Toward a profiling tool for visualizing implicit behavior in X10

Seisei Itahashi
The University of Tokyo
seisei@csg.ci.i.u-tokyo.ac.jp

Yoshiki Sato
The University of Tokyo
yoshiki@ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
chiba@acm.org

Abstract

This paper presents a design sketch of the profiler that we are currently developing for X10. To modify an X10 program for improving the execution performance, a profiler should visualize implicit data transfer among places and synchronization among activities. Since X10 is a PGAS language and its programming is not a SIMD-style, visualizing those kinds of behavior of X10 programs is significant in practice. This paper shows how those kinds of behavior are visualized.

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging

Keywords Performance profiling, visualization, parallel computing.

1. Introduction

X10 provides high productivity for large-scale computing with high-level language abstraction on parallel and distributed operations. However, since they involve various implicit operations, developers often see difficulties in understanding the behavior of their programs with respect to synchronization and data transfer. To address these difficulties, we are currently developing a profiler and a visualizer for X10 programs. For example, unlike a typical MPI program, some data transfers in X10 are implicit. When an activity is moved among places, data transfers are implicitly executed but they are not explicitly described in the source code. Which activities are synchronized by the finish operator is not explicitly described or it is often not statically determined. These difficulties are serious since typical X10 programs are not SIMD-style code but they create a number of activities that behave differently; their program structures are often more complicated than typical SIMD-style programs using MPI.

To illustrate such implicit behavior of X10 programs, our profiler and visualizer perform source-to-source translation of X10 programs. The profiler inserts probe code into an X10 program and logs its behavior during runtime. Then, our plan is that the visualizer presents the logged information of the implicit behavior along with the logs of explicit events, which are explicitly described in the X10 source program such as method calls and the execution of finish. The activities involved in synchronization are identified by showing relevant explicit events such as where the activities are created in the source code. We also plan to provide a scoping mechanism based on aspect orientation. Narrowing the profiled explicit/implicit events and data would be useful to ease the understanding of the behavior of complex programs. These features are currently being implemented by modifying an existing X10 compiler. In this paper, we first present the functionalities demanded when debugging X10 programs. Then we present our design of the profiler and visualizer for providing these functionalities. We also show the current status of our implementation in progress.

```
public static def main(Rail[String]) {
  val BigD = Dist.makeBlock...;
  val D = BigD | ((1..N)*(1..N));
  val A = DistArray.make...;
  val Temp = DistArray.make...;
  val D_Base = Dist.makeUnique...;
  var delta: Double;

  do {
    finish for (z in D_Base.places()) at (z) {
      // kernel computation
      delta = A.map(...).reduce(...);
    }
    finish for (place in D.places()) at (place){
      for (p in D) {
        // copy the array in parallel
        A(p) = Temp(p);
      }
    }
  } while (delta >= epsilon);
}
```

Listing 1. An X10 program for computing heat spreading

2. Requirements for X10 profilers

Since X10 is a PGAS language and provides different features from non-PGAS parallel programming languages, a profiler for X10 programs has to show not only typical kinds of profiles but also ones unique to X10. Such unique kinds of profiles have to help developers understand two aspects of the behavior of X10 programs: implicit data transfer and activities involved in synchronization. X10 programs are not SIMD-style code and hence a large number of activities may act differently. Understanding those aspects is significant when developers are debugging their programs and improving their execution performance.

2.1 Implicit data transfer

When an activity moves to another place, all local variables and arrays declared before the movement will be copied and transferred to the destination place. For example, Listing 1 is a typical X10 program taken from [2]. Since the original program was written in an old style, we rewrote the program to follow the current specifications of X10. In this program, a number of activities are created and moved to different places. Estimating how much data is transferred and when it is transferred would be a complex task for developers as the number of activities increases but they are crucial information for improving the execution performance of the program. An activity might more frequently move if the program includes variables of type GlobalRef. The value of such a variable is often fetched by a short expression surrounded by at, which might be abused and cause a serious performance bottleneck due to data transfer.

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

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

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

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

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

Listing 2. An X10 program in which the number of activities created is not statically known

2.2 Activities synchronized together

If some methods are called within the body of finish, the methods might create several new activities and the number of the created activities could not be statically determined. The developer would want to know how many activities are synchronized by finish and how long the synchronization takes. They would want to know whether or not there is an activity that runs longer than others and thus becomes a bottleneck at the time of synchronization.

Listing 2 shows an example of such a case. This program includes three subclasses of the Scheduler class. The run method in the Sequential class sequentially performs the given tasks. The run method in the Parallel class performs each task by using a different activity. The run method in the last class OptimizedParallel performs the tasks in parallel by using a fixed number of activities. It will avoid excessive parallelism. Letting developers select an appropriate Scheduler object is natural but then the number of activities the doTask method will create and synchronize is not easily estimated without considering the dynamic type of the object.

3. Our profiler and visualizer for X10

To satisfy the requirements mentioned above, we are currently developing a profiler for X10 programs and the visualizer of the profiled data. Our profiler is a modified version of the X10 compiler built with Polyglot [6]. It is a source-to-source translator that inserts probe code into a target X10 program. When the translated program is run, the probe code records various kinds of events such as the creation of an activity, the move of an activity, and synchronization. Here, the synchronization includes barrier synchronization and the termination of an activity. Then our profiler reads the recorded events and visualizes them. When the probe code records an event, it also records information such as the current position in the source code and the name of the method where the event occurs. We also have the visualizer show the current position and the method name for each event. The profiler/visualizer is a Java application using JavaFX.

Listing 3 presents the X10 program after the translation by the current version of our profiler. The original program was presented in Listing 1. It inserts probe code around the language constructs of X10 such as at, async, and finish. For example, in Listing 3, the calls to the methods in CreateEvent and MoveEvent are the probe code. The probe code records when certain language constructs are executed. The call to activitiyCreated in CreateEvent is also the probe code. It records when the main method starts.

3.1 The behavior of activities

The current version of our visualizer can read the event log recorded by the probe code and then it can illustrate the behavior of activities. Figure 1, 2, and 3 show the output by our visualizer.

Figure 1 illustrates the movement of activities. The vertical axis represents which place an activity is located. The digits indicate the place number. The horizontal axis represents the wall time. The colored thick line denotes the trace of the move of each activity till the activity terminates. Each activity is identified by a pair of two digits such as 0-1. The first digit represents the place where the activity is created and the second digit represents a uniquely-assigned number among the activities created in the same place. The yellow line 0-0 denotes the trace of the root activity. The figure shows that it moved from place 0 to other places and created an activity.

Figure 2 illustrates how activities are synchronized by finish and so on. The vertical axis represents the identification of an activity. The horizontal axis represents the wall time. Unlike Figure 1, the colored thick line denotes when an activity blocks for synchronization. For example, Figure 2 illustrates that the activities 0-1, 1-0, 2-0, and 3-0 stopped running till they are synchronized at time 600. It also presents that the activities 0-1, 1-0, and 2-0 were waiting for the activity 3-0. Figure 3 is a complementary chart to Figure 2. It illustrates when an activity is not blocked but running. It also illustrates when an activity is created and when it terminates.

Our visualizer also shows detailed information of events and activities. Figure 4 and Figure 5 are the graphs made by our visualizer by running KMeansDist.x10, that is bundled as sample code with the X10 compiler source files. Figure 4 shows that when you click one of the charts that illustrates the events of synchronization behaviors, our visualizer displays the detailed information of the clicked synchronization event. The detailed information includes not only start time and end time but also the line number in the source code that the event happened, and the names of class and method that the event belongs to. Figure 5 shows the information of the activity that is clicked in the graph, which illustrates the running time of activities. The displayed information includes the activity number of the clicked activity, the place that the activity is created at, the time that it started and ended, the names of the class and method that the activity belongs to when it is created, and the line number of the async that creates the activity.

3.2 Functionalities not implemented yet

Our profiler and visualizer has not been able to show implicit data transfer during runtime but we plan to implement this functionality. For this functionality, the probe code has to be inserted into not only X10 source code but also the X10 runtime written in C++. The probe code records the amount of the transferred data by the X10 runtime along the wall time.

After this modification, the visualizer would be able to present a chart like the one shown in Figure 6. The horizontal axis represents

```
public class MyHeatTransfer {
  public static def main(Rail[String]) {
    finish  {
      val trace0 = new EventTracer();
      val gtrace0 = GlobalRef[EventTracer](trace0);
      // The root activity is created
      CreateEvent.activityCreated(gtrace0,
            "MyHeatTransfer", "main", 9L);
      {
        val BigD = Dist.makeBlock...;
        val D = BigD | ((1..N)*(1..N));
        val A = DistArray.make...;
        val Temp = DistArray.make...;
        val D_Base = Dist.makeUnique...;
        var delta: Double;

        do {
          SyncStartEvent.syncStart(gtrace0,
              "MyHeatTransfer", "main", 18L);
          finish {
            for (z in D_Base.places()) {
              at (z) {
                MoveEvent.activityMoved(gtrace0,
                "MyHeatTransfer", "main", 18L);
                // kernel computation
                delta = A.map(...).reduce(...);
              }
              MoveEvent.activityMoved(gtrace0,
                "MyHeatTransfer", "main", 18L);
            }
          }
          SyncEndEvent.syncEnd(gtrace0,
              "MyHeatTransfer", "main", 18L);

          SyncStartEvent.syncStart(gtrace0,
              "MyHeatTransfer", "main", 23L);
          finish {
            for (place in D.places()) {
              at (place)  {
                MoveEvent.activityMoved(gtrace0,
                "MyHeatTransfer", "main", 23L);
                for (p in D) {
                  // copy the array in parallel
                  A(p) = Temp(p);
                }
              }
              MoveEvent.activityMoved(gtrace0,
              "MyHeatTransfer", "main", 23L);
            }
          }
          SyncEndEvent.syncEnd(gtrace0,
              "MyHeatTransfer", "main", 23L);
        } while (delta >= epsilon);
      }
      TerminateEvent.activityTerminated(gtrace0,
            "MyHeatTransfer", "main", 9L);
    }
    // create a log file
    EventTracer.printResultByPlace();
  }
}
```

Listing 3. The X10 program after the translation

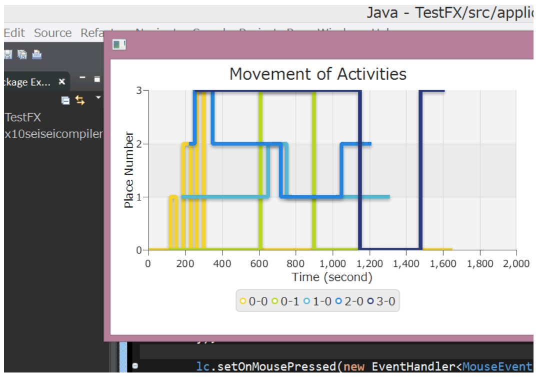


Figure 1. The move of activities

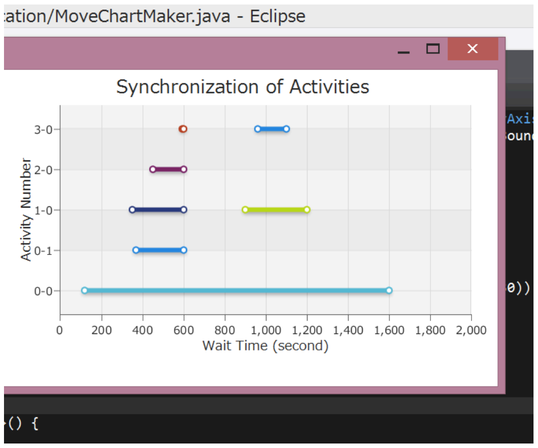


Figure 2. Synchronization among activities

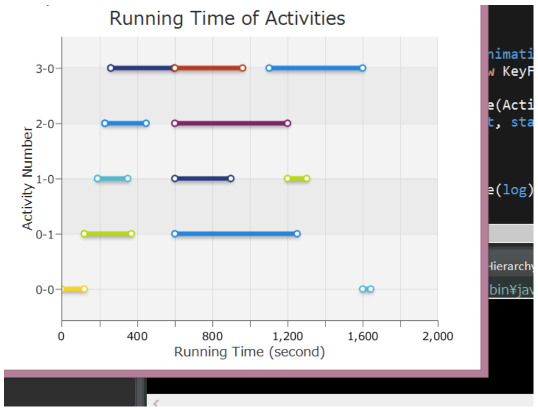
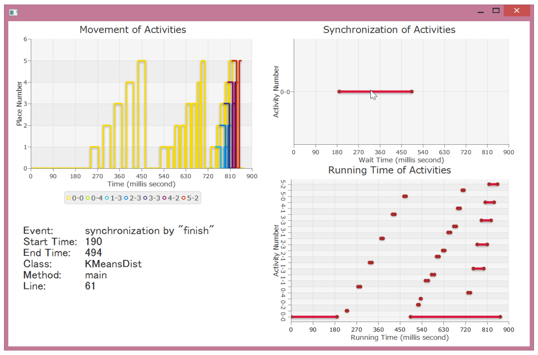


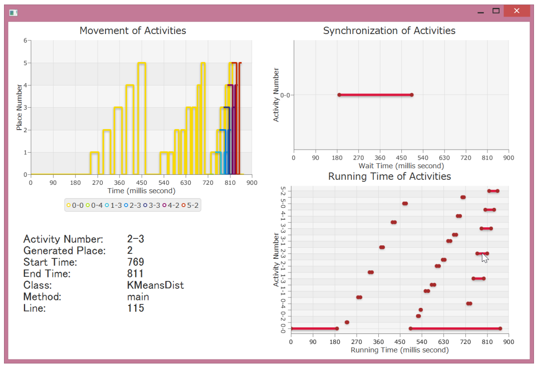
Figure 3. The busy periods of activities

the wall time. The vertical axis represents both the amount of data transferred in total and the identification of activities. This chart overlaps the amount of transferred data over the event logs of each activity. It does not directly present which event at the level of X10 source code causes a large amount of data transfer but it would help developers understand the communication behavior of the X10


Movement of Activities
Synchronization of Activities
Running Time of Activities
Place Number
Time (millis second)
Activity Number
Wait Time (millis second)
Running Time (millis second)
0-0 0-4 1-3 2-3 3-3 4-2 5-2

Event:	synchronization by "finish"
Start Time:	190
End Time:	494
Class:	KMeansDist
Method:	main
Line:	61

Figure 4. Detailed information of a synchronization event


Movement of Activities
Synchronization of Activities
Running Time of Activities
Place Number
Time (millis second)
Activity Number
Wait Time (millis second)
Running Time (millis second)
0-0 0-4 1-3 2-3 3-3 4-2 5-2

Activity Number:	2–3
Generated Place:	2
Start Time:	769
End Time:	811
Class:	KMeansDist
Method:	main
Line:	115

Figure 5. Information of an activity

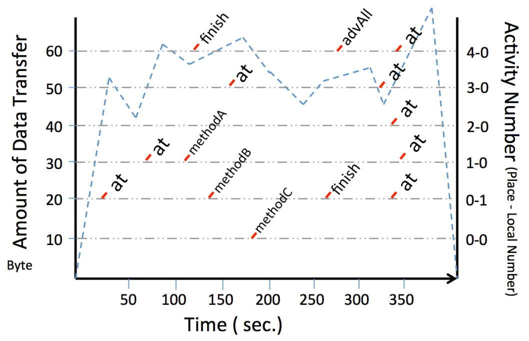


Figure 6. Visualization of implicit data transfer

program and think about how the program should be modified to improve the execution performance.

Another functionality we have not implemented yet is a scoping mechanism to restrict the range of profiling. This mechanism is significant to support development of real-scale software in X10. Our current profiler and visualizer shows the behavior of all the activities created while the program is executed but showing the behavior of a large number of activities in the same chart is not realistic. If the probe code is inserted at a large number of positions, then the execution overheads due to the probe code would be not negligible.

Our plan for the scoping mechanism is to exploit aspect-orientation [3]. We plan to provide an aspect-oriented domain-specific language (DSL) for describing which kinds of events are to be recorded by the profiler. The profiler refers to the description in this DSL and inserts the probe code only at the necessary positions. For example, the DSL will allow developers to specify the activities created within specific contexts so that only those activities will be investigated during the next run. The use of aspect orientation for profilers is not a new idea but it is well known as a killer application of aspect orientation. Several profilers based on aspect orientation, including ours [9], have been proposed.

We also plan to improve the visualization of the behavior of activities. Our current profiler cannot correctly record which activities are synchronized together by the same finish. Like in Figure 2, if a method is called within the body of finish and the method may create an activity, the profiler cannot associate this activity with that finish. Note that the called method cannot be statically determined due to dynamic method dispatch. We must implement a mechanism for recording a call chain as well as activity creation.

4. Concluding remarks

We have presented that a profiler for X10 needs new functionalities and showed a sketch of the profiler we are currently implementing. This profiler records various events during the execution of an X10 program and visualizes implicit data transfer and synchronization among activities. We have already implemented part of the functionality of the profiler and showed it in this paper.

Since a performance profiler is a key component of a tool set for parallel computing, there have been a number of profilers developed. For example, Vampir [4] is a performance profiler for various styles of parallel programs such as MPI, CUDA, and PGAS. For PGAS programs, it collects low-level memory operations such as get, set, and lock of PGAS. Unlike these profilers, we are focusing on X10 and profiler-functionalities uniquely required for tuning X10 programs. The profiler presented in [8] was designed for PGAS programs. However, its target abstraction is global arrays [5] and thus their profiler is different from ours, which is for higher-level language constructs of X10, such as `async` and `finish`. The profiler presented in [7] is also different from ours since it focuses on communication overheads due to the low-level operations of remote memory accesses provided by their PGAS language, Titanium [10]. XAnalyzer [1] is a profiling tool designed for X10. It detects code patterns that will cause a performance problem. The code patterns are predefined and XAnalyzer currently supports eight patterns. Although XAnalyzer is a profiler for higher-level code analysis, our tool provides more basic-level performance data and visualizes them. Combination of the two approaches will be beneficial but this is our future work.

References

[1] P. Jeeva, O. Tardieu, and J. N. Amaral. Guiding x10 programmers to improve runtime performance. In *7th International Conference on PGAS Programming Models*, 2013.

[2] K. Kawachiya. Programming language x10 (in japanese). `https://www.research.ibm.com/trl/people/kawatiya/X10seminar.htm`, 2011.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pages 220–242. Springer, 1997.

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

[5] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. 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.

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

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

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

[9] Y. Usui and S. Chiba. Bugdel: An aspect-oriented debugging system. In *Proc. of 1st Asian Workshop on AOSD*, 12th Asia-Pacific Software Engineering Conf. (APSEC 2005), pages 790–795, 2005.

[10] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, 1998.