東京大学
情報理工学系研究科 創造情報学専攻
修士論文

PGAS 言語 X10 の暗黙的な挙動を可視化するプロファイリングツールの開発

A Profiling Tool for Visualizing Implicit Behavior in X10

板橋 晟星
Seisei Itahashi

指導教員　千葉 滋 教授

2015 年 1 月

概要

本研究では、PGAS 言語 X10 の暗黙的な挙動を解析するためのプロファイラ及びビジュアライザを開発した。PGAS 言語とは、分散メモリを抽象化し、区分化されたメモリ空間でデータの局所性を意識したプログラミングを可能とするプログラミング言語である。PGAS 言語の一つである X10 では非同期処理や分散処理が抽象化され、チューニングやデバッグのために、スレッドの挙動や分散ノード間で暗黙的に送受信されるデータをユーザが把握することが困難な問題がある。また，プロセッサのメニーコア化が進むにつれて X10 の抽象化構文による処理量が増え，ユーザが人手で把握するのはますます困難となる．

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、通信のようなイベント情報を収集する。さらに、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。一方、プロファイラが収集したイベント情報を表示するビジュアライザは、可視化するデータを制限するためのスコーピング用 DSL を備えており、大規模並列分散処理で発生する大量のイベント情報でも適切な範囲を指定して抽出できるようにした。

Abstract

We developed a profiler and visualizer to analyze the implicit behavior of X10, which is a PGAS language. PGAS languages abstract the distributed memory and enables us to do programming with being aware of data locality in the partitioned memory space. In X10, asynchronous processing and parallel processing are abstracted. Thus, it is difficult for users to grasp the activities' behavior and implicit data transfer among distributed nodes, and it causes tuning and debugging to be difficult. Also, as processors have more cores, the number of processes X10 runs will increase, thus it will be more difficult for users to analyze by themselves.

Our profiler inserts tracing codes toward asynchronous and distributed processing construct that are original to X10, and collects the information of events such as thread movement and communication among nodes, and synchronization. Moreover, our profiler can estimate the size of implicitly transferred data among distributed nodes by static analysis. On the other hand, our visualizer visualizes the result of profiling. However, since the quantity of visualization will be larger as the processors have more cores, thus there will be a need that event data are to be filtered. For such a need, event filtering DSL for restricting the scope of event data is implemented to our visualizer. Thus, it can extract the data by designating the appropriate scope even when the number of events are greatly large.

目次

第 1 章

はじめに

プロセッサ等の半導体の集積度が 18 ヶ月から 24 ヶ月周期で倍増するというムーアの法則が長年その正しさを証明してきたが，近年それが限界だと言われていることから，CPU コア単体当たりの性能向上のスピードが遅くなっていくことが予想される。それに伴って特に大規模計算においては、多数の高性能ではないコンピュータに分散させて並列処理したり，HPC でメニーコア環境上で分散させて並列処理する等，分散並列処理の重要性が高まっていくと考えられる．

分散並列処理においては長年，MPI[1] と OpenMP[2] が主に使われてきた．MPI は，C 言語を用いて実装された分散並列処理用のインタフェースであり，複数台のコンピュータを使った分散メモリ型の分散並列処理に主に使われ，コンピュータ同士がメッセージを送受信することで協調動作をすることができる．MPI が実装されているコンピュータであれば，OS の違い等のコンピュータ同士の構成の違いを意識せずに分散並列処理ができる一方，メッセージのやり取りの中でも特にデータの送受信において，ユーザが明示的にかつデータの大きさ等の細かい部分まで書かなければならず，プログラムが複雑化しやすくバグが発生しやすいため，生産性が高くないという問題点がある．OpenMP は 1 台のコンピュータにおいて共有メモリを使った並列処理を行う際に使用されるインタフェースであり，HPC など高機能な計算機内で多数の CPU に処理を分散させて並列処理を行うことができる．しかし，共有メモリであることから，データの局所性が低く，データアクセス速度が遅くなることがある．

X10[3, 4] では，これらの問題を解決するために，同期処理や分散ノード間のデータ通信を高度に抽象化した構文や，区分化されたメモリ空間でのプログラミングを可能としている．抽象化構文によってデータ送受信は自動的に行われ，分散ノード間にまたがった同期処理も MPI より非常に少ないコード量で記述することができるため，生産性が高い．そして分散ノード間で大きな共有メモリ空間を形成し，それを区分化したメモリ空間を実現しているため，データの局所性も高くなっている．しかしながら，低いレベルの処理を抽象化しているために，スレッドの挙動やデータ送受信のタイミング・量など，ユーザがプログラムの暗黙的な挙動を把握しにくく，チューニングやデバッグをしにくいという問題がある．

本研究では，X10 の暗黙的におこなわれる挙動を探知し，可視化することができるプロファイリングツール X-Eye を開発した．X-Eye は，プロファイラとビジュアライザの二つから構

成され，プロファイラは X10 固有の抽象化構文を対象に解析をおこなう．一方，ビジュアライザでは，解析結果をグラフにして可視化し，ユーザに提示する．ビジュアライザには，データの可視化範囲をフィルタリングすることのできるスコーピング用 DSL を備えた，スコーピング機能が実装されている．スコーピング機能では，スコーピング DSL を記述することで，グラフとして可視化する解析データの範囲を適切に制限することができ，X10 プログラムが大量の処理をおこなってグラフが複雑化した場合でもプログラムの挙動を把握しやすいようにした．

本論文では，2 章で X10 という言語のプログラミングモデルと基本的な抽象化構文の説明及び抽象化構文の利点，次に X10 が抱えている問題点，そして関連研究について説明する．3 章では，本研究で開発したプロファイリングツール X-Eye の解析手法と可視化，スコーピング DSL の紹介とシステム構成について説明する．4 章は，X-Eye の実装手法について述べ，5 章では X-Eye が行うプロファイリングのオーバーヘッドや，スコーピング DSL の可視化速度に与える影響についておこなった実験の結果を提示する．そして，6 章では本研究のまとめをおこない，今後の課題を考察する．

第 2 章

PGAS 言語 X10

この章では，PGAS 言語の一つである X10 の独自のメモリモデルや構文の特徴を紹介し，分散メモリ型の MPI や共有メモリ型の OpenMP の問題点をどのように解決しているかを述べる．そして，X10 の特徴から発生する問題点を洗い出し，同じ問題点を持つ他の PGAS 言語の関連研究も交えて，既存手法について述べる．

2.1 X10 の特徴

X10 は，IBM が開発している比較的新しい分散並列プログラミング言語である．X10 で書いたプログラムはコンパイル時に Java もしくは C 言語に翻訳されて実行される．Java に翻訳する X10 コンパイラを Managed X10，C 言語に翻訳する X10 コンパイラを Native X10 と呼ぶ．

マルチコア CPU やコンピュータが相互接続された分散並列環境におけるプログラミングでは MPI や OpenMP が主に使われてきたが，プログラムの複雑化や局所性の問題があり，そのような環境を生かしたプログラミングは容易ではない．PGAS(Partitioned Global Address Space)[5] と呼ばれるメモリモデルはそのような問題点を解決するものとして期待されており，X10 は PGAS を採用した言語の一つである．PGAS 言語には X10 の他に，クレイが開発している Chapel[6, 7]，佐藤 三久 筑波大学教授らによる XcalableMP[8, 9]，Robert Numrich らによる Co-array Fortran[10]，UC Berkeley による Unified Parallel C[11, 12] など他にも多数あるが，PGAS 言語では，個々の言語によって定義されているメモリモデルがわずかに異なるため，ここでは X10 における PGAS メモリモデルについて述べる．

X10 のメモリモデルは，図 2.1 のように単一のアドレス空間を複数の “プレース” と呼ばれる空間に区切る．複数のコンピュータが相互接続されている分散環境では，一般的に 1 ノードが 1 プレースになることが多い．データはどれかのプレースに属し，移動はしないが，アクティビティ (スレッドと同等) がプレース間を移動してデータを取得することができる．その際，アクティビティローカルな変数は移動先のプレースに暗黙的にコピーされるため，MPI でデータ送受信時に必要だった明示的な送受信コードの記述は必要なく，プログラムが複雑化しにくい．また，単一の共有アドレス空間を形成していながら，複数のプレースに区切られ，

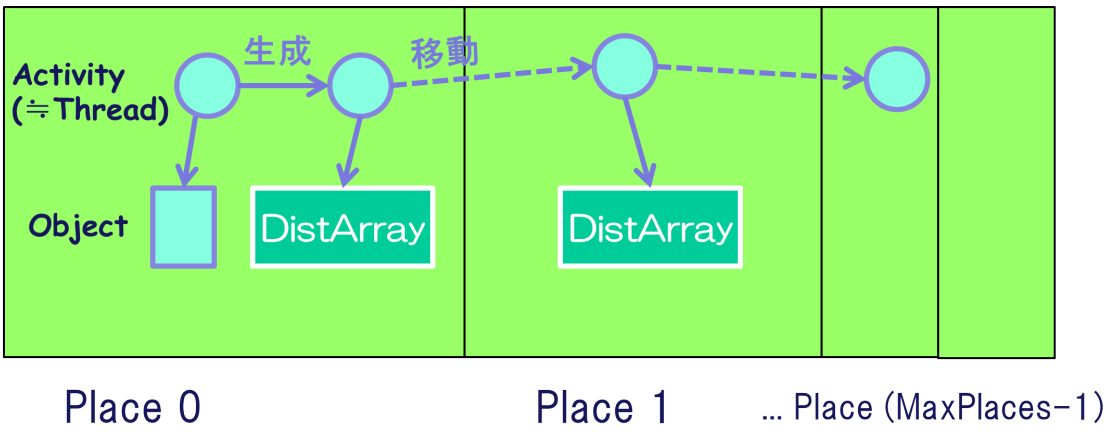

図 2.1. X10 のメモリモデル

データは特定のプレース内に保存されるため，データの局所性が高く，データアクセスのオーバーヘッドが少ない．

次に，X10 の構文を紹介する．

2.1.1　async 構文

```
1  async { S; }
```

図 2.2. async 構文

async 構文はアクティビティ（軽量スレッド）を生成する構文である．図 2.2 のように，async と書き，括弧内に処理 S を書くだけで，新たなアクティビティを生成し非同期的に処理 S を実行する．

2.1.2　at 構文

```
1  at( here.next() ) { S; }
```

図 2.3. at 構文

at 構文はアクティビティが他のプレースに移動して資源を利用するというプログラミングモデルを抽象化したもので，アクティビティの移動を表す．at に引数として渡したプレースへ移動し，処理を実行する．図 2.3 では，現在いるプレースの次のプレースに移動して処理 S を実行した後、現在のプレースに戻ってくる一連の処理を表す．

2.1.3 finish 構文

```
1 finish {
2   async { S1; }
3   async { S2; }
4 }
```

図 2.4. finish 構文

finish 構文は粗粒度なバリア同期処理を抽象化した構文である．図 2.4 では，二つのアクティビティを生成してそれぞれが S1，S2 を非同期的に実行し，両方とも実行が終わるまで，finish 構文を実行しているアクティビティは待つという処理を表している．finish 構文は強力な同期のため，同期に参加している，すなわち finish 構文の中で生成されたアクティビティの一つでも処理が長引けば，finish 構文を実行しているアクティビティはそれだけ長く待たされることになる．

2.1.4 clocked 構文

```
1 val c = Clock.make();
2
3 async clocked (c) { S1; }
4 async clocked (c) { S2; }
```

図 2.5. clocked 構文

clocked 構文は細粒度な同期処理を抽象化した構文である．clock 変数を宣言して初期化し，アクティビティを生成する際に clocked キーワードとともに引数として clock 変数を渡す．同じ clock 変数を持っているアクティビティ同士で同期をすることができる．図 2.5 では，clock 変数 c を渡すことで同期しあうアクティビティを決定している．

同期しあうアクティビティを決定した後，それらのアクティビティ間で細かい同期をする構文が用意されている．そのうちよく使われる Clock.advanceAll() は，Clock.advanceAll() の前の処理が終わると，自分以外のすべての同期に参加しているアクティビティに通知をし，それらのアクティビティからも通知が来るまでブロッキングする．同期に参加しているすべてのアクティビティから通知が来たことが確認されると次の処理へ進む．例えば図 2.6 においては，S1 と P1 が実行された後に S2 と P2 が実行される．なお，アクティビティは非同期に実行を行うので，S1 と P1，S2 と P2 の実行順序は問わない．なお，Clock.advanceAll() 以外にも，同期に参加しているアクティビティのうちのどれか一つにのみ通知を行う Clock.advance() や，

```
1  val c = Clock.make();
2
3  async clocked (c) {
4    S1;
5    Clock.advanceAll();
6    S2;
7  }
8
9  async clocked (c) {
10   P1;
11   Clock.advanceAll();
12   P2;
13 }
```

図 2.6. clocked 構文の細かい同期の取り方

通知のみを行いブロッキングをしない Clock.resumeAll() や Clock.resume() がある．

2.1.5　分散配列

これまで説明した構文とともに，分散配列は X10 プログラミングを容易にする重要な仕組みの一つである．一般的な分散並列処理では，資源を均等に分割してそれぞれのノードに分配する処理を複雑なコードで明示的に書く必要があった．X10 の分散配列の仕組みにおいては，資源を自動で均等に分割し均等に各プレースに分配することができる．分配された資源はプレースローカルなデータとなり，各プレースのデータは，そのプレースにアクティビティが移動することでアクセスすることができる．図 2.1 において DistArray が分散配列にあたる．

2.2　X10 の問題点

前章で述べたとおり，X10 には様々な抽象化された構文があり，これらの構文によって，アクティビティ生成やバリア同期などの処理を簡単にプログラミングすることができるようになり，分散並列処理のコーディングを容易にしている．しかし，プログラミングが容易である一方，データ送受信やアクティビティの挙動を抽象化しているため，プログラムの様々な挙動をユーザが把握しにくく、デバッグやチューニングが困難になりやすいという問題がある．問題点を次に列挙し，その後に詳細を述べる．

- at 構文を実行した際，移動先のプレースと暗黙的に送受信されるデータの量と送受信のタイミング，そして送受信の経路を把握しにくい
- finish 構文でバリア同期をする際，バリア同期に参加しているアクティビティの数を把握しにくい，もしくは静的に数を決定することができないことがある

2.2.1 暗黙的なデータ送受信

X10 では，MPI のようなメッセージパッシングモデルにおけるデータ送受信に関連するプログラムの不雑性の問題を解決するために，データ送受信を暗黙的に行う．OpenMP のデータ局所性の問題点を解決するためにプレースローカル，要するにプレースに紐づけられたデータは送受信されず，アクティビティが移動して参照しに行くが，アクティビティローカルなデータは，アクティビティが他のプレースに移動した際に送受信のコードを明示的に記述しなくても，移動先プレースにコピーされる．

```
1  //a is initialized to array that has 10 elements with value 1
2  val a = new Rail[Long](10, 1);
3
4  at (here.next()) { // place 1
5    at (here.next()) { // place 2
6      doSomeTasks();
7      Console.OUT.println(a(0));
8    }
9  }
```

図 2.7. at 構文による暗黙的なデータ送受信のソースコード例

図 2.7 では，定数 a は 10 個の要素をもつ配列として宣言され，すべての要素が 1 で初期化されている．そして，at 構文でまず place 1 に移動し，その後再度 at 構文で place 2 へ移動し，place 2 の中で配列 a のうちの最初の要素だけが参照されている．このソースコード中の暗黙的なデータ送受信に関して，以下のような問題を提起することができる．

データ送受信量

place 2 では a の最初の要素のみ参照されているが，配列 a のすべての要素が転送されるのか? 最初の要素のみが転送されるのか?

データ送受信の経路

転送されるデータは，place 1 に転送されてから再度 place 2 に転送されるのか? 直接 place 2 に転送されるのか?

チューニングをしてパフォーマンスを上げようと考える際，どれくらいの量のデータが送受信されるのかは致命的な問題である．暗黙的なデータ送受信量を把握することができれば，無駄な送受信によるオーバーヘッドを削減でき，チューニングで大きな成果を得られる可能性がある．そして，データがどの経路をたどって転送されるのかもチューニングにおいて重要な要素である．図 2.7 では，place 1 を経由して転送されてしまう場合，直接 place 2 に転送される場合と比べて送受信回数が 1 回多くなってしまい，オーバーヘッドの原因となる．

2.2.2　抽象化されたアクティビティの挙動

X10 では，アクティビティに関連する挙動が高度に抽象化され，通常では複雑なコードを書かないとできない処理も容易に書くことができる．アクティビティ生成の async 構文，粗粒度バリア同期の finish 構文，細粒度バリア同期の clocked 構文などがその例として挙げられる．プログラミングは容易になる一方，それぞれのプレースで何個のアクティビティが生成されているのか，アクティビティはどのプレースを移動して各プレースでどれだけの時間実行したのか，finish 構文による同期ではどのアクティビティどうしが互いに同期しているのか，また finish 構文内で何個のアクティビティを生成しているのか等，ユーザがチューニングする際に重要な情報を知ることが難しい．finish 構文を例に，重要情報を知ることが難しい場合を以下に説明する．

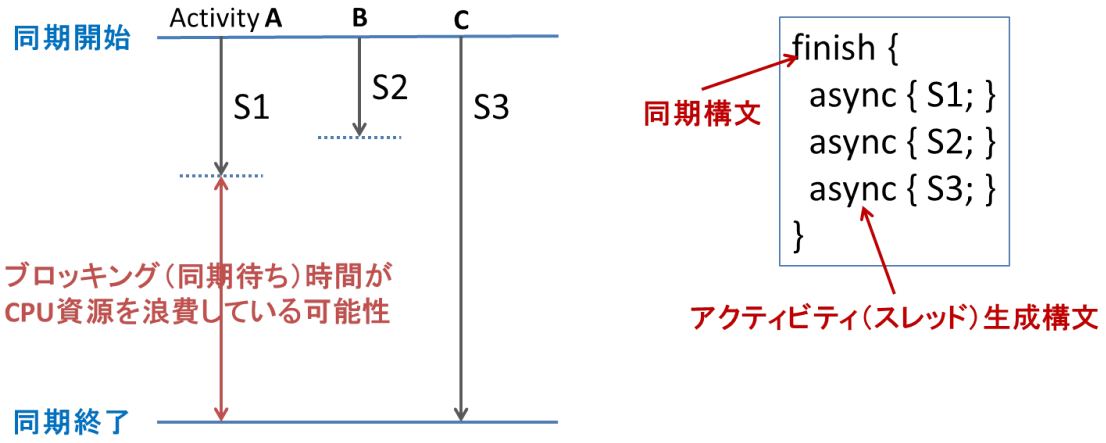

図 2.8. 同期待ちによって CPU 資源が浪費される可能性のある例

図 2.8 は，同期待ちの時間が長くなってしまい，その間 CPU に新たなタスク割り当てがない場合に CPU 資源が浪費されてしまう例を示している．図 2.8 の右側はその例を表すソースコードで，粗粒度同期処理をおこなう finish 構文内に 3 つのアクティビティが生成されている．それぞれのアクティビティを順に A, B, C と名付けた場合，アクティビティ C の処理時間が長くなっている．このような場合，アクティビティ C が同期待ち時間のオーバーヘッドの原因となっている．X10 では同期処理やアクティビティ生成が容易なため、このようなことが起こりやすく，同期制御内にたくさんアクティビティが生成されている場合，オーバーヘッドの原因となっているアクティビティを探し出すのに多くの手間がかかる．

次に，同期制御内に参加しているアクティビティを特定することが困難な例を紹介する．図 2.9 において，Scheduler クラスはアブストラクトクラスであり，doTask メソッドのみ実装されており，finish 構文の中で run メソッドが呼ばれている．前にも述べたように finish 構文は強力なブロッキングを伴うため，run メソッドが何個のアクティビティを生成しているかを知ることがパフォーマンスチューニング上，非常に重要である．しかし，run メソッドはアブス

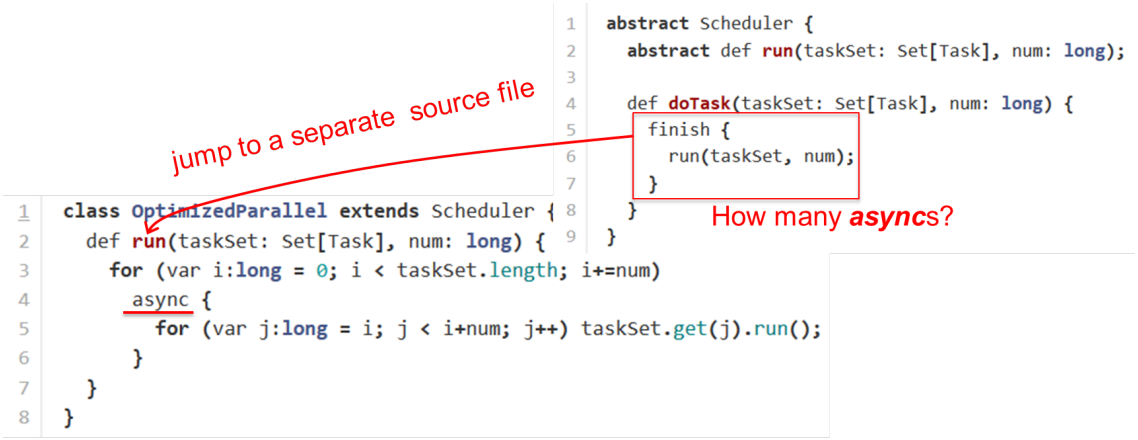

図 2.9. 同期に参加しているアクティビティ個数の決定に手間がかかる例

トラクトメソッドとなっており，サブクラスで実装されることとなる．OptimizedParallel クラスは Scheduler クラスを継承ししたサブクラスであり，run メソッドを実装し，その中で async 構文を使ってアクティビティを生成して非同期処理を行っている．この実装では，変数 num で指定された数のタスク毎にアクティビティを一つ生成している．X10 では，1 クラス 1 ファイルが原則となっており，この場合アクティビティが async 構文によって何個作られているかを知るためにはサブクラスを探し出し，さらにその実装まで読んで num の値を特定・推定しなければならない．これはユーザーにとって非常に手間のかかる作業となる場合がある．

問題はこれだけではない．Scheduler クラスと OptimizedParallel クラスの単体のソースコードをそれぞれ図 2.10(a) と図 2.10(d) に，さらに二つの Scheduler クラスのサブクラスである Sequential クラスと Parallel クラスのソースコードを図 2.10(b) と図 2.10(c) に示す．図 2.11 は，これらのクラスの関係性を表したクラス図である．

Sequential クラスは，タスクを表す変数 taskSet を逐次処理するクラスで，アクティビティは全く生成しない．Parallel クラスはすべてのタスクを非同期的，すなわち並列的に処理するクラスで，タスク一つにつきアクティビティが一つ生成される．OptimizedParallel は先も述べたように，変数 num によってアクティビティ数が変わるため，最適なアクティビティ数でタスクを処理することができるクラスである．

このような継承関係がある状況で，Scheduler クラス型の doTask メソッドを呼び出しても，ダイナミックディスパッチングのために，どのサブクラスの run メソッドが呼び出されるのか静的に決定することは困難である．したがって，アクティビティが何個生成されるのかが実行されるまで判別することができず，ユーザーがパフォーマンスチューニングを行うのはますます困難となる．

```
abstract Scheduler {
  abstract def run(taskSet: Set[Task], num: long);

  def doTask(taskSet: Set[Task], num: long) {
    finish {
      run(taskSet, num);
    }
  }
}
```

(a) Scheduler クラス

```
class Sequential extends Scheduler {
  def run(taskSet: Set[Task], num: long) {
    //run the tasks in sequential
    for (task in taskSet) task.run();
  }
}
```

(b) Sequential クラス

```
class Parallel extends Scheduler {
  def run(taskSet: Set[Task], num: long) {
    //run the tasks in parallel
    for (task in taskSet) async { task.run(); }
  }
}
```

(c) Parallel クラス

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

(d) OptimizedParallel クラス

図 2.10. 同期に参加しているアクティビティ個数の決定が静的には困難な例

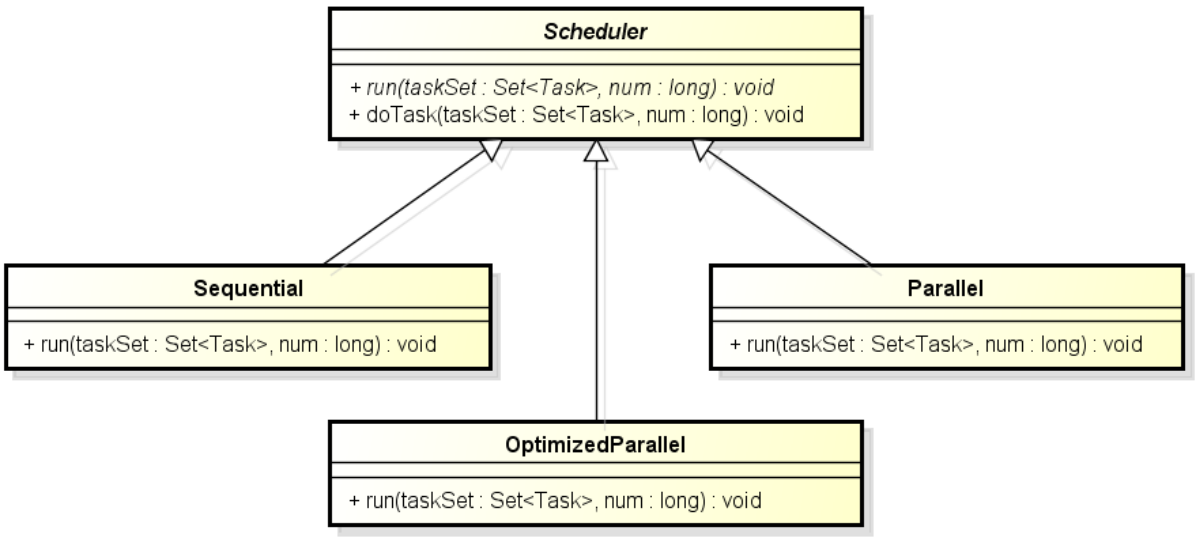

図 2.11. 一連のクラスをまとめたクラス図

2.3　関連研究と既存手法

X10 のプロファイリングツールに関する研究は数少ないものの，他の PGAS 言語や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，数多くの研究が行われてきた．本節では，数少ない X10 用のプロファイリングツールはもちろん，その他の並列分散向け言語のプロファイリングに関する研究を紹介する．

Prospector[13] は，並列プログラム向けのプロファイラである．プログラムの並列化は人が行うとミスが起きやすいが，今日ではコンパイラがコンパイル時に並列化を行うようにもなった．Prospector は，最新のコンパイルでも見逃した，更なる並列化の余地があるループ構造を見つけ出すことができる．だが，我々のツールはループを対象にしているのではなく，X10 固有の構文による暗黙的な動作を対象にして解析を行い，オーバーヘッドを見つけることを支援している．

Vampir[14] や TAU[15] 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 PGAS のプログラムに使うこともできる．PGAS 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．また，関数の開始・終了や，通信先ノードやメッセージサイズを知ることができる．しかし，本研究で開発したツールは X10 にフォーカスしてプロファイリングを行うために，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収集している．また，通信に関しても Vampir や TAU は通信先ノードのホスト名を表示するが，X10 ではメモリモデルが抽象化されプレース番号が割り振られているため，そのような抽象化に対応できておらずプレース番号を知ることが難しい．その上，毎回の通信メッセージのサイズを表示するが，我々のツールでは，通信

の送受信サイズのみならず，送受信された変数と送受信された経路も表示することができる．

Nathan らによる研究 [16] では，PGAS 言語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ことと，対象の PGAS 言語が Global Arrays[17] であるという点で，我々の開発したプロファイリングツールとは異なる．

Ti-trend-prof[18] は，PGAS 言語向けの自動でパフォーマンスデバッグをおこなうツールであり，プログラムの計算複雑性を測定する trend-prof[19] を拡張して開発された．Ti-tred-prof は，PGAS 言語の一つである Titanium[20] を対象に，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，それによりオーバーヘッドを改善する．しかし，プロファイル対象言語が我々のツールとは異なることや，我々のツールではリモートアクセスのオーバーヘッドではなく暗黙的なデータ送受信によるオーバーヘッドを探知する上，プログラム全体の挙動を追跡するという点で異なる．

次に，X10 用のプロファイリングツールの先行研究を紹介する．XAnalyzer[21] は，オーバーヘッドが大きくなるコードのパターンを 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．もし検知された場合は，修正すべきコードも提示する．XAnalyzer は我々のツールと同じく高いレベルでコードを解析するが，我々のツールは決まったパターンではなく，X10 固有のイベントをすべて追跡していくため，XAnalyzer よりも基礎的なレベルでの解析をおこなっているうえ，可視化機能も実装されている．

第 3 章

X-Eye

本研究では，前章で述べた X10 の問題点を解決し，X10 におけるデバッグやチューニングを支援するため，X10 の暗黙的な挙動を可視化するツール X-Eye を提案する．X-Eye は，まずプロファイラ部で X10 のプログラムをコンパイルする時に追跡コードを挿入することで，実行時にイベント情報を収集し，解析結果をファイルに出力する．そして，解析結果を可視化して視覚的にユーザに提示する．これによって，暗黙的なデータ送受信や，抽象化されたアクティビティの挙動をユーザーが把握することを支援することができる．さらに，ビジュアライザにはイベントデータのフィルタリングを行うスコーピング DSL も実装されており，DSL のコードをユーザが書くことによって，プロファイリング結果のデータを柔軟に絞ることができ，可視化範囲を制限することができる．なお，Xnalyzer はプロファイラとビジュアライザの二つの要素から構成され，プロファイリングはプロファイラ，可視化とイベントフィルタリングはビジュアライザに実装されている機能である．

本章では，X-Eye における X10 プログラムの解析手法，そしてスコーピング機能の詳細と X-Eye のシステム構成についてについて述べる．

3.1　解析手法

X-Eye では，プロファイラ部で X10 プログラムのプロファイリングを行ってから，その結果をもとにビジュアライザ部で可視化を行う．まずプロファイラが行うプロファイリングの手法について説明する．

X-Eye のプロファイラは X10 コンパイラを拡張したものである．まず X10 のプログラムに対しコンパイル時に，挙動を追跡するコードを X10 固有の抽象化構文に対して挿入することで，実行時にイベント情報を収集する．なお，暗黙的なデータ送受信に関しては，コンパイル時の追跡コード挿入と並行して静的解析によって行う．したがって、X-Eye のプロファイラでは，X10 プログラムをコンパイルする際に追跡コードの挿入と，データ送受信に関する静的解析の二つを行う．まず追跡コードの挿入手法について説明し，次に追跡コードの挿入において個々のアクティビティを命名する Activity Identifier，そして最後にデータ送受信の静的解析手法について説明する．

3.1.1　イベントに対する追跡コード挿入と実行時の情報収集

X10 プログラムをコンパイルする際，X-Eye は async，at，finish，atomic 等の X10 固有の抽象化構文及び，Clock.advanceAll()，Clock.advance()，Clock.resumeAll()，Clock.resume() などの X10 固有の細粒度同期メソッドの実行開始と実行終了の位置に追跡コードを挿入する．図 3.1 のコードに対して，X-Eye を使って追跡コードを挿入した実例が図 3.2 である．図 3.1 と図 3.2 で，赤色の下線が引かれているコードが追跡対象イベントで，青色の下線が引かれているコードが対象イベントに対して挿入された追跡コードである．図のとおり，async の開始には CreateEvent クラスの activityCreated メソッド，終了には TerminateEvent の activityTerminated メソッド，finish の開始と終了には SyncStartEvent の syncStart メソッド，SyncEndEvent クラスの syncEnd メソッドが挿入されるなど，追跡する対象によって異なる追跡コードが挿入されるうえ，同じイベントでも開始と終了で異なる追跡コードが挿入される．個々の追跡コードがどのイベントに対してどの位置に追加されるかをまとめた表が表 3.1 である．しかし，実行時に収集する情報はどの追跡コードも以下の 1.〜8. であり，同じである．これらの情報を収集し終わった後，収集した情報を加工して最終的には，Activity クラス，Move クラス，Finish クラス，Clock クラス，Atomic クラスのインスタンスとしてまとめられ，ログファイルとして出力される．ログファイルのデータ構造は JSON 形式となっており，図 3.3 は実際のログファイルを，全体の構造が見えるよう途中を省略して示したものである．ビジュアライザによる可視化は，ログファイルからこれらのクラスのインスタンスを生成し，行われる．

```
1  public class Test {
2    public static def main(Rail[String]) {
3      for (p in Place.places()) at(p) async {
4        finish {
5          doTask(p.id());
6        }
7      }
8    }
9
10   public static def doTask(id:Long) {
11     ...
12   }
13 }
```

図 3.1. 追跡コードが埋め込まれる前のソースコード

1. 追跡コードを実行したタイムスタンプ
2. 追跡コードを実行しているアクティビティの番号 (アクティビティ番号については後述する Activity Identifier による)
3. 追跡コードを実行している親アクティビティの番号
4. 現在プレースの番号

```
1  public class Test {
2    public static def main(Rail[String]) {
3      finish {
4        val tracer0 = new EventTracer(); //メインアクティビティ用のトレーサー
5        CreateEvent.activityCreated(tracer0, "Test", "main", 2L, 8L);
6        finish {
7          BeforeMoveEvent.activityMoving(tracer0, "Test", "main", 3L, 7L, 0L,
8                                         TransferListBuilder.create().build());
9          for (p in Place.places()) at(p) {
10           MoveEvent.activityMoved(tracer0, "Test", "main", 3L, 7L, 0L,
11                                   TransferListBuilder.create().build());
12
13           val tracer1 = new EventTracer(tracer0);
14           async {
15             CreateEvent.activityCreated(tracer0, "Test", "main", 3L, 7L);
16
17             SyncStartEvent.syncStart(tracer1, "Test", "main", 4L, 6L);
18             finish {
19               doTask(p.id(), tracer1);
20             }
21             SyncEndEvent.syncEnd(tracer1, "Test", "main", 4L, 6L);
22           }
23           TerminateEvent.activityTerminated(tracer1, "Test", "main", 3L, 7L);
24
25         BeforeMoveEvent.activityMoving(tracer0, "Test", "main", 3L, 7L, 0L,
26                                        TransferListBuilder.create().build());
27         }
28         MoveEvent.activityMoved(tracer0, "Test", "main", 3L, 7L, 0L,
29                                 TransferListBuilder.create().build());
30       }
31       TerminateEvent.activityTerminated(tracer0, "Test", "main", 2L, 8L);
32     }
33     EventTracer.printResultByPlace();
34   }
35
36   public static def doTask(id:Long, tracer:EventTracer) {
37     ...
38   }
39 }
```

図 3.2. 追跡コードが埋め込まれたソースコード

図 3.3. プロファイリング後に出力される解析結果のログファイルの実例 (全体像が見えるよう途中を省略)

5. 実行しているクラス名
6. 実行しているメソッド名
7. 追跡対象イベントのプログラム中での開始行番号
8. 追跡対象イベントのプログラム中での終了行番号

3.1.2　Activity Identifier

X10 ではアクティビティを一意に特定する機能を提供していないため，我々で Activity Identifier (すべてのアクティビティを命名し，実行しているアクティビティを一意に特定する仕組み) を実装し，プログラム実行時もすべてのアクティビティを区別して解析できるようにした．アクティビティの命名方法はプレース番号とそのプレース内で生成された順番によって構成される．例えば，X10 プログラムが実行されて一番最初に実行される main メソッドのメインアクティビティは Activity 0-0，5 番プレース内で 16 番目に生成されたアクティビティであれば，Activity 5-16 と命名される．

これらのアクティビティの区別は，アクティビティ一つにつき一つの専用トレーサーを生成し，各追跡コードにトレーサーを渡すことで実現している．このトレーサーが，例えば Activity 5-16 というように，自身のアクティビティ番号を保持している．図 3.2 における，緑色の下線で引かれた tracer がそれにあたる．アクティビティが一つ生成されるごとに，トレーサーの変

表 3.1. 各追跡コードが使われる状況

追跡コードのクラス	追跡コードが挿入される場所
CreateEvent	async でアクティビティが生成される直後
TerminateEvent	async で生成したアクティビティが終了する直後
BeforeMoveEvent	at で移動する直前
MoveEvent	at で移動した直後
FinishStartEvent	finish で同期を開始する直前
FinishEndEvent	finish の同期が終了する直後
AdvanceAllStartEvent	Clock.advanceAll() が実行される直前
AdvanceAllEndEvent	Clock.advanceAll() の実行が終了する直後
AdvanceStartEvent	Clock.advance() が実行される直前
AdvanceEndEvent	Clock.advance() の実行が終了する直後
ResumeAllStartEvent	Clock.resumeAll() が実行される直前
ResumeAllEndEvent	Clock.resumeAll() の実行が終了する直後
ResumeStartEvent	Clock.resume() が実行される直前
ResumeEndEvent	Clock.resume() の実行が終了する直後

表 3.2. ログ出力のために最終的なクラスにまとめる際に使用される追跡コードの組み合わせ

最終的なクラス名	インスタンス生成の際に加工・使用される追跡コードのクラス
Activity	CreateEvent と TerminateEvent
Move	BeforeMoveEvent と MoveEvent
Finish	FinishStartEvent と FinishEndEvent
Clock	AdvanceAllStartEvent，AdvanceAllEndEvent，AdvanceStartEvent，AdvanceEndEvent，ResumeAllStartEvent，ResumeAllEndEvent，ResumeStartEvent，ResumeEndEvent
Finish	FinishStartEvent と FinishEndEvent

数名は名前の衝突を避けるために，tracer1，tracer2，... と番号が一つずつ大きくなっていく．すべての追跡コードの引数に，対応する tracer を渡すことで，同一メソッド内においてはどの追跡コードも自分自身が実行しているアクティビティを一意に特定することができる．

3.1.3 メソッドの引数拡張による，Activity Identifier のメソッドへの引き渡し

前述した，追跡コードを X10 固有のイベントに対して挿入して実行時にイベント情報を収集することで，“各プレースにアクティビティが何個生成されたのか” や，“finish による同期待ちの時間はどれくらいだったのか”，“アクティビティはどう移動して各プレースでどれくらいの時間実行したのか” などの基本的な情報を知ることができるようになるが，図 2.9 や図

2.10 で紹介した，“同期に何個のアクティビティに参加しているのか”，“どのアクティビティが参加しているのか”，という静的には決定することが難しかった，比較的高度な問題を解決できない．その原因は，実行元の main メソッド以外のメソッドにトレーサーを渡すことができず，他メソッドがどのアクティビティ内で実行されているのか特定することができないことである．その例を図 3.4 に示す．main メソッド中に async 構文によるアクティビティ生成イベントがある．それに対応するトレーサーとして変数 tracer4 が async の直前で生成されている．したがって，この async 内で発生するイベントに対してはこの tracer4 を追跡コードに渡すことで，どのアクティビティで発生しているイベントなのかを特定する．今回は，このアクティビティ内で，青い枠で囲ってあるのところで，run() メソッドの呼び出しが起きている run() メソッドの定義を見ると，メソッド内で finish イベントが発生している．この finish イベントに対して変数 tracer4 を渡すことができないため，run() メソッド内で発生しているイベントは自分がどのアクティビティで実行しているのかを特定することができない．

これを解決するため，X-Eye のプロファイラではコンパイル時に単に追跡コードを挿入するだけではなく，追跡対象となっているすべてのクラスのすべてのメソッド定義を拡張し，引数を一つ増やしてトレーサーを渡せるようにした．そして，メソッドが呼び出されているコードもトレーサーを引数に追加して呼び出すコードに変更する．図 3.1 と図 3.2 の紫色の下線の doTask() メソッドでメソッド引数の拡張の実例を示している．図 3.2 では，図 3.1 では引数が一つしかなかった doTask() メソッドの引数を一つ増やしてトレーサーを渡せるようになっており，doTask() メソッドを呼び出しているところでもトレーサーが引数に一つ追加されて呼び出されるよう変更されている．これによって，同期に何個のアクティビティが参加し，どのアクティビティが参加しているのか，といった静的に決めることが困難だったアクティビティの挙動も把握できるようになった．

3.1.4　コンパイル時のデータ送受信に関する静的解析

次に，分散ノード間で行われる暗黙的なデータ送受信を静的に解析する手法を説明する．我々は，X10 の処理系の実装プログラムを解析した結果，データ送受信に関する以下のような結果を得た．

- at 構文で移動した先のプレースで参照されているオブジェクトのみが送受信される
- オブジェクトが配列であり，配列の一部のみが参照されていても配列全体が送受信される
- オブジェクトが参照される前に他のプレースを経由した場合は，オブジェクトもそのプレースを経由して送受信される（複数回の送受信が行われることになる）
- 送受信量（バイト数）はそのプリミティブ型のバイト数に等しい．int 型であれば 4 バイト，long 型であれば 8 バイト，float 型であれば 4 バイト，double 型であれば 8 バイト，配列であれば各要素のバイト数の合計，オブジェクト型であればそれらのフィールドのバイト数の合計に等しい

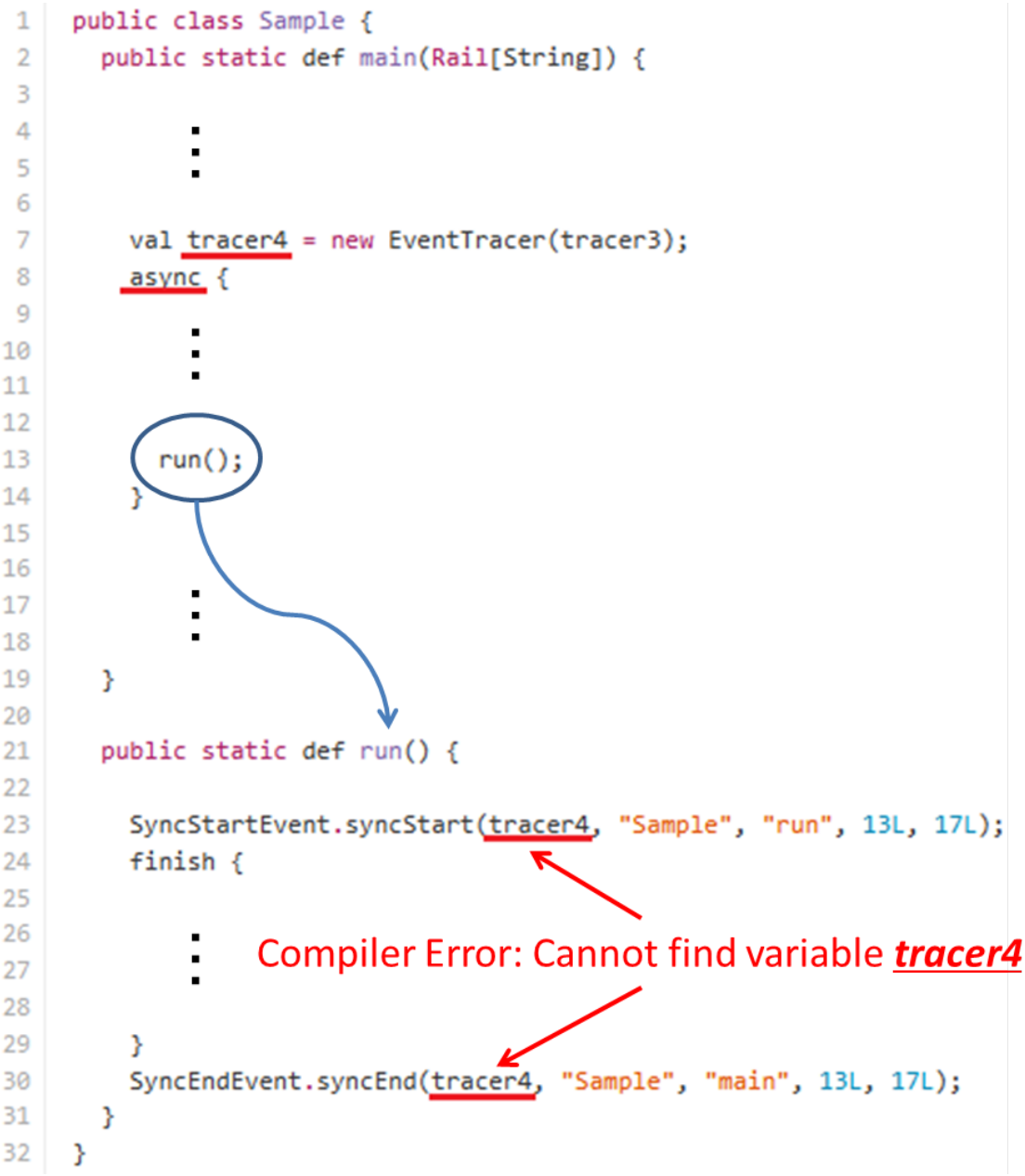

図 3.4. トレーサーが渡せず，イベントがどのアクティビティ内で実行されているのか特定できない例

```
val a:int = 5;

at( here.next() ) {
  Console.OUT.println(a);
}
```

図 3.5. X10 runtime を解析してそれぞれの型の送受信サイズを調べるためのベンチマーク

これらの結果は X10 コンパイラの中の X10 処理系の実装を記述してあるの C++ ソースコードを読み，ソースコード中に printf() メソッドを埋め込んで送受信が発生した際のサイズを出力するようにし，我々が作成した何種類もの送受信ベンチマークを適用することによって解析した．図 3.5 は int 型の送受信サイズを調べる際に使用したベンチマークである．変数 a が送受信される変数である．他の型に関してはこの変数 a の型を変更して調べた．

これらの解析結果をもとに，X-Eye のプロファイラはコンパイル時に静的にプログラムを解析し，送受信されるオブジェクト名，バイト数，送信元プレース，目的地プレース，経由するプレースを見積もる．見積もるために X-Eye のプロファイラは二度コンパイルを行う．一度めのコンパイルではフィールドやローカル変数などのあらゆる変数の名称や型，サイズなどの詳細情報を取得して HashMap として保存する．なお，ここでいう変数とは定数も含む．そして二度めのコンパイル時に，X10 固有のイベントに対して追跡コードを埋め込むが，その際に at 構文中にデータ送受信を行って参照されていると推定される変数を見つけた場合，一度めのコンパイルで取得したデータ (HashMap) から，該当する変数の詳細情報（型やサイズ）を取り出す．その変数は送受信される変数だと推定されているわけなので，at 構文に対して追跡コードを挿入するときに，その変数の詳細データが追跡コード中に String 型の引数として埋め込まれる．図 3.6 は送受信変数が静的に解析され，その詳細情報が追跡コードに埋め込まれるソースコード例である．送受信の詳細情報は最後の引数に埋め込まれており，図 3.6 では，8 バイトの Long 型のオブジェクト p が送受信されることを表している．したがって，X-Eye のプロファイラは，二度コンパイルをおこなって，データ送受信の情報も含めた追跡コードを挿入する手法をとっている．そしてプログラムを実行し，実行終了とともにイベント情報が収集されてログファイルとして吐き出されるという流れになる．

```
MoveEvent.activityMoved(tracer0, "KMeansDist", "main", 79L, 99L, 8L,
                                    TransferListBuilder.create().add("p", "Long", 8L).build());
```

図 3.6. 静的に解析されたデータ送受信情報が追跡コードに埋め込まれる例

3.2　可視化

次に，プロファイリング結果を可視化するビジュアライザについて説明する．ビジュアライザでは，動的プロファイリングが終了した後に，解析結果として出力されるログファイルを読み込むことで可視化を行う．表示するグラフには，「主に移動イベントを見るグラフ」と「主に同期イベントを見るグラフ」の二種類があるが，どちらも移動イベントだけ，同期イベントだけ表示するわけではなく，あらゆるイベントが表示される．しかし，前者は移動イベントを見るのに適しており，後者は同期イベントを見るのに適しているため，そのように呼ぶことにする．両者のグラフとも x 軸は経過時間のナノ秒だが，前者のグラフの y 軸はプレース番号，後者のグラフの y 軸はアクティビティ番号である．なお，グラフのイベント表示は，at 構文による移動イベント (move) で，暗黙的なデータ送受信を伴わない移動であれば M，暗黙的な送受

信を伴う移動であれば D，finish 構文による同期イベントであれば F，atomic 構文によるアトミックイベント（排他計算）は A，clocked 構文による同期イベントはそれ自体だけでは意味を持たないので，実際に細粒度の同期処理を実行する Clock.advanceAll() を上下二本の左向き矢印，Clock.advance() を←，Clock.resumeAll() を上下二本の右向き矢印，Clock.resume() を→として表現している．また，アクティビティの発生を表す始点は右向きの三角形型矢印，アクティビティの終了を表す終点は主に移動イベントを見るグラフでは円となっており，主に同期イベントを見るグラフでは縦長の長方形となっている．なお，グラフの複雑化を避けるためにデフォルトでは，主に移動を見るグラフはデフォルトは移動をおこなったアクティビティのみを，主に同期を見るグラフでは同期を行ったアクティビティのみを初期表示の対象としている．あらゆるアクティビティを表示したい場合には，“Display all activities” のチェックボックスにチェックを入れることで表示することができる．

図 3.7 は，我々がサンプルとして作成したソースコードと，それを X-Eye のプロファイラでプロファイリングした結果をビジュアライザで可視化したものであり，主に移動イベントを見るグラフを表示している．このソースコードは，メインアクティビティがすべてのプレースを回ってそのプレースに子アクティビティを作成していき，作成された子アクティビティは自分より 2 個分先のプレースに移動するというシンプルな処理をしている．本論文のスペースに貼付できる画像サイズの関係でややグラフ内の文字が見えにくが，アクティビティの生成と終了を表す始点と終点，移動イベントの M や D，そして finish 構文による同期が発生したこと表す F も一つ確認することができる．イベントの文字の上にマウスをあてるとそのイベントの開始時間，終了時間，実行したアクティビティ番号，実行したプレース，クラス名，メソッド名，対応するソースコードの行番号などの詳細情報が表示される．グラフの線の上にマウスをあてると，そのアクティビティの詳細情報が表示される．

次に，X10 のコンパイラに同梱されている実際のソースコードを対象に X-Eye でプロファイリングした結果のグラフを紹介し，そのグラフからわかるソースコードの挙動を説明する．なお，以下で使用するソースコードは，X10 のバージョン 2.4.1 に同梱されているソースコードである．

図 3.8 は 8 クイーン問題を一般化した N クイーン問題を計算するプログラム NQueensPar.x10 を我々の開発した X-Eye のプロファイラでプロファイリングした結果をビジュアライザで可視化したグラフである．なお，コマンドライン引数で初期値 11 を与えて計算を行った．ビジュアライザの y 軸では，イベントのアイコンを切れ目なく表示するために，最小値を実際に存在する最少のプレース番号より 1 つ小さく，最大値を実際に存在する最大のプレース番号より 1 つ大きくなるように設定している．図 3.8 は，0〜3 までの 4 つプレースで実行したことを表している．

図 3.8 では，すべてのアクティビティがプレース 0 に生成され，一切移動をしていないことがわかる．したがって NQueensPar.x10 では複数プレースを生成することはメモリを無駄に確保してしまうことになる可能性があり，チューニングの際に参考になる情報だといえる．

次に K 平均法の計算を行うプログラム KMeansDist.x10 について説明する．図 3.9 は，KMeansDist.x10 について，計算を行う初期条件を大幅に小さくしたプログラムをプロファイ

```
1  public class Sample {
2      public static def main(args: Rail[String]) {
3          System.sleep(25); //dealing with tasks
4          finish { //synchronization
5                  System.sleep(50); //dealing with tasks
6                  for (t in Place.places()) at(t) { //move
7                          System.sleep(50); //dealing with tasks
8                          async { //create activity
9                              val value = 10;
10                             System.sleep(100); //dealing with tasks
11
12                             at(here.next().next()) { //move
13                                     Console.OUT.println(value); //refer to "value"
14                                     System.sleep(100); //dealing with tasks
15                             }
16                         }
17                         System.sleep(50); //dealing with tasks
18                 }
19             }
20     }
21 }
```

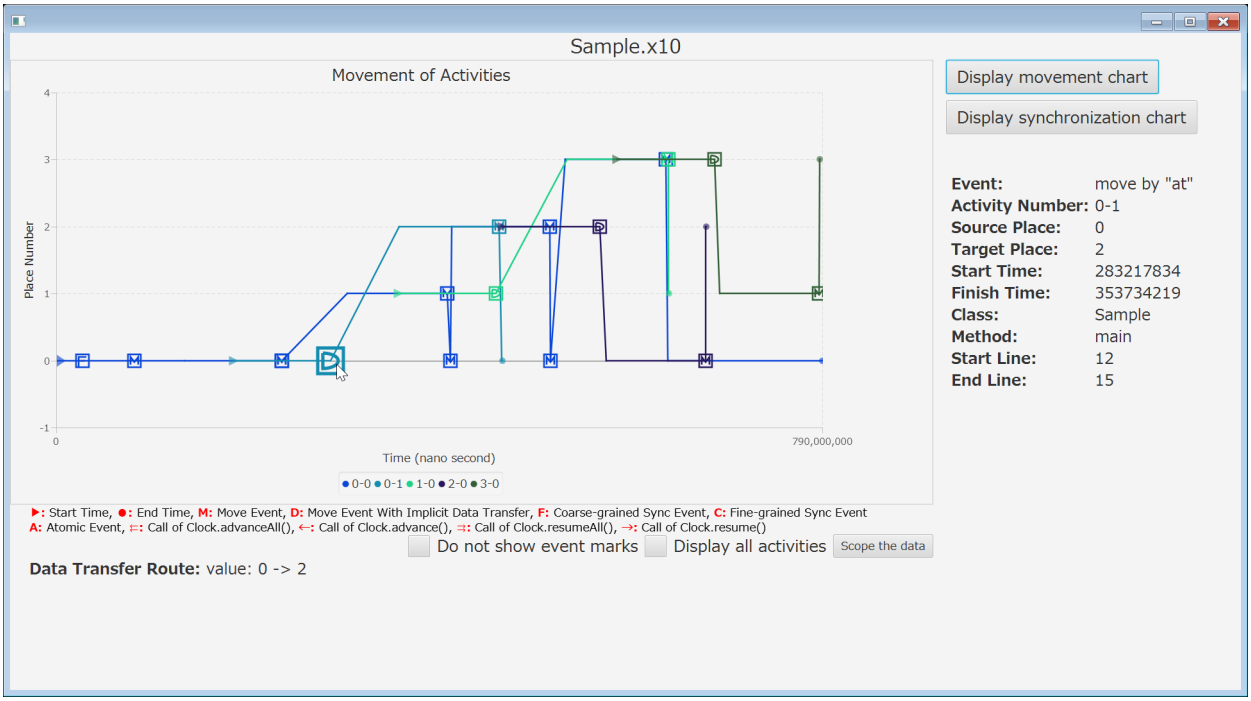

図 3.7. オリジナルで作成したサンプルソースコードとその挙動の可視化のようす

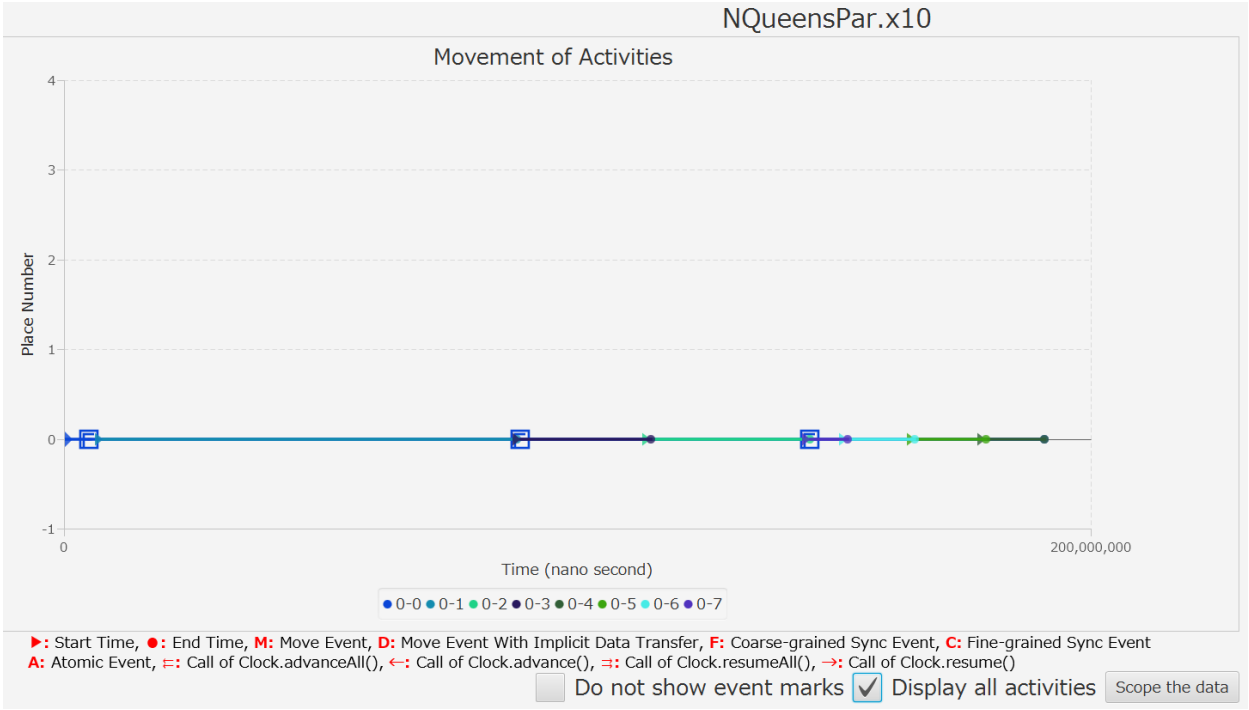

図 3.8. NQueensPar.x10 のソースコードを解析した結果を可視化したグラフ

リングした結果を可視化した，主に移動イベントを見るグラフと，その一部を拡大したグラフである．今回は，KMeansDist.x10 において定数 POINTS=10 に，定数 ITERATIONS=3 に変更してプロファイリングしている．なお，拡大はグラフをダブルクリックすることで行うことができる．このグラフから，KMeansDist.x10 は非常に多くの移動イベントが発生していることがわかる．それだけではなく，赤枠を拡大した下のグラフの緑枠で囲ってあるところを見ると，他のプレースに移動する線がないにも関わらず D や M などの移動イベントが複数存在することが確認できる．この図のとおり，マウスをあてて右側に詳細情報を表示しているが，Source Place（移動元プレース）と Target Place（移動先プレース）が同じ 0 になっていることがわかる．このメインアクティビティであるアクティビティ 0-0 はマウスであてたイベントを皮切りに，その後もデータ送受信を伴う移動である D やデータ送受信を伴わない移動である M を現在いるプレースに対しておこなっている．アクティビティが現在いるプレースに対して移動をおこなうことは意味をもたない上，オーバーヘッドが発生する．特に，マウスをあてたところのような，暗黙的なデータ送受信を伴う移動イベントはオーバーヘッドが大きくなる可能性があり，無駄な移動を削減することがチューニングで非常に重要になる．したがって，自分自身のいるプレースへの移動が発生していることはユーザーにとって非常に有益な情報である．これによって，前章の 2.2.1 の暗黙的なデータ送受信を把握しにくい問題を解決している．

同じ K 平均法プログラム KMeansDist.x10 の同期イベントに関するグラフを図 3.10 に示した．透明度が低くなっている部分は同期によってブロッキング（同期待ち）を行っているこ

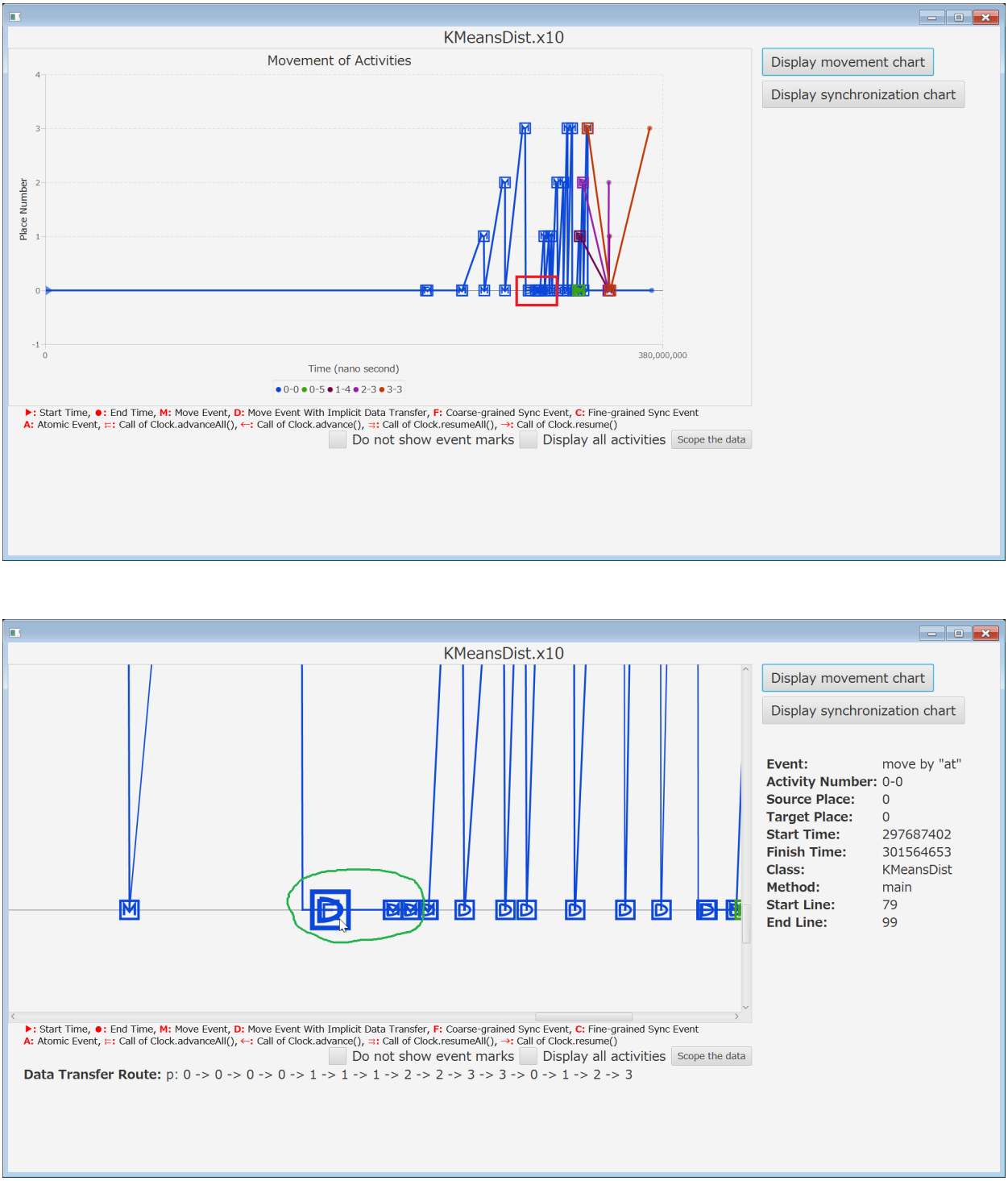

図 3.9. 初期条件を小さくした KMeansDist.x10 のソースコードを解析した結果を可視化した，主に移動イベントを見るグラフと赤枠を拡大表示したグラフ

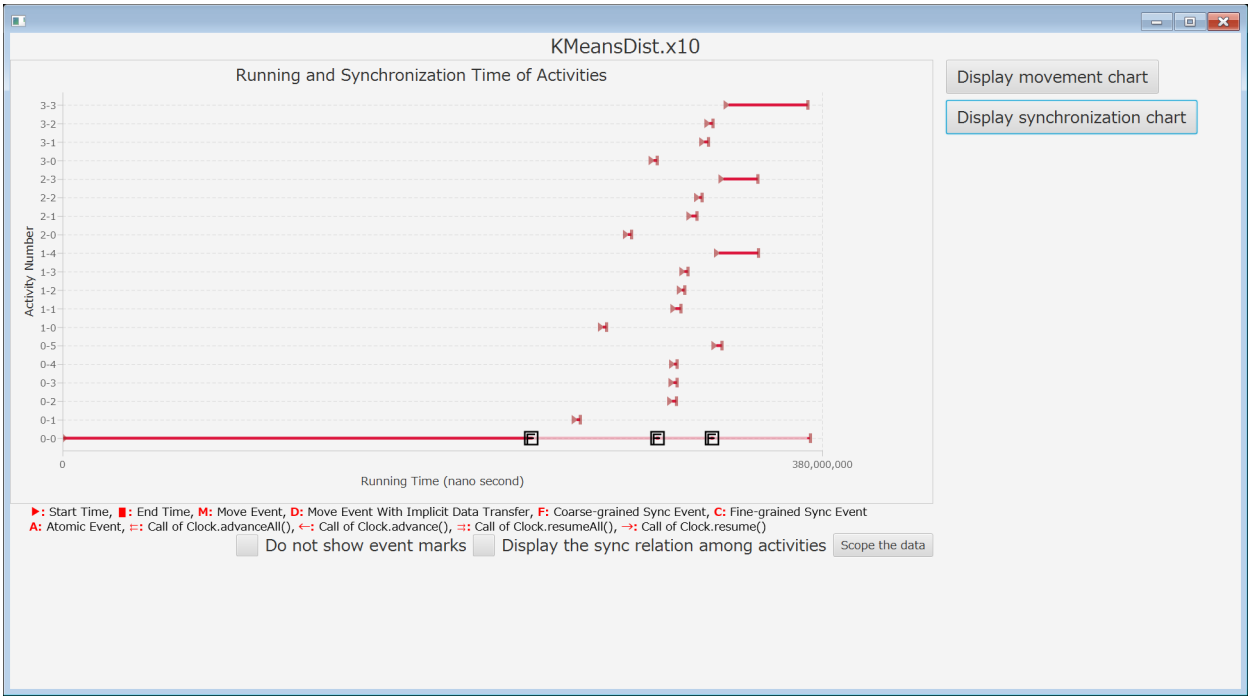

図 3.10. 初期条件を小さくした KMeansDist.x10 のソースコードを解析した結果を可視化した，主に同期イベントを見るグラフ

とを表している．その他の部分は，アクティビティのブロッキング以外の純粋な実行時間である．図 3.10 では，メインアクティビティであるアクティビティ 0-0 が三回に渡って finish 構文を実行してブロッキングしていることがわかる．

次に，図 3.11 では，先ほどの図 3.10 で，どのアクティビティがどの同期に参加しているのか，つまりどの finish 構文内でどのアクティビティが生成されたのかを表している．“Display the sync relation among activities” にチェックを入れることで同期に関するアクティビティの関係性を見ることができる．この図では，1 回目の finish 構文の実行では，アクティビティ 0-1，1-0，2-0，3-0 がその同期に参加しており，同期の実行元アクティビティ 0-0 はこれらのアクティビティの実行が終了するまでブロッキングしていることが読み取れる．2 回目と 3 回目の finish 構文による同期も同じようなことが読み取れる．したがって，このグラフでは，前章の 2.2.2 で紹介した，図 2.9 や図 2.10 などの，抽象化されたアクティビティの挙動を把握することが困難な例を解決している．

3.3　スコーピング DSL によるイベントフィルタリング

前セクションでは，可視化によって様々な X10 の挙動を見ることができ，前章で紹介した X10 の問題点を解決し，ユーザーのチューニングを助けることができることを説明した．しかし，プロセッサがメニーコアになるとともに X10 が行う処理量は増え、イベントの発生量が増えていく．その場合，プロファイリングで収集したイベントデータがあまりにも膨大にな

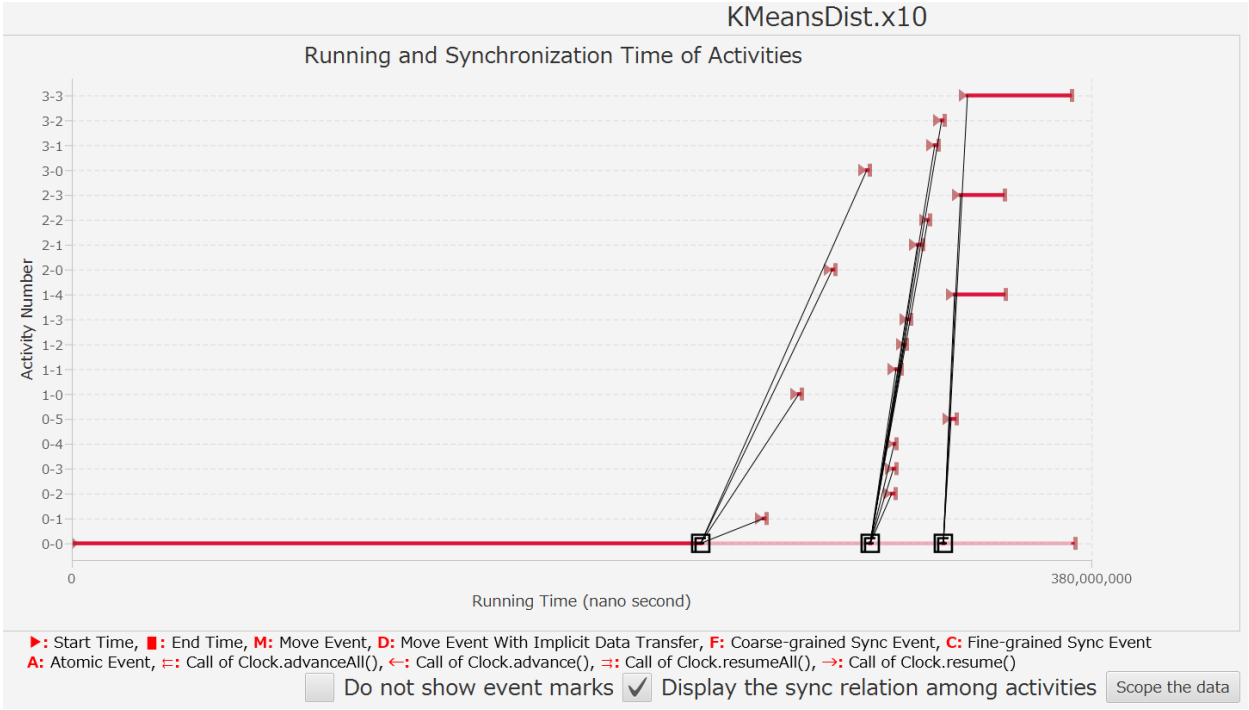

図 3.11. 初期条件を小さくした KMeansDist.x10 のソースコードを解析した結果を可視化した，主に同期イベントを見るグラフにおいて，同期の関係性を表示

り，グラフが複雑化しすぎて読み取れない場合もある．X-Eye のビジュアライザでは，可視化するだけでなく，データをユーザーがフィルタリングすることができるスコーピング DSL も実装している．スコーピング DSL は，最新の Java8 で導入されたラムダ式 [22, 23] を使ったリスト操作ライブラリである Stream API ライクに実装されており，Stream API と同じように書くことでフィルタリングすることができる．また，フィルタリング条件としてイベントの実行時情報を使って記述することでフィルタリングをおこなう．

まず，先ほどの KMeansDist.x10 のソースコードで初期条件をやや大きくすることで，グラフがやや複雑化してしまい読み取りにくくなる例を紹介する．

図 3.12 は，KMeansDist.x10 の初期条件を，定数 POINTS=25，定数 ITERATIONS=5 と，先ほどよりもやや大きくして X-Eye でプロファイリングした結果を可視化したグラフである．移動イベントが非常に多く発生しており，グラフの線やイベント表示が重なって，拡大しなければほとんど見えなくなっている．グラフを拡大していけば見えるようになる可能性はあるが，初期条件をさらに大きくすれば更なる拡大が必要となり，根本的な解決法とはなりえない．

さらに，移動イベントが発生しているアクティビティだけでなく，移動しなかったアクティビティも含めて表示すると，図 3.13 のように，さらに複雑なグラフとなる．

このように，大規模分散並列処理でイベントが大量に生じたときのために，我々は可視化するイベントデータの範囲をユーザーが指定してフィルタリングすることができるスコーピング機能をビジュアライザに実装した．スコーピング機能では，我々の開発したスコーピン

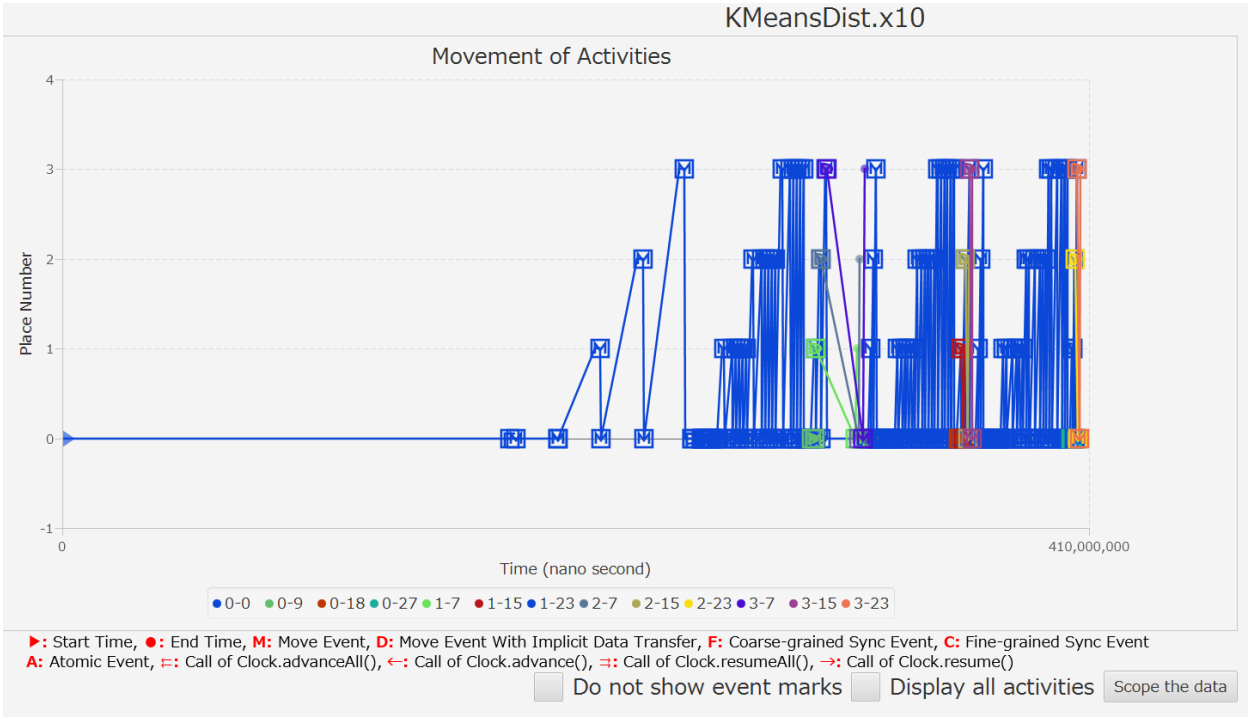

図 3.12. 初期条件をやや大きくした KMeansDist.x10 のソースコードを解析した結果を可視化した，主に移動イベントを見るグラフ

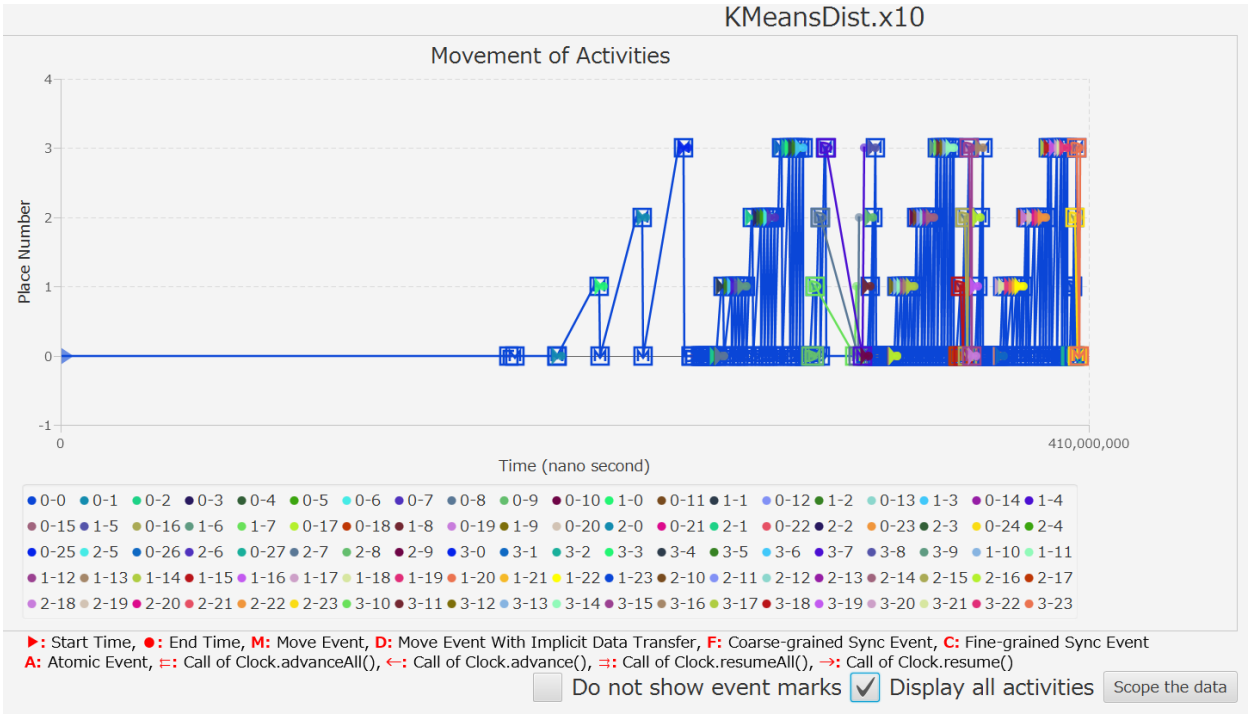

図 3.13. 図 3.12 において，移動しないグラフも表示した結果

グ DSL を用いて記述することでフィルタリングの範囲を指定できる．スコーピング DSL は，Java 8 において登場した Stream API を参考に実装されており，**ストリーム**を生成し，それに対して filter() などのメソッドでフィルタリングをおこなう．フィルター条件は，filter() の引数内にラムダ式の形で記述することができる．図 3.14 はフィルタリングをおこなうシェルで，DSL はその最下段のテキストエリアに記述する．DSL を記述し終えた場合はセミコロン (;）を入力するこで DSL が実行され，まだ DSL を記述し終えておらず途中結果を見て新たなフィルター条件を書きたい場合はドット (.) を入力することでそこまで書いた DSL の途中結果が実行され，そのあとに続けて filter() メソッドを書くことで，逐一途中結果を見ながら対話的にイベントをフィルタリングすることができる．図 3.14 の最上段はコンソールとなっており，DSL に書かれた System.out.println() などの標準出力はそのコンソールに出力されるようになる．中段は履歴となっており，今まで実行した DSL の履歴が表示される．filter() などのフィルタリングをおこなうメソッドの引数内に書く条件として，イベントの開始・終了時間やイベント実行アクティビティ番号など，各イベントの情報を用いる．これらのイベント情報は，プログラム実行時に収集したイベントの実行時情報を加工して，Event.Elem クラスとしてまとめられているため，ラムダ式で記述するフィルター条件の引数は Event.Elem である．表 3.3 は Event.Elem クラスの visible なフィールドである．フィルター条件を書く際には，これらのフィールドを用いて該当するイベントの条件を絞り込む．

表 3.3. フィルター条件の引数としてとる Event.Elem クラスのフィールド

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 →実装は付録参照)	イベントの種類

これから，図 3.12 に対してフィルタリングをおこなう例を紹介する．図 3.12 では，メインアクティビティであるアクティビティ 0-0 が非常に多くの移動イベントを発生していて，それがグラフを複雑にしている大きな要因だとわかる．したがって，メインアクティビティをフィルタリングすれば見やすいグラフになると予想される．図 3.14 は，メインアクティビティをフィルタリングして，メインアクティビティ以外を表示するようにした DSL の記述である．

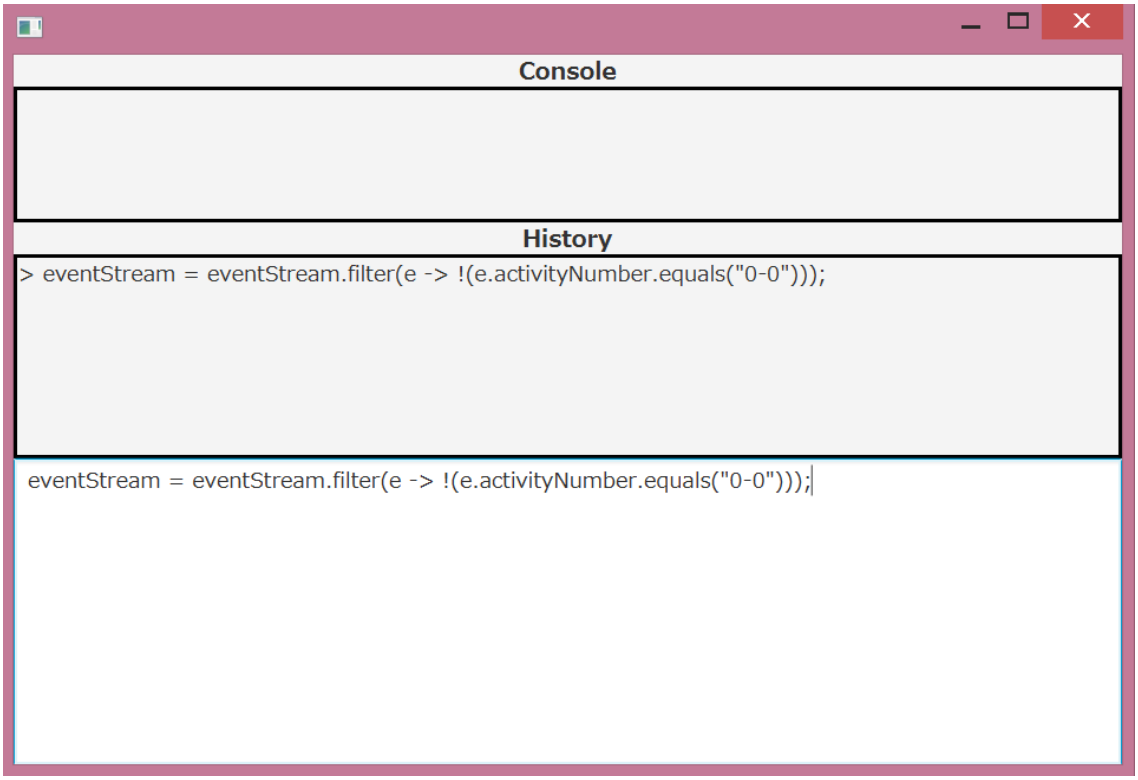

図 3.14. メインアクティビティをフィルタリングする DSL

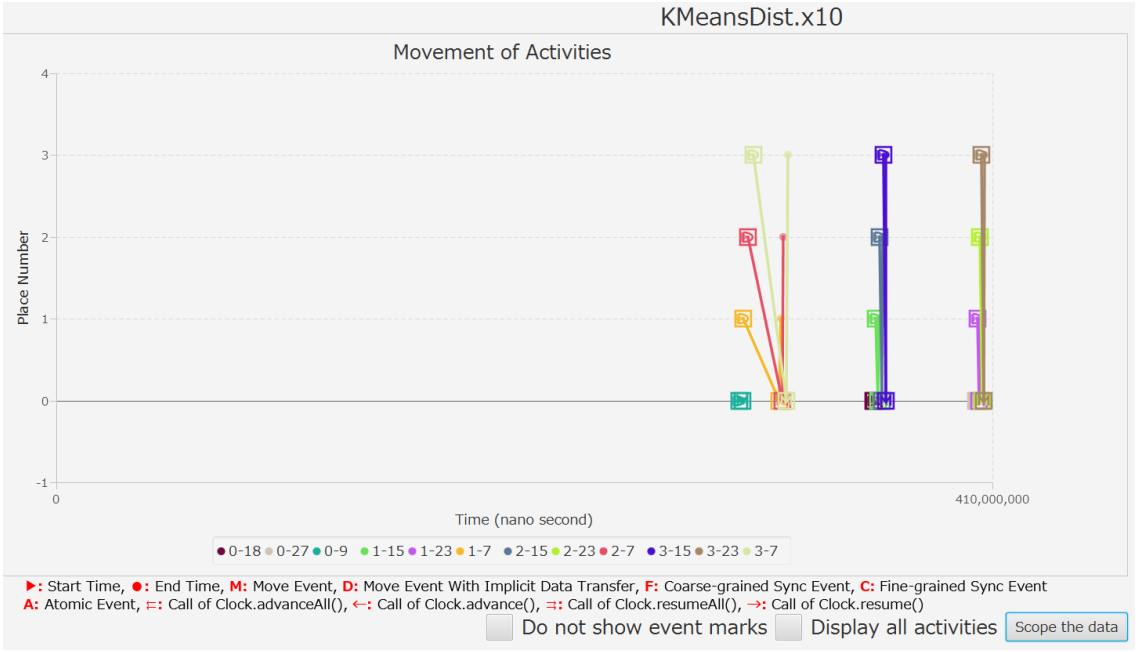

図 3.15. 図 3.14 の DSL 記述によってフィルタリングされた結果のグラフ

条件として，イベントを実行したアクティビティ番号（activityNumber フィールド）を使っている．この DSL を実行すると，図 3.15 のように，メインアクティビティが表示されなくなり，見やすいグラフになったことがわかる．

3.4　システム構成

次に我々のツールのシステム構成について述べる．図 3.16 はシステム構成図である．まず，プロファイルルーチンで X10 プログラムをプロファイリングした後，取得した実行時情報をログファイルに保存する．可視化ルーチンではログファイルを読み込んでイベントデータを生成するとともに，GUI ウィンドウ（GUI 部）を作成する．この時点ではまだグラフは生成されない．ユーザーが GUI ウィンドウ上でクリックした種類のグラフに応じて可視化ルーチンがイベントデータをグラフ作成ルーチンに渡してグラフを取得する．可視化ルーチンは取得したグラフを GUI 部に渡すことでグラフを表示する．

スコーピング機能に関しては，ユーザのクリックに応じて可視化ルーチンがスコーピング用の新たな GUI を生成する．そこでユーザによって書き込まれたスコーピング DSL は DSL 処理ルーチンに渡され，そこでコンパイル・実行され，イベントデータに対してフィルタリングを行う．フィルタリングされたイベントデータは可視化ルーチンを経由してグラフ作成ルーチンに渡され，再度グラフの描画が行われる．

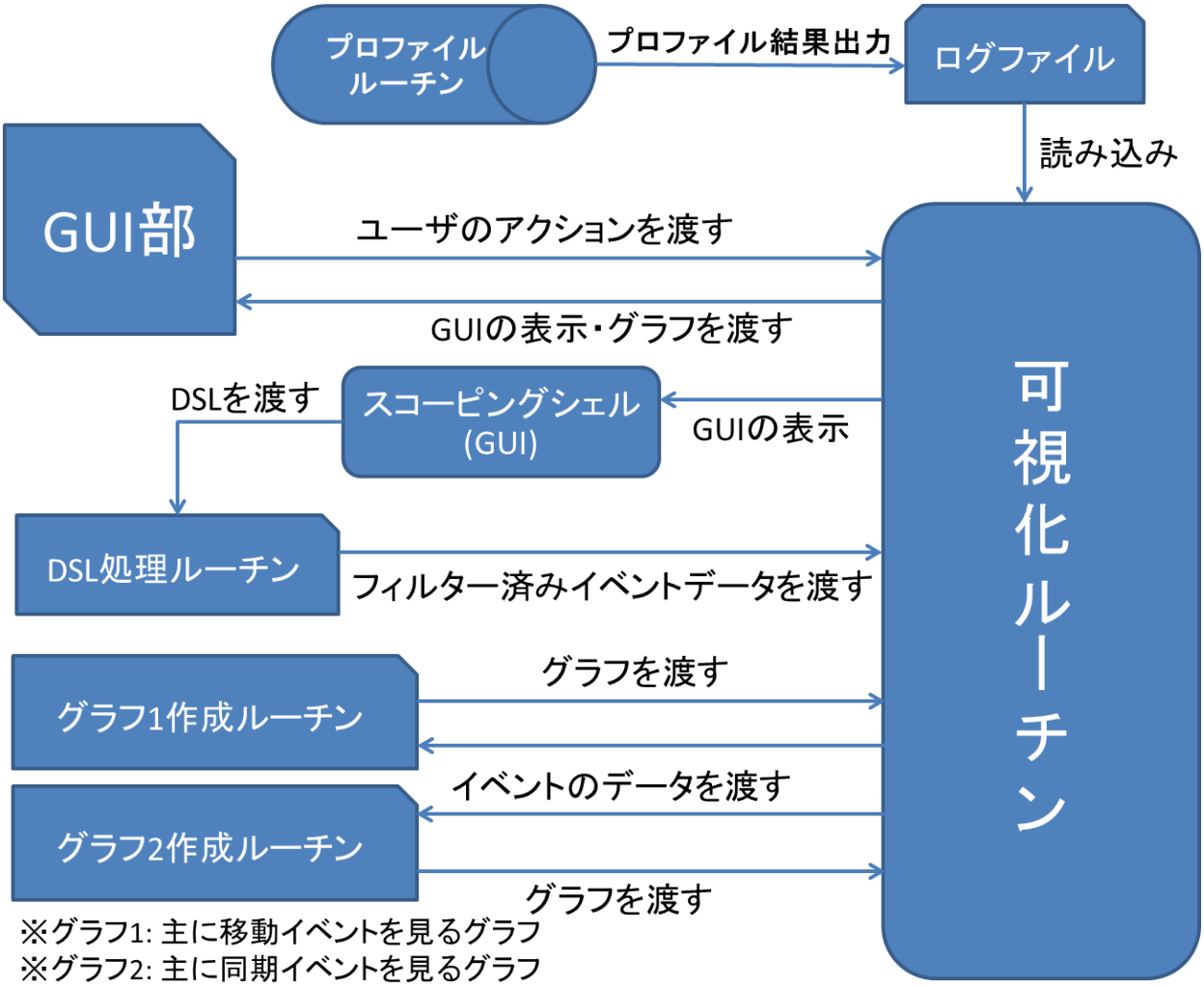

図 3.16. X-Eye のシステム構成図

第 4 章

X10 コンパイラの拡張による実装

我々は，X10 コンパイラをさらに拡張する形でプロファイラを実装し，スコーピング機能は，記述された DSL をオンメモリーでコンパイル・実行する形で実装した．X10 コンパイラは Polyglot[24, 25] と呼ばれる Java コンパイラ拡張フレームワークを拡張して実装されているため，プロファイラの開発では Polyglot を使用した．また，Java の最新 GUI ライブラリである JavaFX[26] を用いてビジュアライザを実装した．

この章では，まず Polyglot について紹介し，その後，プロファイラとスコーピング DSL の実装方法を述べる．

4.1　X10 コンパイラ

Polyglot は，Java 言語の拡張可能なコンパイラフロントエンドフレームワークであり，デザインパターン，特にビジターパターン [27] を用いて一連のコンパイル処理が記述されているため，構文解析過程以降では，自分でビジターを作ることで Polyglot が使われているコンパイラを拡張することができる．また，LALR 法を用いた構文解析を行う PPG が構文解析器生成系として使われている．初代 Polyglot は Java1.4 をサポートしており，それを拡張する形で現在は Java 7 もサポートされ，それだけではなく X10 をはじめとする様々な言語が Polyglot の拡張として開発されている．

Polyglot では，構文解析以降のコンパイルをビジターパターンを使っておこなっている．その解析のようすを表したのが図 4.1 である．型チェック，例外チェックなどのそれぞれ役割を持ったビジターが順番に，構文解析で作られた抽象構文木のルートノードから順番に木を走査していくことで意味解析，そして最適化などの構文解析以降のコンパイル処理を行う方式をとっている．

4.2　X-Eye のための X10 コンパイラの拡張

プロファイラは，X10 コンパイラを拡張して開発した．先にも述べたとおり，X10 は Polyglot を拡張して作られた言語であるため，Polyglot を独自に拡張した．今回，X10 コンパ

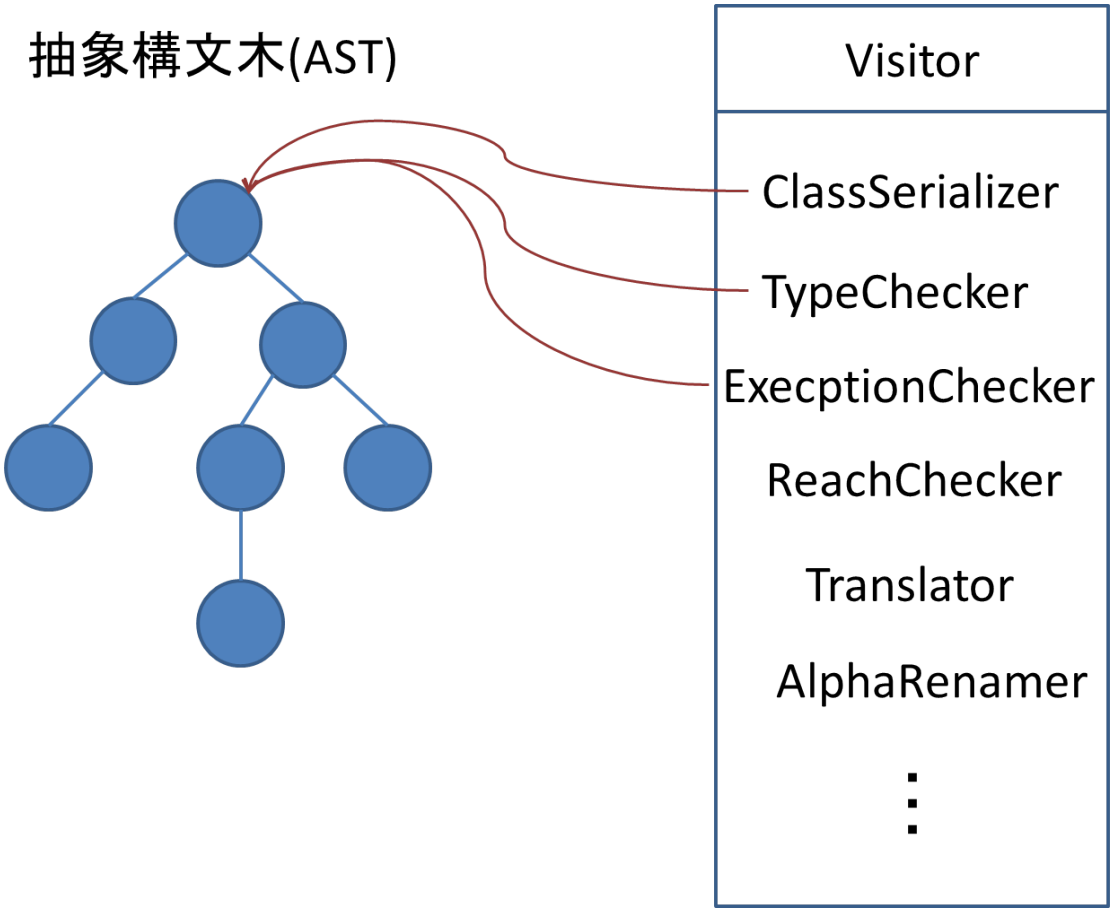

図 4.1. Polyglot における AST 解析

イラを拡張した目的は，コンパイル時に追跡コードを X10 のイベント毎に埋め込むためである．Polyglot ではビジターパターンを使って個々の役割を担うビジターが抽象構文木のノードを訪問していって処理することで構文解析以降のコンパイル処理を行う．したがって，今回の X10 コンパイラの拡張では，X10 固有の構文を認識してその前後に追跡コード埋め込む新たなビジター ProbeAdder を作り，ビジター ProbeAdder が抽象構文木（AST）のノードを訪問していき，対象となるイベント（X10 の固有の構文によるもの）を認識したときに，イベントの種類に合った追跡コードを埋め込むという手法でおこなった．追跡コードを埋め込んだ後，実行することでイベントの実行時情報を収集することでプロファイリングを行う．なお，分散ノード間の暗黙的なデータ送受信の解析はコンパイル時に静的解析によって送受信量を見積もったり送受信されるデータの経路を特定するなどして，その結果を追跡コード中に反映させる．この静的解析も，送受信されるデータを発見するビジター DataTransferResearcher を新たに作り，コンパイル時にそのビジターがプログラムの抽象構文木を走査することで暗黙的なデータ送受信量を見積もる．しかし，我々のツールは 2 回コンパイルを行い，これまでに述べた処理はすべて 2 回目のコンパイル時に行う処理である．1 回目のコンパイルでは，あらゆる変数の名称や型などの情報を取得し保存するビジター ObjectGetter が木を走査する．それ

によって得た変数情報をもとに 2 回目のコンパイル時にデータ送受信量を見積もる．

4.3　スコーピング DSL の実装

スコーピング DSL によるイベントフィルタリングの実装について述べる．図 4.2 はそれを模式化したものである．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る．同じく String 型の文字列として保持してある Scope クラスのソースコードの中にある filter() メソッド内に文字列として埋め込まれる．これにより，フィルタリングをおこなう Scope クラスのソースコードが完成し，その完成したソースコードが文字列として変数 classData によって保持された状態となる．次に，その Scope クラスのソースコードが文字列として保存してある変数 classData から，仮想の Java ファイルである Scope.java をメモリ上に生成する (ディレクトリ上に実際のファイルとして生成されるのではなく，あくまでメモリ上に仮想的に Scope.java が生成される)．文字列から仮想 Java ファイルを生成するために我々が作成したクラス StringJavaFileObject のソースコードとその説明を図 4.3 に示す．そして，この仮想 Java ファイルである Scope.java をコンパイルするために，Java ライブラリから Java コンパイラ (シェルコマンドの javac に相当) を呼んでオンメモリーでコンパイルすることによって Scope.class が生成される．Java ライブラリから Java コンパイラを読んでオンメモリーでコンパイルをおこなうソースコードを図 4.4 に示す．そしてそのあとに Scope.class をロードして Scope クラスのインスタンスを生成し，そのインスタンスから filter() メソッドが呼ばれるようにしてあり，それによってイベントデータがフィルタリングされる．

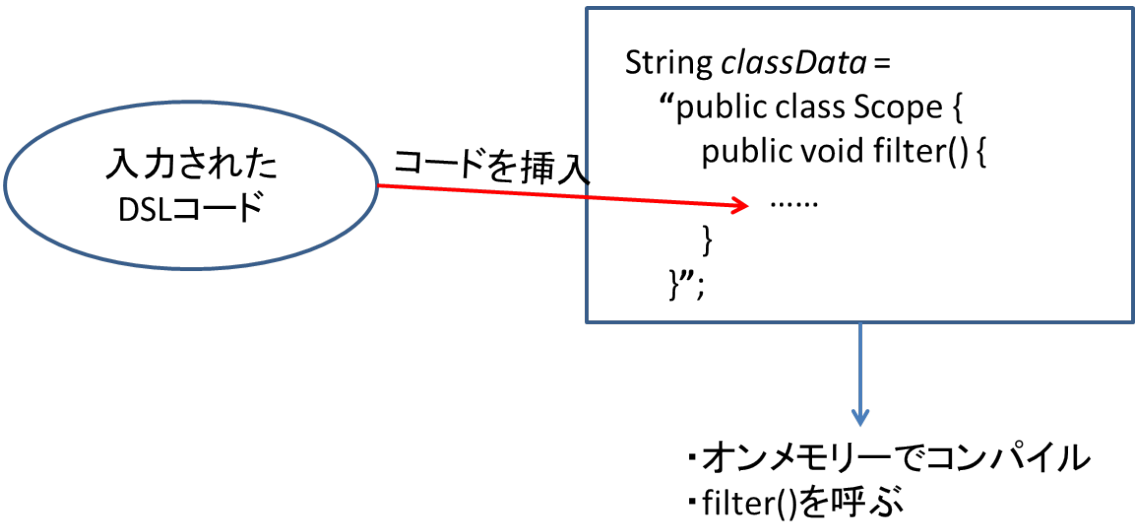

図 4.2. スコーピングの実装

```java
public class StringJavaFileObject extends SimpleJavaFileObject {
  private final String code;

  public StringJavaFileObject(String name, String code) {
      super(URI.create("string:///" + name.replace(".", "/") + Kind.SOURCE.extension), Kind.SOURCE);

      this.code = code ;
  }

  @Override
  public CharSequence getCharContent(boolean ignoreEncodingErrors) throws IOException {
      return code;
  }
}
```

メモリ上に仮想のjavaファイルを生成

生成する仮想javaファイルの中身のソースコード

仮想javaファイルを生成する際に中身のソースコードを取得するメソッド

図 4.3. 文字列から仮想の java ファイルをメモリ上に生成する，自作のクラス

Javaコンパイラの取得

メモリ上に仮想的に
生成されるScope.javaを取得
のための仮想ファイルマネージャ

```java
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

DiagnosticCollector<JavaFileObject> collector = new DiagnosticCollector<JavaFileObject>();

List<String> options = Arrays.asList("-d", "bin");

StandardJavaFileManager fileManager = compiler.getStandardFileManager(collector, null, null);

List<? extends JavaFileObject> fileobjs = Arrays.asList(new StringJavaFileObject("Scope", classData));

JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, collector, options, null, fileobjs);
```

コンパイルエラーのメッセージの保存先

コンパイルオプションの設定

コンパイルを開始

メモリ上に仮想的にScope.javaを生成

図 4.4. メモリ上に生成された仮想ファイル Scope.java をオンメモリーでコンパイルするソースコード

第 5 章

実験

この章では，我々のツールがコンパイル時に挿入する追跡コードで実行時にどれくらいのオーバーヘッドが出るのかを，KMeansDist.x10 と NQueensPar.x10 の二つのサンプルプログラムでおこなった結果を示す．

次に，プロファイル結果を読み込んで可視化する際に，スコーピング DSL を使った場合と使わなかった場合で表示スピードにどのような差が出るのかを実験した結果を示す．

5.1　追跡コードによる実行時情報収集のオーバーヘッド

本ツールのプロファイラでは，コンパイル時に追跡コードをソースコードに挿入することで、実行時に情報を収集する．しかし，追跡コード自身の実行時間が長いと，正しいプロファイル結果が取れなくなる可能性もある．そのため，本節では追跡コードによるオーバーヘッドがどれくらいあるのかを実験した．

なお，実験環境は以下の通りである．

- CPU: Intel Xeon E5-2687W 3.10GHz 8 cores
- RAM: 64GB
- OS: CentOS release 6.2
- X10 コンパイラ及び処理系のバージョン: 2.4.0
- 使用したコンパイルコマンド： x10c
- コンパイルオプション: -O -NO_CHECKS -classpath gson-2.2.4.jar
- 実行オプション： -classpath gson-2.2.4.jar

図 5.1 は，KMeansDist.x10 において，初期条件の定数 POINTS=10 に，定数 ITERATIONS=3 とした図 3.9 や図 3.11 で用いたのと同じプログラムで，プレース数を 1，4，8，... , 16，20 と変更させながら，追跡コードが入っていない場合と追跡コード入っている場合の実行速度の比較実験をおこなった結果である．なお，結果は追跡コードが入っている場合も入っていない場合も，各プレース条件で 10 回ずつ実行し，その平均値を載せている．図 5.2 は，図 5.1 におけるオーバーヘッドをパーセンテージでグラフ化したものである．

プレース数が少ないときには，追跡コードありの場合となしの場合の速度の差が小さく，オーバーヘッドも小さい．しかし，プレース数が多くなるにつれて追跡コードありの場合となしの場合の速度の差が大きく開いていることがわかる．それとともに，プレース数が多いほど，オーバーヘッドが大きくなっている傾向が読み取れる．プレース数が 16 個以降ではオーバーヘッドが 80% 台と高くなっている．図 5.3 は，各プレース数で実行した場合のイベント発生件数を測定したものである．プレース数 1 のときだけイベント数が多いが，プレース数 4 以降は同期の finish イベント意外のイベントは右肩上がりで増えている．これは，KMeansDist.x10 の計算においてループを使用しており，プレース数 1 のときでは何回もループしないと計算が終わらないが，プレース数 4 以降では 1 回のループで計算が終わるためにイベント発生件数がプレース数 4 のときに減っているものと考えられる．

これらの結果より，プレース数が多くなればなるほどオーバーヘッドも大きくなる原因として，移動イベントが挙げられる．我々のツールではアクティビティごとにトレーサーが生成され，その個々のトレーサーが個々のアクティビティ内で終了まで受け渡されていく．そのため，移動が発生したときにはトレーサーも移動先プレースへ送受信される．トレーサーのインスタンスのクラスは様々なフィールドを持っており，サイズが比較的大きい．そのため，プレース数が増えれば増えるほど，様々なプレース間でトレーサーが受け渡される頻度が増えるがためにオーバーヘッドが大きくなっていると考えられる．図 5.3 において，プレース数 1 のときは自分のいるプレースへの移動がおこなわれるため実質的な移動とは異なり，除外するものとするが，プレース 4 以降では移動イベントの発生件数も右肩上がりで増えている．

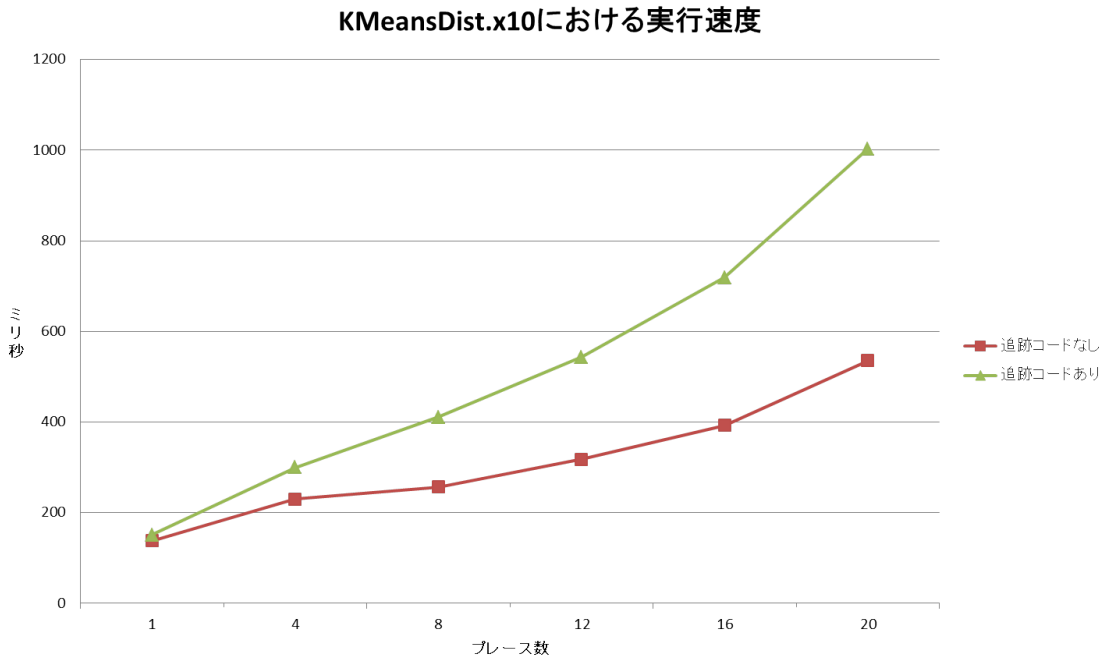

図 5.1. KMeansDist.x10 における追跡コードの有無による実行速度の違い

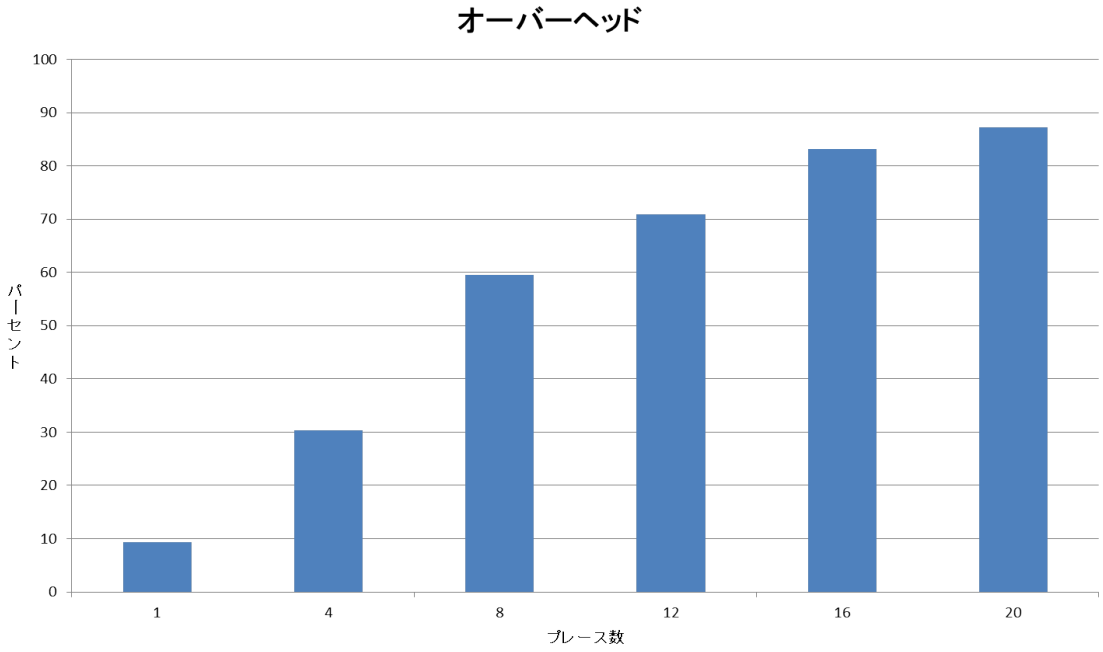

図 5.2. KMeansDist.x10 における追跡コードによるオーバーヘッド

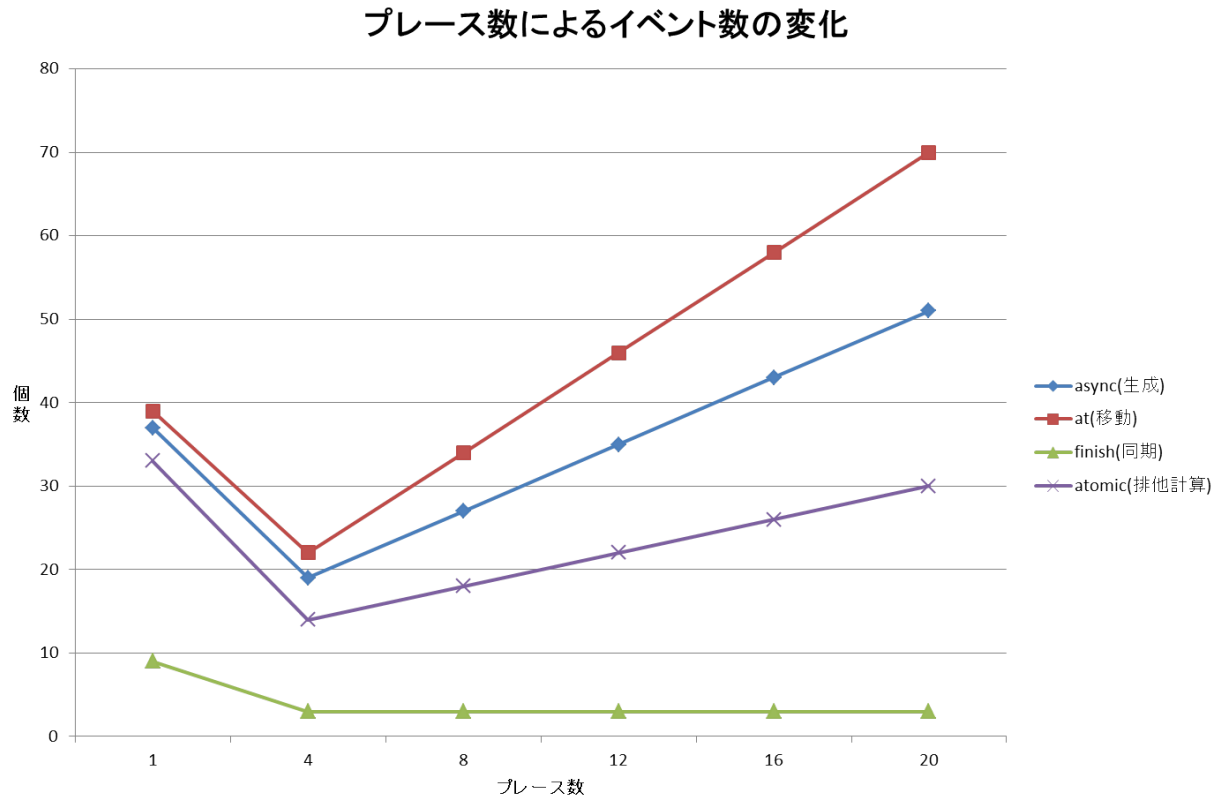

図 5.3. KMeansDist.x10 におけるプレース数によるイベント発生件数の変化

次に，N クイーン問題を計算するプログラム NQueensPar.x10 において，我々のプロファイラを使った実験を示す．なお，NQueensPar.x10 では移動イベントがまったく無いため，プレース数を実験条件としても無意味であるため，実験条件とはせず，コマンドライン引数に与える N の値（つまりコマンドライン引数として 8 を与えれば 8 クイーン問題となる）を実験条件とした．

図 5.4 は，N の値を 8，9，10，11 と変更していった際の，追跡コードがある場合とない場合の実行速度を示したグラフである．N の値が増えれば増えるほど問題サイズが大きくなるため，どちらの場合も実行速度が大きくなっている．このグラフをもとに，図 5.5 に追跡コードによるオーバーヘッドの割合を示した．N の値が 8 と 11 の場合は 10% 台，9 と 10 の場合も 20% 台で，移動イベントがない場合にはトレーサーインスタンスの送受信がないために，オーバーヘッドが少ないことがわかった．なお，N=11 のときにオーバーヘッドが少なくなっているのは，イベント数が増え具合よりも個々のイベントが処理を行う時間の増え具合が大きかったことによるものと考えられる．それと同じく，N=9，10 のオーバーヘッドが N=8 のときよりも大きくなっているのは，イベント数の増え具合よりも個々のイベントの実行時間の増え具合が小さかったことにより，追跡コードそのものの実行時間がオーバーヘッドとして表れたものだと考えられる．

以上の KMeansDist.x10 と NQueensPar.x10 の実験結果より，移動イベントの有無がオーバーヘッドに影響をあたえやすく，移動イベントがない場合はオーバーヘッドがじゅうぶんに小さいことがわかった．

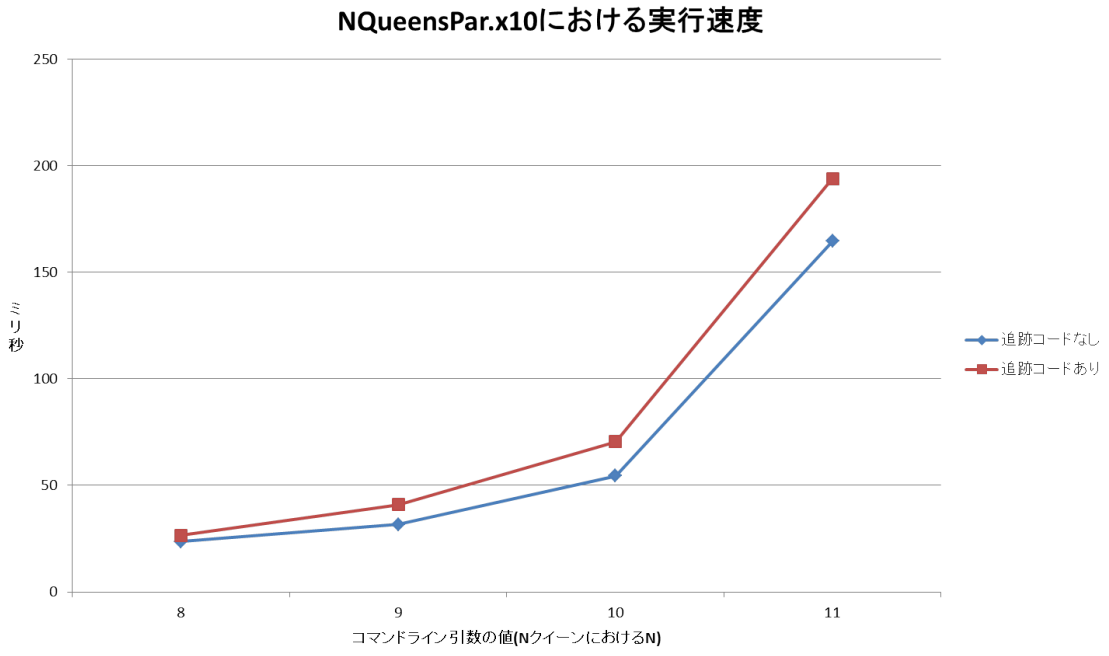

図 5.4. NQueensPar.x10 における追跡コードの有無による実行速度の違い

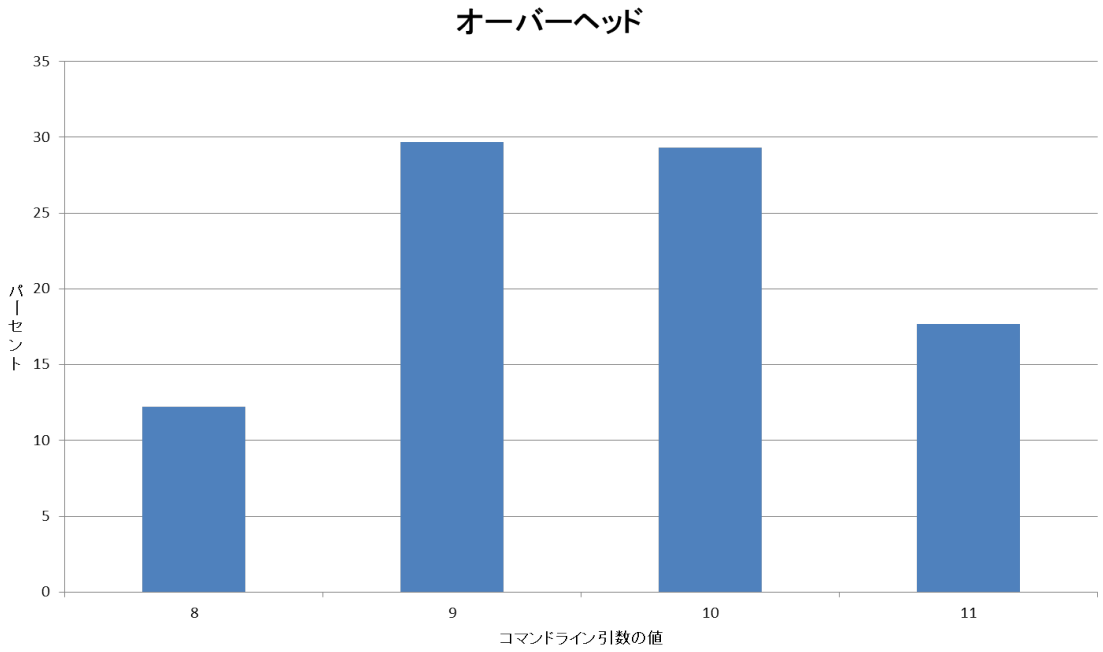

図 5.5. NQueensPar.x10 における追跡コードによるオーバーヘッド

5.2　スコーピング DSL のイベントフィルタリングによる可視化スピードへの影響

次に，我々のツールのビジュアライザに関しておこなった実験結果について紹介する．ビジュアライザでは，スコーピング機能によって表示するイベントを適切に絞ることができる．イベントが大量に発生している場合，すべてを可視化すると適切なチューニング及びデバッグ情報が得られない可能性があるだけでなく，可視化に要する時間も大きくなり，ログのサイズにたいしてメモリ量が小さい場合には可視化が困難なこともある．したがって，そのような場合にも，スコーピング DSL によりイベントをフィルタリングすることが求められるため，イベントフィルタリングした場合としていない場合の可視化速度への影響実験をおこなった．なお，可視化の実験は主に移動イベントを見るグラフで行い (移動イベント以外も可視化はされる)，それぞれ 5 回実行をしてその平均を実験結果として載せている．

実験環境は以下の通りである．

- CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 4 cores
- RAM: 16GB
- OS: OpenSUSE 12.3
- ビジュアライザに使用した Java: Java(TM) SE Runtime Environment (build 1.8.0_25-b17) Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)

図 5.6 は KMeansDist.x10 において，プレース数 4 を維持したまま，定数のうちの POINTS の値を変化させながら，ITERATIONS は 3 にし，それ以外の定数はサンプルコードから変更していない．DSL によるフィルタリングしていない場合とプレース 0 で起きたイベントのみを表示するようにフィルタリングした場合の表示速度を測定した結果である．定数 POINTS の値が大きくなり処理量が大きくになるにつれて，フィルタリングなしの場合は可視化速度が非常に遅くなっていっているが，フィルタリングをした場合はわずかしか遅くなっていない．フィルタリングなしで POINTS=100 の場合は 6 分以上もの時間を要しているのに対し，フィルタリングしたほうはごくわずかな速度の上昇でとどまっている．

したがって，DSL でフィルタリングした場合は，適切な情報を得られやすいだけでなく，可視化のオーバーヘッドも非常に少ないことがわかった．

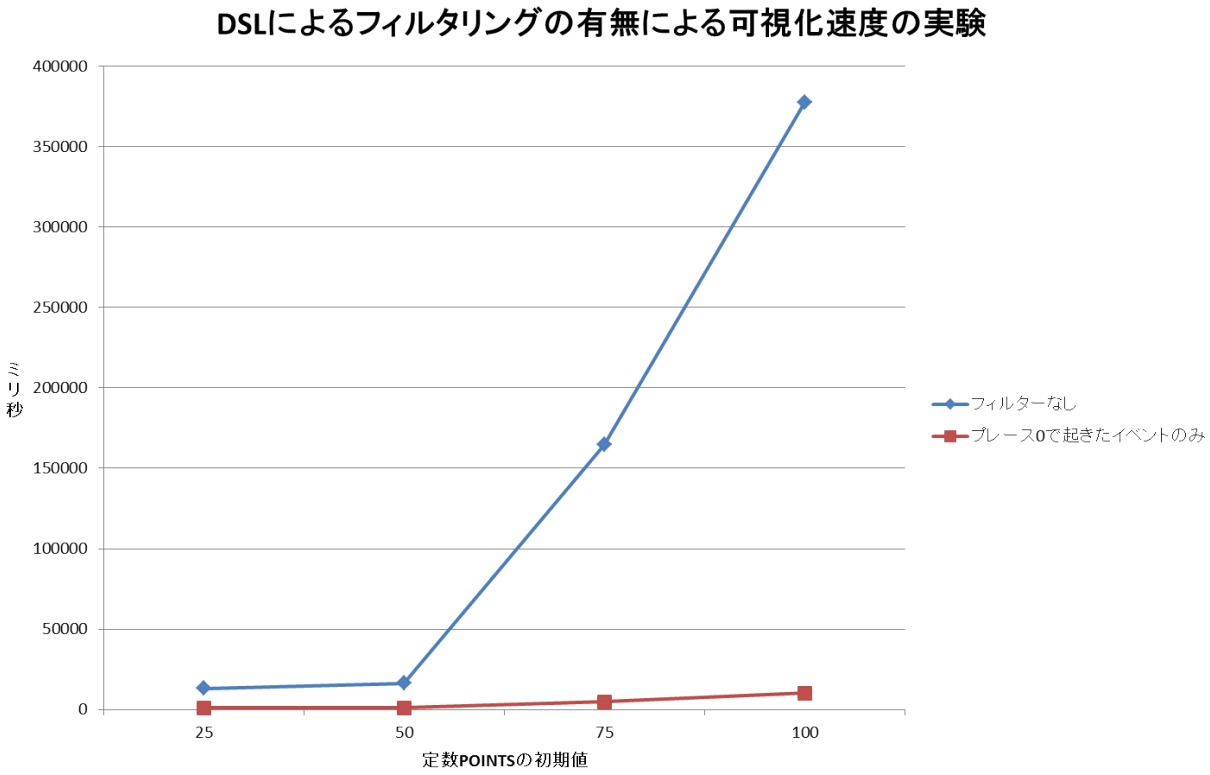

図 5.6. KMeansDist.x10 におけるスコーピング DSL の使用有無による可視化速度の違い

第 6 章

まとめと課題

本研究では，X10 の暗黙的な挙動を可視化するためのプロファイリングツール X-Eye を開発した．X10 は並列分散処理用に様々な抽象化された構文や抽象化されたメモリモデルを使った，独特のプログラミングモデルを持っている．抽象化された構文によってデータ送受信を抽象化したり，スレッド生成などの並列分散処理に欠かせないローレベルな処理を容易にプログラミングできるようになった一方，データ送受信のタイミング・量や同期などの挙動の詳細を把握することが難しい．我々の開発したツール X-Eye の中のプロファイラは，X10 コンパイラを拡張して開発したもので，コンパイル時に追跡コードを X10 プログラムに挿入することで実行時にそれら X10 固有のイベントの実行時情報を取得して，JSON 形式のログファイルとして保存する．そしてビジュアライザによってそれらのログ情報をグラフとして可視化するだけでなく，ビジュアライザにはスコーピング機能が実装されており，Java 8 のリスト処理ライブラリである Stream API をモデルとして我々が設計したスコーピング DSL を記述することで可視化するイベントを適切に，かつリアルタイムにフィルタリングすることができ，大規模分散処理などで大量のイベントが発生した場合にも，自分の求めている情報を取得しやすくなった．プロファイリングは移動イベントが大量に発生している場合にプレース数が多くなればなるほどオーバーヘッドが大きくなり，移動イベントがない場合にはオーバーヘッドが低くなることがわかった．また，スコーピング DSL を使ってイベントをフィルタリングすれば可視化速度も上がることもわかった．

今後の課題としては，移動イベントが大量に発生している場合でもプロファイリングのオーバーヘッドを低くすることである．これはアクティビティごとに生成しているトレーサーを送受信するために発生していることが原因だと考えられる．そのため，トレーサークラスのフィールドなどを必要最小限にすることでトレーサーインスタンスのサイズを小さくするか，もしくはトレーサーを使わない追跡方法を確立する必要がある．また，async 構文は追跡コードだけでなくトレーサーインスタンスも生成するがために追跡コードそれ自体の実行時間がやや大きくなるため，トレーサーをアクティビティごとではなく，プログラムの最初に 1 つ作っておき，それがすべての追跡を行うような方式でもオーバーヘッドを減らせる可能性がある．今後の研究の方向性は，移動イベントやアクティビティ生成イベントが多い場合でも，プロファイリングオーバーヘッドが小さくなるよう，オーバーヘッド削減に努めていくことに注力

するとともに，スコーピング DSL の機能を増やすことでより多彩なイベントフィルタリングができるようにすることである．

発表文献と研究活動

(1) 板橋 晟星, 佐藤 芳樹, 千葉 滋．PGAS プログラムのためのプロファイリングツールの開発，PPL2014，2014.03.05-07 (ポスター)

(2) Seisei Itahashi, Yoshiki Sato and Shigeru Chiba. Toward a profiling tool for visualizing implicit behavior in X10, The 2014 X10 Workshop co-located with PLDI'14，2014.06.12 (発表)

(3) Seisei Itahashi, Yoshiki Sato and Shigeru Chiba. X-Eye: A Profiling Tool For Visualizing Implicit Behavior in X10 , X10 Day Tokyo 2015，2015.01.29 (講演発表)

(4) 板橋 晟星, 佐藤 芳樹, 千葉 滋．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プロファイリングツールの開発, IPSJ PRO，2015.03.09-2015.03.10 (発表)

参考文献

[1] Jack J.Dongara and David Walker. MPI: A Standard Message Passing Interface. In *Supercomputer12(1)*, pages 56–68, 1996.

[2] L. Dagum, Silicon Graphics Inc., USA, and R. Menon. OpenMP: an industry standard API for shared-memory programming. In *Computational Science & Engineering, IEEE*, volume 5, pages 46–55, 1998.

[3] 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. In *OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[4] X10 language web page. `http://x10-lang.org/`.

[5] PGAS web site. `http://www.pgas.org/`.

[6] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[7] Chapel language web page. `http://chapel.cray.com/`.

[8] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420. IEEE, 2010.

[9] XcalableMP language web page. `http://www.xcalablemp.org/ja/index.html`.

[10] Robert W Numrich and John Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[11] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[12] Unified Parallel C language web site. `http://upc.lbl.gov/`.

[13] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar' 10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

[14] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications*, pages 637–644. IOS Press, 2008.

[15] Jens Volkert Bruno Buchberger. Tau: A portable parallel program analysis environment for pc++. In *Parallel Processing: CONPAR 94 - VAPP VI*, pages 29–40. Springer Berlin Heidelberg, 1994.

[16] Nathan Tallent and Darren Kerbyson. Data-centric performance analysis of pgas applications. In *2nd Int'l Workshop on High-performance Infrastructure for Scalable Tools (WHIST 2012)*, 2012.

[17] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int'l Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[18] Jimmy Su and Katherine A. Yelick. Automatic communication performance debugging in pgas languages. In *LCPC*, LNCS 5234, pages 232–245. Springer, 2007.

[19] Goldsmith, Simon F., Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404. ACM, 2007.

[20] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, 1998.

[21] Paudel Jeeva, Olivier Tardieu, and José Nelson Amaral. Guiding x10 programmers to improve runtime performance. In *7th International Conference on PGAS Programming Models*, 2013.

[22] OpenJDK's Project Lambda web site. `http://openjdk.java.net/projects/lambda/`.

[23] Oracle's Project Lambda pdf. `http://www.oracle.co.jp/javaone/2012/download-docs/file.php?name=JS1-31_DL.pdf`.

[24] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th Int'l Conf. on Compiler Construction*, LNCS 2622, pages 138–152, 2003.

[25] Polyglot web site. `http://www.cs.cornell.edu/Projects/polyglot/`.

[26] JavaFX oracle web site. http://www.oracle.com/technetwork/jp/java/javafx/overview/index.html.

[27] Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998.

謝辞

本研究を進めるにあたり，千葉滋教授からは研究活動の心構え，研究の方針，論文の作成方法，そして様々な研究のアイデアについて指導を賜り，大変感謝しております．元千葉研究室所属で現在は情報基盤センターの佐藤芳樹氏には毎週のように進捗を聞いていただき，アドバイスをいただくことで研究を円滑に進めることができました．国内外での学会発表に参加できたのみ千葉教授および佐藤氏の助力によるところがとても大きく，私の人生の糧となる経験をさせていただくことができました．また，卒業してしまいましたが，元千葉研究室博士課程の武山文信氏には X10 コンパイラに関して指導をいただくことができ，研究活動の大きな山を越えることができました．ならびに現在の研究室の先輩方にも普段から様々な助言をいただくことができ，同期の学生，そして両親にも研究生活の様々な支援をいただきました．この場を借りて，感謝いたします．ありがとうございました．

付録 A

ソースコード

Listing A.1. EventStream.java

```
public enum EventType {
  MOVE, //移動イベント
  EXEC, //移動を除いた、アクティビティの純粋な実行を表すイベント
  FINISH, // 粗粒度同期イベント
  CLOCK, // 細粒度同期イベント
  ATOMIC; //アトミックな計算のイベント
}
```

Listing A.2. EventStream.java

```
public class EventStream<E extends Event.Elem>{
  private EventList<E> eventList;
  Stream<E> stream;
  private Map<String, Activity> activityMap;

  public EventStream(EventList<E> eventList, Map<String, Activity>
      activityMap) {
    this.eventList = eventList;
    this.stream = this.eventList.stream();
    this.activityMap = activityMap;
  }

  public EventStream<E> filter(Predicate<? super Event.Elem> predicate) {
    List<E> restList = new ArrayList<E>();

    this.stream.forEach(e -> restList.add(e));
    List<E> deletedList = new ArrayList<E>(restList);

    deletedList.removeAll(restList.stream().filter(predicate).collect(
        Collectors.toList()));

    for (Event.Elem event : deletedList) {
      event.isAvailable = false;
      if (event.type == EventType.MOVE) {
                  activityMap.get(event.activityNumber).isUseExec = false;
              }
    }
```

```
    restList.addAll(deletedList);

    restList.sort(new Comparator<Event.Elem>() {

      @Override
      public int compare(Elem e1, Elem e2) {
        if (e1.activityNumber.equals(e2.activityNumber)) {
          if (e1.type == e2.type) {
            return (int) (e1.startTime - e2.startTime);
          } else {
            return e1.type.number - e2.type.number;
          }
        } else {
          return e1.activityNumber.compareTo(e2.activityNumber);
        }
      }
    });

    this.eventList = new EventList<E>(restList, activityMap);
    this.stream = restList.stream();

    return this;
  }

  public ActivityStream<Activity> mapToActivity() {
    return new ActivityStream<Activity>(this, this.activityMap);
  }

  public EventList<Event.Elem> toList() {

    List<Event.Elem> list = new ArrayList<Event.Elem>();
    this.eventList.forEach(e -> list.add(e));

    EventList<Event.Elem> eventList = new EventList<Event.Elem>(this.
        activityMap);

    for (Event.Elem event : list) {
      eventList.add(event);
    }

    return eventList;
  }
}
```

Listing A.3. ActivityStream.java

```
public class ActivityStream<T extends Activity> {
  private Stream<Activity> stream;
  private EventStream eventStream;

  public ActivityStream(EventStream eventStream, Map<String, Activity>
      activityMap) {
    this.eventStream = eventStream;
    stream = mapToActivity(this.eventStream.stream, activityMap);
```

```
}

@SuppressWarnings("unchecked")
public ActivityStream<Activity> filter(Predicate<? super Activity>
    predicate) {
  List<Activity> restList = new ArrayList<Activity>();
  stream.forEach(e -> restList.add(e));

  List<Activity> deletedList = new ArrayList<Activity>(restList);
  deletedList.removeAll(restList.stream().filter(predicate).collect(
      Collectors.toList()));

  for (Activity activity : deletedList) {
    activity.isAvailable = false;
  }

  restList.addAll(deletedList);

  restList.sort(new Comparator<Activity>() {

    @Override
    public int compare(Activity e1, Activity e2) {
      return e1.activityNumber.compareTo(e2.activityNumber);
    }
  });

  this.stream = restList.stream();

  return (ActivityStream<Activity>) this;
}

public void forEach(Consumer<? super Activity> predicate) {
  stream.forEach(predicate);
}

public Stream<Activity> mapToActivity(Stream<? extends Event.Elem>
    eventStream, Map<String, Activity> activityMap) {
  Set<Activity> activitySet = new TreeSet<Activity>(new Comparator<
      Activity>() {

    @Override
    public int compare(Activity o1, Activity o2) {
      return o1.activityNumber.compareTo(o2.activityNumber);
    }
  });

  eventStream.forEach(e -> {
    Activity activity = activityMap.get(e.activityNumber);

    switch (e.type) {
    case EXEC:
      activity.exec.add((Exec.ExecElem) e); break;
    case MOVE:
      activity.move.add((Move.MoveElem) e); break;
```

```
      case FINISH:
        activity.finish.add((Finish.FinishElem) e); break;
      }

      activitySet.add(activity);
    });

    return activitySet.stream();
  }

  public ActivityStream() {
    super();
  }

  public EventList<Event.Elem> toList() {
    Set<? extends Activity> activitySet = stream.collect(Collectors.toSet())
        ;
    EventList<Event.Elem> list = eventStream.toList();
    list.isUseEvent = true;
    list.activitySet = activitySet;

    return list;
  }
}
```