ВИКОРИСТАННЯ МОВИ RUBY ТА ТЕХНОЛОГІЙ DRB І RINDA ДЛЯ ПРОВЕДЕННЯ РОЗПОДІЛЕНИХ РОЗРАХУНКІВ ІЗ ВИКОРИСТАННЯМ ҐРАТОК ІЗІНГА

О.В. Дробнич, М.П. Дробнич, Т.В. Матяшовський, В.М. Різак

Ужгородський національний університет, 88000, Ужгород, вул. Волошина, 54

Обґрунтовано актуальність використання динамічної об'єктно-орієнтовної мови програмування надвисокого рівня Ruby для розподілених наукових розрахунків. Проаналізовано особливості технологій DRB та Rinda та наведено приклад їх використання для розподілених розрахунків у двовимірній гратці Ізінга.

Ключові слова: розподілені розрахунки, методи Монте-Карло, моделювання з перших принципів, Ruby, DRB, Rinda, Tuple Spaces.

Мова Ruby

Традиційними мовами для організації наукових розрахунків є мови C, C++. Це обумовлено високою лінійною швидкістю, яку мають скомпільовані додатки на цих мовах. Що, в свою чергу, обумовлено низкорівневістю цих мов. Поступові кількісні зміни по прискоренню процесорів, перехід на багатоядерні архітектури, створення і розповсюдження грідів сформували з часом якісно іншу картину, коли на перший план починають виходити простота кодування та підтримка паралельних процесів, в той час як лінійна швидкість виконання відступає на задній план.

Останнім часом ми бачимо підвищення цікавості до "супер високорівневих" мов програмування типу Ruby, Python, Groovy як основи для розробки і постановки чисельних експериментів [1, 2].

Порівняємо 2 фрагменти програмного коду на C++ і Ruby. Обидва фрагменти накопичують цілі числа, що вводить користувач з клавіатури, а потім виводять їх відсортовано на екран.

Приклад коду такої програми на мові C++ можна подивитись на лістингу L1. Порівняйте цей приклад із фрагментом тої самої програми мовою Ruby (див. лістинг L2).

У даному прикладі Ruby виграє по простоті коду завдяки потужності вбудованих колекцій. На цьому переваги у виразових можливостях не закінчуються. Динамічна типізація, автоматичні перетворення типів, динамічне добавлення елементів класу, робота із кодом як із даними, динамічне виділення пам’яті для роботи з надвеликими числами, чиста ООП-концепція - дозволяють розробляти більш короткий і зрозумілий код мовою Ruby ніж мовами C, C++. Лаконічність і простота коду, у свою чергу, скорочує витрати часу на відладку і оптимізацію. Це приносить науковому програмісту додатковий час, що є найважливішим ресурсом для чисельних експериментів.

Ruby та кластерність

Взагалі кажучи, перед написанням паралельного програмного коду програміст має зробити два стратегічних рішення:

1. Як він буде розподіляти код - по легковажних процесах - нитках (threads) або по системних процесах?
2. Як окремі екземпляри коду будуть між собою спілкуватись?

Нитки, або легковажні процеси підтримують більшість сучасних мов програмування. Нитки дозволяють використовувати "локальну" кластерність апаратного забезпечення - кластерні та багатоядерні процесорні архітектури, які переважно використовуються у сучасних персональних комп’ютерах. Тобто цей підхід дозволяє запускати паралельний код на окремому комп’ютері.

Існують 3 різновиди організації підсистеми ниток у мовах програмування:

1:1, 1:N, N:M, де перше число означає кількість задіяних виконавчих блоків процесора, а друге - кількість ниток у програмному коді. Слід зауважити, що Ruby використовує найбільш просту із вказаних моделей - 1:N. Тобто всі нитки виконуються на одному виконуючому блоці. Для розробки розподіленого коду із використанням ниток більш підходять мови Java, JRuby та Erlang, що використовують модель N:M.

Системні процеси відповідають як "локальній" так і "глобальній" кластерності - комп'ютерним кластерам, грідам. У даній статті ми будемо використовувати саме цей підхід до організації паралельного коду на мові Ruby.

Технологія роздільної пам'яті

Друге стратегічне питання - "як окремі екземпляри коду будуть обмінюватись інформацією?" - має три основних підходи для вирішення:
- обмін повідомленнями;
- пам'ять, що розділяється;
- зовнішній сервіс.

Обмін повідомленнями - це підхід, що з'явився першим - веде свою історію від створення операційної системи Unix. Цей підхід забезпечує найбільшу швидкість розподілених розрахунків за рахунок складності програмного коду. Сигнали, мережеві сокети і файли каналів - перші інструменти, на яких можна було будувати розподілений системний код. Згодом з'явились спеціалізовані програмні бібліотеки PVM та MPI. З найбільш сучасних реалізацій цього підходу можна перерахувати бібліотеки RabbitMQ, ActiveMQ, імплементації ідеї демонів-агентів для мови Ruby: Nanite, Revactor, Journeta.

Пам'ять, що розділяється - більш високорівневий підхід, де один із програмних модулів емулює загальнодоступну ділянку пам'яті, з якою працюють незалежні (у тому числі і віддалені процеси). Цей підхід забезпечує посередню швидкість паралельних розрахунків і потребує нескладного програмного коду. Memcached і DRb - сучасні представники такої технології. Більш детально ми зупинимось на бібліотеці DRb нижче.

Зовнішній сервіс забезпечує високорівневий інтерфейс для зовнішнього зберігання даних. Цей підхід є найбільш повільним у плані швидкості розподілених розрахунків і одночасно потребує найбільш простих рішень у плані програмного коду. Найбільш популярний варіант використання такого підходу - робота з реляційною базою даних типу MySQL або Oracle. Останнім часом з'явилось багато цікавих реалізацій для сховищ даних типу ключ - значення: Google Datastore, Redis, MemcachedDB, MongoDB.

Технологія DRb

Distributed Ruby (DRb) дозволяє створити віддалену колекцію програмних об'єктів, робота з якою виконується як з локальною колекцією, хоча дані фізично знаходяться на віддаленому комп'ютері. Ця бібліотека асоціює таку колекцію з мережевою адресою та портом віддаленого комп'ютера (URI), які згодом використовуються процесами-клієнтами для прив'язки до колекції та розміщення в ній програмних об'єктів та викликів їх методів.

Нижче приведено приклад простого сервера і клієнта, розроблених із використанням DRb. Серверна частина представлена на лістингу L3. Клієнтська частина – на лістингу L4.

У цьому прикладі ми прив'язуємо пустий масив до першого з доступних мережевих портів локальної машини-сервера. Потім URI цього масиву ми передаємо як параметр командного рядку в програму-клієнт, що завантажується на іншій машині.

Реєстрація сервісів: Rinda та TupleSpaces

Недоліком бібліотеки DRb є те, що вона жорстко прив'язується на IP адреси та порти мережевих станцій, що означає ускладнене конфігурування і підтримку такого гріда. Цю проблему вирішує технологія Rinda, що використовує концепцію пошуку сервісів по символічному імені.

Далі наведена програма, яка моделює процеси випадкового голосування "за" чи

"проти" певного твердження або публікації в мережевому блозі. Серверна частина представлена на лістингу L5. Клієнтська частина - на лістингу L6. Слід зауважити, що на відміну від попереднього прикладу, в цій програмі відсутні звертання до конкретних мережевих адресів і портів. Замість цього ми використовуємо псевдонім **Rinda::RingFinger.primary**, який пов'язаний із мережевим сховищем, де зберігається значення **:carma**.

Приклад реалізації розрахунків Монте-Карло на простій 2D-гратці Ізінга з допомогою технології Rinda

На цьому етапі ми спробуємо реалізувати конкретний приклад чисельного експерименту із використанням технологій, що обговорювались. Як приклад візьмемо просту модель Онзагера [3] - двовимірну модель Ізінга, що використовується для елементарного моделювання фазових переходів у моделях магнетиків та сегнетоелектриків.

Серверна частина програмного коду завантажує у Rinda-сховище дві 2D підгратки із псевдонімами lattice1 та lattice2, що мають одну суміжну сторону. В ході розрахунків ця частина програми також періодично виводить значення лічильників кроків від двох паралельних процесів Монте-Карло (див. лістинг L7).

Далі наведено код програми першого процесу Монте-Карло, що виконується над підграткою lattice1 (Лістинг L8). І другого процесу Монте-Карло над підграткою lattice2 (Лістинг L9).

Оскільки гратки lattice1 і latice2 мають спільну сторону, обом процесам Монте-Карло приходиться час від часу блокувати суміжну підгратку, щоб дістати не спотворене значення псевдоспіну, що знаходиться на границі (див. рядки коду, що працюють із значенням **extnode**).

Проведений чисельний експеримент виявив наявність фазового переходу в околі Tc = 2.269 із невпорядкованої фази у впорядковану, що є підтвердженням правильної роботи програмного коду. Далі наведено хід температурної залежності параметра порядку за результатами розрахунків (рис. 1).

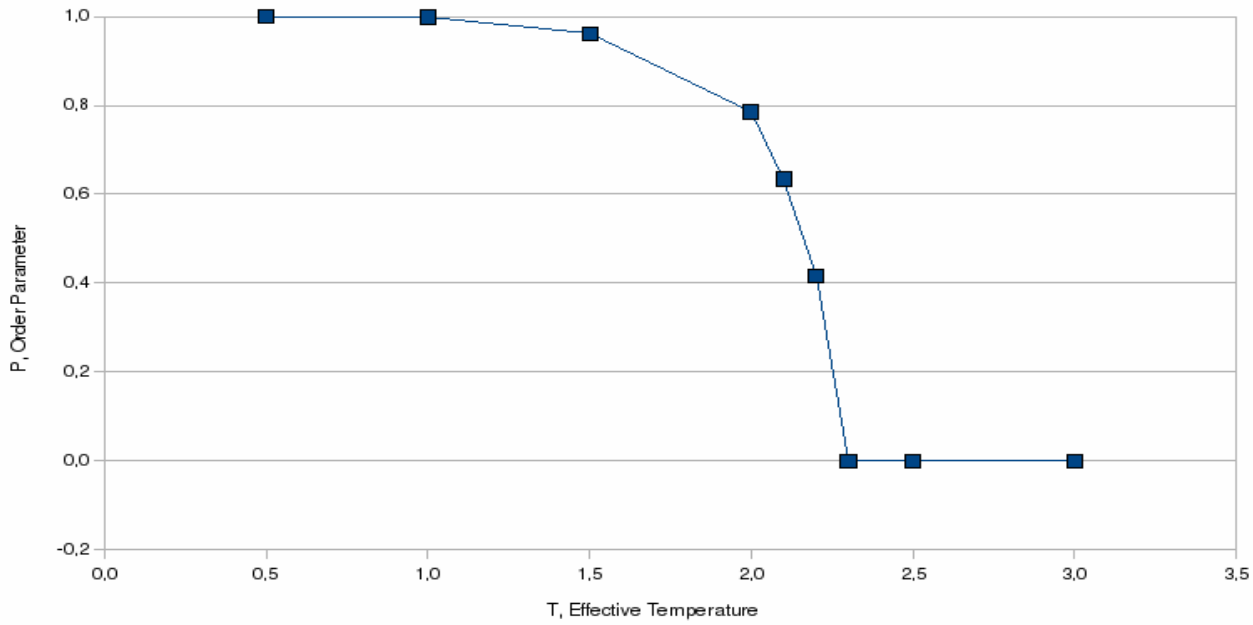

Рис. 1. Температурна залежність параметру порядку.

Даний підхід можна використати для проведення чисельних розрахунків методом Монте-Карло у тривимірних моделях типу Ізінга, що відтворюють геометрію реальних феромагнетиків або сегнетоелектриків та дозволяють відтворювати їх фазові діаграми.

Література

1. Ong. E. MPI Ruby: Scripting in a Parallel Environment // Computing in Science and Engineering, 4(4):78–82, 2002.
2. Bates. M. Distributed programming with Ruby. - Addison-Wesley Professional Ruby Series, 2009. - 273 p.
3. Onsager, Lars. Crystal statistics. I. A two-dimensional model with an order-disorder transition", Phys. Rev. (2) 65: 117–149, 1944.

Додатки

Лістинг L1

```
#include <iostream>
using namespace std;

class ListItem{
  public:
   int data;
   ListItem* next;
};

class List{
  public:
   ListItem* root;
   int size;
   List (){
      root = new ListItem;
      size = 0;
   }
   ListItem* getItem( int position ){
      ListItem* current = root;
      for (int i=0; i<position; i++)
         current = current -> next;
      return current;
   }
   void addItem( int x ){
      getItem(size) -> next = new ListItem;
      size++;
      getItem(size) -> data = x;
   }
   void swap(int x, int y){
      int buf = getItem(x) -> data;
      getItem(x) -> data = getItem(y) -> data;
      getItem(y) -> data = buf;
   }
   void sort(){
      for (int j=0; j<size; j++)
         for (int i=2; i<=size; i++)
            if( getItem(i-1)->data < getItem(i)->data )
               swap(i-1, i);
   }
};

int main(){
   List L;
   char answer;
   do{
      int buf;
      cout<<"Input value...";
      cin>>buf;
      L.addItem(buf);
      cout<<"Is there new item? (y/n)";
      cin>>answer;
   } while (answer == 'y' or answer == 'Y');
  for ( int i=1; i<=L.size; i++)
      cout<<L.getItem(i)->data<<",";
   cout<<"\n";
   L.sort();
   for ( int i=1; i<=L.size; i++)
      cout<<L.getItem(i)->data<<",";
   return 0;
}
```

Лістинг L2

```
def collector
   list = []
   answer = "y"
   while answer == "y" or answer == "Y"
      puts "Input next value..."
      list = list + [gets.to_i]
      puts "Is there new value? (y/n)?"
      answer = gets[0,1]
   end
   puts list.join(", ")
end
collector
```

Лістинг L3

```
#!/usr/bin/env ruby -w
# simple_service.rb
# A simple DRb service

# load DRb
require 'drb'

# start up the DRb service
DRb.start_service nil, []

# We need the uri of the service to connect a client
puts DRb.uri

# wait for the DRb service to finish before exiting
DRb.thread.join
```

Лістинг L4

```
#!/usr/bin/env ruby -w
# simple_client.rb
# A simple DRb client

require 'drb'

DRb.start_service

# attach to the DRb server via a URI given on the
command line
remote_array = DRbObject.new nil, ARGV.shift

puts remote_array.size

remote_array << 1

puts remote_array.size
```

Лістинг L5

```
#!/usr/bin/env ruby

require 'drb/drb'
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service

tuple_space = Rinda::TupleSpace.new
ring_server = Rinda::RingServer.new tuple_space

tuple_space.write [:carma, 0]
views = average = max = min = 0

4.times { system "./voter.rb &" }

# observer

while true
  current = tuple_space.take([:carma, nil])[1]
  tuple_space.write [:carma, current]
  views = views + 1
  average = ( average + current ) / views.to_f
  max = current if current > max
  min = current if current < min
  puts "carma: #{current} max: #{max} min: #{min}
average: #{average} views: #{views}"
end
```

Лістинг L6

```
require 'drb/drb'
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service

tuple_space = Rinda::RingFinger.primary

while true
  current = tuple_space.take([:carma, nil])[1]
  tuple_space.write [:carma, current +  2 * rand(2) - 1]
end
```

Лістинг L7

```
#!/usr/bin/env ruby
require 'drb/drb'
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service

tuple_space = Rinda::TupleSpace.new
ring_server = Rinda::RingServer.new tuple_space

tuple_space.write [:lattice1, [
[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,
1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,
1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1] ] ]
tuple_space.write [:lattice2, [
[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,
1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,
1,1,1],[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1] ] ]

tuple_space.write [:settings, {:T => 2.0}]
tuple_space.write [:counter1, 0]
tuple_space.write [:counter2, 0]

system "./processor1.rb &"
system "./processor2.rb &"

# observer

show_number = 0
while true
  sleep 5
  p show_number
  show_number = show_number + 1
  current1 = tuple_space.read([:lattice1, nil])[1]
  current2 = tuple_space.read([:lattice2, nil])[1]
  p current1
  p current2
  counter1 = tuple_space.read([:counter1, nil])[1]
  counter2 = tuple_space.read([:counter2, nil])[1]
  puts "counter1: #{counter1}"
  puts "counter2: #{counter2}"
end
```

Лістинг L8

```
#!/usr/bin/env ruby
require 'drb/drb'
require 'rinda/ring'
require 'rinda/tuplespace'
include Math

DRb.start_service
```

```
tuple_space = Rinda::RingFinger.primary

T = tuple_space.read([:settings, nil])[1][:T]

while true
  current = tuple_space.take([:lattice1, nil])[1]
  tuple_space.write [:lattice1, current]

  x = rand 10
  y = rand 10

  extnode = 0

  if x == 0
    current2 = tuple_space.read([:lattice2, nil])[1]
    extnode = current2[9][y]
  end

  dE = -2 * current[x][y] * ((x >= 1 ? current[x-1][y] :
extnode) +
                        (y >= 1 ? current[x][y-1] : 0) +
                        (x <= 8 ? current[x+1][y] : 0) +
                        (y <= 8 ? current[x][y+1] : 0))
  if dE >= 0
    current[x][y] = - current[x][y]
  else
    w = exp(dE / T)
    r = rand
    if r < w
      current[x][y] = - current[x][y]
    end
  end
 tuple_space.write [:lattice1, current]
 counter = tuple_space.take([:counter1, nil])[1] + 1
 tuple_space.write [:counter1, counter]
end
```

Лістинг L9

```
#!/usr/bin/env ruby
require 'drb/drb'
require 'rinda/ring'
require 'rinda/tuplespace'
include Math

DRb.start_service

tuple_space = Rinda::RingFinger.primary

T = tuple_space.read([:settings, nil])[1][:T]

while true
  current = tuple_space.take([:lattice2, nil])[1]
  tuple_space.write [:lattice2, current]
  x = rand 10
  y = rand 10

  extnode = 0

  if x == 9
    current2 = tuple_space.read([:lattice1, nil])[1]
    extnode = current2[0][y]
  end

  dE = -2 * current[x][y] * ((x >= 1 ? current[x-1][y] :
0) +
                        (y >= 1 ? current[x][y-1] : 0) +
                        (x <= 8 ? current[x+1][y] : 0) +
                        (y <= 8 ? current[x][y+1] : extnode))
  if dE >= 0
    current[x][y] = - current[x][y]
  else
    w = exp(dE / T)
    r = rand
    if r < w
      current[x][y] = - current[x][y]
    end
  end
 tuple_space.write [:lattice2, current]
 counter = tuple_space.take([:counter2, nil])[1] + 1
 tuple_space.write [:counter2, counter]
end
```

USING OF RUBY LANGUAGE, DRB AND RINDA FRAMEWORKS FOR CONDUCTING OF DISTRIBUTED CALCULATIONS OVER ISING-LIKE LATTICES

O.V. Drobnych, M.P. Drobnych, T.V. Matyashoskii, V.M. Rizak

Uzhhorod National University, 88000, Uzhhorod, Voloshina Str., 54

We're considering the relevance of the use of Ruby, the high-level dynamic object-oriented programming language for purpose of distributed scientific computations. Also the peculiarities of DRB and Rinda frameworks discussed and an example of their use for the distributed calculations in two-dimensional Ising lattice was shown.

Key words: distributed calculations, Monte-Carlo methods, simulations from first principles, Ruby, DRB, Rinda, Tuple Spaces.

ИСПОЛЬЗОВАНИЕ ЯЗЫКА RUBY И ТЕХНОЛОГИЙ DRB И RINDA ДЛЯ ПРОВЕДЕНИЯ РАСПРЕДЕЛЕННЫХ РАСЧЕТОВ С ИСПОЛЬЗОВАНИЕМ РЕШЕТОК ИЗИНГА

А.В. Дробнич, М.П. Дробнич, Т.В. Матяшовский, В.М. Ризак

Ужгородский национальный университет, 88000, Ужгород, ул. Волошина, 54

Обоснована актуальность использования динамического объектно-ориентированного языка программирования сверхвысокого уровня Ruby для распределенных научных расчетов. Проанализированы особенности технологий DRB и Rinda и приведен пример их использования для распределенных расчетов в двухмерной решетке Изинга.

Ключевые слова: распределенные расчеты, методы Монте-Карло, моделирование из первых принципов, Ruby, DRB, Rinda, Tuple Spaces.