УДК 004.42

Гороховський С. С., Кравченко М. В.

ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ЗАСТОСУВАННЯ МОВ SCALA, ERLANG І HASKELL В УМОВАХ БАГАТОЯДЕРНИХ АРХІТЕКТУР

У роботі проведено порівняння основних можливостей мов Scala, Erlang і Haskell та їх ефективності при розв'язанні практичних задач на багатоядерних архітектурах. На прикладі задачі знаходження досконалих чисел на певному проміжку показано, як за допомогою акторів отримати значне прискорення програми на багатоядерних архітектурах.

Ключові слова: функціональні мови програмування, актори, багатоядерні архітектури.

Вступ

Протягом останніх років тактова частота процесорів майже не збільшується. Водночас набирає поширеності явище нарощування кількості ядер у процесорах, що дозволяє виконувати декілька машинних інструкцій одночасно.

Новий напрям розвитку процесорів ставить серйозні задачі перед програмістами, які прагнуть, щоб їхнє застосування було якнайефективнішим і використовувало всі новітні можливості апаратного забезпечення. Щоб максимально використовувати потужність багатоядерних систем, необхідно проектувати програми таким чином, аби певні задачі були незалежними від інших, що дозволило б виконувати їх паралельно.

Зазвичай цього дуже важко досягти в імперативних мовах програмування, де в основному передбачається послідовне виконання програмного коду. Водночас, у функціональних мовах програмування підхід суттєво відрізняється. Оскільки програмісту необов'язково вказувати конкретну послідовність дій, які необхідно виконати, то компілятор здатен проаналізувати і відділити взаємно незалежні інструкції, які можна виконати паралельно. До того ж, функціональні мови програмування, на відміну від імперативних, позбавлені сторонніх значень (side effects), що дозволяє набагато легше їх розпаралелити.

Зважаючи на актуальність таких мов, як Scala, Erlang і Haskell, мета даної роботи полягає у проведенні порівняння основних можливостей цих мов та їх ефективності при розв'язуванні практичних задач на багатоядерних архітектурах.

Основні характеристики мов

Scala

Ім'я Scala походить від словосполучення «масштабована мова» («scalable language»). Вона спроектована для того, щоб легко розширюватися відповідно до вимог користувачів. Scala можна застосовувати для вирішення широкого кола задач з програмування, від написання невеликих скриптів до побудови великих систем.

Програмний код мови виконується на стандартній платформі Java і, як більшість мов на базі JVM (Віртуальної машини Java), без проблем запускається з усіма бібліотеками мови Java.

Фактично, Scala – це суміш об'єктно-орієнтованої та функціональної парадигми програмування зі статичною типізацією, більш строгою ніж, наприклад, в Java, що дозволяє виключити чимало помилок ще на етапі компіляції.

Архітектура мови має дуже багато елементів, запозичених з інших мов програмування. Наприклад, синтаксично Scala дуже подібна до Java та C#, які в свою чергу дотримуються багатьох принципів мов C та C++. Функціональна частина мови дуже подібна до мов сімейства ML (SML, OCaml, F# тощо). Ідеї для написання однієї з найвагоміших частин мови, базованої на акторах бібліотеки багатопоточності, взяті з мови програмування Erlang.

Erlang

Як і мова програмування C++, Erlang створена невеликою групою ентузіастів дослідницьких центрів великих телекомунікаційних компаній. Для архітекторів мови C++ головна проблема складає: як писати програми на рівні вищому, ніж мова асемблеру, водночас, на апаратному забезпеченні з обмеженими ресурсами. Для творців мови Erlang мета була інакшою: дозволити програмістам створювати складне, високопродуктивне, суттєво розпаралелене і надзвичайно стійке до відмов програмне забезпечення.

Мову Erlang розроблено в Лабораторії комп'ютерних наук компанії Ерікссон у Сток-

гольмі (Швеція). Від моменту створення у 1988 році і до перетворення на проект з відкритим кодом, Erlang 10 років успішно застосовувалася компанією і показала себе ефективною складовою багатьох великих систем.

Спочатку ядро системи писалося мовою Prolog. Але вже на ранніх тестах системи стало очевидним, що це ядро хоч і є набагато ефективнішим, ніж інші програмні системи, але все одно вимагає покращення і оптимізації під конкретні цілі мови Erlang. Тому від початку 1990 року розробники мови вирішили реалізувати першу абстрактну машину для Erlang. Вона була написана на C і стала в 70 разів швидшою, ніж реалізація ядра на Prolog. Її назвали JAM. Erlang – це функціональна мова програмування, в якій фундаментальною частиною є «процеси», що дозволяють ізольоване і паралельне виконання певних частин коду.

Haskell

Haskell (Хаскель) – це стандартизована, цілком функціональна мова програмування з сильною статичною типізацією. Своєю назвою вона завдячує американському математику і логіку Хаскелу Каррі, який зробив вагомий внесок у розвиток комбінаторної логіки.

Ідею створення мови проголошено на конференції з питань функціональних мов програмування і комп'ютерних архітектур 1987 року. Поштовхом стало створення мови Miranda, цілком функціональної мови на базі «лінивих обчислень» і багатьох її похідних. Haskell повинна був стати відкритим стандартом для таких мов.

Першу версію випущено у 1990 році як «Haskell 1.0».

На сьогодні мова має відкриту специфікацію і численні реалізації, наприклад Глазго Хаскель Компілятор (Glasgow Haskell Compiler – GHC) та Утрехт Хаскель Компілятор (Utrecht Haskell Compiler – UHC).

Порівняння основних концепцій мов. Багатопоточне програмування

Для того, щоб запускати ефективні програми, які використовують усі ресурси процесорів, необхідно виконувати велику кількість багатопоточного коду. Це досягається кількома способами. Перший – використання потоків, другий – процесів. Різниця між ними в тому, що потік має спільну пам'ять з іншими потоками, тоді як процеси не ділять між собою жодні ресурси. Це означає, що потоки потребують певних механізмів блокування, таких, наприклад, як м'ютекси, щоб запобігти ситуації, коли два потоки маніпулюватимуть однією ділянкою пам'яті одночасно. Процеси не мають цієї проблеми, оскільки вони використовують механізми передачі. Водночас, процеси, зазвичай, вимогливіші до ресурсів процесора і пам'яті, і тому більшість програмістів частіше обирають потоки, хоча їх складніше програмувати.

Модель багатопоточності в Erlang має такі особливості:

- ізольовані «легкі» (lighweight) процеси;
- процеси не мають ніяких спільних ресурсів, не допускають побічних продуктів (side-effects);
- виключно функціональний механізм спілкування на базі передачі повідомлень, які проходять через «поштову скриньку» (mailbox);
- процеси можуть бути розподілені між різними машинами.

Бібліотека Акторів Scala дуже схожа за функціональністю на модель процесів в Erlang. Як і процеси, актори в Scala відправляють і отримують повідомлення через «поштову скриньку». Scala використовує техніку зіставлення зі зразками для отримання повідомлень, що дозволяє писати код надзвичайно елегантно і чітко.

Водночас, порівняно з Erlang, бібліотека Акторів не гарантує такої важливої для багатопоточного програмування особливості, як незмінність об'єктів. В Erlang декілька процесів мають доступ до одних і тих самих даних у межах віртуальної машини, адже мова гарантує, що колізій не відбудеться через незмінність даних. У Scala таких гарантій немає, тому треба бути дуже обережним, а іноді навіть використовувати допоміжні синхронізаційні механізми для того, щоб забезпечити доступ до спільної пам'яті.

Також характерною відмінністю акторів Scala від процесів Erlang є те, що актори інкапсулюють в собі стан, тоді коли процеси передають стан як параметр.

Приклад акторів відправлення і отримання повідомлення у Scala:

```
a! msg  //відправлення повідомлення
receive { //обробка повідомлень
        case msg_pattern_1 => action_1
        case msg_pattern_2 => action_2
        case msg_pattern_3 => action_3}
```

У Haskell паралелізм досягається за допомогою використання розширення стандартного компілятора під назвою Glasgow parallel Haskell (GpH).

Це розширення додає дві анотації:
– par (par :: a ->b ->b) – з її допомогою можна вказати, що наступний код буде виконуватися паралельно;
– pseq (pseq :: a ->b->b) – з її допомогою можна вказати, що код буде виконуватися у точно визначеному порядку. Це необхідно для правильної організації паралельних обчислень.

Існують інші засоби, скажімо, монада Par, DPH та Eden.

Scala пропонує один з найзручніших і корисніших механізмів для забезпечення паралельної роботи, це паралельні колекції. За допомогою ключового слова ***par*** будь-яку колекцію можна перетворити на паралельну. Програмісту не треба вручну виконувати забезпечення безпеки багатопоточності, Scala інкапсулює це всередині себе. Вона використовує фреймворк Fork/Join в Java і пул потоків, який розподіляється серед доступних процесорів.

Збирач сміття (Garbage collection)

У порівнянні з імперативними мовами програмування, функціональні мови накопичують у пам'яті дуже багато «сміття» (ділянки пам'яті, що займаються об'єктами, які потім не використовуються у програмі). Наприклад, оскільки всі дані в програмі незмінні, то єдиним способом, яким можна зберегти результат виконання наступної операції, буде створення нових значень і збереження їх у пам'яті. Крім того, кожна ітерація рекурсивних обчислень створює нове значення в пам'яті. Тому для ефективності таких мов і створена концепція «збирачів сміття».

Збирач сміття – це форма автоматичного контролю за ресурсами пам'яті, при якій відбувається спроба звільнити «сміття». Перша реалізація такого збирача представлена Джоном МакКарті в 1959 році як вирішення проблем з пам'яттю в Ліспі.

Основним аргументом проти мов, у яких реалізовано автоматичне збирання сміття, є те, що вони неефективні для застосувань, що вимагають низької затримки відповіді на запит (latency). Це зумовлено потенційно довгими «чистками» збирача сміття, що на певний час призупиняє роботу віртуальної машини, так зване «замороження». Сучасні оптимізації в плані збирання сміття дозволяють до певної міри зменшити кількість таких станів, але не повністю.

Erlang було розроблено для створення застосувань, які повинні досягати ефективності виконання реального часу, і тому збирач сміття віртуальної машини Erlang оптимізовано заради досягнення цього результату. Наприклад, у кожного процесу є своя купа, яка «очищується» збирачем сміття окремо, незалежно від інших процесів, що мінімізує час, коли процес може «заморозитися» для виконання операції збирання сміття. Також в Erlang є ділянка пам'яті ***ets***, яка використовується для зберігання великих об'ємів даних, що не потребують збирання сміття.

Одна з останніх версій віртуальної машини Erlang надає збирачу сміття ще такої переваги – розподіленості між процесорами. Тепер на кожне ядро процесора існує окремий процес збирача сміття, і якщо один з них у певний момент часу не займається чисткою сміття, то він може бути задіяним в очистці сміття процесів, які обробляються іншим збирачем.

У свою чергу, віртуальна машина Java, в байт-код якої Scala компілює свої програми, трішки поступається своїм збирачем сміття в рамках багатопоточного застосування. Для того, щоб зменшити час очікування повного збирання сміття, використовується багатопоточна версія збирача, що досягає своєї мети, але від цього страждає продуктивність і збільшуються витрати на пам'ять.

GHC може ефективно виконувати збирання сміття, що дозволяє заповнити близько 1 Гб даних за секунду, більша частина яких буде звільнено миттєво. Незмінність даних стає причиною генерування великих проміжних результатів у пам'яті, крім того, це може допомогти швидко збирати сміття. Ідея полягає в тому, що на результати пізніших обчислень не можуть вказувати більш ранні значення. Тому в будь-який момент часу можна сканувати останні значення, що були створені, і звільняти ті, які вже не використовуються.

GHC використовує тимчасове сховище для даних. Коли воно вичерпується, відбувається «маленьке очищення», що сканує це сховище і видаляє сміття. Наприклад, якщо є рекурсивний алгоритм і він швидко заповнює це проміжне сховище новими «поколіннями» індуктивних змінних – тільки останнє покоління змінних, тобто найновіші значення, буде збережено і скопійовано в основну пам'ять.

Але збирач сміття GHC не є паралельним. Він, як і в Java, може використовувати всі потоки для виконання збирання сміття, але повинен зупинити на деякий час усі потоки, щоб завершити цю операцію.

Заміна коду під час виконання (Hot Code Swapping)

Процес, який називається заміна коду під час виконання, є дуже корисною можливістю мови програмування. Він не тільки усуває необхідність призупиняти роботу системи для внесення змін у програмний код системи, а й надає мові більшої продуктивності, оскільки дозволяє досягти інтерактивності програмування. Використовуючи цю можливість, можна миттєво тестувати результати зміни коду без зупинок сервера, перекомпіляції модулів, перезапуску сервера (включно з усіма поточними станами застосування) і повернення до «чистого стану» системи.

Заміна коду під час виконання надзвичайно важлива для застосувань реального часу, які дозволяють синхронному обміну повідомленнями між користувачами. Перезавантаження таких серверів спричинить від'єднання клієнтських сесій, що погано вплине на зручність користування.

JVM певний час не мала підтримки заміни коду під час виконання програми, пізніше такі проекти, як JRebel (комерційний) та HotSwap (з відкритим кодом) надали цю можливість віртуальній машині Java. Але їх не порівняти за повнотою можливостей з моделлю Erlang.

Основною концепцією заміни коду в Erlang є те, що коли завантажується новий код, віртуальна машина зберігає попередню версію програмного коду. Це дозволяє запущеним процесам отримувати повідомлення виконати заміну коду до того, коли стара версія буде остаточно видалена (що може спричинити зупинку процесів, які не зробили оновлення коду). Це є унікальною рисою мови Erlang серед інших.

У Haskell також є обмежена версія заміни коду під час виконання за допомогою пакету ***dyre.*** Хоча для оновлення коду усе ще необхідний перезапуск системи, є можливість зберегти стан застосування і завантажити його після успішного оновлення.

Оптимізація хвостової рекурсії (tail recursion)

Одним з фундаментальних елементів функціонального програмування є хвостова рекурсія. Насправді дуже важко написати застосування мовою Erlang без використання хвостової рекурсії, оскільки в ньому немає підтримки циклів. Також вона дозволяє реалізувати заміну коду під час виконання. Серверний процес (gen_servers) рекурсивно викликає функцію loop() між викликами до «receive». Коли процес отримує повідомлення про оновлення коду, він виконує низку операцій, щоб увійти у свій основний цикл з новим кодом. Без оптимізації хвостової рекурсії така модель призвела б до швидкого переповнення буфера.

У Scala підтримка оптимізації хвостової рекурсії обмежена, в основному, через особливості байт-коду в різних віртуальних машинах Java. Як описано в документації Scala:

> «Теоретично, хвостова рекурсія завжди може бути оптимізована шляхом повторного використання стеку функції, яка її викликає. Проте деякі середовища виконання (такі як Java VM) не мають достатніх засобів для ефективної імплементації повторного використання стеку викликів. Тому в Scala є тільки підтримка повторного використання стеку викликів функції, остання дія якої є виклик самої себе».

Оптимізація хвостової рекурсії в Scala працює тільки у тому випадку, якщо функція *x()* викликає в кінці свого виконання функцію *x()*. Водночас, якщо *x()* викликає *y()*, то виклик в *y()* функції *x()* вже не буде оптимізовано.

У Haskell до стандартної оптимізації хвостової рекурсії приєднується ще один вид оптимізації, який називається регульована рекурсія (guarded recursion), де будь-який рекурсивний виклик відбувається в конструкторі даних. Це дозволяє результату функції обчислюватися «ліниво», відкладаючи рекурсивний виклик на реальний момент виконання.

Підтримка розподіленого програмування

Однією з сильних сторін мови Erlang є поєднання багатопоточного і розподіленого програмування. Erlang дозволяє надсилати повідомлення в процес на локальній чи на віддаленій віртуальній машині, використовуючи однакову семантику («незалежність від місцезнаходження», або location transparency). Крім того, створення процесів і їх зв'язування/моніторинг працює бездоганно між вузлами. Це дозволяє уникнути труднощів при побудові розподіленого і відмовостійкого застосування.

На відміну від Erlang, у Scala немає вбудованих засобів для конструювання розподілених систем. Існує кілька варіантів, завдяки яким можна розширити можливості мови для забезпечення розподіленості. Наприклад, використання фреймворку Akka:

> Актори в Akka є «незалежними від розташування та розподіленими за ідеєю. Це означає, що ви пишете своє застосування, не вказуючи явно в коді, як воно буде заванта-

жено і розподілено. Ви змушені лише при реальній потребі сконфігурувати систему акторів на відповідній топології, яка буде використовуватися.

Є успішні приклади зв'язування моделі акторів Scala з такими проектами, як Terracotta та GridGain, які використовуються також для досягнення розподіленості між вузлами на різних віртуальних машинах Java.

Щодо підтримки розподіленості в Haskell існує декілька варіацій:

1) GdH: Glasgow Distributed Haskell – підтримує ефективну взаємодію між багатьма вузлами;
2) Eden – надає достатньо контролю для імплементації паралельних розподілених алгоритмів, звільняючи від необхідності слідкувати за низькорівневими деталями реалізації;
3) Cloud Haskell – надає можливість створювати програми для розподілених програмних систем, використовуючи модель повідомлень, яка схожа на Erlang.

Ефективне використання багатоядерності

Розглядається приклад того, як збільшується продуктивність розв'язання задачі при використанні всіх можливостей багатоядерних архітектур. Прикладом буде розв'язання задачі пошуку «досконалих чисел на проміжку» мовою програмування Scala.

Досконале число – це натуральне число, що дорівнює сумі всіх своїх власних дільників (усіх додатних дільників, які не дорівнюють цьому числу).

Перше досконале число 6 має такі дільники: 1, 2, 3, сума яких дорівнює 6. Друге досконале число – 28 має такі дільники: 1, 2, 4, 7, 14, сума яких дорівнює 28.

За означенням, досконалим числом є таке, що сума всіх його дільників, включаючи себе самого, дорівнюватиме подвоєному добутку цього числа:

```
def factorsSum(number: Int) =
{(0 /: (1 to number)) { (acc, i) => if (number % i ==
0) acc + i else acc }
def isPerfect(number: Int) = 2 * number ==
factorsSum (number).
```

Цей код обчислює суму всіх дільників для даного числа послідовно. Він має кілька недоліків. По-перше, для великих чисел послідовне виконання буде повільним. По-друге, якщо запускати цей код на багатоядерній машині, то додаткові ядра не використовуватимуться, бо всі обчислення будуть спрямовуватися на одне ядро.

Можна отримати ефективніше виконання на багатоядерній архітектурі за допомогою розбивання обчислення суми дільників на декілька потоків. Навіть на одноядерній машині застосування отримає більше процесорного часу і працюватиме швидше при такій реалізації.

Тому можна розбити ряд чисел від 1 до шуканого числа на певне число проміжків і делегувати знаходження суми для кожного проміжку різним потокам.

```
import scala.actors.Actor._
def factorsSumRange(lower: Int, upper: Int, number:
Int) =
        {(0 /: (lower to upper)) { (acc, i) => if (number
        % i == 0) acc + i else acc }

def isPerfect2(number: Int) = {val N = 1000000
        val partitions = (number.toDouble /N).ceil.toInt
        val caller = self
        for (i <- 0 until partitions) {val lower = i * N + 1;
                val upper = candidate min (i + 1) * N
                actor {caller ! factorsSumRange  (lower,
                upper, number)}
        val sum = (0 /: (0 until partitions)) {
        (partialSum, i) =>
                receive {case sumInRange : Int =>
                partialSum + sumInRange }
      2 * number == sum}
```

Для кожного проміжку обчислення часткових сум дільників делегувалося окремим акторам. Коли актор завершує своє завдання, він надсилає повідомлення з частковою сумою у так званий «збирач» (combiner). У ньому методом foldLeft() (скорочено в Scala – /:) обчислюється сума всіх часткових сум у чисто функціональному стилі.

Для знаходження 33550336 п'ятого досконалого числа, різниця між виконанням послідовного і паралельного алгоритму не стала дуже вагомою. Тому, щоб збільшити інтенсивність обчислень, поставлено таку задачу: знайти досконалі числа на проміжку значень.

Код для розв'язання цієї задачі:

```
def countInRange(start:Int, end:Int, f:Int =>
Boolean)={
        val startTime = System.nanoTime()
        val numberOfPerfectNumbers = (0 /: (start
        to end)) { (count, number) =>
                if (f (number)) count + 1 else count}
        val endTime = System.nanoTime()
```

```
        println("Found" +
        numberOfPerfectNumbers + "perfect
        numbers in given range, took" + (endTime -
        startTime)/1000000000.0 + "secs")}
val startNumber = 33550300
val endNumber = 33550400
countPerfectNumbersInRange(startNumber,
endNumber, isPerfect)
countPerfectNumbersInRange(startNumber,
endNumber, isPerfect2).
```

Порівняння ефективності мов Scala, Erlang та Haskell

Для порівняння ефективності мов Scala, Erlang і Haskell розглянемо задачу множення матриць. Оскільки багато обчислень відбуваються незалежно одне від одного, їх можна легко розпаралелити.

Кожна комірка результуючої матриці повністю незалежна від інших комірок. Вся необхідна інформація для обчислення значення певної комір-

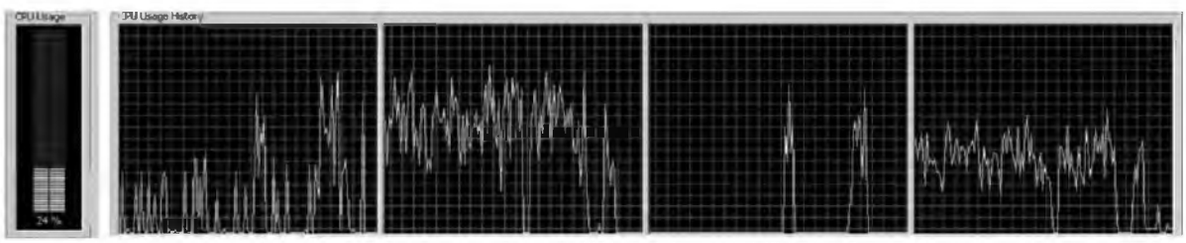

Рис. 1. Виконання програми без використання акторів

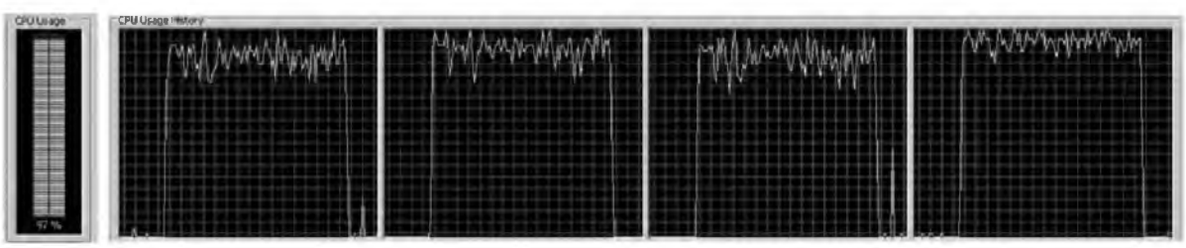

Рис. 2. Виконання програми з використанням акторів

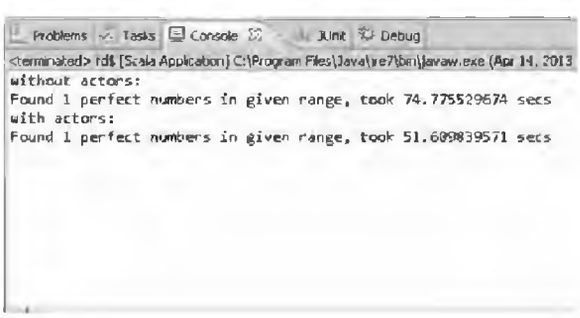

Рис. 3. Порівняння часу виконання

У методі countPerfectNumbersInRange обчислюється кількість досконалих чисел в заданому проміжку від start до end. Третім параметром є об'єкт-функція, що відповідає методу пошуку досконалого числа, який буде передаватися. Час виконання обчислюється методом System.nanoTime(), що входить до стандартного пакету Java.

Результати показали, що програма без застосування акторів не використовує всі можливості процесорів і завантажує їх лише на чверть, тому час виконання довший порівняно з іншою реалізацією. Програма, яка використовує можливості багатоядерності Scala, завантажує процесор майже на максимальну потужність, зменшуючи істотно час виконання (з 75 секунд до 52)

ки складається з відповідного рядка і стовпчика двох матриць-операндів. Це те, що дозволяє легко розпаралелити обчислення добутку матриць.

Було створено реалізацію цього алгоритму мовами Erlang, Scala і Haskell. Результати представлено на графіках. Протестовано алгоритм на матрицях 400x400 та 500x500, на одно- , двох- і чотириядерних режимах чотириядерного процесора (Intel® Core™ i5 CPU 650 @ 3.2 GHz) на комп'ютері з 4 Гб пам'яті.

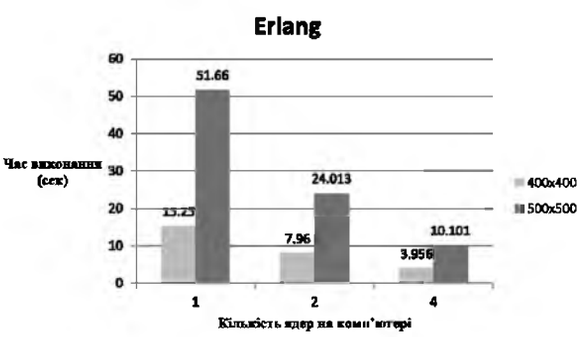

Рис. 4. Результати тестування алгоритму множення матриць мовою Erlang

За результатами роботи, найшвидшою реалізацією виявився варіант мовою Erlang. Слід зазначити, що Haskell і Scala теж показали гарні

результати у порівнянні. Наприклад, реалізація на Scala дозволила отримати найкращий приріст ефективності при переході з одноядерної до чотириядерної архітектури.

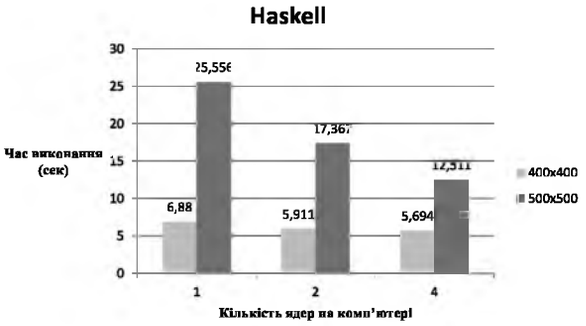

Рис. 5. Результати тестування алгоритму множення матриць мовою Haskell

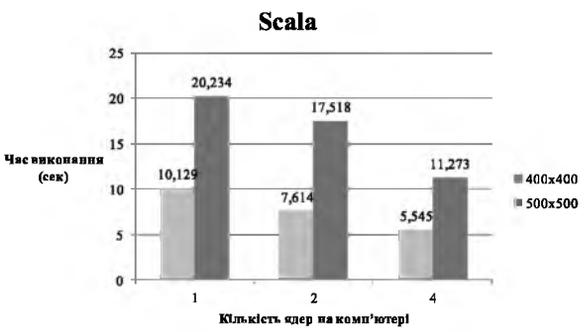

Рис. 6. Результати тестування алгоритму множення матриць мовою Scala

Отже, після детального розгляду основних можливостей таких мов, як Erlang, Scala і Haskel, та порівняння їх ефективності на прикладі розв'язання задачі множення матриць, досліджено, що всі ці мови дають суттєвий приріст ефективності при збільшенні кількості ядер, доступних у системі.

Кожна з мов пропонує свої варіанти вирішення складних питань, які виникають при використанні багатоядерності (забезпечення багатопоточності, збирач сміття, підтримка розподіленості тощо). Важко сказати, яка з цих мов є найефективнішою, адже кожна з них має як сильні сторони, так і слабкі. Erlang, наприклад, дозволяє легко будувати розподілені застосування, має ефективну систему передачі повідомлень, що забезпечує підтримку багатоядерності. Haskell підтримує концепцію «лінивих обчислень», що відкладає виконання складних обчислень до того часу, коли вони справді стануть необхідними. Scala дозволяє взаємодію з іншим кодом, написаним для віртуальної машини Java, що є дуже зручним для програмістів, які знають Java; також є можливість використовувати численні бібліотеки, доступні для мови Java.

На прикладі задачі знаходження «досконалих» чисел на певному проміжку показано, як за допомогою акторів отримати значне прискорення програми на багатоядерних архітектурах.

На прикладі задачі множення матриць порівнювалися основні можливості підтримки багатоядерності в Erlang, Scala і Haskell та ефективність програм при збільшенні ядер, доступних у системі.

Список літератури

1. Lipovaca M. Learn You a Haskell for Great Good! A Beginner's Guide [Електронний ресурс] / M. Lipovaca. – 2011. – Режим доступу: http://learnyouahaskell.com. – Назва з екрана.
2. Odersky M. Programming in Scala. Second Edition / M. Odersky, L. Spoon, B.Venner. – Walnut Creek, California : Artima Press, 2010. – 883 p.
3. Logan M. Erlang and OTP in Action / M. Logan, E. Merritt, R. Carlsson. – Manning Publications Co, 2009. – 397 p.
4. Subramaniam V. Programming Scala: Tackle Multi-core Complexity on the JVM. / V. Subramaniam. – The *Pragmatic Bookshelf*™, 2009. – 250 p.
5. Odersky M. Functional Programming Principles in Scala [Електронний ресурс] / M. Odersky // École Polytechnique Fédérale de Lausanne. – Режим доступу: https://www.coursera.org/course/progfun-002/class/index. – Назва з екрана.
6. Nilsson N. The multicore crisis: Scala vs. Erlang. [Електронний ресурс] / N. Nilsson. – 2008. – Режим доступу: http://www.infoq.com/news/2008/06/scala-vs-erlang. – Назва з екрана.

S. Gorokhovsky, M. Kravchenko

COMPARISON OF EFFICIENCY OF LANGUAGES SCALA, ERLANG, AND HASKELL UNDER MULTICORE ARCHITECTURES

In this paper, a comparison of the main features of Scala, Erlang, and Haskell languages and efficiency in the practical problems solving using multicore architectures. Example of finding the "perfect" numbers for various intervals has been shown as by the actors to get significant acceleration.

Keywords: multicore architectures, actors, languages Scala, Erlang, Haskell.

Матеріал надійшов 01.10.2013