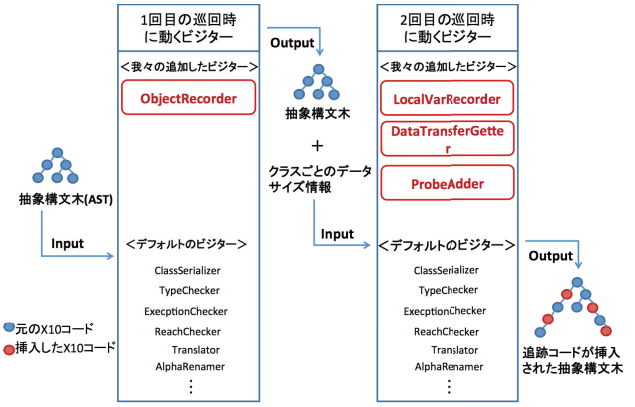

図 14　AST 解析のために追加したビジター

Fig. 14　Visitor classes for analyzing AST.

データサイズを見積もる．次に，それらの情報を保持し，AST 構造に従った 2 回目の巡回を行う．2 回目では，すべてのローカル変数について詳細情報（変数名，型，クラス名，メソッド名，行番号）を取得する LocalVarRecorder クラス，送受信される変数を推定する DataTransferGetter クラス，AST 中の X10 固有のイベントのノードの前後に追跡コードのノードを挿入する ProbeAdder クラス，などのビジターが AST を巡回し，追跡コードが挿入された X10 の AST を出力する．なお，at 構文に対する追跡コードは，DataTransferGetter クラスが推定した送受信される変数の情報を追跡コード中に埋め込んだ後に挿入する．2 回目の巡回後に出力した追跡コードが挿入された AST を X10 コンパイラでコンパイルしてバイナリファイルを出力する．

X-Eye ビジュアライザに実装されているイベントフィルタリング用 DSL は，Java コード片としてオンメモリーでコンパイルされ，出力されたクラスファイルをロードすることで実行される．**図 15** のように，書き込まれた DSL のコードは文字列として String 型の変数に保存され，同じ

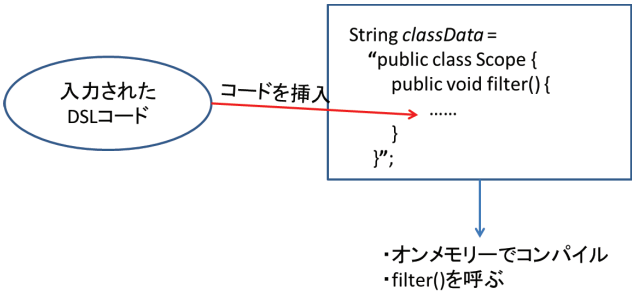

図 15　イベントフィルタリング用 DSL の実行過程
Fig. 15　An overview of processing the event filtering DSL.

く String 型の文字列として保持している Scope クラスのソースコードの中にある，イベントをフィルタリングするための filter() メソッド内に文字列として埋め込む．こうして得られた Scope クラスのソースコードをメモリ上に展開し，Java コンパイラ API を呼び出すことでオンメモリで Java バイトコードを生成する．メモリ中の Scope.class を動的にロードして Scope クラスのインスタンスを生成し，そのインスタンスの filter() メソッドを呼ぶことで，イベントデータがフィルタリングされる．

4.　実用性の検討

X-Eye の実用性を検討するために，X10 のベンチマークに適用して性能ボトルネックを探すとともに，X-Eye プロファイラが挿入する追跡コードによるオーバヘッドを測定した．また，X-Eye ビジュアライザについても，イベントフィルタリングの有無による可視化速度の変化を測定した．

4.1　ケーススタディ

ケーススタディとして，X10 コンパイラ（バージョン 2.4.0）のプログラムに同梱されているサンプルプログラム分散版 K 平均法（KMeansDist.x10）と並列版 N クイーン問題（NQueensPar.x10）に X-Eye を適用した．その結果，不要なアクティビティの移動やデータ送受信，バリア同期待ちを確認することができた．**図 16** に示した KMeansDist.x10 の結果からは，同一プレースでのアクティビティの移動や，主アクティビティが全プレースを巡回して子アクティビティを生成していることを確認できる．下側のグラフは拡大したグラフであり，右側に表示されているイベント情報とグラフの下に表示されている変数名の送受信経路は，グラフ中のイベントをクリックして表示した詳細情報である．データ送受信をともなう移動は D マークで表示され，詳細情報の移動元 Source Place と移動先 Target Place が同じプレース 0 であることが分かる．また，下段に表示されている変数の送受信経路を見ると，変数 p が同じプレースへ何度も送受信されている．実際の KMeansDist.x10 のコードを**図 17** に示す．2 行目において，分散配列要素の存在するプレースを巡回している．また，1 行目で現在の

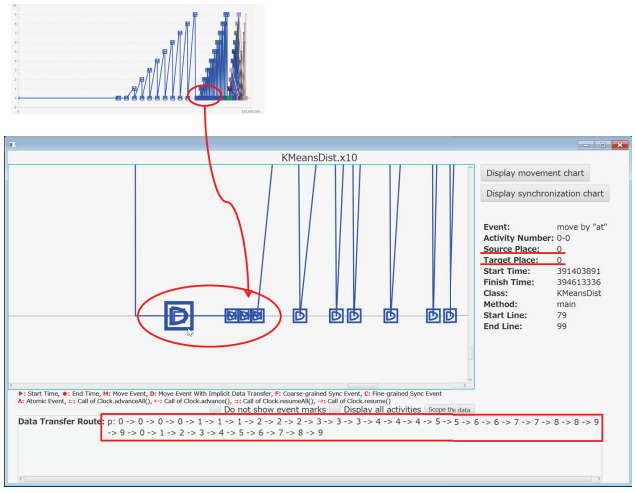

図 16　同一プレースへのデータ送受信をともなう移動
Fig. 16　Local activity movements causing unnecessary data communication.

```
1  val there = here;
2  for (d in points_dist.places()) at (d) async {
3    val tmp_new_clusters = local_new_clusters();
4    val tmp_cluster_counts = local_cluster_counts
       ();
5
6    at (there) atomic {
7      for (j in 0..(DIM*CLUSTERS-1)) {
8        central_clusters_gr()(j) +=
             tmp_new_clusters(j);
9      }
10     for (j in 0..(CLUSTERS-1)) {
11       central_cluster_counts_gr()(j) +=
             tmp_cluster_counts(j);
12     }
13   }
14 }
```

図 17　KMeansDist.x10 のコード片
Fig. 17　A piece of code of KMeansDist.x10.

プレース情報を変数 there に代入し，2 行目で移動をした後さらに 6 行目で there に対して移動をする際に同一プレースへの移動が発生している．これは，2 行目での移動先プレースの中に変数 there に格納されているプレースと同じプレースが含まれているためである．

図 18 に示す NQueensPar.x10 の結果からは，バリア同期に参加しているアクティビティの一部によって同期待ちの時間が長くなったことが分かった．丸で囲んだ 2 回目のバリア同期では，アクティビティ 0–2 の処理開始のタイミングや処理時間が長いため，同期待ち時間を増加させており，以降の処理の開始も遅れさせていることが分かる．

4.2　性能測定

X-Eye プロファイラが挿入する追跡コードによるオーバヘッドを測定するため，KMeansDist.x10，NQueensPar.x10 と分散版 N クイーン問題（NQueensDist.x10）を用いて実

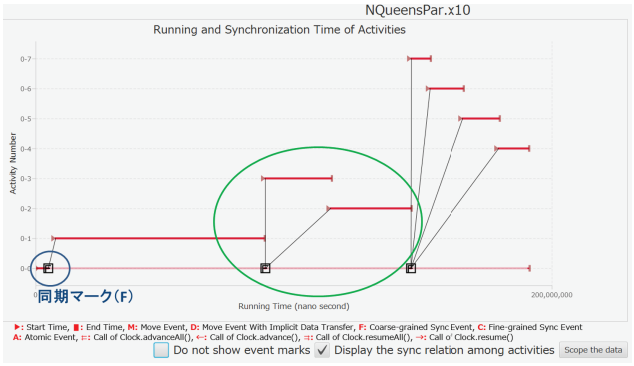

図 18　バリア同期によるブロッキング時間
Fig. 18　A blocking time caused by barrier synchronization.

表 2　1 イベントあたりの追跡コード自体の実行時間（マイクロ秒）
Table 2　Elapsed time of the profile code per event.

async	at	finish	atomic
3.50	11.81	0.58	0.6

験を行った．さらに，ビジュアライザに実装されているイベントフィルタリング機能の有無による描画速度の変化を比較をした．

まず，X-Eye プロファイラが挿入する追跡コードのオーバヘッドを確認するためにマイクロベンチマークで実験を行った．実験環境は以下のとおりである．

- CPU：Intel Xeon E5-2687W 3.10 GHz 8 cores
- RAM：64 GB
- OS：CentOS release 6.2
- Java：1.7.0_25, OpenJDK Runtime Environment
- X10：2.4.0, Managed X10
- X10 コンパイルオプション：-O -NO_CHECKS -classpath gson-2.2.4.jar
- X10 実行オプション：-classpath gson-2.2.4.jar

表 2 に結果を示す．結果は，JIT コンパイラの影響を排除し，10 万回の連続実行の平均実行時間である．結果から，at に対する追跡コードの実行時間が最も大きくなり，次に async が続き，finish と atomic は非常に小さい値になった．at の場合，送受信プレースの双方で追跡コードが実行され，またイベントデータの送受信をともなうため，async, finish, atomic と比べて実行時間が大きくなる．また，async の追跡時にはイベントデータを管理するオブジェクトを生成する時間が含まれる．

次に，X-Eye プロファイラ利用時の KMeansDist.x10, NQueensDist.x10 と NQueensPar.x10 のオーバヘッドを測定した．JIT コンパイラの影響を排除するために 10 回の予備実行の後に連続して 50 回測った結果の平均値を用いた．単一ノード上での結果を**図 19** と**図 21** に，プレース数と同数の物理ノードを利用した結果を**図 20** に示す．また，**表 3** と**表 4** にイベント発生件数を示す．実験環境は以下のとおりである．

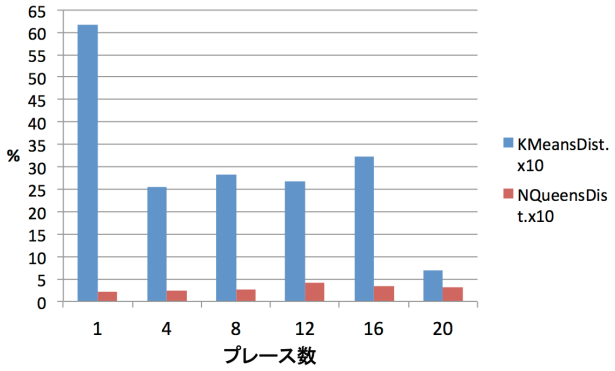

図 19　KMeansDist.x10（POINTS=2000）と NQueensDist.x10 (N=14) でのオーバヘッド（各プレース数で実行したうちの最大値）
Fig. 19　The maximum overheads measured in KMeansDist.x10 (POINTS=2000) and NQueensDist.x10 (N=14).

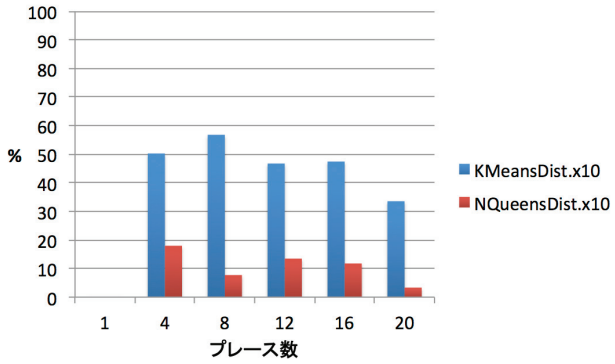

図 20　図 19 をプレース数と同数の物理ノードで行った場合のオーバヘッド
Fig. 20　Fig. 19's benchmarks processed an equal number of nodes as places.

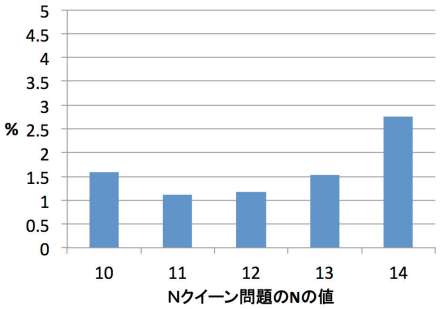

図 21　NQueensPar.x10 でのオーバヘッド
Fig. 21　The overhead of NQueensPar.x10.

- TSUBAME2.5 1〜20 ノード
- CPU：Intel Xeon X5670 2.93 GHz 6 cores
- RAM：54 GB
- OS：SUSE Linux Enterprise Server 11 SP3
- Java：1.6.0, IBM J9 VM (build 2.4)

単一ノード上で分散した KMeansDist.x10 では，at イベントの発生件数の多いプレース数では，大きなオーバヘッドが計測された．一方，NQueensDist.x10 のイベントの大

表 3　KMeansDist.x10 と NQueensDist.x10 における N の値とプレース数の違いによる追跡コードのオーバヘッドとイベント発生数

Table 3　The overhead of the profile code and the number of events with changing N and the number of places in KMeansDist.x10 and NQueensDist.x10.

プレース数	プログラム名	イベント発生回数（回）			
		async	at	finish	atomic
1	KMeansDist	44,044	44,066	66	44,022
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184
4	KMeansDist	26,104	26,156	39	26,052
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184
8	KMeansDist	26,208	26,312	39	26,104
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184
12	KMeansDist	28,337	28,504	42	28,168
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184
16	KMeansDist	34,544	34,816	51	34,272
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184
20	KMeansDist	12,241	12,360	18	12,120
	NQueensDist (N=12)	12	12	1	14,200
	NQueensDist (N=13)	13	13	1	73,712
	NQueensDist (N=14)	14	14	1	365,596
	NQueensDist (N=15)	15	15	1	2,279,184

表 4　NQueensPar.x10 におけるイベント発生数

Table 4　The number of events in NQueensPar.x10.

N の値	async	at	finish	atomic
10	7	0	3	2,172
11	7	0	3	8,040
12	7	0	3	42,600
13	7	0	3	221,136
14	7	0	3	1,096,788

```
1 eventStream = eventStream.filter(
2         e -> e.execPlace == 0);
```

図 22　プレース 0 で発生したイベントのみを表示する DSL の記述

Fig. 22　A DSL code to display the events only created at the place 0.

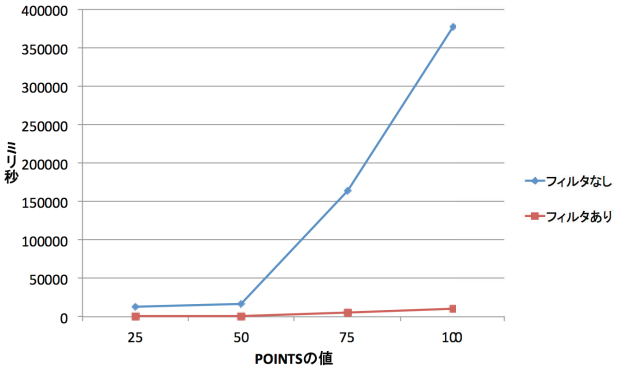

図 23　KMeansDist.x10 での，フィルタリング有無における描画速度の違い

Fig. 23　Differencies of the rendering speeds with and without event filtering in KMeansDist.x10.

部分は atomic であり，at イベントはわずかであるためオーバヘッドも 0.1〜4%程度に抑えられた．at イベントの発生しない NQueensPar.x10 に関しても，同様の結果が得られた．一方，複数ノード上で分散した実験においても，単一ノードと同様の傾向が見られ，KMeansDist.x10 では at 回数に相関し，NQueensDist.x10 では小さく抑えられていた．

最後に，イベントフィルタリングの有無による X10 ビジュアライザの描画速度を比較測定した．実験は，KMeansDist.x10 の結果を用いて，以下の環境で行った．

- CPU：Intel Core i7-3770 CPU 3.40 GHz 4 cores
- RAM：16 GB
- OS：OpenSUSE 12.3
- Java：1.8.0_25-b17, Oracle Java SE

フィルタ条件としては，プレース 0 で発生したイベントのみを表示する図 **22** を用いた．Movement Chart の描画速度とグラフノードの生成回数を図 **23**，図 **24** に示す．問題サイズが大きくなるにつれて，大量のグラフノードを生成するため描画速度は悪化する．一方，フィルタリングによって可視化範囲を絞れば，グラフノード生成を抑えられ効率的な解析が可能となる．

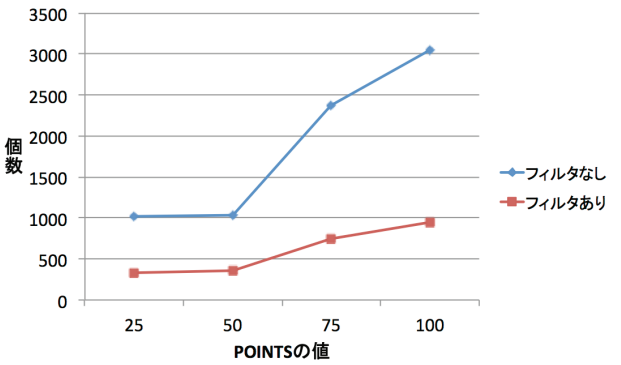

図 24　KMeansDist.x10 での，フィルタリング有無におけるグラフノードの生成件数の違い

Fig. 24　Differencies of the number of graph node creating with and without event filtering in KMeansDist.x10.

4.3　議論

本節では，プローブ効果と再現性について議論する．一般に，並行プログラムのプロファイルでは，追跡コードによる干渉がプログラムの振舞いを変更するプローブ効果と，非決定性による再現性の低さが問題となる．前節の性能測定から，X-Eye プロファイラの追跡コードによる測定オーバヘッドは，無視できないほど大きい場合があった．また，測定オーバヘッドによって同期タイミングが変化すれば，システム全体の挙動へ影響する可能性も出てくる．X-Eye プロファイラの追跡コードは，X10 固有の並列分散構文にのみ挿入し，プログラムの意味的な振舞いを変更しないよう設計しているが，大量の追跡コード実行によるプローブ効果は避けられない．したがって，X-Eye ビジュアライザの可視化結果の分析では，過大なイベント数を把握し，それによるプローブ効果の可能性を十分に考慮する必要がある．一方，アクティビティ間のデータ送受信量や移動，同期回数のような挙動は，オーバヘッドの多寡にかかわらないため，X-Eye を利用した分析がより適していると考えられる．

非決定性の問題により，アクテビティ識別子のような再現性の欠如したデータを使ったフィルタリングでは意図した調査ができないことがある．X-Eye はコンパイル時にアクティビティ生成箇所を特定して識別子の情報を埋め込み，メソッド呼び出しの引数として受け渡すようにしている．そのため，アクティビティの生成順序が異なれば，同じアクティビティに対しても別の識別子が割り付けられてしまう．すると，識別子が動的に決定されるアクティビティに対しては，アクティビティ識別子をキーとしたフィルタリングができなくなってしまう．X-Eye では，表 1 に示したように，イベントに関する様々なソースコード情報や実行時情報を収集している．これらを利用すれば，識別子が静的に決まらないアクティビティであっても，複合的な条件を与えてフィルタリングのキーとすることが可能で，非決定性にともなう問題を回避できる．たとえば，送信元と送信先のプレースや，移動回数のような再現性の高いデータを組み合わせれば，アクティビティの特定が可能となる．また，識別子以外のプログラム実行の再現性については，X10 プログラムに依存し，乱数や経過時間のような非決定的要因をもとにプログラムを実装しない限り，決定的な動作が期待できる．一方，再現性が担保されたプログラムであっても，X10 処理系のタスクスケジューラの非決定性によってアクティビティの実行順序が変化する可能性がある．したがって，これらの非決定的要因に依存した処理を認識し，アクティビティの移動や同期，データ送受信量などの分析対象の再現性を注意した利用が求められる．

5.　関連研究

X10 のプロファイリングツールに関する先行研究は少ないものの，MPI，Global Arrays [2]，Ti-trend-prof [3] など他の並列分散向け言語のプロファイリングツールの開発も含めると，数多くの研究がある．Vampir [4] や TAU [5] は，MPI でよく用いられるプロファイリングツールであるが，CUDA や X10 のプログラムに使うこともできる．X10 プログラムで使った場合には，get，set や lock などの低いレベルでのメモリオペレーションを収集する．しかし，X-Eye は async や finish などの高レベルな X10 固有の抽象化構文に関するイベントを解析している．

Nathan らによる研究 [6] では，Global Arrays においてグローバルな範囲のオブジェクトを対象に時間・ランク・呼び出し文脈の観点からプロファイリングを行うツールを開発しているが，プロファイリング対象がオブジェクトの動きのみであるという点で X-Eye よりもプロファイリングできる範囲が小さい．

Ti-trend-prof [3] は，Titanium 向けの自動でパフォーマンスデバッグを行うツールであり，共有メモリを介したリモートメモリへの暗黙的な read や write を探知し，それによりオーバヘッドを改善する．遠隔参照を探知する点では X-Eye と同じであるが，プロファイル対象言語が我々のツールとは異なる．また X-Eye は X10 固有のイベントの実行時情報を解析しているため，同期制御など様々な種類のオーバヘッドを見つけることができる．

XAnalyzer [7] は，X10 向けのパフォーマンスチューニングツールである．オーバヘッドが大きくなるコード断片のパターンを 8 つ準備しておき，それらのパターンを含むかどうかを検知する．XAnalyzer は X-Eye と同様に高いレベルでコードを解析するが，X-Eye は決まったパターンではなく，X10 固有の構文を対象にプログラムの全体の挙動を追跡し可視化することで，XAnalyzer よりも多くの種類のオーバヘッドを見つけられる可能性がある．

6.　まとめ

本研究では，X10 ユーザによるパフォーマンスチュー

ニングを支援するため，X10 の抽象化された並列分散構文による暗黙的な挙動の可視化ツール X-Eye を開発した．X-Eye を使うことで，分散ノード間で暗黙的に発生するデータ送受信や，同期制御内の挙動などが見えるようになっただけでなく，専用 DSL によるイベントフィルタリング機能によって，大規模処理で大量の数のイベントが発生したときでも可視化する範囲を対話的に絞り込めることで，適切なチューニング情報をユーザが取得できる．

ベンチマークを使ってプロファイリング時のオーバヘッドを測定した結果，at 構文が多いプログラムでは，追跡コード自身の一部が送受信されるためにオーバヘッドが大きくなる傾向にあるが，at 構文が少ないプログラムではオーバヘッドは低く抑えられることが分かった．また，DSL によるイベントフィルタリング有無によって，可視化速度が大きく異なることも分かった．

今後の課題として，at イベントが多いときでもオーバヘッドを小さく抑えられるように，イベント追跡方法を改良していくことと，イベントフィルタリング機能を拡張し，可視化した後に実行コンテキスト情報に基づいて対話的にプロファイリング範囲を絞ってプロファイリングしてできるようにすることがあげられる．また，分散配列のプロファイリングも今後の課題であり，配列の各プレースへの分散を実行するコードに対して追跡コードを挿入することで実現することができると考えられる．

参考文献

[1] Nystrom, N., Clarkson, M.R. and Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java, *Proc. 12th Int'l Conf. on Compiler Construction*, LNCS 2622, pp.138–152 (2003).

[2] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Aprà, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *Int'l Journal of High Performance Computing Applications*, Vol.20, No.2, pp.203–231 (2006).

[3] Su, J. and Yelick, K.A.: Automatic Communication Performance Debugging in PGAS Languages, *LCPC*, LNCS 5234, pp.232–245, Springer (2007).

[4] Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H. and Nagel, W.E.: Developing Scalable Applications with Vampir, VampirServer and VampirTrace, *Parallel Computing: Architectures, Algorithms and Applications*, pp.637–644, IOS Press (2008).

[5] Bruno Buchberger, J.V.: TAU: A portable parallel program analysis environment for pC++, *Parallel Processing: CONPAR 94 — VAPP VI*, pp.29–40, Springer Berlin Heidelberg (1994).

[6] Tallent, N. and Kerbyson, D.: Data-centric Performance Analysis of PGAS Applications, *2nd Int'l Workshop on High-performance Infrastructure for Scalable Tools* (*WHIST 2012*) (2012).

[7] Jeeva, P., Tardieu, O. and Amaral, J.N.: Guiding X10 Programmers to Improve Runtime Performance, *7th International Conference on PGAS Programming Models* (2013).

板橋 晟星

1989 年生．2013 年早稲田大学人間科学部人間情報科学科卒業．2015 年東京大学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．