Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

Design and Implementation of a DSL based on Ruby for Parallel Programming

並列プログラミングのための Ruby ベースの DSL の設計と実装

Tetsu Soh
曹　哲

Supervisor: Assistant Professor Sasada Koichi

January 2011

Abstract

Finding a high productivity and high efficient(HPHP) parallel programming language is a key to achieve popular parallel programming and it has been an active field of research for over 3 decades.

To achieve HPHP parallel programming language, we proposed to design a DSL based on Ruby, which is a popular sequential langauge among mainstream developers for its high productivity, and then translate the DSL to X10 language, which is a high performance parallel language, to perform parallel execution. Our proposed DSL can express concurrent and distributed programming with syntax that is consistent with Ruby. The DSL code is converted to X10 program by our code translator and is executed in parallel with some runtime libraries developed by us. The code translator and runtime libraries that we developed implement most of Ruby's features and bridge the large gap between the Ruby and X10 language. Therefore, the DSL can keep the productivity of Ruby and achieve high performance. We employ HPC and Ruby benchmarks to evaluate the productivity and scalability of the DSL. The results show that our DSL has better or equal productivity than the X10 language and similar scalability to the X10 language.

概要

並列プログラミング言語として実用されるためには，高い生産性と高い性能を併せ持つ必要があり，これらを達成するための研究が，30 年に渡って続いている．本研究の目標は，生産性と性能を併せ持つ並列プログラミング言語を実現することである．この目標を達成するために，本研究では，生産性の高いプログラミング言語として知られる Ruby をベースとした DSL を設計し，これを高速な並列プログラミング言語である X10 へと変換する．

本研究で設計した DSL は，Ruby の文法ベースに，並列実行のための拡張を加え，並列分散プログラミングを表現することができるようにしたものである．この DSL を，本研究で開発したトランスレータを用いることで，Ruby とは大きく異なる性質を持つ X10 へと変換する．変換後の X10 コードは，本研究で開発したランタイムライブラリを用いて動作することで，Ruby の持つ様々な性質を実現する．これにより，Ruby の持つ高い生産性を失わずに，並列に動作するプログラムを記述し，実行することができる．HPC，および Ruby のベンチマークを用いて DSL の生産性とスケーラビリティの評価した結果，提案した DSL は X10 と同等のスケーラビリティを持ち，かつ X10 以上の生産性を持つことが確認できた．

Contents

Chapter 1 Introduction 1

Chapter 2 Anylysis 3
2.1 Parallel Programming Overview 3
2.2 Overview of X10 5
2.3 Overview of Ruby 7
2.4 Goal 9

Chapter 3 Language Design 10
3.1 Example of DSL 10
3.2 Object Oriented Features 11
3.3 Control Structures 18
3.4 Parallel Programming Module 18
3.5 Features Summary 21

Chapter 4 System Design and Implementation 23
4.1 System Overview 23
4.2 Parsing and AST 24
4.3 Converting DSL to X10 25
4.4 Parallel Programming Model 32

Chapter 5 Evaluation 33
5.1 Productivity Evaluation 33
5.2 Scalability Evaluation 35

Chapter 6 Discussion 39
6.1 Productivity of DSL 39
6.2 Dynamic Typing and Parallel Programming 39
6.3 Scalability 40
6.4 Parallel Programming Model 40
6.5 Hybrid of Ruby and X10 41

Chapter 7 Related Work 42
7.1 DSL for Parallel Programming 42
7.2 Hybrid of Productivity and Efficient Language 42
7.3 Removing GVL in Ruby 43
7.4 Others 43

Chapter 8 Conclusion 44

Publications 46

References 47

Chapter 1

Introduction

Finding a high productivity and high performance (HPHP) programming language for parallel computing has been an active research field for over 3 decades[1]. The motivation for parallel programming originally comes from scientific programming and high performance computing (HPC) and most of parallel programming languages are designed for this purpose. However, recently, as multi-core processors become prevalent and the awareness of crisis for sequential programming [2], mainstream software development is under the pressure to adapt to parallel programming, which is known as popular parallel programming[3]. One key to achieve the popular parallel programming is parallel programming language.

On the other hand, people are expecting more for a parallel programming language. The requirement for high performance is just a base line. As the parallel computing hardware becomes various, such as the cluster computers, GPGPU, SMP, people require that a parallel programming language should be hardware independent. Moreover, as there are many different programming algorithms, the language should be not limited to a particular parallel algorithms. So an ideal parallel programming language should be high productivity, high performance, hardware and parallel algorithms independent.

However, we observed that it is very difficult for a programming language to achieve all these requirements. There are several issues for current parallel programming languages:

1. Most parallel programming languages are designed for HPC. Thus, they are very difficult to use for mainstream programmers. These languages lack productivity.
2. Most mainstream programming languages are sequential. They are inadequate for parallel programming because either the language does not support parallel execution, such as Ruby/Python, or lack parallel model to achieve high performance.[4]
3. Some languages adopt parallel programming models to achieve parallel programming. For example, the Go language[5] employs Communicating Sequential Processes (CSP) model and the Scala language[6] employs the Actor model. However, these programming models have sharp learning curves because they are every different from the shared memory programming model which programmers are used to. The result is less productivity.
4. Developing new languages has very high implementation cost and it is difficult and time consuming to make programmers to migrate to a new language and master it.

Our proposed solution is to use a source-to-source converter to combine 2 languages together to achieve high performance and high productivity. It means that programmers write parallel programs in a high productivity language, and then translate the code to a high performance language to execute.

Our approach is to define a domain specific language (DSL) based on a high productivity language (in this paper the language is Ruby) to express parallel programming model. And

implement the DSL by developing a source-to-source compiler to translate the DSL code to a high performance language (in this paper the language is X10).

Generally speaking, scripting languages are good at productivity and system languages are good at performance[7]. Ruby[8] is a popular scripting language and is famous for its high productivity. X10[9] is a recently developed HPC parallel language that has characters of high performance, hardware and algorithms independence. Therefore, by using our approach to combine these 2 languages together, we can achieve productivity, performance, hardware and algorithm independence for parallel programming.

This approach has several advantages. First, the DSL is based on a mainstream language so that lots of users can avoid learning a new language to adapt to parallel programming. Second, the DSL can inherit the high productivity from the based language. Third, the implementation cost is low. By using source-to-source converter, the implementation is easier than implementing a new language.

To achieve our proposed solution, we designed a DSL based on Ruby. The DSL is designed to express primitives for parallel programming model in Ruby-style. Thus, users can write parallel program in Ruby-style. And we developed a converter to translate the DSL code to X10 code. We also developed X10 runtime libraries to implement some of Ruby's most important features, such as dynamic typing, open class, mix-in and so on. Because there is no Ruby Virtual Machine in the execution of the DSL, so some Ruby features such as dynamic evaluation are not supported.

Our research has following contributions: 1) By using the DSL, users can write parallel program in Ruby-style. It benefits Ruby users. 2) It benefits HPC users by providing an alternative parallel programming language. The DSL is based on Ruby, so the HPC programmers can enjoy the productivity of the Ruby language. 3) The experience of developing a DSL based on a sequential, mainstream, scripting language for parallel programming can be referred by other developers. DSL seems to be a promising solution for building a parallel language. We put this approach into practice and gain experience on how to design the DSL and how to solve challenges in implementation.

This paper gives an overview of parallel programming models and X10 and Ruby programming languages (chapter 2), the language design of our proposed DSL (chapter 3), the implementation of the language system (chapter 4), the result of evaluation for the DSL (chapter 5) and the discussion (chapter 6). We end with a summary of future work.

Chapter 2

Anylysis

In this chapter, we give an overview of parallel programming. And analyze the pros and cons of the X10 and Ruby programming languages. Through the analysis, we want to clarify following questions.

- Why is DSL a possible solution for parallel programming?
- Why is Ruby inadequate for parallel programming?
- Why is the DSL based on Ruby?
- Why is X10 not good for hosting a DSL?

Based on the analysis, we define goals and make some design decisions for our proposed DSL.

2.1 Parallel Programming Overview

In this section, we give an overview of parallel programming. It gives a background on parallel programming and clarify some terminologies that will be referred in later parts of this paper.

2.1.1 Popular Parallel Programming

The parallel programming originally is mainly used for researches, or academic fields. And the parallel programming languages were almost designed for scientists to perform

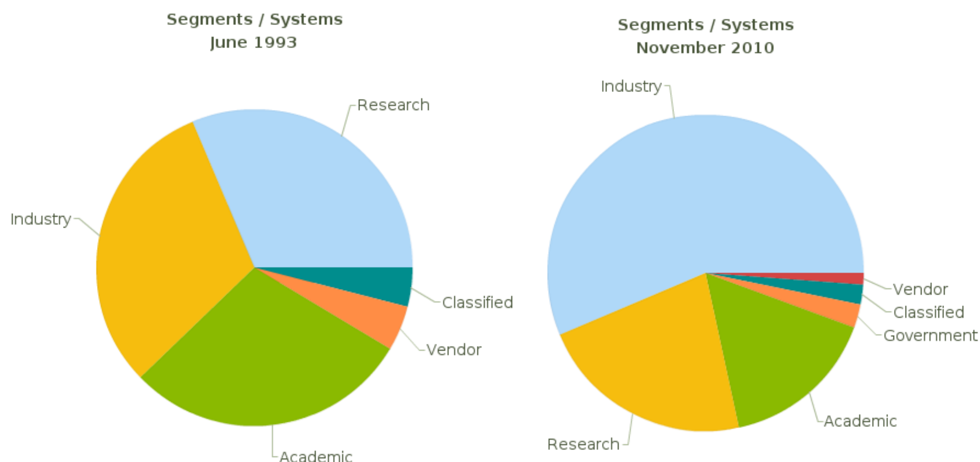

Fig. 2.1. Comparison of Parallel Computing Users(Data from Top500.org)

numerical computing.

However, more and more parallel applications are developed in industry fields. Figure 2.1 shows the statistics for parallel computing users at year of 1993 and 2010*1. From these figure we can clearly see the tendency of popular parallel programming, which means to make parallelism pervasive and parallel programming mainstream. To achieve this goal, it requires parallel programming languages that are easy to use for most general programmers. However, current parallel languages are far from this goal.

2.1.2 Approach to Parallel Programming

For a language to achieve parallel programming, there are generally 2 approaches: 1) a language-based approach, which means to develop a new language from scratch, such as CILK[10], Chapel[11], and X10; 2) library-based approach, which means to implement the parallel programming model as library to extend existing sequential languages, for example MPI[12], OpenMP[13]. Because an internal DSL can be treated as a kind of library, so internal DSL is also the second approach to achieve parallel programming.

Each approach has its merit and disadvantage. For the language-based approach, the advantage is that new language can define precise semantics for synchronization constructs and the shared memory model and compiler can do related optimization. The disadvantage is that the development cost is high and it is difficult for users to accept a new language. For library approach, the advantage is that the low learning curves for users. However, the disadvantage is that it is hard to make big breakthrough. The expressiveness and the performance is limited by the hosting language.

2.1.3 Parallel Computing Environments

From the memory architecture perspective, the parallel computing environments can be classified as: 1) shared memory system, such as multi-core system, 2) distributed memory system, such as cluster system, and 3) hybrid distributed-memory system, such as multi-core cluster system, in which each node is a multi-core system.

Need to point out that the hybrid distributed-memory system is prevalent recently. And this kind of system always requires a hybrid parallel programming model. For parallel computing on each node, the shared memory programming model is used, such as Pthread or OpenMP. Parallel execution over nodes is achieved by using distributed memory programming model, such as MPI. Moreover, the merging of hybrid architecture swaps out most of current parallel programming languages because they cannot provide a hybrid programming model.

2.1.4 Classification of Parallel Programming Models

There are many different ways to classify parallel programming models. We classify the models from abstract level perspective because this effects how we decide to design the DSL. Generally, the parallel programming consists of 4 subtasks[14, 15]:

Decomposition Means to find code that can run in parallel and assign them to thread.
Mapping Means to specify where the thread should run.
Communication Means to send/receive messages among threads.
Synchronization Means to manage synchronization among threads.

*1 The data are provided by Top500.org

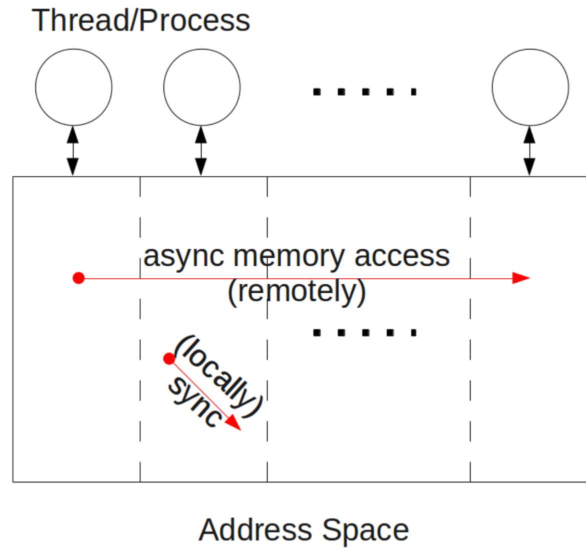

Fig. 2.2. APGAS Model

We measure the abstract level of a parallel programming language by how many tasks can be done automatically by the language. More tasks are handled automatically, higher abstract level the language is. For example, the MPI is a very low level parallel programming model because all tasks are handled directly by users.

We also know, typically, higher abstract level languages have better productivity because there are less details that need programmers to bother about. However the trade-off is that it is more difficult to achieve good performance. So for a parallel programming language, the balance of abstract level is critical.

2.2 Overview of X10

X10 is a new developed, statically typed, parallel, distributed, object-oriented programming language designed for HPC[16, 17, 4] . The design goal of X10 is to provide safety, scalability and flexibility for parallel computing. The target of X10 originally is HPC world, such as scientific applications development. However, recently, it tries to cover the general application development too.

2.2.1 Main Features of X10

As a new developed language, X10 has some advantages on parallel programming.

APGAS Model

The background of X10 development is the merging of hybrid multi-core cluster computing environment as mentioned previously. So the X10 language employs the Asynchronous Partitioned Global Address Space (APGAS) memory model. The figure 2.2 shows what APGAS looks like.

From the programmers perspective, the PGAS offer a global address space similar to shared memory. It means that it is possible for programmers to create an object that is visible from any other processes/threads of program. Contrast to MPI on distributed memory architecture, which is said to has a local address space. An object allocated by

one process is not visible to others processes. Thus users have to explicitly communicate to other processes to access the object.

However, the address space is divided into parts. The access of object crossing partition is exposed to programmers with special syntax. In such a way, the locality of object is visible for programmers. Because the cost of memory access to local partition and to remote partition is totally different, the locality-awareness is very important for achieve performance.

To put it together, the APGAS model is suitable for the hybrid cluster of shared memory architecture because it provides the programming convenience of shared memory and the locality awareness for controlling data layout which is critical to achieve high performance and scalability

Modest Abstraction Level

As mentioned in previous section, a parallel programming model/language can be classified by its abstract level. X10 provides automatic communication and synchronization but leaves the decomposition and mapping for programmers to do manually. Because the decomposition and mapping are depended on parallel algorithms and underlying hardware, leaving these tasks to programmers gives the opportunities of manually tunning to maximum performance. The lock manipulation and message sending/receiving can be handled by compiler very well. Therefore, automatically do these tasks will relief programmers from onerous tasks and good for productivity.

Hardware and Algorithms Independence

X10 language is design as an general purpose parallel programming language. It can run on different hardwares. It can run on SMP, cluster or CUDA environment. Moreover, X10 does not specify what parallel algorithm should be used. Users can do SPMD or data parallel with X10.

New Programming Primitives

X10 defined some primitives and built-in objects for concurrent and distributed programming. For concurrent programming, X10 offers the `async`, `finish`, `atomic` and `when` primitives to instead threads and locks. For distribute programming, X10 offers the `Place`, `Dist`, `Region` and `DistArray` objects[18].

2.2.2 Demerits of X10

However, as a parallel programming language designed for HPC, there are disadvantage of X10 that make it less attractive for general purpose programmers.

First of all, X10 is less expressive compared to modern dynamic programming langauges[19]. Because X10's syntax is consistent with Java, so it has similar expressiveness to Java. For example, users have to close every statement with a semicolon. Moreover, X10 is original designed for HPC programming which is mainly numerical oriented computing. To handle general application development, X10 is less advantage.

Second, X10 is a statically typed language. The type system in X10 goes further than the type system in Java. X10 has type constrain and generic type. These features are powerful. However, these features are only make sense to framework developers. For general programmers they are just make code is hard to read and write. For example, to define a lambda(function) in X10 looks like:

```
val fun:(int)=>void = (i:int) => {// body}
```

Compared to do the same thing in Ruby:

```
fun = lambda {|i| #body}
```

Although statical type and dynamic type which is more productivity is still arguable and hard to give a objective conclusion, recently research shows that the type system is not really as helpful as it seems to be[20]. We believes that modest type system can make code more understandable and maintainable (type declaration can be used as document), but dynamic type is easier to get hands on.

2.3 Overview of Ruby

Ruby is a scripting language and is famous for its ease to use and high productivity. Ruby is always used as "glue" for connecting different components written in other languages during software development or to write prototype in early phase of development. However, recently, Ruby is also used to develop some large-scale and long running applications, such as web applications, even enterprise development[21].

2.3.1 Main Features of Ruby

Although it is very difficult to clearly define why Ruby is high productivity, there are some features make Ruby is very easy to use.

Dynamic Typing

The variables in Ruby has no compile type. So users don't need to write the type for variables. Although statically typed language can benefit from compile-time verification and better IDE support, dynamic typing make it is easier to start coding – users do not need to deeply consider and design the interface of a object(such as the type of method). It is especially useful in the early stages of a project because the implementation may be changed very often.

Duck Typing

For static object-oriented language (OOL) the interface of a object is defined through type. And users need to confirm the type of an object then can call a method on it. For example, in Java, following code is often seen.

```
if (a instanceof String) { // do something on string }
```

In Ruby, users do this by check whether an object has expected method by using `respond_to?` method and if it has the method, then invoke the method using `send(name)` method.

This feature reduces the necessary of inheritance. In Java, if you want a class to behavior like a String, you have to inherit the `String` class. In Ruby, you just define methods that a string object should have.

Mixins and Singleton Methods

Ruby does not support multiple inheritance but it achieves the same effect with mix-in. If a class includes a module, then all methods defined in that module will be mixed into the class. Users can use the included methods as they as defined on the class.

The mix-in is possible because that Ruby allows the dynamic modification of the class at runtime. This character is known as open-class. Singleton method is another feature

that benefits from the open-class character of Ruby.

Singleton method means a method defined on an instance. It breaks the general think that the behavior of an instance is determined by its class. It enables programmers to specify behavior for a particular instance.

Blocks and lambdas

The block/lambda in Ruby is first-class object. Users can treat them as data and pass them among objects. Moreover, Ruby support anonymous lambdas. This features make it possible to implement block-scoped constructors that look like language features. It is also one reason that Ruby is good for creating DSL.

Literals

Ruby has literals expression for almost every build-in types.

- Strings, e.g., `"string"`
- Numbers, including binary, octal, decimal, hex
- Arrays, e.g., `[1,2]`, `%w(each word is element)`
- Hashes, e.g., `{key=>value}`
- Blocks, e.g., `{block body}`
- Regular expressions, e.g., `/regexp/`

These literal expressions simplify the creation of new objects.

Syntax Sugars

Ruby defined lots of syntax sugars so that users can use whatever they favor to. For example, In X10, users can write a loop structure with either `for` or `while`. In Ruby, except `while` and `for`, users can also express loop by using `loop`, `until`, or `each/times` methods.

2.3.2 Demerits of Ruby

As a popular mainstream programming language, it is would be very impact if Ruby was adequate for parallel programming. However, there are some issues for applying Ruby in parallel programming.

First of all, Ruby cannot support parallel execution. Ruby do support threads. From C Ruby 1.9, the thread model in Ruby is implemented by native thread. So the Ruby code can run in concurrent. However, due to the use of Giant VM Lock (GVL) in implementation, the Ruby VM cannot support multiple threads to run simultaneously. Ruby VM employs a mutual exclusion lock to avoid sharing code that is not thread-safe with other threads. It means, when a thread in Ruby try to run, it has to first require the GVL. Only after the thread gains the GVL, it can run. The result is that at any time there are only one thread is running.

Second, in shared memory environment, as most mainstream programming language, the concurrent coding in Ruby is achieved by using the thread/lock mechanism. However, the practical development experience tells us that it is very difficult to write correct and efficient parallel program with thread and lock. The thread/lock is too low level. Programmers have to manage the lock and synchronous the threads by themselves. It is easy to suffer from the dead lock problem or over synchronization that hurt performance.

Third, Ruby itself does not support distributed programming. There are some libraries, such as dRuby[22], that implement the distributed programming using technique similar to remote method invocation(RMI). However, the lesson we learned from Java RMI tells that it is hard to achieve good performance and scalability[23]. Moreover, the basic problem

with the RMI or RPC techniques is that they does not effective on cluster of multicore systems.

2.4 Goal

X10 is a powerful parallel programming language, while Ruby is a easy-to-use sequential programming language. The main goal of our DSL is to bridge the gap between the Ruby and the X10 language to combine the advantages of the 2 languages to achieve parallel programming. To achieve the goal, we make following key design decisions.

Develop a DSL Based on Ruby

As we pointed in previous chapter, to achieve parallel, we can introduce a new language or implement as libraries(frameworks, DSL) to extends a host language. Because we want to combine the expressiveness of Ruby and parallel power of X10, a DSL is a promising approach. Furthermore, because X10 is less expressive and concise than Ruby, it is not fit to host a DSL. Therefore, we decide to build a DSL on Ruby to use the Ruby syntax.

Be Consistent with Ruby

The DSL program should looks like original Ruby program. It means that 1)for existing syntax, the semantics in the DSL should be the same as host language, 2) new introduced syntaxes should follow Ruby's conventions, 3) the DSL should implement most of Ruby's main features.

Following the Ruby-style make the converter is more difficult to develop, but we believe that it is convenient for the Ruby users because they need to pay less effort to adapt the DSL.

Introduce PGAS Model

Because Ruby lacks of programming model to address programming challenges on shared-memory cluster environment, we decide to introduce the PGAS model in our DSL. This model is also used in X10 and other parallel languages, such as UPC[24], and CAF[25], and has been proved to be an effective solution for hybrid memory architecture.

Introduce New Primitives for Parallel Programming

Ruby uses thread and lock to express concurrent program. These primitives are too low level. Thread-based parallelism on shared-memory system has been proved to be a hard way to do parallel. On the other side, X10 has some high level primitives for concurrent programming[17]. We decide to express concurrent using high level primitives from X10 to instead the thread and lock in Ruby.

Moreover, Ruby has not program model for distribute programming but X10 has. So we decide to introduce the distributed programming model from X10 into our DSL.

Don't Raise Abstraction Level

Considering the performance, we decide to keep the abstraction level of parallel programming in our DSL at the same level as X10 language, which means that decomposition and mapping tasks are left to users to handle explicitly.

So far, automatic parallelization has not been proven successful except in the scientific fields. Because our main goal is to bridge the Ruby and X10, making a smarter compiler is out of this research.

Chapter 3

Language Design

In this section, we describe language features of the DSL. The DSL combine some features from Ruby and some features from X10 language. We are aiming at writing parallel program in Ruby style. The syntax of DSL is almost consistent with Ruby and most of Ruby features are preseved. So Ruby users won't feel discomfort with the DSL. However, due to implementation limition, there are some features are not supported in the DSL.

3.1 Example of DSL

Before we discuss the design details, in this section, we present 2 example programs to illustrate some of the DSL's features. Through these programs, we can know what the DSL looks like, how it expresses objected-oriented features and how parallel programming is handled in the DSL.

Concurrent Programming Example

The example program 1 calculates the Fibonacci number by using the naive recursive algorithm. For Ruby users they may be very familiar with the code. Because except for 2 statements, other statements are the same as Ruby. However, Ruby users won't code Fibonacci in the way that we showed here. Generally, Ruby users prefer more functional programming style. To show the DSL's object-oriented feature, the program uses a more Java-style.

In the program, we define a class named `Fib` to represent a Fibonacci number. The class has a field (instance variable) `r` to hold the value and a method `run` to calculate the value. In the example, we calculate the Fibonacci 24 and print out the result.

In the `run` method, there are 2 new syntax, `finish` and `async`, to achieve calculating the Fibonacci value in parallel. The `async {block}` spawns a new thread to evaluate the following block and return immediately. So the evaluation of the block is done asynchronously. The `finish {block}` waits all `async` evaluations that occur in the block are finished. In the example, the Fibonacci (N-1) and (N-2) are calculated in two different threads. When all calculations are finished, the value Fibonacci N can be calculated.

The most important thing demonstrated by this example program is that the DSL achieves concurrent programming while the syntax is consistent with Ruby.

Distributed Programming Example

The example program 2 shows how to use distributed array to perform $1^2 + 2^2 + \ldots + N^2$ in parallel. In the DSL, we use Ruby-like syntax to express distributed constructs that are imported from X10 language. The basic constructs are `Point`, `Region`, `Dist` and `DistArray`. More details on distributed constructs are explained in later sections.

In this program, a distributed array is created and each element in the array is initialized

```
1   class Fib
2       attr_accessor :r
3       def initialize(x)
4           @r = x
5       end
6       def run
7           return @r if @r < 2
8           f1 = Fib.new(@r - 1)
9           f2 = Fib.new(@r - 2)
10
11          finish {
12              async { f1.run }
13              f2.run
14          }
15          @r = f1.r + f2.r
16      end
17  end
18
19  puts Fib.new(24).run
```

Example 1: Example of Concurrent Programming in DSL

```
1   dist = Dist.makeBlock(1..1000)
2   dist_ary = DistArray.new(:int, dist) { |pt| pt[0] }
3
4   sum = lambda {|x,y| x+y}
5
6   result = dist_ary.map{|i| i*i}.reduce(sum, 0)
```

Example 2: Example of Distributed Programming in DSL

to the value that equals to the element's index. The index of a distributed array is represented by the `Point` object in the DSL. Similar to creating a Ruby `Array` object, the `new` method of `DistArray` can take a block to perform initialization. Different from Ruby `Array` object, the `DistArray` can only contain same type objects. The first argument of `new` method is used to specify the type of the `DistArray`.

Other point showed in this example is that the function definition is the same to Ruby. By using the `lambda` syntax, users can create a function and then pass them around as value. In the program, 2 functions, `product` and `sum`, are defined. The `map` method on `DistArray` object will perform the passed function at each distributed places and generate a new array to store values. So the calculation of `product` is done in parallel. The `reduce` method perform the passed function on all points in a `DistArray` in an undetermined order (depends on X10's compiler) and return the result. Therefore, the `sum` function will sum all elements in the array.

3.2 Object Oriented Features

In this section, we describe the object oriented features that the DSL should support.

3.2.1 Classes, Inheritance and Open-class feature

The DSL support object oriented programming(OOP). The constructs for OOP are class. The usage class are the same as Ruby. Following sample code shows how to define a class with a method and then create a instance of that class.

```
class Foo       # Define a class.
    def foo     # Define a instance method.
    end
end
a = Foo.new     # Create a instance.
a.foo           # Invoke method on the instance.
```

Inheritance is also supported in the DSL. Same as Ruby, each class can has only one parent class. If the child and parent class both has a method with same name, the method defined in the child class overrides the method in parent class. Some other OOP languages also support method overloads, such as Java, X10, but the Ruby does not. Therefore, the DSL does not support method overload, too. Following DSL code continue the example above and shows how to inherit a class and override a method.

```
class Bar < Foo
    def foo     # Override Foo#foo method.
    end
end
```

Open-class is a feature that differentiate Ruby from other OOP languages. The DSL supports this feature. Open-class means that the class defined in a program can be changed at runtime. Users can append new methods or define new fields to a class that has already been loaded at runtime. Compare to Java, a loaded class is not allowed to be changed[*1]. Following DSL code clarifies what open class means.

```
a = "string"
a.foo           # Error! String class does have foo method.

class String
    def foo     # Append method foo to String class.
        "foo"
    end
end
puts a.foo      # Prints "foo".
```

3.2.2 Modules and Mixins

Modules and mix-in feature in Ruby is used to break through the limitation of single-root inheritance structure but also avoid the trouble of multiple inheritance. The DSL support these features and the usage is the same as Ruby.

Just like a class, a module can contain constants, methods and classes. But a module cannot be instantiated and inherited. To access the methods defined in a module, users

---

[*1] Bytecode instrumentation can bypass this limitation.

need to include that module into a class. After that, the class can use methods defined inside a module as they were its own members. A class can include multiple module. Following DSL code demonstrates the usage of modules and mixins.

```
1   module A            # Define a module.
2       def foo         # Define a method inside the module.
3           "foo"
4       end
5   end
6
7   class B
8       include A       # Include the module.
9   end
10
11  puts B.new.foo      # Prints "foo".
```

3.2.3 Type Annotation

Because Ruby is a dynamically typed language, there is not type information can be used at compile time. X10 is a statically typed language and it require type information to pass the compilation. Although dynamic typing can be handled by the DSL implementation, explicitly defining types can help to reduce the overhead of dynamic method dispatch at runtime. Therefore, the DSL provides `type_def` and `type_var` to annotate type in source code.

The `type_def` defines types for the next statement. Annotation in the following example, means that method `foo` takes 2 int type argument and return a int type value.

```
1   type_def :int, :int, :int   # DSL code to annotate type
2   def foo(a, b)
3       a + b
4   end
```

The `type_var` defines a variable that store the type values and can be used only inside code translator. Following code define a variable whose name is `a` and value is a list of types.

```
type_var :a, :int, :int, :void
```

3.2.4 Methods

We have shown how to define methods in classes or modules. This section gives more details and describe differences between DSL and Ruby on methods.

In Ruby, there are 3 kinds of methods: class/module methods, instance methods and singleton methods. In the DSL, we implemented the latter 2 types.

Instance Methods

The instance method is defined on a class and belongs to instances of the class. An instance method is also called class member because it is defined in the body of a class. The definition and usage of an instance method in DSL is same as Ruby. Following code shows how to define a method and invoke it in the DSL.

```
class A
    def foo(s)          # Define a method on class A.
        puts s
    end
end
A.new.foo("hello")      # Print "hello"
```

Singleton Methods

The singleton method is supported in the DSL. A singleton method is a method defined to a specific object rather than to a class. Many of the methods in the Ruby library are singleton methods. The syntax of singleton method in the DSL is same as Ruby. Here is an example that shows how to use singleton methods in the DSL.

```
class A; end
a = A.new

def a.foo       # Define a singleton method to the instance a.
    "foo"
end

class << a      # Other format for define singleton methods
    def bar
        "bar"
    end
end

puts a.foo      # Prints "foo".
puts a.bar      # Prints "bar".

A.new.foo       # Error! Method foo is not defined in class A.
```

Class Methods

The instance method is what we have seen in previous section. A class (or module) method belongs to the class/module itself. And users can invoke a class method by using the syntax `Classname.method`.

We temporally give up the implementation of class/module method because it is less used than the other 2 kind of methods.

Passing Arguments and Returning Values

In DSL, passing arguments into methods and returning values from methods are similar to Ruby but some features are not supported. We list these unsupported features below to avoid confusion.

1. Default arguments. In Ruby, users can specify default values for arguments. Currently, we does not implement it.
2. Multiple arguments. In Ruby, a method is capable of receiving an uncertain number of arguments. This action can be simulated by passing an array containing arguments to a method. However, we do not implemented it in the DSL.

3. Parallel assignment. In Ruby, the code `a,b = 1,2` will assign value 1 to variable a, 2 to variable b. We does not support this feature in DSL due to thread safety considering.
4. Returning multiple values. A method in Ruby can return more than one values. Returned values are wrapped into an array. Because the parallel assignment is not available in the DSL, the returning multiple values feature is less useful. So we does not implement it in the DSL.

All these features are useful. We plan to support them in the future.

Visibility

In Ruby, methods has 3 level of visibilities: `public`, `protected` and `private`. In the DSL, we support `public` and `protected` level.

As the name suggests, `public` methods are totally public and they are accessible outside the object in whose class methods are defined. They are default visible level in Ruby.

`private` methods are only accessible inside the object in whose class methods are defined.

`protected` methods are similar to `private` methods but the limition is a little loose. A `protected` method can be invoked 1)inside the object in whose class methods are defined, or 2) if the receiver object is an instance of the class who defined the `protected` method and the receiver object is inside the class's instance.

The `protected` level is equal to `private` level in the X10 language. The `public` level in Ruby and X10 are same. There are no corresponding level in X10 to Ruby's `private` level. The `private` level in Ruby is meaningless because users can access them by using the `send` method to invoke by name.

3.2.5 Blocks

In Ruby, a chunk of codes between curly brackets or `do..end` is treated as block. Blocks is known as anonymous lambda or anonymous function in other languages. The DSL support the block features and the usage of block is almost the same as in Ruby.

A block can take parameters like method. Users can also turn them into named `Proc` object. Users can `yield` a block or invoke `call` method on `Proc` object to execute it. And blocks are closures, which means they can store the values of local variables that are in the scope in which the block is declared. Following DSL code shows how to use them.

```
1    a = lambda { puts "a" }     # Literal Proc definision.
2    a.call                      # Prints "a"
3
4    def foo
5        yield 1
6    end
7    foo{|x| puts x}             # Invoke an anonymous block
```

In Ruby there are 2 syntax to create an `Proc` object, which are `lambda {block}` and `proc {block}`. However, in the DSL, we only support the `lambda` syntax to avoid confusion. The 2 kinds of block syntax are almost the same except for the `return` statement in the block has different meaning. The `return` statement in `lambda` block will return from the lambda to the place of invocation. However, the `return` statement in `proc` block will return to the location where the `Proc` object is defined.

3.2.6 Variables

The variables in the DSL are quite different from Ruby. First, the DSL has the conception of immutable variable, which Ruby does not have. Second, mutable class variable and mutable global variable in the Ruby are not supported in the DSL. Third, the DSL introduce the global reference conception from X10.

Immutable Variables

First of all, all variables in the Ruby are mutable. It means that a variable can be reassigned many times, even for constant. The DSL is differ from Ruby here by introducing the immutable variable conception. An immutable variable cannot be reassigned after initialization. Reassigning an immutable variable will get compile time error.

In the DSL, users can declare a variable is immutable by prefixing `val_` to a variable name. Similarly, by prefixing `var_` to a variable can make the variable is mutable. Further more, if the mutable variables' type can be confirmed, either by type annotation or type inference of translator, then reassigning to different type object is not allowed. Here is an example to use local variable in the DSL.

```
immu = 1
immu = 2 # compile error

var_mut = 1
mut = 2 # OK

type_def :int
var_mut = 1
mut = "string" # Compile error
```

The immutable/mutable rules for different kinds of variable in the DSL are:

1. Local variables are immutable by default.
2. Parameters are always immutable.
3. Fields (instance variables) are mutable by default.
4. Constant are always immutable.
5. Reassigning immutable variables gets compile error.
6. It is allowed to reassign different types value to a mutable variable only when the type cannot be confirmed at compile time. Otherwise, compile error occurs.

There are 2 reasons why we employ immutable variables in the DSL. First, X10 language distinguishes between immutable and mutable variables. When we implement the DSL in X10, we have to face this problem. It is convenient for code generation to employ immutable variable because the definition of immutable variables in X10 does not require type.

Second, the local variables and parameters are generally used as immutable. Reusing a local variable for different purpose is a bad programming habit and in practical it is rare. Therefore, we believe this exception would not effect the Ruby style programming.

Class Variables, Global Variables and Constants

There are class variable, global variable and constant in the Ruby language. However, in the DSL, these tree kinds of variables are less different.

In Ruby, a class variable (`@@var`) is defined for a class and is shared among all in-

stances of that class. A global variable (`$var`) is accessible from anywhere in the Ruby program. The scope of constant in Ruby depends on the declaration position. Constants declared within a class or module are accessible within the context of that class or module. Constants declared outside of a class or module are accessible globally.

In the DSL, first of all, mutable global or class variables are not supported. Because the DSL is supposed to support parallel programming and shared mutable variables are not thread safe. For the similar reason, in the DSL, reassigning a constant is not allowed, which is acceptable in Ruby language. Therefore, in the DSL, immutable global and class variables are less different from constant.

Moreover, because the DSL employ the PGAS moduel, so the meaning of global is different from Ruby. In the DSL, a global variable is only visible in the place (The place conception is explained in the section **??**) where the variable is defined but not available from entire program. To make a variable available from different places, users must use global reference that described in next section.

Following code shows how these variables are used in the DSL.

```
1  $g = "g"
2  $g = "f"        # Error! "global" variables are immutable.
3
4  C1 = "c1"
5  class Foo
6      @@cls = 1; C2 = "c2"
7      def foo
8          return @@cls
9      end
10
11     def bar
12         puts $g, C1, C2     # C1's scope is the same as $g
13     end
14 end
15
16 a = Foo.new; b = Foo.new
17 a.foo == b.foo      # true
18 a.bar               # print g, c1, c2
```

In brief, the syntax of global and class variable is preserved in the DSL but the usage of variables defined by these syntax is different from Ruby: 1) Variables defined like Ruby global variables and constants declared outside a class or module can be visible for any code in the place where they are declared; 2) Variables defined like Ruby class variables are visible among all instances of a class; 3) Constants declared inside a class or module can be visible within the context of that class or module. All these three kinds of variable are immutable.

Global References

The DSL employs the PGAS module that is used in the X10 language. The address space is partitioned. Each address partition is represented by `Place` object in the DSL. An object exists only in the place where it is declared and cannot be refered from other places. To make a variable referable cross multiple places, in the DSL, users must use the `GlobalRef` object to wrap the variable.

Underlying, a global reference (not the value wrapped in it) will be serialized by X10 runtime so that it can be transmitted to different places. However, to use the value that

Table. 3.1. Control Structures in DSL

category	syntax
Selection Structures	`case..when`; `if`; postfix `if`; `if..then..else`; `if..elsif`; `begin..end if`; `unless`; postfix `unless`; `unless..else`; `begin..end unless`;
Repetition Structures	`for..in` `while`; postfix `while`; `begin..end while`; `until`; postfix `until`; `begin..end while`; `loop` block
Exception Handling	`rescue`; `ensure`; `else`; `retry`; `raise`
Others	`redo`; `next`; `break`;

wrapped inside a global reference, users need to shift back to the place (Place changing is explained in section 3.4.2) where the global reference is created. The value can be retrieved by calling the `apply` method on the global reference. Following code shows the usage of global reference in the DSL.

```
s = "string"
gref = GlobalRef(s)     # Wrapper a object

at(gref.home) {         # Shift place
    s = gref.apply()    # Retrieve the value
}
```

3.3 Control Structures

Ruby has many control structures(about 13). They are all supported in the DSL. For clarity, in table 3.1 we summary the control structures that are available in the DSL.

Both Ruby and X10 are structured programming language, so the control structures in 2 languages are similar. For example, both languages support exception handling. So exception handling structures can be implemented in the X10 language.

One difference between our proposed DSL and Ruby is that the `redo` and `break` in the DSL can only be used inside a loop. In the Ruby, `redo` goes back to the begin of a block and `break` leaves the block. Because they may be misused to break the object initialization, we decide only allow them be used inside loop. In the Ruby and DSL, users can pass a block to the constructor to perform initialization. Bad use of `redo` or `break` in such blocks will cause the initialization never finish (`redo` cause dead loop) or finish unexpected (`break` leave block).

3.4 Parallel Programming Module

The parallel programming module in the DSL consists of primitives for concurrent programming and structures for distributed programming.

3.4.1 Concurrent Programming Primitives

The concurrent programming in the DSL can be achieved by using the following primitives instead thread and lock in Ruby. These primitives comes from X10 language. We express

them in a Ruby style in the DSL.

async {S} Spawn a new thread to perform S and return immediately. So the S is evaluated asynchronously. Different from the Thread object in Ruby, the `async` cannot be aborted or cancelled. And the `async` cannot return value. In following example code, method `foo` and `bar` are executed in parallel.

```
1   async { foo() }
2   bar()
```

finish {S} The execution of `finish {S}` will be blocked until all threads spawned by `S` are finished. It is similar to the effect of invoking `Thread#join` method on each thread defined in a block in Ruby. In following code, the method `bar` is executed only when the thread spawned in the second line terminate.

```
1   finish {
2       async { foo() }
3   }
4   bar()
```

atomic {S} The statement `S` in `atomic` block will be executed as one step. The execution cannot be interrupted. Users can use `atomic` blocks to guarantee that invariants of shared data. Following example show how to use `atomic` to achieve thread safety in the DSL.

```
1   @size = @size + 1             #not thread-safe
2   atomic {@size = @size + 1}  #thread-safe
```

when (cond) {S} : The execution of statement `S` will be suspended until the condition is satisfied. The condition evaluation is atomic with the execution of the block. Following code implement a block `pop` method by using `when` statement.

```
1   def pop()
2       ret = nil
3       when(@ary.length > 0) {
4           ret = array.delete_at(0)
5       }
6       ret
7   end
```

Notice that because the X10 contains these primitives, so the implementation is just simply mapping DSL syntax to corresponding X10 code.

3.4.2 Distributed Programming Structures

We introduce following structures from X10 into the DSL to express distributed program.

Place The `Place` object represents a virtual computing resource. A place approximately equal to a process. For example, it may represent a core in a SMP environment or a node in a cluster environment. The variable access in local place is always synchronized. And reference to remote place requires asynchronous action. Communication may occurs for object access between different places. The `Place` object cannot be created at runtime by users. The `Place` is the concrete programming structure to express PGAS module.

Point The `Point` object is used to represent the index of an element in an distributed array. In the DSL, we use the same syntax as Ruby array to access coordinate of

a point. For example, following DSL define a 3-dimension point and access to the first coordinate and the rank(n-dimension) of the point.

```
1    p = Point.new(4,5,6)
2    p[0]    #=> 4
3    p.rank  #=> 3
```

Region The `Region` object represents a collection of `Point` objects. It is used to control the shape of an array. In the DSL, the `Region` is represented by using the same syntax as the `Range` object in Ruby(`(n .. m) defines a Range object from n to m`). Multiple dimensions region can be expressed by using product operator(`*`). Following DSL code define a 2-dimension region and iterate over the point in the region.

```
1    r = (1..3) * (1..3)     # Define a Region object
2    r.each do |pt|
3        puts pt             # Print [1,1], [1,2],[1,3], ...
4    end
```

Dist The `Dist` object represents a distributed object that specify the mapping between points and places. It is used to control what computing should be performed at which place. In the DSL, users cannot define their own `Dist` object. There are three kinds of predefined `Dist` object and users can get instance through factory methods: `makeBlock`, `makeUnique`, and `makeConstant`. Following DSL code creates a `Region` object and distribute points to different places.

```
1    r = (1..2) * (1..2)     # Define a Region object
2    d = Dist.makeBlock(r)   # Suppose there 2 places, then
3                            # Points [1,1],[2,2] => place1
4                            # Points [2,1],[2,2] => place2
```

DistArray The `DistArray` is used to represent an array that contains objects across multiple places. It has three basic methods, `scan`, `map` and `reduce`, to perform action over elements in the distributed array. To create a `DistArray` object in the DSL, users can use the constructor shown in following code. Different from general `Array` object in the DSL, the `DistArray` object can only contain objects that have the same type. The first parameter in the constructor specifies the element type.

```
1    d = Dist.makeBlock(1..5)
2    # Create a DisArray object and
3    # initialize the element by the index value.
4    dis_ary = DistArray.new(:int, d) { |pt| pt[0] }
```

at (place) S The `at` primitive is used to perform place shifting. In the DSL, code can only directly access variables defined in the place where the code is evaluated. Therefore, to access variables defined in other places, must change the place. Suppose there are 2 places p1 and p2 and current the code is execution at p1. The code `at (p2) {S}` will change place to p2 and evaluate `S` at p2. Finally return the value back to p1. The `at` is synchronous construct.
Notice that the mechanism underlying the place shifting is serialization. X10 will perform a deep copy of the value transmitted. So the variable referred inside `at` block is different from the object referred by local variable, although the variable names are the name.

Similar to the concurrent programming primitives, the implementation of distribution programming model in the DSL is simply converting the DSL code to corresponding X10

object.

3.5 Features Summary

In this section, we summarize the features supported in the DSL. Table 3.2 compare the features among Ruby, X10 and the DSL.

From the results we can know that the feature defined for DSL are basically the same to Ruby and plus concurrent and distributed structures that we learn from X10. The syntax of the DSL are consistent with Ruby's. Therefore, Ruby users won't feel discomfort when programming with the DSL.

In addition to some detailed syntax in the DSL are different from Ruby, the dynamical evaluation is not supported in the DSL. Dynamic evaluation means a string can be evaluated as Ruby code by interpreter at runtime. To support this feature, need an available Ruby VM at X10 runtime environment. Compare the implementation cost to the benefits of dynamic evaluation, we decided to drop this feature in our DSL.

Table. 3.2. Features comparison among Ruby, X10 and the DSL(– stands for unavailable)

Feature	example in Ruby	example in X10	example in DSL
		Object Oriented Features	
Class	`class A`	`class A`	`class A`
Headerless class	–	`struct A`	–
Module	`module B`	`abstract class B`	`module B`
Interface	–	`interface A`	–
Inheritance	`class A < String`	`class A extends String`	`class A < String`
Mix-in	`include B`	–	`include B`
Open-class	(support)	–	(support)
Final-class	–	`final class A`	–
Visibility	`public`	`public`	`public`
	–	`protected`	–
	–	`package`	–
	`protected`	`private`	`protected`
	`private`	–	–
		Methods	
Singleton method	`def a.foo`	–	`def a.foo`
Class method	`def A.foo()`	`static def foo()`	–
Default arguments	`def foo(a=1)`	–	–
Multiple arguments	`def foo(*a)`	–	–
Parallel assignment	`a,b=1,2`	–	–
Return multi-values	`return 1,2`	–	–
		Variables	
Immutable	–	`val a = 1`	`a = 1`
Mutable	(always mutable)	`var a = 1`	`var_a = 1`
Local	`a = 1`	`val a = 1`	`a = 1`
Instance	`@a`	`var a`	`@a`
Class	`@@a`	static val a	`@@a`(immutable)
Global	`$g`	–	`$g`(immutable)
Constants	`Cons`	`static final val`	`Cons`
		Literal expressions	
Sring	`"string"`	`"string"`	`"string"`
Number	`123; 1.23`	`123; 1.23`	`123; 1.23`
Regexp	`\regexp\`	–	`\regexp\`
Array	`[1,2,1+2]`	`[1,2,1+2]`(fixed size)	`[1,2,1+2]`
Hash	`{1=>"a"}`	–	`{1=>"a"}`
Block	`{|i|body}`	`(a:int)=>{body}`	`{|i|body}`
Region	–	`(1..3)*(1..3)`	`(1..3)*(1..3)`
		Concurrent Programming	
Thread	`Thread.new{}`	`async{}`	`async{}`
Lock	`Mutex.new`	`atomic{}`	`atomic`
		`when(cond){}`	`when(cond){}`
	`Thread#join`	`finish{}`	`finish{}`
		Distributed Programming	
Point	–	`new Point(1,2)`	`Point.new(1,2)`
Region	–	`Region.make(1,3)`	`Region.new(1,3)`
Dist	–	`Dist.makeBlock`	`Dist.makeBlock`
DistArray	–	`new DistArray`	`DistArray.new`
Place	–	`at(place) {}`	`at(place) {}`
		Dynamically Evaluation	
eval	`eval "1+1"`	–	–

Chapter 4

System Design and Implementation

4.1 System Overview

The core components of the DSL implementation consist of the following parts:

An abstract Syntax Tree and AST builder The C Ruby release contains a Ruby syntax parser `ripper`. However, the output of the `ripper` is a parse tree. For convenience of later code generating, we define an AST and develop a AST builder which consume the parse tree and generate the AST.

A code generator The code generator output X10 source code from based on the AST. The assistant system consists type infer, return statement infer and scope checker. First of all, the assistant system will analysis the AST and append necessary information to it. Then the code generator will emit X10 code from the AST.

Runtime libraries The generated X10 code should be compiled by `X10c` with the runtime libraries together in X10 environment. The runtime libraries contains the extension functionalities we developed for X10, for example, the reflection feature, the array and hash object, the regular expression libraries, etc.

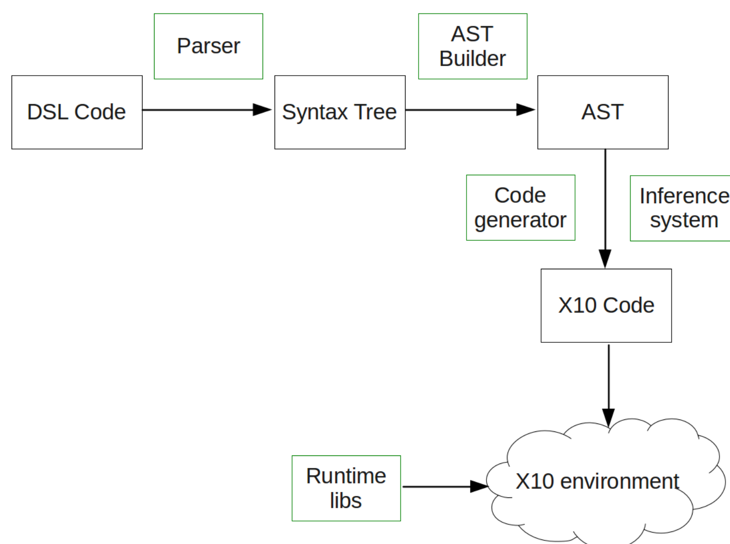

Fig. 4.1. Structure of DSL implementation

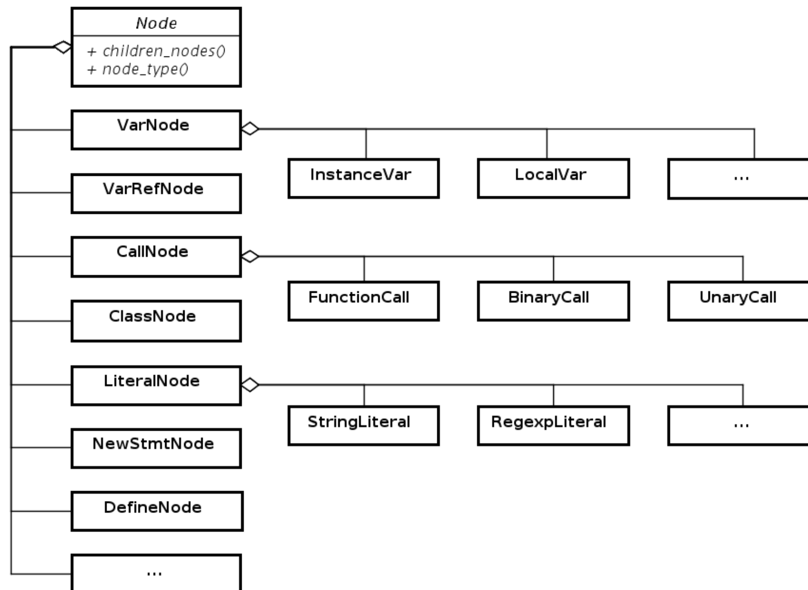

Fig. 4.2. Layout of AST

4.2 Parsing and AST

In this section, we will explain how we parse the source code of DSL and to build an abstract syntax tree.

4.2.1 Parsing

C Ruby 1.9.x release contains a parser for Ruby script – `ripper`. We use the `ripper` to generate a parse tree (also known as syntax tree), and then travels the parse tree to generate an abstract syntax tree (AST).

The `ripper` is a event driven parser. It has some predefined events, such as `stmts_new` which means starting a new statement, `string_literal` which means a literal string statement and so on. The `ripper` steps through a Ruby script and when meets a predefined event, it tries to invoke method associated to that event. In such a way, users can register methods to interested events to get meta data, such as the line number, the column number, the token name, etc.. With these data, it is easy to construct a parse tree, in which the input symbols are grouped into subtrees. Subtrees represent the structure of phrases.

Because the parse trees are specific to grammars and the grammar of Ruby is very flexible, it turns out the parse tree is full of noise. For example, the conditional statement in Ruby has if, postfix if, unless and postfix unless 4 different formats. And each format generates different parse tree. So the parse trees are inconvenient to walk and transform. To overcome this problem, we generate a abstract syntax tree from the parse tree.

4.2.2 AST Layout

The UML in figure 4.2 shows the basic layout of we defined AST. The root object is named as Node, in which some basic methods are defined. Other main nodes are explained in

Table. 4.1. Main AST Nodes and Explanation

Node	Explain
VarNode	Represents definition of a variable.
VarRefNode	Represents a reference to variable.
ClassNode	Represents definition of a class.
DefineNode	Represents definition of a method.
NewStmtNode	Represents a single statement(logically).
CallNode	Represents invocation of a method.
LiteralNode	represents literal expressions.

the table 4.1.

4.2.3 Generating AST

The AST builder is implemented by using the external tree visitor pattern[26], in which the tree walking and action execution code are separated from the AST node definitions. The visitor object in this pattern is generally implemented with object-oriented feature such as method overload and need a big switch case to switch on token type. For example, to execute code on a node, the visitor object would be implemented as (in Java):

```
#Java code
public void parse(Node n) {
    switch (n.token.type) {
       case Token.ASSIGN:   parse(AssignNode)n); break;
       case Token.ID:       parse(VarNode)n); break;
       //and so on ...
       default:
            throw new Exception("Unsupported Node type");
    }
}
```

However, thanks to Ruby's introspection features, we can avoid the big switch case by invoking method by name. The visitor object in Ruby will be implemented as following:

```
def parse(node)
    __send__("parse_" + node.node_type, node)
end
```

This implementation is conciser than the former one that dispatch with switch case. And the tree walking can be simply implemented by recursively invoke the parse method in visitor.

4.3 Converting DSL to X10

In this section, we describes how DSL's features are implemented in X10 language.

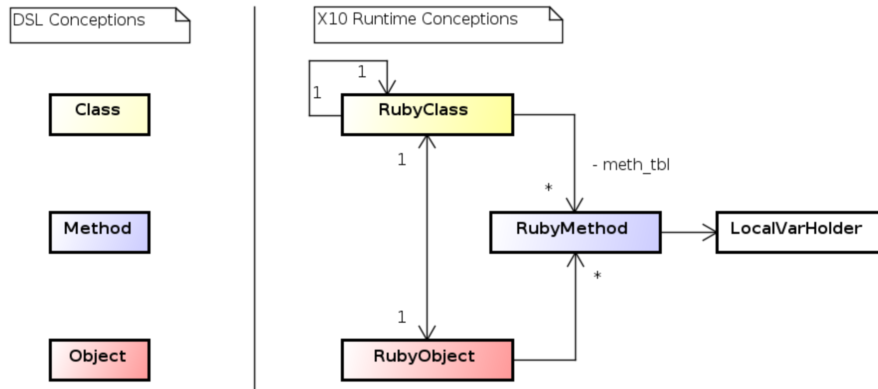

Fig. 4.3. Object Oriented Conceptions Mapping between Ruby and X10

4.3.1 Classes, Objects and Methods

In this section, we describe how we convert DSL classes, objects and methods into X10. They are the cornerstone of implementing object oriented features. To put simple, we define `ruby.RubyClass`, `ruby.RubyObject` and `ruby.RubyMethod` to represents DSL classes, objects and methods, respectively. The diagram 4.3 shows the relationships.

Classes

We cannot directly convert DSL class to X10 class. The classes in X10 are static. The methods and fields defined in a class are fixed at runtime. But classes in DSL can be dynamically modified. Users can add or change methods or fields in a class at runtime. Moreover, the static class feature of X10 comes from Java. X10 is implemented by Java language and run on JVM. JVM does not allowed to modify a loaded class.

We defined an X10 class (`ruby.RubyClass`) to represent DSL class. The parent class is hold by one field in the `RubyClass`. Therefore the instances of `RubyClass` can be chained together to implement the hierarchy of class inheritance. `RubyClass` also hold a method table that is a hash table with method names and method as key/value pairs. Therefore, each class defined in our DSL is converted to an instance of `RubyClass`. And modifying methods in a class is equal to adding or updating method table entries.

Objects

Like all Ruby objects are inherited from a Ruby class named `Object`, all DSL objects are inherited from an X10 class named `ruby.RubyObject`. The `RubyObject` has a field that store the class object(`RubyClass` object) of the object. So the class-instance(object) relationship can be satisfied. The `RubyObject` also has a methods table and a fields table. The methods table is same as the one in `RubyClass`. The fields tables is a hash table that maps the instance variable names to values. The method table in the `RubyObject` is used for implement singleton method. We will describe it later.

Creating an object in DSL is converted to create an instance of `RubyObject`, set the object class and initialize the filed table.

Methods

DSL methods are represented by X10 class `ruby.RubyMethod`. `ruby.RubyMethod` is implemented as an abstract class. Each method in DSL corresponds to one subclass of

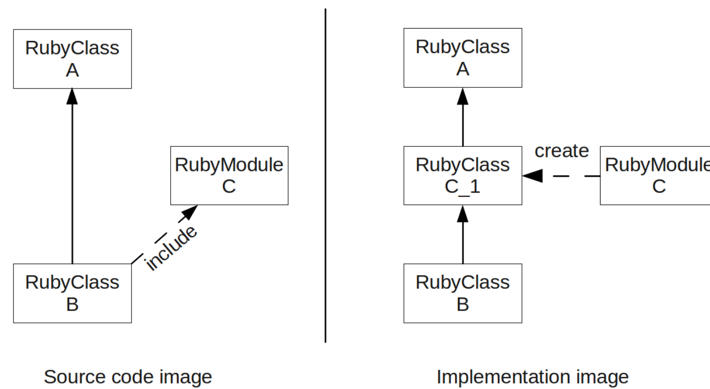

Fig. 4.4. Implementation of mixin

RubyMethod. Singleton pattern is employed to make sure there is only one instance for each class. Therefore each method at runtime is unique. The instances of RubyMethod are stored in methods table of RubyClass or RubyObject.

Method objects can be invoked by invoke method that defined in RubyMethod class. Because the method object does not belong to any class, the first argument of invoke methods must refer to the receiver object. The arguments of a method invocation can be passed to the invoke method by wrapping in a list. To save the overhead of creating list, and decompose arguments from the list, the RubyMethod class defined a set of abstract invoke methods that take from 0 to 10 arguments. For the method with more than 10 arguments, there is a general purpose invoke method that takes an array as argument. The subclass of RubyMethod can override a invoke method according to the number of argument it required.

When store into the table method, the method name is mangled to be used as key. The rule of name mangling is that method name follows a $ character and follows the number of arguments. Because the DSL does not support method overload, the method name with argument number can make sure the mangled name is unique in side a class.

4.3.2 Mixins and Open class

Before explain how mixin and open class feature are implemented, we need to clarify how method is resolved.

Resolving Methods

The procedure of finding the correct method to be invoked is known as resolving a method. The rules for resolving a method in our implementation is simple.

First find the method on the receiver's method table. If not found,

Next retrieve the receiver's class object and then find method on the class object. If still not found,

Finally repeatedly find method on the class' parent class until reach the root class whose parent field refer to null object.

With the knowledge of how a method is resolved, it is easy to understand how to implement override, mixin and singleton class.

Mixin

In DSL, a class can include modules to bring methods that are defined in modules into the class' scope. Similar to DSL classes, DSL module is represented by an X10 class `x10.RubyModule` that maintain a method table. `RubyModule` can generate `RubyClass` object that contain a reference to the `RubyModule` object's method table. Therefore, mixin is implemented by inserting the `RubyClass` object generated from module object into the class's inherited class chain. Figure 4.4 shows the image of mixin. Notice that, mixin a module, won't change the class' parent class.

Open Class

Open class feature represents as at runtime 1) defining new method to a class and 2) defining new method to an object. The later one is also known as singleton method. Because the implementation approach behind are similar, here we only explain the singleton method.

In DSL, singleton methods are defined on an object but not in class. So other instance of the object's class, cannot use these methods. Singleton methods is easy to implement with our class structure. Because each instance of X10 `RubyObject` class corresponds to a object in DSL, defining singleton methods just means adding methods to the method table hold by `RubyObject` object. The translation between singleton method definition in DSL to add method in method table is handled by code converter automatically.

4.3.3 Method Invocation

Method invocation in our implementation is handled by 2 ways. First, with dynamic typing, methods are invoked dynamically by the method name. Second, with type annotation, methods are called directly.

Invoking by Name

Ruby is a dynamically typed language, so no type information is available at compile time. When compile the DSL code to X10, the converter has no idea on the object's type and what methods are available. In this case, method is invoked by method name. Here is an example.

```
1    a + b   # DSL code
2    invoke$(a, "sum", b); # Generated X10 code
```

The `invoke$` method will first resolve the method through its name as described in section 4.3.2 and retrieve a `RubyMethod` object. If the method is not found, then an `UndefinedMethodException` will be throw. If the method is found, then execute the method by calling `invoke` method defined on the method object.

Directly Invoking

Dynamic invocation is flexible, but slow. Each time to invoke a method, must first find the method and then invoke it. To speed up the method invocation, the converter will generate native X10 method and call them directly when following conditions are all satisfied.

1. The type of object is known at compile time. The converter can infer some simple types. And for those that cannot be inferred, users can use type annotation.
2. The method invoked on the type is visible at compile time.

3. No open-class feature is used on the class.

■Type Annotation To know the type information at compile time, there are 2 approaches. First is type inference. Concurrent implementation of converter perform easy type inference from assignment. For example, given DSL code `a = 1`, then the converter can infer that variable `a` is an integer. Because it is impossible to completely infer type, currently, we make little effort on type inference.

Second, to use type annotation to mark type explicitly. This is a very straightforward solution. It gives programmers opportunity to use annotation to attach type information in the DSL code. We defined the following two syntax to annotate type as mentioned in previous chapter.

```
1    type_def <types>, <type2>, ...
2    type_var <var_name>, <type1>, <type2>, ...
```

The `type_def` will define type(s) for the next statement, either a method definition or a variable assignment. To define types for a method definition, the arguments of `type_def` will be treated as types for the method arguments and return value sequentially. Here is an example.

```
1    type_def :int, :int, :int   # DSL code to annotate type
2    def foo(a, b)
3        a + b
4    end
5
6    def foo(a:int, b:int):int { # Generated X10 code
7        return a + b
8    }
```

The `type_var` is similar to `type_def` but define a variable whose value is types. It is used to avoid repeatedly annotating the same type. For example, following DSL code equals to the above code.

```
1    type_var :a, :int, :int, :void
2    type_def :a
3    def foo(a, b); end
```

Obversely, annotation is easy to implement but dirty the code. It also against the Ruby's philosophies that to make code as concise as possible.

■Check Open-class Feature Even if the type is known at compile time, it is not enough to call method directly. Because the open-class feature, the method invoked may does not exist at compile time. In this case, the converter will generate X10 code to invoke the method by its name.

Furthermore, even if the method is visible at compile time, it is still not safe to call method directly. Because the method may be overridden at runtime. To handle this problem, the converter will check whether open-class feature is used on a class and set a flag in each generated class to indicate whether the open-class feature is used. The direct invoked methods will check this flag before execution. If the flag is true, which means open-class feature is used on the class, and then the method invocation will fall back to dynamic invocation.

4.3.4 Local Variables

Basically, the local variable in the DSL will be converted to X10 local variable. However, there 2 special case need to handle: 1) immutable/mutable local variables and 2) local variable enclosed in the block.

Immutable Local Variables

As mentioned in the chapter 3.2.6, in our DSL, the local variables are immutable until it is explicitly declare as a mutable variable. The mutable variable is declared by prefixing `var_` to the variable name. The type of value that is reassigned to the mutable variable will be checked. If the type is different from previous, compile time exception will be thrown. Reassignment to a immutable value will get a error at compile time.

Enclosed Local Variables

In Ruby, the local mutable variable is accessible in the block but in X10 it is not. In X10, only immutable local variable or mutable filed can be access in a block.

In the DSL, we follow the Ruby's convention that mutable variable should be accessible inside a block. So instead store local variable on stack, we store the local variables that are used inside a block on heap.

First of all, the code generator will check how the local variable is used in the code. If it finds mutable variables defined outside a block are used in the block, it will mark the variable to be replaced. Next, the code generator will define an inner class, which has a public field, inside the class in which the replaced variable lives. And then, the definition of the mutable variable will be replaced by creating an instance of the inner class and assign the value to the public file inside the inner class. Finally, all access to the mutable variable will be replaced by the reference to the public field. Following is a pseudo code show the implementation.

DSL code:	Generated X10 code:
	`static final class valueHolder { public i:Any; }`
	`val ins = new valueHolder();`
`var_i = 1`	`ins.i = 1;`
`finish async {i = foo()}`	`finish { async { ins.i = foo(); }}`
`i += 1`	`ins.i += 1`

4.3.5 "Global" Variables

First of all, as mentioned in chapter 3, the global variables here are not really global but just accessible in local place. However, we still use the name for convenience.

The Ruby's global variables and constant defined outside any module or class, are visible to all code in the place where the variable/constant are declared. Moreover, the global variables and constants are immutable for the sake of thread safe.

Global variables and constants are compile to X10's `public static val` variables that contain in a `public static final` class. The class has a private constructor so that no instance can be created. Here is an example.

```
#DSL code
$value = 1
Const = 2

# Generated X10 code
public final class GlobalVar {
    private def this () {//no instance}
    public static val value = 1;
    public static val Cons = 2;
}
GlobalVar.value = 1;
GlobalVar.Cons = 2;
```

4.3.6 Wrapper Classes for Built-in Types

Ruby has some built-in types, such as Fixnum, String, etc.. In our implemented runtime, these built-in types are wrapped to objects that are inherited from `RubyObject`. For example, the X10 `ruby.rtt.RubyFixnum` wraps the Ruby `Fixnum` object and `ruby.rtt.RubyString` wraps the Ruby `String` object.

Providing the built-in type wrappers has 2 advantage. First, all runtime object are a subclass of `ruby.RubyObject`. It is convenient for managing objects in runtime. Second, we can simulate the behavior of Ruby built-in types. For example, string object in ruby is mutable whereas string in X10 is immutable. Methods that change the string itself, such as `gsub!`, `reverse!`, etc., can be simulated by changing the underlying value field in the `RubyString` object. Currently, we provide wrapper class for `String`, `Fixnum`, `Regexp`, `Match`, `Array` and `File` type in Ruby.

Moreover, we also provide wrapper classes for some of types defined in X10 language, such as `Point`, `DistArray` and so on. These wrapper class define some new API to make usage of them is like to Ruby. For example, Ruby users are used to use `each` method to iterate an array. So the `DistArray` wrapper class provides the `each` method.

4.3.7 Control Structures

Most of control structures in DSL can be simply mapped to X10's controrl structures. However, there are 2 exceptions. First is `case...when` selection structure. Second is exception handling.

Implement case..when

`case..when` is similar to X10's `switch..case` but much more powerful. The `switch` statement in X10 is same as C language that the switch condition can only be integer value. However, the `case` statement can accept any value, such as string or regular expression. Moreover, each `when` statement can have multiple values. If any of the value match the `case` condition, then the following block will be evaluated.

So `case..when` in Ruby cannot be mapped to `switch..case` in X10. We convert the `case..when` into `if..else if..else` block. Each `case` statement is converted to a `else if` statement. Multiple match values are converted to multiple if conditions connect with `OR` operator.

Exception Handling

X10 provides exception handling primitives and basic functionalities are similar to Ruby. The `begin..rescue..ensure` in Ruby is converted to `try.catch..finally` in X10. Multiple exceptions in one `resuce` statements is translated to multiple `catch` statements with same block.

However, Ruby code can use `retry` to go back to the start of block and use `else` to run code when no exception occurs in block. These two primitives are implemented in X10 by wrapping the `try..catch` block in a loop and using labeled `continue` and `break` statements.

4.4 Parallel Programming Model

In this section, we shows how parallel programming model is implemented in the DSL.

4.4.1 Concurrent Programming Primitives

In our proposed DSL, we represent some X10 primitives (`finish`, `async`, `when` and `atomic`) with Ruby syntax to replace the thread and lock for concurrent programming. Thanks to the Ruby syntax feature that a method invocation can omit parentheses of arguments and can be attached with a bock, we can represent new primitives in a way that looks like build-in features for Ruby. For example, following DSL code represents the `async` usage.

```
async { 1 + 1 }
```

In fact, Ruby interpreter treats the code as a method invocation with no arguments, as following:

```
async() { 1 + 1 }
```

So the code translation is a easy. By using pattern match, the code generator filters out these special methods from general method invocation and then emit corresponding X10 code. This kind of representation is also close to the grammar in X10. So either Ruby users or X10 users can adapt to the syntax easily.

4.4.2 Distributed Programming Structures

The distribute programming model in the DSL is achieved by using 4 new type objects – `Point, Region, Dist` and `DistArray` The construction of object follows the Ruby convention, which is calling the `new` method on the class and the initial status can be set by the attached block. For example, following code will create a distribute array over a 4x4 matrix and initialize all elements to 0.

```
1    val region = (1..4) * (1..4)
2    DistArray.new(:int, region) { 0 }
```

Three things need to notice. First, different from literal array, which is translated to `RubyArray` runtime type, the `DistArray` can only contain same type objects and the type is specified by the first argument. Second, to emit X10 code, code generator will translate the initialization block to a function object in X10 and then pass it to the constructor as an argument. 3) the literal for Region object is implemented by asterisk production of two range literal as showed in the first line in the above example.

Chapter 5

Evaluation

This chapter presents a evaluation of the DSL. The focus of the evaluation is on the productivity and the scalability.

5.1 Productivity Evaluation

We evaluate the productivity of the DSL from 2 aspects: a) the resemblance between the DSL and Ruby, and b) the code size of the DSL program.

5.1.1 Resemblance to Ruby

The major target of the DSL is to enable Ruby users to write parallel program in Ruby style and one of the design principles is to make the DSL as resemble as possible to Ruby. If the DSL is resemble to the Ruby, it is easy for Ruby users to learn and use the DSL. Hence, the likeness to Ruby benefits the productivity of write parallel program.

The likeness to Ruby is measured by comparing the main features of the DSL to the Ruby language. Table 5.1 shows the major functionalities of Ruby and the support status in the DSL. As we state in the chapter 3, the dynamic evaluation is impossible to implement without a Ruby VM at runtime and the DSL does not support it. Other features are all supported by the DSL.

Table. 5.1. Functionalities supported by DSL

Ruby Features	DSL Support
Dynamic evaluation	No
Open class	Yes
Singleton class	Yes
Dynamic typing	Yes
Mix-in	Yes
Blocks and lambda	Yes
Literals	Yes
Exception and GC	Yes
Object-Oriented	Yes
Implicit return	Yes

This result shows that users can write the DSL as they write Ruby script. Little effort is required for learning the DSL.

5.1.2 Code size

By measuring the code size, we can measure the effort required for writing a parallel program. A program with a smaller code size will require less effort to develop and maintain. Moreover, to achieve a specific functionality, smaller code size means better expressiveness of the programming language. Therefore, the code size can be used for productivity analysis.

We implement 5 benchmark programs with both X10 and the DSL and then compare the code size of each implementation. The benchmarks we used are 4 benchmark programs from X10 release and 1 benchmark program from Ruby release. The 4 benchmarks from X10 are:

HeatTransfer A program that simulates 2-dimension heat transfer problem by using the SPMD style implementation.

FSSimpleDist A program that implement the STREAM problem in the HPCC benchmark[27] with distributed array.

NQueenPar A program that solves N-Queen problem is implemented with control flow parallel style.

NQueenDist This program is identical to NQueenPar except that it is implemented with distributed array.

We implement these benchmarks in the DSL.

The benchmark from Ruby release is `regex-dna` that reads in some DNA sequences and then uses regular expression to count matched patterns in the DNS sequence. It is original a sequential program, we implement the parallel version with the DSL and X10.

We use 2 metrics for measuring code size:

Byte size Measuring the size in bytes that a program source-code file occupies after remove comments, duplicate whitespace characters and then compress with GZip.

Effective Lines of Code(eLOC) The lines of code but does not count stand-alone braces {} or parenthesis ().

The byte size of source code shows the real quantity of code for programmers to input. It may be effect by the programming convention, for example, how to name a variable. In the evaluation, to reduce noisy, all function name, variable, class name are same in different implementations.

The eLOC is used widely. Comparing to the source line of code (SLOC) the eLOC is more precise. It is defined by Resource Standard Metrics(RSM)[28] as a realistic code metric independent of programming style. For example, given following code segment, eLOC will be 2 but SLOC will be 4.

```
1    if (x<10)
2    {
3        //comment
4
5        y = x + 1;
6    }
```

The result of code size evaluation shows in table 5.2.

Table. 5.2. Code size comparison for X10 and DSL

Benchmark	Code size metric	X10 impl.	DSL impl.	DSL(annotation)
Heattransfer	byte size	559	536	578
	eLOC	21	24	26
FSSimpledist	byte size	575	438	459
	eLOC	35	31	31
NQueenDist	byte size	866	708	756
	eLOC	57	51	56
NQueenPara	byte size	812	625	684
	eLOC	52	46	51
regex-dna	byte size	412	222	222
	eLOC	27	16	16

5.2 Scalability Evaluation

Scalability for parallel program refers to the ability that decrease the program execution time by involving more processors. It is critical for parallel program and the major motivation for programming in parallel. This section represents a evaluation of the scalability for the program implemented with DSL and compare to the X10 implementation.

5.2.1 Evaluation Environment

All evaluation were performed on a system with 4 Intel Xeon(R) 2.40GHz CPUs that each CPU has 6 cores. The system has 16GB memory. The OS is 64-bit Linux with kernel version 2.6.26. The JDK is ORACLE JDK 1.6.0_21. The GCC version is 4.3.2. The X10 language version is 2.1.0. The execution time is measured with the `time` command.

We used 3 benchmarks: 1) NQueensPar, 2) NQueensDist, and 3) the regex-dna. All of them are described in previous section. We measure the wall-clock execution time of the X10 and the DSL implementation for each benchmark. For each benchmark, we measure following 3 cases:

- X10 implementation.
- DSL implementation without dynamic typing.
- DSL implementation with dynamic typing.

To show the scalability, the computing set for the program must big enough. After all, short running program does need to bother with parallelization. For the NQueenPar benchmark, we calculate the problem with N = 12. And for the regex-dna, the program matches 20 patterns on a 49MB input file.

The X10's current Java back-end implementation does not support real distributed execution. All calculation are performed in one JVM. So the distribution in benchmark NQueensDist is simulated on the multi-core environment. This simulation cannot show real execution time but can reflect the scalability. So it won't be problem in our evaluation.

5.2.2 Results

Figures 5.1, 5.2 show scalability of NQueensPar and NQueensDist benchmark. We can see, both parallel and distribute implementation have good scalability. For parallel style

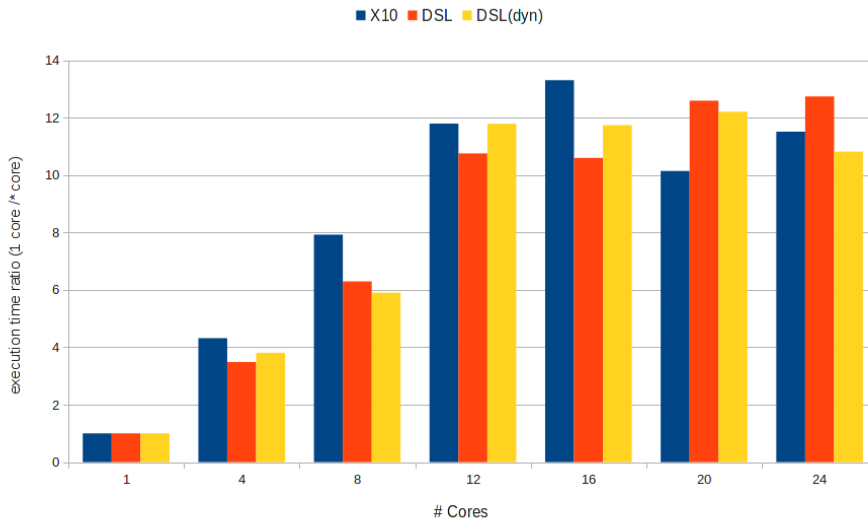

Fig. 5.1. NQueensPar Benchmark (N = 12)

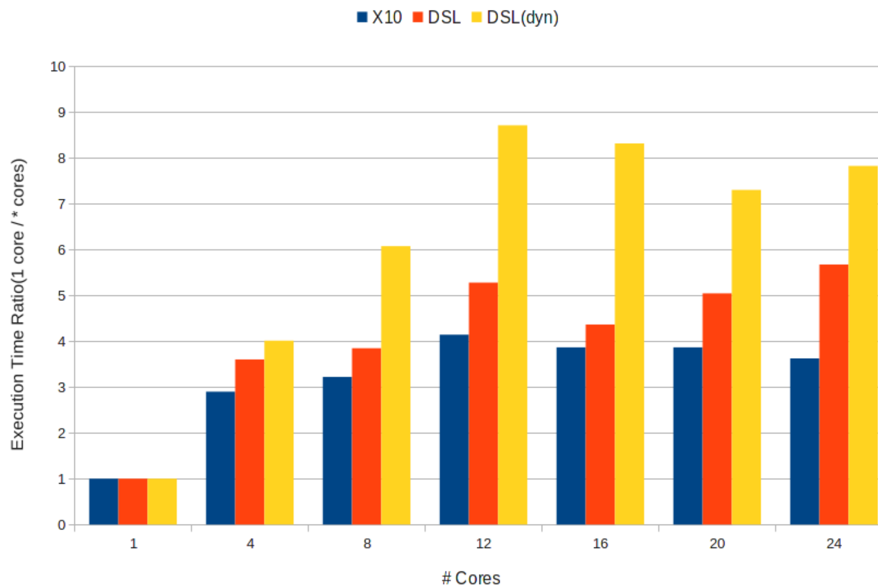

Fig. 5.2. NQueensDist Benchmark (N = 12)

implementation, the best performance comes when the number of cores are 16. And for distributed style implementation, the best performance comes when the number of cores are 12. The DSL implementations of these 2 benchmarks are slower than the X10 implementation. However, scalability is similar.

Figure 5.3 shows the scalability of regex-dna benchmark. The figure shows the execution time ratio of sequential Ruby version to parallel X10 and DSL versions. Execution with 1 core is slower than sequential Ruby version. As more cores are used to computing, the parallel version becomes faster than sequential version. The best performance comes when there 8 cores to compute. The benchmark does not scale well.

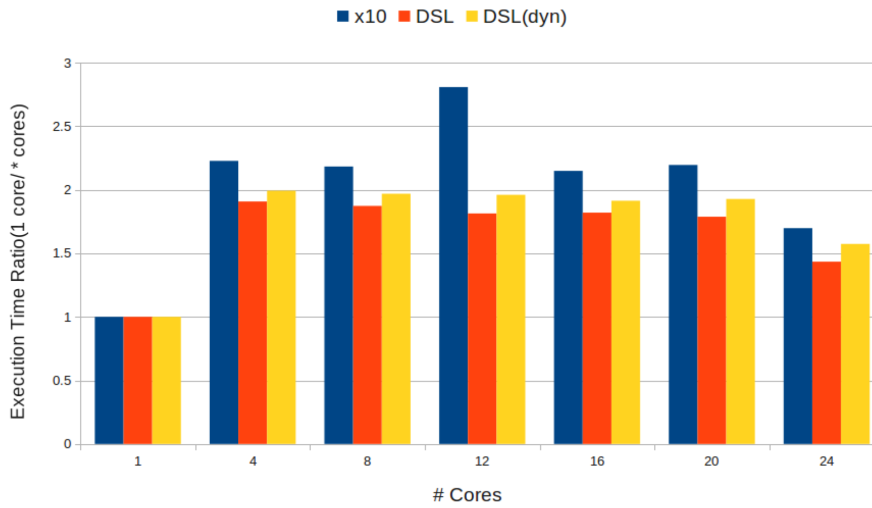

Fig. 5.3. Regex-dna Benchmark

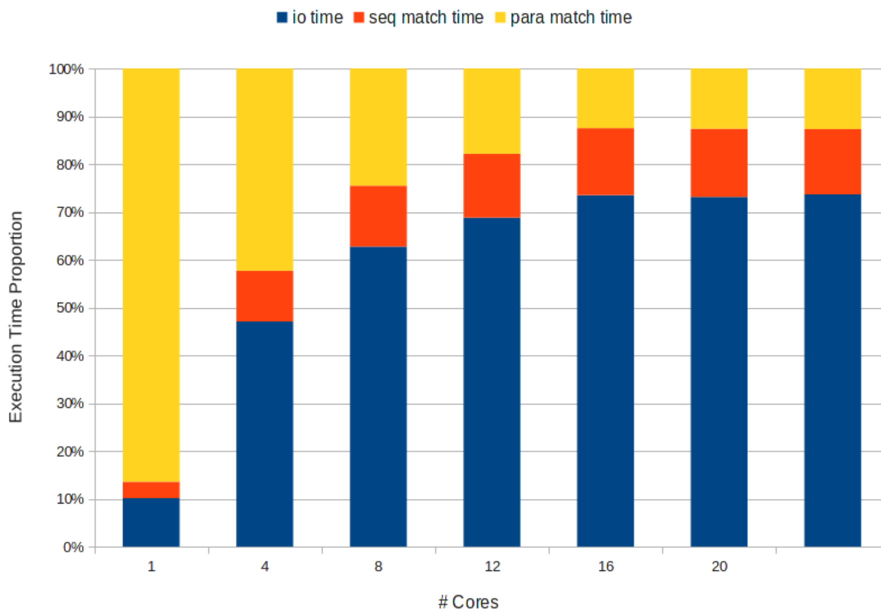

Fig. 5.4. Time Consuming Proportion

5.2.3 Scalability Analysis

To figure out why the regex-dna benchmark does not scale well, we measure the execution time of each step in the program. The regex-dna program redirects a large DNA sequence from `stdin` and uses some regular expression patterns to find how many are matched in the input DNA sequence. The program consists of 3 steps: I/O reading, formatting the read string and regular expression matching. Only the 3rd step is implemented in parallel.

Figure 5.4 shows the execution time proportion of each steps. From the figure, we can know, although the regular expression match step gets speedup as more cores are used to

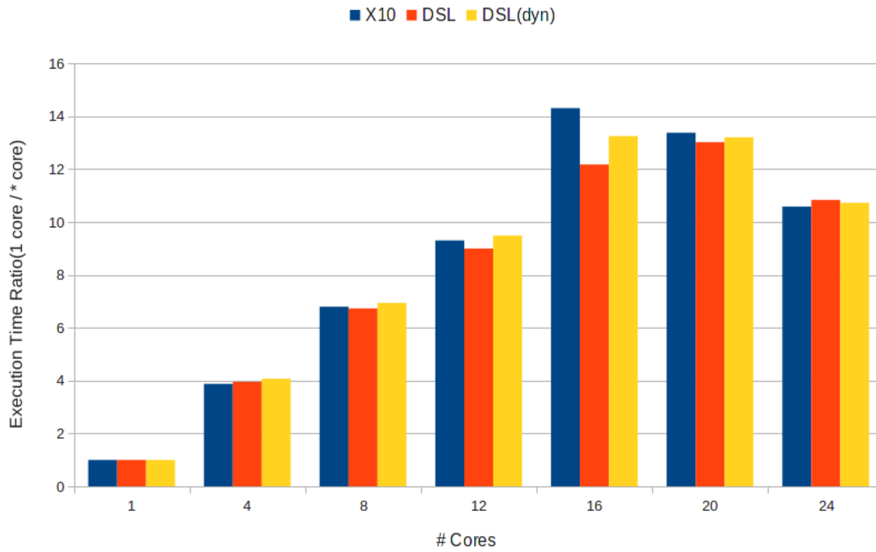

Fig. 5.5. Parallel Match Speed Comparison

computing, the I/O operation speed drops down. The figure 5.5 shows the ratio between execution time of matching step in the parallel version and the sequential version. We can see that the parallel parts in the program has a good scalability and the best performance shows up at 16 cores.

We suspect that the I/O performances in X10 is effected by current X10's thread scheduler. X10 creates a set of worker threads and put them into a thread pool at the initialization phrase of X10 execution environment. During the execution of an X10 program, "idle" worker threads are constantly searching for work. The idle thread tries to steal jobs from a random selected thread. It is known as job stealing. It means that the thread that is performing I/O action will be interrupted by other idle threads. More idle threads there are, more interruption occurs on the I/O thread. Therefore, I/O performance decreases as there are more cores.

Chapter 6

Discussion

6.1 Productivity of DSL

The code size evaluation in previous chapter shows 2 interesting results.

1. The byte size of the DSL is smaller than the X10 for all benchmark.
2. The eLOC depends on benchmarks. For the HPC benchmark program, eLOC of DSL and X10 implementation has no big difference. For the Ruby benchmark program, the DSL implementation has smaller eLOC than the X10 implementation.

These 2 results demonstrate that our proposed DSL has equal or better productivity than the X10 program. Further, smaller byte size of the DSL implementation mainly benefits from the dynamic typing character of Ruby. Even for the program that two implementation have similar eLOC, the DSL implementation has less code. This results shows that our decision of supporting dynamic typing in the DSL is correct and receives good payoffs. The overhead of dynamic typing is covered by the scalability of program as the evaluation shows.

The fact that DSL has same abstract level as the X10 language explains why the eLOC of HPC benchmarks are similar. We borrow the programming model and primitives from X10 language for concurrent and distributed programming. Therefore, for the HPC benchmarks that concentrate on parallel problem, the structures that used to construct program are the same. From another aspect, the X10 language is also a modern, high level language. The major features of Ruby, such as closure, exception handling, GC and so on, are also supported by the X10. So Ruby has less advantage on this side.

The result that the eLOC depends on benchmark program points out that the requirement for popular parallel programming is different from HPC parallel programming. The HPC program alway focus on how to productively and efficiently achieve parallelism. However, for the general program, the requirement from sequential and parallel code are mixed together. The different code size between the DSL and the X10 implementation is largely due to the sequential part implementation. Therefore, a productivity language for popular parallel programming should have good expressiveness for both sequential program and parallel program. Our proposed DSL inherits the advantages of Ruby and X10. Thus, it satisfies this requirement.

6.2 Dynamic Typing and Parallel Programming

When comparing dynamic typing language to static typing language, the performance is alway be mentioned as a disadvantage of dynamic typing language. It is true that runtime type check and dynamic method invocation has more overhead.

However, the scalability evaluation in previous chapter shows that the dynamic version

and statically typed version has similar scalability. This result points out that in the parallel environment, the scalability can cover the dynamic invocation overhead. And the prevalence of multi-core CPU makes sure that the computing system is dominated by parallel is not far anymore. So we believe that the language developers should turn their focus from making a language run faster to making a language more scalable. And the performance of a dynamic programming language is no more a big problem in parallel computing environment.

6.3 Scalability

The results of scalability evaluation demonstrate that the DSL has similar scalability to X10 language. Enabling the dynamic typing feature will introduce runtime overhead, but does not hurt the scalability. With the good scalability, the DSL is adequate for parallel programming.

However, the scalability is very difficult to achieve. First of all, programmers has to choose suitable parallelism algorithms and model for the problem. And then programmers have to implement the program carefully and correctly. Even all these are done perfectly, the implementation of language may also effect the scalability. So it requires that the programmers should have a deep understand of the language that they are using and some tools to tuning the scalability. In our case, due to the X10 implementation, although the parallel processing part of regex-dna benchmark scale very well, the entire program does not scale.

Right now, there are less tools to help users to debug and tune the programs on X10 environment. Therefore, it is very tough to solve the scalability problem. This situation reminds us that we have developed a Ruby memory profiler to help Ruby users to solve memory related problems. It would be helpful if we could port the Ruby memory profiler to the X10 in future.

6.4 Parallel Programming Model

Same as X10, the DSL employs the PGAS model to express parallel programming. Generally speaking, there are 3 models to express parallel: shared memory, message passing and PGAS. In our case, we choose the PGAS because the PGAS model has similar programming experience to the shared memory model, which general programmers are used to.

Recently, the messaging passing model has been adopted by lots languages. For example, the Go language uses the CSP model, the Scala language implemented the Actor library. However, some programmers argue that it is hard to program in message passing style. Gorlatch even pointed out that the `send/receive` in message passing is harmful to parallel programming as the `goto` is harmful to sequential programming[29].

One benefit we experienced from PGAS model is that PGAS model requires less efforts to adapt program to different styles of parallel programming. For example, the benchmarks that solve NQueens problem uses two different styles. One achieve parallel by using concurrent control structures, which is shared memory style and other is using distributed array. These two versions of implementation only have 3 different lines. From programmers perspective, they may start programming aiming at an shared memory system but later they may want to change the program to a cluster system. The PGAS model make the adaption work less effort.

6.5 Hybrid of Ruby and X10

Our proposed DSL can be considered as a hybrid of Ruby and X10 language. Ruby is a dynamic scripting language. X10 is a static system language. In our research, we uses DSL and source-to-source translator to bridge the large gap between these 2 languages.

From the development, we have learned that although completely converting all Ruby's features to X10 is difficult, most of Ruby's features can be supported to preserve the productivity of Ruby while achieve efficient parallel execution. It applies the Pareto principle[*1]. The main challenge is converting Ruby's dynamic features.

The dynamic character of Ruby represents at 3 perspectives: 1) the dynamic typing, 2) open-class(class can be modified dynamically), and 3) dynamic evaluation. We implemented the first 2 features. The dynamic typing can be handled by combining type inference, type annotation and dynamic method dispatch. And the overhead of dynamic typing can be covered by the speedup gained from scalability.

What makes the implementation hard is the open-class feature. Because the class itself is a mutable shared object at runtime, the implementation has to consider thread safety. What makes the situation worse is dynamic evaluation. Because the input string can be executed as Ruby source code, if users combine open-class with dynamic evaluation, it would be very difficult to detect when a class is modified.

In our case, because the DSL does not support the dynamic evaluation, the action that modified the class can be detected at the converting phrase and non-necessary overhead of dynamic dispatch can be avoided. Therefore, we believe what we gained is more than what we lost by dropping the support of dynamic evaluation feature.

[*1] Also known as the 80-20 rule.http://en.wikipedia.org/wiki/Pareto_principle

Chapter 7

Related Work

7.1 DSL for Parallel Programming

The approach we proposed to achieve parallel programming is to build a DSL on Ruby and implement it by employing a source-to-source compiler. DSL have been employed to improve domain-expert programmer productivity for a long time. And the embedded (or internal) DSL proposed by Hudak[30] simplifies building a DSL. For an embedded DSL, the DSL is implemented by using the constructs provided by the host language. Therefore, a embedded DSL is always treated as libraries or a framework to the host language. Our proposed DSL borrowed the syntax from the host language so it looks like an embedded DSL. However, the DSL is implemented by converting to another language which is different from traditional embedded DSL's definition.

OpenMP is an embedded DSL hosted on C language by using directives to mark out the code that can run in parallel. OpenMP is effective for shared memory system only. Our proposed language is capable for hybrid SMP cluster environment.

Ypnos[31] is a DSL hosted on Haskell programming language for structured grid programming. The Ypnos defined special syntax to express computing on an element and its neighborhood elements in the grid. The implementation is hold by Haskell itself. Different from the Ypnos, our proposed DSL is not limited to the structured grid computing problem but for general parallel programming problems.

Hassan Chafi[32] points out that embedded DSL is a promising approach for heterogeneous parallel programming but the key is how to overcome the limitation (expressiveness, the implement efficiency, etc.) from the host language. In our research, we bypass the limitation that Ruby is not inadequate for implementing parallel DSL by converting the DSL to the X10 language. The source-to-source translating is a general strategy for implement a new language. In fact, the X10 itself is implemented by converting to Java language.

7.2 Hybrid of Productivity and Efficient Language

The `SEJITS`[33] is a project that tries to achieve high performance computing by using high productivity language. The `SEJITS`, first, will defined some special tasks as library in a high productivity language, such as Ruby or Python. And then at runtime, these predefined tasks will be translated to a low level parallel programming language, such as C language with OpenMP. Next, the translated code will be linked to the original program and can be executed through the foreign-function interface(FFI) that is provided by the high productivity language.

First of all, the aim of `SEJITS` is different from ours. They are aiming at making a high productivity language runs faster by parallel executing parts for the code. Our

purpose is to write parallel program using a high productivity language. Therefore, in our research, we introduce parallel programming mode and primitives to the original high productivity language but they does not. Second, the `SEJITS` is similar to our approach at the point that compiling a high productivity language into a high efficient language. However, they try to do this at runtime, or called JIT compiling and they only compile the predefined tasks. Our proposed solution compiles the entire program before execution, or called AOT compiling. Third, the `SEJITS` Ruby code is 1 $\tilde{3}$ times slower than the program written in efficient language mostly due to overhead of the JIT code generation and execution through FFI. Our proposed solution does not have this performance issue. And we cannot compare the scalability of `SEJITS` because their work does not show the scalability evaluation.

7.3 Removing GVL in Ruby

As mentioned in previous, one limit of current Ruby implementation is that the giant VM lock (GVL) prevents Ruby code executing in parallel. In fact, not only Ruby, the Python language has a similar issue. The lock in Python implementation is called global interpreter lock (GIL). Therefore, there are many efforts to remove the lock, GVL or GIL. To remove the GVL or GIL means using fine-giant locks on all mutable data structures instead a global lock.

However, the problems of removing GVL/GIL are: 1) Lots of locking/unlocking slow down the single-threaded execution. In 1999, Greg Stein created a fork of Python(1.5) with the patch that remove the GIL in Python and benchmarking showed that the single-threaded execution is slowed down nearly two-fold. 2) Without global lock complicates the extension development. With the protection of global lock, the C extension is thread safe. So the developers do not have to worry about protecting their mutable data with lock by themselves. Sasada Koichi has implemented a Ruby VM without GVL and has similar conclusions[34].

7.4 Others

`dRuby`[22] is a implementation of distributed object system by pure Ruby. It employs some technique similar to remote method invocation(RMI) in Java to achieve distribute computing in Ruby. However, it is hard for this approach to achieve performance and scalability.[23].

`Diamondback Ruby`[35, 36] is an extension to Ruby that uses annotation to attach type system to Ruby. By using this extension, users can mark Ruby program like a static typing language program. We also use annotation to mark type in our implementation but not as serious as Diamondback Ruby. The type annotation in our research is optional feature to help reduce the cost of dynamic invocation.

Chapter 8

Conclusion

We design and implement a DSL based on Ruby for parallel programming. To achieve the popular parallel programming, a parallel programming language that is productive, efficient, hardware and algorithms independent and easy to be accepted by mainstream programmers is necessary. We observed that these requirements are difficult to be satisfy by one language and propose the approach that bridge the gap between popular, productive language and efficient parallel language by building a DSL.

We designed a DSL based on Ruby to use Ruby's syntax to express parallel programming. The DSL is consistent with Ruby's syntax and programing conversion. It supports most of Ruby's main features including object oriented, dynamic typing, open-class, mix-in and flexible control structures. We introduce the primitives and structures original in X10 to express parallel programming. In the DSL, the concurrent programing is expressed by using primitives of `finish`, `async`, `atomic` and `when` instead of thread/lock. The distributed programming can be constructed by using the `Point`, `Region`, `Dist` and `DistArray` structures with place shifting primitive `at`. Although the DSL support dynamic typing, users can also use the `define_type` syntax to annotate type explicitly to reduce the overhead of dynamic typing.

The DSL is implemented by compiling to X10 language. We design and implement a source-to-source translator to convert the DSL code to X10 program. With runtime libraries that we developed, the converted code can be executed in X10's environment. The code translator and runtime libraries work together to implement most of Ruby's features. We implement the dynamic method dispatch in runtime libraries, so that the dynamic typing, open-class and mix-in features can be supported. Runtime libraries also wrap the built-in types of X10 to provide Ruby-like API. For example the `each` method iteration. The new defined syntax and lots of difference on syntax between Ruby and X10 is handled by translator. For example to automatically insert return statements which is required in X10 but not in Ruby. The effort of code translator and runtime libraries make sure the DSL is consistent with Ruby's syntax and programming conversion.

We evaluated the productivity and scalability of the DSL. The results show that the DSL has equal or better productivity than the X10 language and has similar scalability to the X10 language. We measured the byte size and effective lines of code(eLOC) to evaluate the productivity. The DSL has smaller byte size than X10 and similar or smaller eLOC than X10 depends on benchmarks. We discuss the result and conclude that for HPC applications that focus on numerical computing, due to the abstract level of the DSL is same as X10, the expressiveness of both language is also similar; for general purpose applications, thanks to Ruby's productivity, the DSL has better expressiveness than X10.

We evaluated the scalability of DSL with and without dynamic typing and compare the results to X10. Although both DSL implementations are slower than X10, the scalability is similar to X10. Therefore, we conclude that our DSL has good enough scalability for par-

allel computing. Furthermore, because the dynamic typing overhead can be compensated by speedup gained from the scalability, we argued that language developers should make more efforts on chasing scalability rather than performance in sequential environments.

The DSL benefits the Ruby programmers by making them able to write parallel program in Ruby style. Ruby is a popular language with lots of users. By make Ruby users easy to adapt to parallel programming, we also contribute to the popular parallel programming. The DSL also provides an alternative to HPC users. With the DSL that we proposed, HPC users can enjoy the productivity of the Ruby language. Achievements of our research show that the approach of using DSL to achieve high productivity and high performance parallel programming is possible and effective. And the source-to-source translator strategy is an effective solution to bypass the limit from host language on implementing the DSL.

In future, we plan to use the DSL as base language and build a more high level DSL on it. Our proposed DSL has a modest abstract level to enable programmers to handcraft parallel code for performance. However, for some specific parallel problem domain, a higher level DSL is possible to achieve better productive while guarantee the performance. One good example is structured grid computing.

Consider to the implementation, we want to improve the type inference of code translator in future. Current implementation only infer type on assignment. More aggressive type inference can reduce the overhead of dynamic typing or the necessary of type annotation.

Another direction for future work is to develop performance analysis tools, such as CPU or memory profiler. We have developed a memory profiler for Ruby. It would be helpful if we could port it for parallel programming.

Publications

(1) Soh Tetsu, Sasada Koichi. Design and Implementation of Memory Profiler for Ruby. 情報処理学会プログラミング研究会, Mar 16, 2009.
(2) Soh Tetsu. Memory Profiler for Ruby. 日本 Ruby 会議 2010, Aug, 2010.
(3) Soh Tetsu. Ruby to X10 Translator. X10 BoF at SPLASH2010, Oct 20, 2010.

References

[1] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal*, 13, 2010.

[2] The free lunch is over. `http://www.gotw.ca/publications/concurrency-ddj.htm`, 2005.

[3] J. Shen M. Irwin. Revitalizing computer architecture research. Technical report, Computing Research Association, Dec 2005.

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

[5] The go programming language. `http://golang.org`.

[6] The scala programming language. `http://www.scala-lang.org/`.

[7] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31:23–30, March 1998.

[8] The ruby programming language. `http://www.ruby-lang.org/`.

[9] The x10 programming language. `http://www.x10-lang.org/`.

[10] The cilk project. `http://supertech.csail.mit.edu/cilk/index.html`.

[11] The chapel programming language. `http://chapel.cray.com/`.

[12] Messange passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi/`.

[13] The openmp api specification for parallel programming. `http://www.openmp.org/`.

[14] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20:133–158, Apil 1991.

[15] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30:123–169, June 1998.

[16] Vijay Saraswat Kemal Ebcioc glu and Vivek Sarkar. X10: Programming for hierarchical parallelism and nonuniform data access(extended abstract). *In Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004)*, October 2004.

[17] K Ebcioglu, V Saraswat, and V Sarkar. X10: an experimental language for high productivity programming of scalable systems(extended abstract). *In Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[18] Vijay Saraswat. *Report on the Programming Language X10.* x10-lang.org, 2.1.0 edition.

[19] Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17:35–75, December 1991.

[20] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010.

ACM.
[21] Steve Vinoski. Enterprise integration with ruby. *IEEE Internet Computing*, 10:91–95, 2006.
[22] druby. `http://www2a.biglobe.ne.jp/~seki/ruby/druby.en.html`, 2005.
[23] Sanjay P. Ahuja and Renato Quintao. Performance evaluation of java rmi: A distributed object architecture for internet based applications. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '00, pages 565–569, Washington, DC, USA, 2000. IEEE Computer Society.
[24] Unified parallel c. `http://upc.lbl.gov/`.
[25] Co-array fortran. `http://www.co-array.org`.
[26] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.
[27] Hpc challenge benchmark. `http://icl.cs.utk.edu/hpcc/`.
[28] Resource standard metrics. `http://msquaredtechnologies.com/m2rsm/docs/index.htm`.
[29] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26:47–56, January 2004.
[30] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996.
[31] Dominic A. Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10, pages 15–24, New York, NY, USA, 2010. ACM.
[32] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM.
[33] Bryan Catanzaro, Shoaib Ashraf Kamil, Yunsup Lee, Asanovi Krste, James Demmel, Kurt Keutzer, John Shalf, Katherine A. Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.
[34] SASADA KOICHI, MATSUMOTO YUKIHIRO, MAEDA ATSUSHI, and NAMIKI MITARO. An implementation of parallel threads for yarv: Yet another rubyvm. **情報処理学会論文誌. プログラミング**, 48(10):1–16, 2007-06-15.
[35] Diamondback ruby. `http://www.cs.umd.edu/projects/PL/druby/`.
[36] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

Acknowledgements

Many people contributed to the success of this work. I am indebted to all of them. I would like to express my particular thanks to Sasada Koichi, who is my instructor, for contributing the original idea of this project and giving me lots of advice to guide me along the development of this project. He also reviewed and commented on the construction of this paper. Without his energy, time and interest, this project would never be possible. I am appreciate Kiyokuni Kawachiya, who is in charge of this project from IBM Research Tokyo, for providing tutorial and answering my questions on X10 language. He also provided me opportunities to know many X10 related developers so that I can gather different information on X10 language. I am also grateful to Shiba Satoshi for his discussion enlightening me on the mechanism behind Ruby interpreter and helping me to finish the Japanese abstraction of this paper, to the developers of X10 language for their helpful information and advice on how to use X10 language. Last but not the least, I would like to thank all members in Sasada laboratory for the useful discussion and stimulating feedbacks to support me to the end of this project.