



**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
“Києво-Могилянська Академія”**

Магістерська програма:
“Інформаційні управляючі системи та технології”

**Кваліфікаційна робота
на здобуття академічного ступеня магістра**

на тему:

„Проблеми координації в розподілених обчисленнях”

Виконав:
Студент II року навчання
Микола Стукало

Науковий керівник:
доц.. С. С. Гороховський

Київ 2009

Зміст

1. Вступ.....	4
1.1.Структура роботи.....	6
1.2.Вимоги до програмного середовища.....	7
2. Основні підходи до координації.....	8
2.1.Вимоги до систем координації.....	10
2.2.Модель data-driven координації.....	11
2.3.Модель control-driven координації.....	13
2.4.Області застосування.....	15
3. Координаційна модель Linda.....	16
3.1.Простір кортежів.....	16
3.2.Кортеж.....	17
3.3.Операції.....	18
3.3.1.out.....	18
3.3.2.in.....	18
3.3.3.rd.....	19
3.3.4.eval.....	19
3.4.Правила відповідності.....	20
3.5.Програмовна бібліотека пошуку відповідей.....	21
3.6.Проблема багатьох операцій rd.....	22
3.7.Імплементация.....	23
3.7.1.Law-governed Linda.....	24
3.7.2.Bonita.....	25
3.7.3.JavaSpaces та GigaSpaces.....	27
4. Мова програмування Scala.....	28
5. Linda для Scala.....	30
5.1.Типи даних.....	30

5.1.1.Наперед визначені типи даних.....	31
5.1.2.Створення нових типів даних.....	32
5.2.Особливості реалізації операцій.....	33
5.3.База даних для кортежів.....	35
5.4.Linda-клієнт.....	36
5.5.Linda-сервер.....	37
5.6.Простір кортежів.....	37
5.7.Додаткові операції.....	38
6. Висновки.....	40
7. Джерела.....	41

1. Вступ

Використання розподілених паралельних систем – важливий напрямок розвитку комп'ютерної техніки. Складні обчислення, такі як, наприклад, прогнозування змін клімату, вимагають виконання величезної кількості арифметичних операцій. Окремий комп'ютер не в змозі надати необхідної потужності для виконання подібних операцій. В таких випадках використання паралелізму є перевіреним засобом. За такого підходу проблема розбивається на маленькі частини і розподіляється на мережу комп'ютерів. Кожен комп'ютер працює над своєю частиною проблеми, а загальний розв'язок зводиться з цих маленьких частинок. Такий підхід простіший і дешевший ніж створення та підтримка суперобчислювальних кластерів чи суперкомп'ютерів.

Такий підхід до обчислень використовується в популярному проекті вчених зі Стенфордського університету Folding@home¹. Основна задача проекту – за допомогою моделювання процесів згортання/розгортання молекул білку отримати краще розуміння причин виникнення хвороб, що викликаються дефективними білками, таких як хвороби Альцгеймера, Паркінсона, діабет типу II, хвороба Крейтцфельдта – Якоба та склероз. Для виконання обчислень використовуються сотні тисяч персональних комп'ютерів з усього світу. Для участі в проекті необхідно завантажити невелику програму-клієнт, яка працює в фоновому режимі, коли ресурси процесора не повністю використовуються іншими застосуваннями.

Прогнозування змін клімату чи проект Folding@home – лише окремі приклади складних обчислень, де можна застосувати розпаралелювання задач. Крім цих застосувань, паралелізм може бути використаний для:

¹ <http://folding.stanford.edu/>

- вирішення проблеми керованого термоядерного синтезу;
- розв'язання задач медичної науки та молекулярної біології, генної інженерії;
- вивчення і моделювання змін клімату;
- створення нових напівпровідникових та композитних матеріалів; робіт у галузі машинного інтелекту і робототехніки;
- фінансового моделювання та торгівлі;
- створення складної тривимірної анімації та комп'ютерних спецефектів. [Barker 2000]

В будь-якому випадку, існує необхідність якось розпаралелити проблему і забезпечити комунікацію між комп'ютерами, що беруть участь у обчисленнях. Існують спеціалізовані API (Application Programming Interface, прикладі програмні інтерфейси), серед яких виділяється Message Passing Interface (MPI², інтерфейс передачі повідомлень). Назва "інтерфейс передачі повідомлень" говорить сама за себе. Це добре стандартизований механізм для побудови паралельних програм в моделі обміну повідомленнями. Існують стандартні "прив'язки" MPI до мов C/C++, Fortran 77/90. Безкоштовні та комерційні реалізації є майже для всіх суперкомп'ютерних платформ, а також для мереж робочих станцій UNIX і Windows. В даний час MPI найбільш широко використовується і динамічно розвивається серGelernter - Generative communication in Linda .pdfед представників свого класу.

В середині 1980-х років Девід Гелернтер з Йельського університету запропонував скоріше підхід, ніж окрему мову, для паралельного програмування, що називалася Linda [Gelernter 1985]. Ідея побудови системи Linda виключно проста, а тому красива і дуже приваблива. Паралельна програма складається з багатьох паралельних процесів, кожен

2 <http://www.mcs.anl.gov/research/projects/mpi/>

з яких працює як звичайна послідовна програма. Усі процеси мають доступ до спільної пам'яті, одиницею зберігання в якій є кортеж. Для зв'язку процесів і спільної пам'яті існує всього 6 операцій і в більшості випадків цього цілком достатньо. В цій тезі описано координаційну модель Linda і її імплементацію для мови Scala.

1.1. Структура роботи

Розділ 2 описує загальні поняття координації, вимоги до систем координації та основні підходи для задоволення цих вимог

Розділ 3 описує координаційну модель Linda, організацію простору кортежів, структуру кортежу, операції, необхідні для комунікації між процесами та простором кортежів. Крім того, розглянуто розширення початкової моделі щодо двох нових операцій та методу пошуку кортежів. На останок цей розділ дає опис декількох попередніх імплементацій моделі Linda.

Розділ 4 дає короткий огляд мови Scala, її особливостей та можливості щодо паралельних обчислень.

Розділ 5 описує імплементацію Linda для мови Scala, розроблену для цієї тези

1.2. Вимоги до програмного середовища

Бібліотека, написана в процесі роботи над цією тезою потребує для роботи середовище розробки Eclipse³ зі встановленим плагіном для роботи з мовою Scala або окремий пакет з інтерпретатором командного рядка та компілятором, який можна знайти на сайті Scala⁴.

Розроблена бібліотека є кросплатформенною, оскільки мова Scala використовує Java для своєї роботи і компілюється в байт-код Java. Бібліотека була розроблена в середовищі Mac OS X версії 10.5.7 зі встановленою Java™ 2 Runtime Environment версії 1.5.0_16. Версія Eclipse SDK 3.4.2, версія Scala 2.7.4.

3 <http://www.eclipse.org/>

4 <http://www.scala-lang.org/>

2. Основні підходи до координації

Використання всього потенціалу масштабних розподілених систем вимагає наявності програмних моделей, які явно оперують поняттями паралельної співпраці між дуже великим числом активних сутностей, які складають єдине застосування. Така потреба призвела до проектування та впровадження декількох координаційних моделей разом з окремими мовами, які підтримують ці моделі. Майже всі ці схеми створювалися для того, щоб надати розробникам каркас (framework), який би покращував модульність, сприяв би повторному використанню компонентів (послідовних чи вже паралельних), підвищував би переносимість та мовну сумісність. Однак, ці моделі відрізняються в тому, як саме визначається поняття координації, що конкретно координується, якими засобами досягається координація, і які метафори застосовані для представлення цих понять.

Розробка та впровадження великих і складних систем, таких як авіадиспетчерська служба, транспортна навігація, інтелектуальний пошук та обробка даних, показали, що єдина мова програмування чи системна архітектура не в змозі охопити всі можливі ситуації, які виникають при розробці такого складного та багатофункціонального застосування. Підхід до програмування великих застосувань було знайдено в ідеях багатомовності та гетерогенності. Багатомовне або мультипарадигменне програмування підтримує декілька різних парадигм програмування, забезпечує зв'язок між ними та ізолює небажану взаємодію. Під гетерогенністю мається на увазі те, що мова програмування для гетерогенних систем повинна підтримувати багато різних моделей обчислень. [Papadopoulos 1991]

Концепція координації тісно пов'язана з багатомовністю та гетерогенністю. Розробку паралельного чи розподіленого застосування можна представити як розробку двох окремих частин: власне обчислювальна частина об'єднує процеси, відповідальні за обробку даних, а координаційна частина бере на себе комунікацію та кооперацію між процесами. Тому координацію можна використати для відділення обчислень в розподіленому чи паралельному застосуванні від власне комунікації, що дозволяє проводити розробку двох частин окремо і потім об'єднати результати роботи. Оскільки координаційний компонент системи відділено від обчислювального, то останній можна вважати “чорною скринькою”, і тому конкретна мова, використана для обчислень, не має значення для координаційного механізму. І оскільки мови для обчислень можуть бути різними, то координація заохочує використовувати гетерогенні архітектури.

Загалом, концепція координації стосується не тільки програмування чи комп'ютерних наук. В своїй роботі Мелоун (Melone) та Кроустон (Crowston) говорять про координацію як про міждисциплінарний підхід, який відіграє важливу роль в таких галузях як економіка, дослідження операцій, біологія. За їх загальним визначенням,

Координація – це управління залежностями між процесами.

(Coordination is managing dependencies between activities) [Malone 1994]

Щодо комп'ютерних наук, то Девід Гелернтер та Ніколас Кареро дали найбільш поширене та загальноприйняте визначення координації так:

Координація – це процес створення програм за допомогою склеювання активних шматочків. (Coordination is the process of building programs by gluing together active pieces.) [Gelernter 1992]

Відповідно, координаційна модель – це той клей, який склеює окремі процеси в єдиний ансамбль [Papadopoulos 1991].

Координаційні моделі можна розділити по багатьом критеріям, але найбільш поширена класифікація на дві категорії: data-driven (орієнтовані на задачу) або control-driven (орієнтовані на процеси) координаційні підходи.

2.1. Вимоги до систем координації

В своїй роботі Девід Гелернтер визначає різні вимоги до координаційних моделей. Серед них він згадує ортогональність, простоту, загальність. [Gelernter 1985]

В галузі програмування під *ортогональністю* мається на увазі поєднання незалежних концепцій. Наприклад, для паралельного програмування необхідна обчислювальна та координаційна моделі, які є ортогональними або незалежними. Кожна модель відповідає за свою частину роботи, і вони не втручаються в сферу відповідальності іншої моделі. Такого виду ортогональність робить можливим те, що процес-споживач інформації не мусить мати якихось наперед визначених знань про відправника інформації. Наслідком комунікаційної ортогональності виступає розділення часового та просторового аспектів обміну інформацією, тобто використання розподіленої спільної пам'яті. Просторове розділення має на увазі те, що процеси можуть обмінюватися інформацією через спільний простір даних без жодної інформації про те, хто ввів кортеж в цей простір. Часове розділення дозволяє двом процесам спілкуватися через спільний простір даних без вимоги одночасної їх

активності. Наприклад, процес P1 може помістити в простір даних інформацію I1 для процесу P2. Коли процес P2 стане активним, він може скористатися I1, навіть якщо P1 вже закінчив своє виконання.

Вимога *простоти* стосується в основному комунікації між процесами. Моделі координації, орієнтовані на задачі (такі як Linda), використовують спільний простір даних, що надає розробникам простий спосіб для досягнення різних цілей, головна серед яких – спілкування між процесами. Комунікація процесів між собою через спільний простір даних підтримується завдяки невеликій кількості операцій, які можуть бути легко використані програмістом.

Загальність описує можливість і бажання визначати координаційну модель таким чином, щоб її можна було застосовувати в будь якій множині асинхронних застосувань, таких як розподілені системи.

2.2. Модель data-driven координації

Основною особливістю моделей та мов data-driven координації є той факт, що стан обчислень на будь-який момент часу визначається даними, що отримуються або передаються та фактичною конфігурацією скоординованих компонентів. Іншими словами, координатор чи координований процес відповідає як за перевірку і маніпулювання даними, так і за координацію себе і/або інших процесів за допомогою координаційних механізмів, які забезпечує кожна мова. Це не обов'язково означає, що не існує чіткого поділу координаційних механізмів та чисто обчислювальних функцій деякого процесу. Але, як правило, це значить, принаймні стилістично і лінгвістично, що існує суміш координації та

обчислень в кодї процесу. Мови data-driven координації, як правило, надають деякі координаційні примітиви (у поєднанні з координаційною метафорою), які змішуються з чисто обчислювальною частиною коду. Ці примітиви зручно інкапсулюють комунікаційні і конфігураційні аспекти деяких обчислень, але вони повинні бути використані в поєднанні з чисто обчислювальною обробкою даних, пов'язаних з процесом. Це означає, що про процес не можна легко сказати, координаційний він чи обчислювальний. Як правило, на програміста покладається визначати як відбудеться розрізнення координації та обчислень, хоча у більшості випадків таке чітке розділення не носить обов'язкового характеру на синтаксичному рівні координаційної моделі. [Papadopoulos 1991]

В основу моделі control-driven координації покладено спільний простір даних, який являє собою асоціативну структуру даних. Спільний простір даних відповідає за зберігання інформації, такої як кортежі. Кортеж – це набір значень з мінімальною довжиною в одне значення і теоретичною максимальною довжиною в n значень, наприклад,

(“координата”, 23, 18)

Крім того, кортеж є атомарною одиницею зберігання інформації, не можливо маніпулювати окремим значенням кортежу. Весь кортеж повинен бути отриманий зі спільного простору даних, модифікований і поміщений назад в спільний простір даних. Процес може отримати кортеж за допомогою так званого антикортежу. Спільний простір даних (наприклад, сервер) порівнює цей антикортеж з наявними кортежами і передає відповідний кортеж на початковий процес.

(“координата”, ? x , ? y) чи (“координата”, ? x , 5)

Отриманий кортеж може бути вилучений з серверу або скопійований. Будь який процес може помістити процес в спільний простір даних за

допомогою примітивів, які підтримуються координаційною моделлю. Крім того, через спільний простір даних реалізовано непрямий зв'язок між процесами. Зміст спільного простору даних не залежить від поточного стану всіх учасників координації, тобто процес може помістити кортеж на сервер та припинити своє виконання, а інший процес може одержати цей кортеж. Тому спільний простір даних також допомагає реалізувати просторове та часове розділення між процесами. Деякі моделі координації з таким принципом описані Джорджем А. Пападопулосом і Фархабом Арбабом в [Papadopoulos 1991]. Першим і найстаршим представником цієї координаційної моделі є Linda. Існує декілька розширень оригінальної моделі Linda, таких як Bonita, Bauhaus Linda, Objective Linda та інші, які привносять додаткову функціональність або альтернативне бачення роботи моделі. Функціонування моделі Linda детальніше описано в наступному розділі 3.

2.3. Модель control-driven координації

Основна різниця між моделями data-driven та control-driven координації полягає у тому, що в останньому випадку існує майже повне розділення координації та власне обчислень. Стан розрахунків у будь-який момент часу визначається лише в термінах координаційного шаблону (pattern) до якого належить процес, що бере участь в певних обчисленнях. Фактичні значення даних, над якими працюють процеси, майже ніколи не беруться до уваги. Стилiстично, це означає, що координаційна компонента майже повністю відділена від обчислювальної. Це зазвичай досягається шляхом розробки цілком нової координаційної мови, де обчислювальні

частини розглядаються як "чорні скриньки" з чітко визначеними вхідними/вихідними інтерфейсами. Отож, на відміну від data-driven координації, де існує набір наперед визначених примітивів, control-driven координація майже завжди результує у створення повноцінної мови. Це також означає, що координаційна control-driven модель стимулює розробника розбивати процеси на дві окремі групи, а саме чисто обчислювальні і суто координаційні процеси.

Принцип роботи control-driven координації ґрунтується на взаємодії постачальника (producer) та споживача (consumer) даних. Ця взаємодія визначається як потік або канал даних між вихідними портами постачальників і вхідними портами споживачів. Такі відносини можна уявити у вигляді конвеєра, де вихідні дані постачальників є вхідними даними споживача. Більше того, процес може пересилати у своє навколишнє середовище подійні або контрольні повідомлення для інформування інших процесів щодо зміни свого стану або запиту про стан інших процесів.

Такого виду взаємини між постачальниками та споживачами часто призводять до кількох видів залежностей, які описані Малоуном і Кроустоном у [Malone 1994]. Першою і найбільш поширеною залежністю між постачальником та споживачем є те, що постачальник повинен завершити свою роботу перш ніж споживач зможе почати працювати. Такого виду залежність називається "обмеженнями передумови" (prerequisite constraints). Для таких залежностей має велике значення повідомлення про те, що процес-споживач може почати роботу. Більш того, необхідні послідовне виконання процесів і наявність окремих процесів, які стежать за тим, що процес-постачальник був завершений. Другий вид залежності називається "передача" (transfer) і відповідає за передачу даних

від постачальника до споживача. В даному випадку дані, що передаються – це інформація від постачальника до споживача, і таким чином, передачу даних також можна вважати комунікацією. Третя і остання залежність називається "застосовність"(usability). Ця залежність описує той факт, що результатом роботи постачальника мусить змогти скористатися споживач. Для цієї залежності існує два способи розв'язання: використання запитів до споживача про потрібну йому інформацію чи стандартизація вхідних та вихідних даних.

Яскравим представником control-driven моделі координації є мова програмування Manifold. [Papadopoulos 1991]

2.4. Області застосування

Крім стилістичних відмінностей між двома основними координаційними моделями, які впливають на ступінь розділення обчислювальних та координаційних частин, кожна категорія підходить для різних типів застосувань. Модель control-driven координації, як правило, використовується в основному для розпаралелювання обчислювальних задач. Підхід control-driven координації зазвичай використовується для моделювання систем. Це може бути пояснено тим фактом, що в рамках конфігураційного компонента програміст має більший контроль над даними в разі використання мов, що підтримують data-driven координацію, ніж у випадку control-driven координації. Таким чином, представники першої категорії, як правило, намагаються координувати дані, у той час представники другої намагаються координувати сутності (які, можуть бути не тільки звичайними процесами, а й пристроями, компонентами системи і т.д.).

3. Координаційна модель Linda

Цей розділ дає опис координаційної мови Linda як моделі:

“Linda краще вважати не окремою мовою, а розширенням, яке може бути додано практично до будь-якої мови для полегшення створення процесів, комунікації, та синхронізації.” [Gelernter 1992]

Ця модель складається зі спільної пам'яті, яка називається простором кортежів і декількох операцій для доступу до нього. Ці операції використовуються різними процесами для розміщення, читання чи видалення кортежу з простору кортежів.

Операції моделі Linda, які часто також називають примітивами, дозволяють розробнику координувати процеси, запущені на одній чи на декількох машинах, які працюють в мережі.

3.1. Простір кортежів

Простір кортежів – це спільний простір даних і єдиний посередник для комунікації між процесами. Кожен процес має повний доступ до будь-якого кортежу, може вільно додавати, читати, видаляти всі кортежі. За допомогою метафори спільного простору даних відбувається розділення процесів в просторі і часі (space and time decoupling). Одні процеси можуть розмістити в просторі даних якісь кортежі і продовжувати працювати над обчисленнями чи навіть припинити виконання, а в цей же час інші процеси можуть асинхронно отримати доступ до цих даних. Споживачу даних не

потрібно знати про постачальника даних і навпаки. Таким чином, спільний простір даних виступає ядром комунікації в моделі Linda.

Простір кортежів можна класифікувати на три типи: локальний, розподілений та складений. До локального простору кортежів має доступ лише один процес, який може використати цей простір щоб прочитати чи видалити кортежі з розподіленого чи складеного простору, розмістити їх в локальному просторі і виконати якісь операції над цими кортежами. Складений простір кортежів складається з n просторів кортежів. Наприклад, можна використати 2 простори кортежів, доступних всім процесам, для виконання якихось обчислень, коли m кортежів зберігаються в одному просторі, а результати – в іншому. Розподілений простір складається з одного простору кортежів, доступних всім процесам.

3.2. Кортеж

Кортежі необхідні для комунікації між процесами та простором кортежів. Кортеж являє собою послідовність типізованих значень довільної довжини. Кожне значення має бути з підтримуваного типу даних. Процес створює такі кортежі і розміщує їх в просторі кортежів. Кортежі можуть бути пасивними (містити дані) чи активними (являти собою окремий процес). Процес має змогу створювати так звані антикортежі чи шаблони для знаходження кортежів даних в просторі кортежів. Кортеж називається антикортежем, якщо одне чи більше полів в кортежі містить символ-замісник, наприклад :

("Int", ?x, 20) чи ("divide", 15, Int)

Перші два поля є актуальними параметрами, а останнє поле з

символом-замісником називається формальним параметром.

Способів реалізації кортежів існує багато. В одному випадку це може бути одновимірний масив чи запис, а в іншому випадку вкладені кортежі теж можуть підтримуватися. [Papadopoulos 1991]

3.3. Операції

3.3.1. out

Операція out () є єдиним способом помістити кортеж в простір кортежів .

out (23, "значення", 1)

Кожне поле в кортежі є актуальним параметром. Після розміщення в просторі кортежів кортеж може бути вилучений або прочитаний з будь-якого процесу. Порядок слідування значень в кортежі довільний. Якщо простір кортежів повністю заповнений, то поведінка системи не визначена для операції out(). Наприклад, операцію out() може перервати своє виконання, і/або вивести повідомлення про помилку. В деяких реалізаціях, таких як TCP Linda розмір простору кортежів обмежений тільки наявною віртуальною пам'яттю.

3.3.2. in

Операція in() намагається видалити кортеж з простору кортежів, що відповідає антикортежу. Кожне поле антикортежу може бути фактичним

чи формальним параметром. З наступним антикортеж можна видалити кортеж даних, наведений вище

`in(Int, String,1)`

Процес, який використовує `in()` призупиняє своє виконання до тих пір, поки відповідний кортеж не буде знайдено (тобто якийсь інший процес має розмістити потрібний кортеж в простір кортежів за допомогою операції `out()`). Після цього символи-замісники у антикортежі будуть замінені на відповідні значення в кортежі даних. Наприклад, операцію `in()` можна використовувати для видалення кортежу, його зміни і розміщення назад в простір кортежів.

3.3.3. rd

Операція `rd()`⁵ тотожна операції `in()` вище, з тією лише різницею, що ця операція не видаляє кортеж з простору кортежів. Це значить що кортеж, як і раніше, доступний для інших процесів. Операцію `rd()` можна використовувати таким чином:

`rd(Int, String,1)`

3.3.4. eval

Операція `eval ()` створює активний кортеж шляхом запуску нового процесу, результат роботи якого повертається в даний кортеж після завершення цього процесу. Фактично, якесь місце в кортежі займає виклик

⁵ `rd` означає `read`

певної функції. Наступна операція `eval()` створює простий активний кортеж:

`eval(23, "значення", sin(90))`

Цей активний кортеж перетвориться в наступний кортеж даних

`(23, "значення", 1)`

Варто зазначити, що існують предикатні форми операцій `in()` та `rd()` – `inr()` та `rdp()`. Функціональність `in()` та `inr()` еквівалентна, так же як і `rd()` та `rdp()`. Єдина відмінність полягає в тому, що предикатні форми не призупиняють виконання процесу якщо відповідний кортеж не буде знайдено, вони повернуть значення `хиба` або `(істина, кортеж)` у випадку якщо знайдуть цей кортеж.

3.4. Правила відповідності

Linda використовує асоціативний механізм пошуку відповідностей (Associative Matching Mechanism) для знаходження кортежів в просторі кортежів. Цей механізм базується на таких простих правилах:

- антикортеж і кортеж даних повинні мати однакову кількість полів;
- формальні параметри мають той же тип даних;
- фактичні параметри мають той же тип даних з тим же значенням;
- якщо антикортеж відповідає декільком кортежів даних, один кортеж вибирається довільним чином

Перевага цього механізму – в його простоті, а недолік полягає в недостатній гнучкості в деяких випадках. Наприклад, неможливо отримати максимум чи мінімум значення в кортежі:

`rd (max[Int])` або `in(max[Int])`

У такому випадку процес повинен одержати всі відповідні кортежі, знайти максимальний і, нарешті, відправити усі кортежі до простору кортежів. Слабкістю асоціативного механізму пошуку відповідностей є пошук числових даних в кортежах.[Wells 2004]

3.5. Програмовна бібліотека пошуку відповідностей

Джордж Велс розробив для своєї дисертації [Wells 2001] "Програмовну бібліотеку пошуку відповідностей" ("The Programmable Matching Engine", PME), щоб позбутися слабкості асоціативного механізму пошуку відповідностей, описаного вище. PME входить до складу проекту eLinda і забезпечує розширений механізм для операцій введення з використанням більш гнучких критеріїв для пошуку кортежів. У PME підтримуються інтерфейси та класи Java для розробки нових знаходжувачів збігів (matchers). Тим не менш, деякі запити, які підтримує PME, (наприклад, пошук максимуму) можна здійснюватися з існуючими операціями Linda. На жаль, цей підхід в цілому неефективний.

У PME використовуються трошки інакші операції введення в поєднанні з матчерами, наприклад

`in.maximum`

Крім того, PME використовує розширений синтаксис, щоб позначити поля для знаходжувача збігів. Наступне твердження

`in.maximum("pixe",?x, ?y,? = z)`

використовує знаходжувач максимумів в поєднанні з операцією `in`, щоб знайти максимальне значення `z`. Таким чином, це поле позначене для матчера, а поля `x` та `y` не використовують цей розширений синтаксис. Через

це простір кортежів використовує асоціативний механізм пошуку для цих двох полів.

Такий підхід зводить до мінімуму мережевий трафік, оскільки процесам не потрібно отримувати всі відповідні кортежі. Порівняння здійснюється на боці простору кортежів, а процесам просто передається відповідний кортеж.

3.6. Проблема багатьох операцій rd

За допомогою операції `rd()` можна прочитати кортеж з простору кортежів. Якщо більше ніж один кортеж відповідає антикортежу, то кортеж вибирається довільно і не видаляється з простору кортежів. Це є причиною “проблеми багатьох операцій `rd()`”. Ця операція не деструктивна, отож, немає ніякої гарантії, що всі кортежі які відповідають антикортежу, будуть прочитані з простору кортежів. У гіршому випадку можливе кількаразове повернення одного і того ж кортежу. Простий підхід до вирішення цієї проблеми полягає у видаленні кожного відповідного кортежу за допомогою операцій `in()` та `inpr()` в циклі, здійсненні якихось дій над цими кортежами і поверненні їх назад. Це дуже неефективно, в цей час кортежі недоступні для інших процесів (наприклад, деякі можуть призупинити своє виконання). Для кожного кортежу необхідно виконати 5 операцій:

1. Клієнт повинен надіслати запит до простору кортежів;
2. Простір кортежів повинен зіставити наявні кортежі з антикортежем;
3. Відіслати кортеж до процесу;
4. Цей процес повинен розмістити кортеж в локальному просторі кортежів;
5. Процес повинен відправити назад кожен кортеж.

Щоб вирішити цю проблему, Ентоні Роустон та Алан Вуд запропонували в [Rowstron 1996] новий примітив `copy-collect()`. Синтаксис цього примітиву такий:

`copy-collect(TS t1, TS t2 , <шаблон>)`

Це примітив – недеструктивна операція, що копіює всі кортежі, які відповідають шаблону, з простору кортежів `t1` до простору кортежів `t2`. Ця операція по самій своїй природі вимагає наявності кількох просторів кортежів, наприклад, локального і спільного. У просторі кортежів без будь-якої активності (тобто там не виконують операції інших процесів, які можуть змінити зміст простору кортежів), ця операція копіює всі кортежі з `t1` до `t2` за умови відповідності шаблону. Якщо інші процеси виконують операції, які можуть змінити зміст простору кортежів, результат невизначений. Крім того, різні процеси можуть використовувати `copy-collect()` незалежно і асинхронно для читання кортежів з одного й того ж простору кортежів.

Операція `collect()` була запропонована в [Butcher 1994] Бутчером, Вудом і Аткінсоном. Це деструктивний варіант операції `copy-collect()`, який видаляє всі відібрані кортежі з простору кортежів.

3.7. Імплементациї

Окрім класичної Linda існує багато її розширень та різних варіантів імплементациї. У своїй роботі [Papadopoulos 1991] Джордж Пападопулос згадує багато з них, в тому числі Law-Governed Linda, Bonita, Objective Linda, Sonia, Bauhaus Linda та інші.

3.7.1. Bauhaus Linda

Bauhaus Linda є безпосереднім продовженням простої моделі Linda з з реалізацією складених просторів кортежів у формі мультимножин (multisets, msets). Bauhaus Linda не розрізняє кортежі і простори кортежів, кортежі і антикортежі, активні і пасивні кортежі. Замість додавання, читання або видалення кортежів з одного “плоского” простору кортежів, ця реалізація Linda використовує операції `out`, `rd` та `in` щоб додавати, видаляти чи читати мультимножини з іншої мультимножини. Як наслідок, механізм асоціативного пошуку відповідностей Linda, заснований на порядку та позиції елементів у кортежі, замінюється на невпорядковане включення до множин. До того ж ця мова представляє новий примітив `move`, який використовується для переміщення кортежів вгору чи вниз по рівнях мультимножини, додатковими варіантами якого є операції `up` та `down`. Таким чином можна організувати дані в ієрархічні структури:

```
{“world”
  {“europe”...
    {“ukraine”...
      {“kyiv”...
        {“naukma”...}
      ...}
    ...}
  ...}
}
```


3.7.2. Bonita

Bonita описана в [Rowstron 1998] Вудом та Роустроном. Ця праця пропонує нові примітиви як з метою покращення функціональності моделі так і для збільшення продуктивності. Перша мета досягається за допомогою впровадження складених просторів кортежів та агрегованої маніпуляції кортежами. Для досягнення другої цілі водиться більш точний запис для отримання кортежу, коли запит від якогось процесу на знаходження кортежу обробляється окремо від перевірки чи отримав процес запитувані дані. Таким чином, ці дії можуть бути виконані паралельно і як наслідок зменшуються накладні витрати.

В своїй роботі автори виводять таку формулу для обчислення часових затрат операцій:

$$T_{\text{Pack}} + T_{\text{SendRequest}} + T_{\text{Queue}} + T_{\text{process}} + T_{\text{Block}} + T_{\text{SendReply}} + T_{\text{Unpack}}$$

T_{Pack} – це час, потрібний для виділення необхідної інформації, створення повідомлення та ініціалізація передачі.

$T_{\text{SendRequest}}$ описує час, необхідний для надсилання кортежу до простору кортежів.

T_{Queue} дає значення часу простою, до того моменту як простір кортежів обробить запит.

T_{process} визначає час для знаходження підходящого кортежу, підготовки відповіді та відправлення її до початкового процесу.

T_{Block} – це час до того моменту як потрібний кортеж стає доступним.

$T_{\text{SendReply}}$ описує час для досягнення повідомленням початкового процесу через мережу

T_{Unpack} – це час, необхідний процесу для того, щоб інтерпретувати кортеж.

Значення T_{Block} для всіх не блокуючих операцій, таких як `rdp` або

`inr` завжди дорівнює нулю

Bonita зокрема підтримує такі операції:

`rqid=dispatch(ts, tuple)`

Не блокуючий примітив, розміщує кортеж `tuple` в `ts` і повертає ідентифікатор, який може бути використаний іншими процесами для отримання кортежу.

`rqid=dispatch(ts, template, d | p)`

Не блокуючий примітив, дає доступ до простору кортежів. Видаляє кортеж з простору кортежів, якщо вказано `d`, або повертає копію, якщо вказано `p`. Також повертає ідентифікатор запиту.

`rqid=dispatch_bulk(ts1, ts2, template, d | p)`

Цей не блокуючий примітив копіює (`p`) чи переносить (`d`) всі кортежі, що відповідають шаблону, з `ts1` до `ts2`. Функціональність цього примітиву порівнювана з функціональністю `copy-collect` та `collect`.

Для того щоб перевірити, чи доступний кортеж чи результат, автори пропонують такий не блокуючий примітив:

`arrived(rqid)`

Він повертає `true` чи `false` якщо кортеж вже доставлено. Якщо примітив використано з неправильним `rqid`, то повертається `false`.

Останній з запропонованих примітивів це `obtain` з таким синтаксисом:

`obtain(rqid)`

Він блокує виконання процесу до того як кортеж, визначений `rqid` стане доступним.

3.7.3. JavaSpaces та GigaSpaces

Про JavaSpaces говориться в [Wells 2003]. Це імплементація Linda для Java від Sun Microsystems і частина проекту Jini. Об'єктний простір в цій реалізації використовується як простір кортежів Linda, а об'єкти виступають кортежами. JavaSpaces використовує такі операції для доступу до простору кортежів:

- write розміщує новий об'єкт в об'єктному просторі;
- read повертає копію відповідного об'єкту з об'єктного простору;
- take повертає копію відповідного об'єкту з об'єктного простору і видаляє її.

Ці операції такі самі, як і в оригінальній Linda, тільки з іншими іменами.

Для роботи JavaSpaces необхідні такі сервіси:

- вебсервер;
- RMI activation server;
- сервіс Jini lookup або сервіс реєстру RMI;
- менеджер транзакцій Jini;
- сервер JavaSpaces

JavaSpaces використовує нестандартну серіалізацію об'єктів для передачі через мережу, при якій серіалізуються лише публічні поля класів. Порівняння об'єктів відбувається побайтно, а не за допомогою методу equals.

GigaSpaces, також згадана у [Wells 2003] – комерційна імплементація JavaSpaces з використанням нових можливостей, таких як складені кортежі, чи можливість ітерації по кортежах, які відповідають антикортежу.

4. Мова програмування Scala

Ідеал компонентного програмування, його основна ціль – щоб програми збиралися з бібліотек попередньо написаних компонентів, так само, як апаратура збирається з попередньо виготовлених чіпів.

Автори мови Scala вважають, що, принаймні, частково відсутність прогресу в компонентному програмному забезпеченні пояснюється недоліками мов програмування, які використовуються для визначення та інтеграції компонентів. Більшість існуючих мов пропонує лише обмежену підтримку абстрагування і композиції компонентів. Це стосується, зокрема, таких статично типізованих мов, як Java і C #, які широко використовуються при створенні компонентного ПЗ.

Мова Scala була створена у 2001-2004 рр. в лабораторії методів програмування EPFL⁶. Вона стала результатом досліджень, спрямованих на розробку більш кращої мовної підтримки компонентного програмного забезпечення. На думку авторів Scala, мова програмування компонентних застосувань, по-перше, повинна бути масштабованою в тому сенсі, що має існувати можливість за допомогою одних і тих же концепцій описати як маленькі, так і великі частини. Тому робота над мовою сконцентрувалася на механізмах абстракції, композиції та декомпозиції замість введення великої кількості примітивів, які можуть бути корисними тільки на якомусь одному рівні масштабування. По-друге, масштабована підтримка компонентів може бути надана мовою програмування, яка уніфікує і узагальнює об'єктно-орієнтоване і функціональне програмування. Деякі з основних технічних нововведень Scala – це концепції, які являють собою сплав цих парадигм програмування. У статично типізованих мовах, до

6 <http://www.epfl.ch>

яких відноситься Scala, ці парадигми до цих пір були майже повністю розділені.

Мова Scala як така, не надає жодних примітивів для паралельного програмування. Замість цього ядро мови створювалося так, щоб спростити створення бібліотек, що підтримують різні моделі паралелізму, які будуються поверх поточної моделі мови-основи. [Odersky 2005]

Варто згадати такі ключові аспекти Scala:

- Scala-програми багато в чому схожі на Java-програми, і можуть вільно взаємодіяти з Java-кодом.
- Scala включає уніфіковану об'єктну модель в тому сенсі, що будь-яке значення є об'єктом, а будь-яка операція – викликом методу.
- Scala – це також функціональна мова в тому сенсі, що функції – це повноправні значення.
- У Scala введено потужні і уніфіковані концепції абстракцій як для типів, так і для значень.
- Вона містить гнучкі симетричні конструкції домішок (mixin) для композиції класів та traits.
- Scala дозволяє робити декомпозиція об'єктів шляхом порівняння зі зразком.
- На поточний момент Scala реалізована на платформах Java і .NET.

Основні матеріали по цій мові можуть бути знайдені на сайті <http://www.scala-lang.org/>, серед них варто виділити книгу одного з авторів мови, Мартіна Одерського, “Programming in Scala”. [Odersky 2008]

5. Linda для Scala

Імплементація Linda для Scala, розроблена в рамках роботи над цією тезою, базується на об'єктно-орієнтованому підході. Для представлення типів даних використовуються класи замість вбудованих типів даних Scala. Абстрактний базовий клас для всіх цих класів називається `LindaType`. Кожен клас, який використовується в кортежах, повинен успадковуватися від цього базового класу. Цей абстрактний базовий клас дозволяє розробнику використовувати типи даних, визначені в цій бібліотеці і, що важливо, створювати нові типи даних.

В `LindaType` визначено лише булеву змінну `formal`, яка вказує на те, чи є клас актуальним або формальним параметром. Для пересилання по мережі застосовується вбудована в Java серіалізація об'єктів.

5.1. Типи даних

Кожен тип даних в цій бібліотеці має успадковуватися від базового класу `LindaType`. Це дозволяє реалізувати просте перетворення зі визначених в Scala класів `TupleN` (n -мірний кортеж, $n \in 1, 2, \dots$) до більш зручного з точки зору обробки кортежу у вигляді списку `List[LindaType]`. Необхідність в такому представленні виникла тому, що клас `TupleN` не підтримує ітерацію, що сильно ускладнює порівняння кортежу з антикортежем. Ще одним аргументом для використання списку як структури даних для кортежу було те, що в Scala вже визначено багато функцій для роботи зі списками.

Для порівняння типів даних, похідних від `LindaType`,

використовується перевизначений оператор рівності `==` та допоміжний метод `canEqual`:

```
override def equals (other : Any) = other match {
  case that: LindaInteger =>
    (that canEqual this) &&
    (this.Value == that.Value)
  case _ =>
    false
}

def canEqual (other: Any) = other.isInstanceOf[LindaInteger]
```

Лістинг 5.1. Визначення методу рівності для `LindaInteger`.

5.1.1. Наперед визначені типи даних

Імплементация Linda для Scala має набір наперед визначених типів даних, які наведені в таблиці

клас	тип даних
LindaDouble	Double
LindaFloat	Float
LindaInteger	Int
LindaLong	Long
LindaString	String

Наступний фрагмент коду показує використання типу даних `LindaInteger`:

```
val lint:LindaInteger = 20 //актуальний тип

val lint2 = new LindaInteger (15) //актуальний тип
```

```
lint.toFormal //тепер lint - це формальний параметр, який можна
використати для операції in чи rd
```

Лістинг 5.2. Використання типу `LindaInteger`

Остання лінія коду показує важливу особливість бібліотеки – будь який актуальний параметр можна перетворити на формальний і навпаки за допомогою методів `toFormal` та `toActual`, які визначені в базовому класі `LindaType`.

5.1.2. Створення нових типів даних

В цьому підрозділі описано як створити нові типи даних та використовувати їх в Linda для Scala. Для початку треба визначити новий тип даних, що є похідним від `LindaType`.

```
case class LindaPoint (coord1 : Int, coord2 : Int) extends
LindaType {
  val x = coord1
  val y = coord2

  override def getType = "point" //метод для отримання сигнатури
типу
  override def toString = x+" "+y //метод для перетворення значень
на рядок
}
```

Лістинг 5.3. Визначення класу `LindaPoint`

Ключове слово `case` перед визначенням класу вказує на те, що для цього класу застосовний вбудований в Scala механізм пошуку відповідностей (pattern matching). Для того щоб зберігати кортежів в базі

даних, потрібно отримувати сигнатуру кожного елементу кортежу за допомогою функції `getType`. Потім з цих сигнатур формується рядок, потрібний для знаходження кортежів в базі даних. Функція для отримання такого рядка визначена в об'єкті-синглетоні (singleton object) `LindaTuple`.

```
def getSignature (tuple : List[LindaType]) : String ={
    var output = ""
    tuple.foreach(arg => output = output + " " + arg.getType)
    output.drop(1)
}
```

Лістинг 5.4. Метод для отримання сигнатури кортежу

Також для нових класів потрібно визначити метод для порівняння об'єктів даного класу між собою, як це зроблено в лістингу 5.1.

5.2. Особливості реалізації операцій

Як вже згадувалося раніше, координаційна модель `Linda` містить невелику кількість операцій для комунікації процесів та розподіленого простору даних. Оскільки комунікаційні операції мають приймати на вхід будь-який тип даних і мати простий синтаксис, то логічно оголосити, що вхідний параметр цих операцій має бути `Any` – базовий тип в ієрархії класів `Scala`. Фактично на вхід операцій подається кортеж `Scala` (`TupleN`), який потрібно перетворити на кортеж `Linda`. Тому на кількість вхідних об'єктів накладається обмеження в 22, що пов'язано зі способом реалізації кортежів у `Scala`. Перетворення кортежу в потрібну форму робиться за допомогою функції `createLindaTuple`, що знаходиться в об'єкті-синглетоні

LindaTuple. Ця функція використовується в системі один раз – коли клієнт передає запит від процесу на сервер. Після цього кортеж являє собою список значень типу LindaType.

Для того щоб процес міг зв'язатися з простором кортежів, він має в своєму коді створити новий екземпляр клієнта і вже за його допомогою працювати з серверною базою даних:

```
package org.linda.Test
import org.linda.LindaTypes._
import org.linda.Client._

object process extends Application {
    val i1 = new LindaInteger (28) //створення нових об'єктів даних
    val i2 = new LindaInteger (34)
    val i3 = new LindaInteger (14)
    val s = new LindaString ("hello")

    val client = new Client //створення нового клієнта
    client.out (i1, i2, s) //виконання операції out
    client.out (i1, i3, s)
    i2.toFormal           //перетворення актуального параметру на
формальний
    client.in(i1,i2,s) //виконання операції in
}
```

Лістинг 5.5. Робота з клієнтом

В лістингу 5.3 видно основні моменти роботи з клієнтом – створення нових об'єктів значень, створення клієнта, виконання операцій. Варто зауважити, що остання операція in поверне будь-який з двох кортежів - (28, 34, “hello”) чи (28, 14, “hello”), бо актуальний параметр i2 перетворився на формальний, фактично став шаблоном типу <Int>.

5.3. База даних для кортежів

Під час роботи системи зв'язка клієнт-сервер використовується для передачі запитів до бази даних, яка реалізує спільний розподілений простір кортежів та відповідає за пошук кортежів в цьому просторі.

Структура для зберігання даних – відображення Map, виду [signature : String -> List [List [LindaType]]], де кожному рядку-сигнатурі ставиться у відповідність список кортежів, які мають таку ж сигнатуру. Такий спосіб організації кортежів спрощує пошук потрібного кортежу. Для знаходження відповідності з антикортежем спочатку отримується сигнатура антикортежу. Потім з відображення вибирається список кортежів з відповідною сигнатурою, за допомогою функції find шукаються всі кортежі, які задовольняють шаблон, отриманий з антикортежа. Якщо кортежів більше ніж один, то з них випадковим чином вибирається один за допомогою функції chooseTuple. Вибраний кортеж видаляється з бази даних, оновлений список записується у відображення.

```
def in (antiTuple : List[LindaType]) : List[LindaType]= {
  println(" db in")
  val signature = LindaTuple.getSignature(antiTuple) //отримання
сигнатури кортежу
  val tuple = chooseTuple(find(antiTuple)) //пошук і вибір кортежу
  if (db(signature).length == 1) //якщо вибраний кортеж єдиний з
такою сигнатурою
    db - signature //то видалити сигнатуру з відображення
  else
    db(signature) = deleteTuple(tuple,db(signature)) //видалення
кортежу з бази даних
  tuple //повернути кортеж
}
```

Лістинг 5.6. Реалізація операції in в базі даних

5.4. Linda-клієнт

Клієнт відповідальний за передачу запитів від процесу до серверу і за повернення отриманого результату. Для зв'язку між сервером та клієнтом використовуються Java-сокети. При виконанні якогось методу клієнта, наприклад операцій `in` чи `out`, створюються новий сокет для з'єднання з сервером, потоки (streams) для запису та читання об'єктів з серверу, передається запит на сервер, очікується відповідь і процесу повертаються отримані значення.

```
def in (input : Any) : List[LindaType] = {
    val socket = new Socket(ia, port) //створення нового сокету
    val outputStream = new ObjectOutputStream(new
DataOutputStream(socket.getOutputStream())) //створення нового
вихідного потоку
    val inputStream = new ObjectInputStream (new
DataInputStream(socket.getInputStream())) //створення нового
вхідного потоку
    val antiTuple = LindaTuple.createLindaTuple(input) //створення
кортежу Linda
    outputStream.writeObject(("in", "", antiTuple)) //передача запиту на
сервер
    val tuple = inputStream.readObject().asInstanceOf[List[LindaType]]
//читання об'єкту від сервера
    outputStream.close() //закриття потоків
    inputStream.close()
    tuple //повернення знайденого кортежу процесу
}
```

Лістинг 5.7. Реалізація операції `in` на клієнті

5.5. Linda-сервер

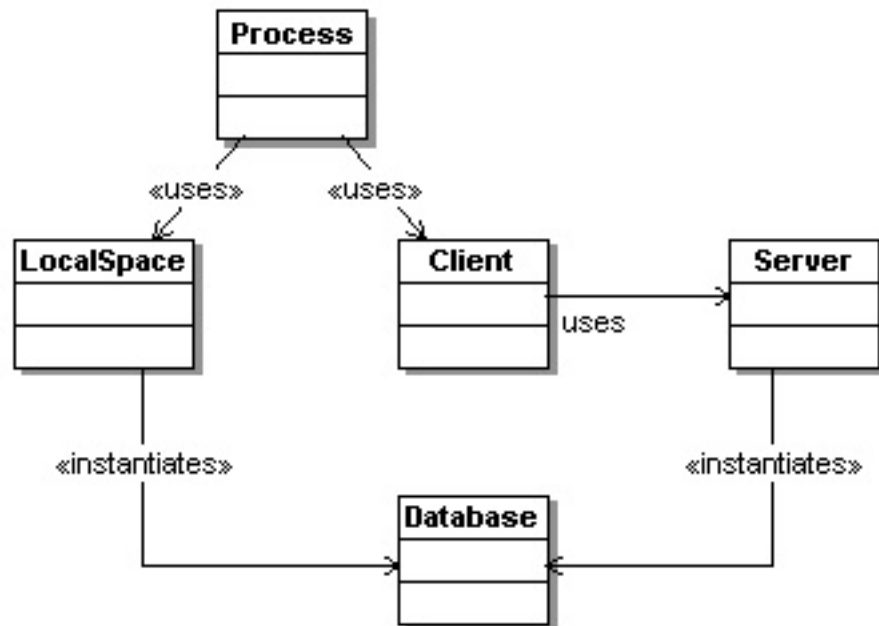
Головна задача сервера – очікувати на запити від клієнтів та виконувати ці запити. Для виконання кожного запиту запускається окремий потік, який і робить запити до бази даних. Потоки можуть виконуватися паралельно. Кожен потік відкриває streams для читання та запису об'єктів з/на клієнт. Саме в потоках реалізовано те, що деякі операції, такі як `in` чи `rd`, призупиняють виконання процесу до тих пір, поки в базі даних не з'явиться потрібний кортеж. Синхронізація між потоками відбувається по об'єкту бази даних за допомогою методу Java synchronized: `db.synchronized {...}`, а коли в базі даних немає відповідного кортежа, виконання потоку призупиняється за допомогою команди `db.wait()`.

```
case "in" => {while (db.find(tuple).isEmpty)
               db.wait()
               outputStream.writeObject(db.in(tuple))
            }
```

Лістинг 5.7. Реалізація операції `in` на сервері

5.6. Локальний простір кортежів

Локальний простір кортежів використовується для підтримки додаткових операцій на зразок `collect` чи `copy-collect`, коли з розподіленого простору кортежів повертається не окремий кортеж, а список кортежів, які необхідно зберегти. Загалом схему взаємодії між клієнтом та сервером з огляду на локальний простір кортежів представлено на діаграмі 5.1.



Діаграма 5.1. Взаємодія в системі Linda для Scala

З діаграми 5.1 видно, що локальний простір даних та сервер використовують один і той же клас бази даних для створення простору кортежів. Для процесу локальний простір виконує ті ж самі функції, що й сервер для клієнта, а саме передає запити процесу до бази даних.

5.7. Додаткові операції

В цій імплементації Linda для Scala реалізовано такі додаткові примітиви чи операції:

- collect
- copy-collect
- query
- copyquery

Дві останні операції також мають предикативну форму, яка не

призупиняє виконання процесу, а повертає true чи false. Операції `coru-collect` та `collect` вже описувалися раніше. Операція `query` схожа на операцію `collect` за винятком того, що вказується кількість елементів, яку необхідно повернути з простору кортежів. Відповідно операція `coruquery` несе ту ж саму функціональність, окрім того що вона не видаляє знайдені кортежі з простору кортежів.

6. Висновки

На сьогодні використання складних паралельних розподілених систем є тим шляхом, по якому пішов розвиток комп'ютерних наук в спробі знаходження розв'язку складних і масштабних обчислювальних проблем. Серед цих проблем можна згадати пошук ліків від хвороб Альцгеймера та Паркінсона, моделювання клімату та його глобальних змін, отримання керованого термоядерного синтезу. Розрахунки, які вимагаються для цих питань, не можуть бути виконані на окремому комп'ютері, тому постає питання про методи розпаралелювання і координації окремих обчислювальних задач. Просте, і в той же час елегантне рішення запропонував Девід Гелернтер у 1985 році, представивши координаційну модель Linda, яка може бути додана практично до будь-якої мови програмування.

Модель Linda базується на концепції спільного простору даних, для доступу до якого у процесів є невеликий, але достатній набір примітивів або операцій. За допомогою цих операцій процес може розмістити, прочитати або вилучити кортеж з спільного простору даних. Процеси спілкуються лише через спільний простір даних, що дозволяє рознести їх у просторі та часі (space and time decoupling).

В цій тезі описано створену бібліотеку Linda для нової мови програмування Scala. Бібліотека використовує клієнт-серверну архітектуру, базу даних для представлення простору кортежів; в ній імплементовано механізм пошуку відповідностей, “примітивні” та додаткові операції. Для подальшого вдосконалення, звичайно, потрібно використання цієї бібліотеки для розробки розподілених паралельних додатків.

7. Джерела

[Barker 2000] Barker, M. (Ed.) (2000). Cluster Computing Whitepaper
<http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/>

(<http://web.archive.org/web/20020203041201/http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/>)

[Gelernter 1985] David Gelernter, Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS), pp 80-112
 Volume 7, Issue 1 (January 1985), ISSN:0164-0925

<http://www.ece.rutgers.edu/~parashar/Classes/03-04/ece572/papers/gencommmlinda.pdf>

[Papadopoulos 1991] Coordination models and languages. George A. Papadopoulos, Department of Computer Science University of Cyprus, Nicosia, Cyprus; Farhad Arbab, Department of Software Engineering CWI Amsterdam, The Netherlands, 1991

<http://www.cse.msu.edu/~stire/cse891s04/Papadopoulos.pdf>

[Malone 1994] T. W. Malone and K. Crowston, The Interdisciplinary Study of Coordination, – ACM Computing Surveys 26, 1994

<http://ccs.mit.edu/papers/CCSWP157.html>

[Gelernter 1992] David Gelernter, Nicholas Carriero. Coordination Languages and their Significance, – Communication of the ACM, 1992 (February), Vol35, No.2

<http://www.caip.rutgers.edu/~virajb/readinglist/coordinationlang.pdf>

[Wells 2004] George C. Wells, New and improved: Linda in Java. - Science of Computer Programming Volume 59, Issues 1-2, January 2006, Pages 82-96.

<http://homes.cs.ru.ac.za/csgw/Research/wellspppj04.pdf>

[Wells 2001] George Clifford Wells, A Programmable Matching Engine for Application Development in Linda University of Bristol, July 2001

<http://homes.cs.ru.ac.za/csgw/Research/ThesisFinal5Oct.pdf>

[Rowstron 1996] A. Rowstron and A. Wood. Solving the Linda multiple rd problem. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Model*, volume 1061 of *Lecture Notes in Computer Science*, April 1996.

<http://www.research.microsoft.com/~antr/papers/coord.ps.gz>

[Butcher 1994] PaulButcher, Alan Wood,Martin Atkins, Global Synchronization in Linda, - Department of Computer Science University of York, Heslington, 1994

<http://web.archive.org/web/19990503000511/http://www.cs.york.ac.uk/linda/ps/collect.ps.gz>

[Rowstron 1998] A. Rowstron and A. Wood. Bonita: A set of tuple space primitives for distributed coordination. - Department of Computer Science, University of York, October 1998.

<http://research.microsoft.com/en-us/um/people/antr/papers/bonita.ps.gz>

[Wells 2003] Linda implementations in Java for concurrent system.

G.C.Wells from Rhodes University, A.G. Chalmers from University of Bristol and P.G. Clayton from Rhodes University. - Concurrency : Practice and Expertise. 2003; 00:1-7

<http://www.cs.bris.ac.uk/Publications/Papers/2000380.pdf>

[Odersky 2005] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, Matthias Zenger. An Overview of the Scala Programming Language, Second Edition. École Polytechnique Fédérale de Lausanne (EPFL) 1015 Lausanne, Switzerland

<http://scala.epfl.ch/docu/files/ScalaOverview.pdf>

[Odersky 2008] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala, First Edition. Published November 17, 2008, 754 pages (eBook)

http://www.artima.com/shop/programming_in_scala