УДК 004.4'24

ПЕРЕТВОРЕННЯ УСПАДКОВАНОГО КОДУ НА FORTRAN ДО МАСШТАБОВАНОГО ПАРАЛЕЛІЗМУ І ХМАРНИХ ОБЧИСЛЕНЬ

А.Ю. Дорошенко, В.Д. Хаврюченко, Є.М. Туліка, К.А. Жереб

Запропоновано підхід до автоматичного перетворення успадкованого коду на мові Fortran для виконання на платформах для хмарних обчислень. Запропоновано архітектуру системи із використанням хореографії веб-сервісів, яка дозволяє необмежене масштабування системи та зменшує накладні витрати із обміну повідомленням. Проведено дослідження підходу на прикладі програми із квантової хімії.
Ключові слова: віртуалізація, хмарні обчислення, масштабований паралелізм, хореографія веб-сервісів.

Предложен подход для автоматического преобразования унаследованного кода на языке Fortran для выполнения на платформах облачных вычислений. Предложено архитектуру системы с использованием хореографии веб-сервисов, которая позволяет неограниченное масштабирование системы и уменьшает затраты на обмен сообщениями. Проведено исследования подхода на примере программы для квантовой химии.
Ключевые слова: виртуализация, облачные вычисления, масштабируемый параллелизм, хореография веб-сервисов.

Proposed an approach to automatic transformation of the legacy code on Fortran for execution on cloud computing platforms. Proposed architecture of the system based on web-services choreography, which allows infinite scalability of the system and reduces overhead on message passing. Investigated an approach of the approach on example of the program from the quantum chemistry field.
Key words: virtualization, cloud computing, scalable parallelism, web-services choreography.

Вступ

Мова Fortran існує з 1950-х років і за цей час зарекомендувала себе як найкращий інструмент для наукових досліджень. Мова має величезну підтримку з боку індустрії програмних систем, постійно виходять нові бібліотеки та компілятори, що дозволяє своєчасно застосовувати новітні технології та стандарти. Прикладом можуть бути компілятор від Potland Group[1] для Fortran з підтримкою GPGPU та CUDA; група High Performance Fortran Forum [2], яка займається розробкою високо продуктивного Fortran; підтримка стандартів OpenMP та MPI для паралельного та розподіленого програмування які застосовуються у сучасних кластерних обчисленнях; Intel займається популяризацією Fortran задля підтримки власних компіляторів та програм написаних на цій мові [3]; відгалуження мови Coarray Fortran увійшло до стандарту мови 2003 року і дозволяє підтримувати роботу із розподіленими масивами [4]; існує величезна кількість бібліотек [5] для чисельних методів обчислення на мові Fortran, які економлять час на імплементацію алгоритмів. До цього також варто додати консервативну політику зворотної сумісності, завдяки якій код, який був написаний із використанням старих стандартів мови Fortran, працюватиме на нових версіях. Все це робить Fortran привабливою платформою і немає жодної ознаки скорого зникнення Fortran із переліку мов для наукових досліджень. Але мова має і свої вади. Основна з них – це застарілість стандартів, що роблять її погано пристосованою до написання ефективного коду для обчислень із розподіленою пам'яттю. Іншою проблемою є велика кількість успадкованого коду що була написана без врахування розподілених архітектур.

Хмарні обчислення дістали розповсюдження в результаті потреби зменшення вартості обчислень. Фундаментальна відміна хмарних обчислень від попередніх підходів до масового паралелізму полягає у використанні масштабованості на противагу продуктивності. Продуктивність – це можливість конкретного компоненту системи виробити певний рівень потужності на виході. Масштабованість це можливість системи до розширення із метою задовольнити попит на додаткові обчислювальні потужності. Як правило масштабованість вимірюють за агрегованою продуктивністю індивідуальних компонент системи, тобто масштабованість вимірює можливість системи до зростання, щоб задовольняти попит на більш вибагливі обрахунки. Масштабовані системи можуть складатись з окремих компонент із низькою продуктивністю. Хмарні обчислення, сфокусовані на оптимізації витрат та балансу ціни/продуктивності за рахунок використання максимальної кількості найдешевших компонент, разом з тим сукупні ресурси систем хмарних обчислень є на порядок більшим ніж ресурси кластерів побудованих навколо Грід: сучасні комерційні системи хмарних обчислень будуються з використанням від 500 тис. до 10+ мільйонів ядер, на відміну від них кластери на основі Грід використовують до 320 тис. ядер.

Мета цієї роботи є вибір моделі розпаралелення прикладної задачі із метою запуску на хмарній платформі, побудова методу напівавтоматичного розпаралелення на основі вибраної моделі та створення необхідної інфраструктури.

1. Опис прикладної задачі та постановка проблеми

В рамках роботи розглянемо задачу із квантової хімії для обрахунку орбіталей електронів [6]. Вхідними даними задачі є геометрія молекули яка складається із координат атомів. В залежності від

складності молекули обсяг вхідних даних варіюється від декількох атомів, до декількох тисяч атомів. Задача є складною проблемою обчислень з високими вимогами до швидкодії. Час обробки зростає квадратично в залежності від кількості вхідних молекул. В той же час невисокі вимоги до обсягу даних – від кількох кілобайт до сотні мегабайт. Програма написано мовою Fortran, з використанням декількох відкритих бібліотек [5]. Програма є прикладом успадкованого коду: еволюціонувала впродовж багатьох років, з використанням різних діалектів мови, Fortran 77 Fortran 90 та Fortran 2003. Початковою метою програми, була ефективна робота на комп'ютерах із одним ядром, без вимог до паралельного виконання. Основною проблемою використання програми є її невисока швидкодія із збільшенням кількості атомів у вхідних даних. Враховуючи, що всі чисельні оптимізації програми були задіяні ще в перших версіях програми, проблема швидкодії має вирішуватись на рівні оптимізації коду та використання кращої обчислювальної техніки.

Логіка програми складається з вводу даних та викликів підпроцедур для виконання обчислень, виводу результату та записів проміжних результатів у файл. Профайлинг програми перед початком оптимізації вказує на наступний розподіл часу для основних кроків

● Зчитування даних із файлів та виділення пам'яті під результуючі змінні. Ініціалізація константних значень в залежності від обраного методу обрахунку – близько 1 %.

● Підпроцедури hcore, яка для кожного атому з вхідних даних підраховує інтеграли є основною за часом і займає 60 % часу.

● Iterc – Оптимізація просторової будови молекули, що вимагає побудови Хессіану, збір Фокіану та побудову two-center repulsion інтегралів, використовує 30 % загального часу. Більше половини цього часу втрачається на діагоналізації симетричної матриці в підпроцедурі Givens.

Виходячи із цих даних, основний фокус паралелізації програми буде присвячено підпроцедурам hcore та iterc. Часова складність алгоритму обчислень в обох підпроцедурах лінійно залежить від кількості атомів у вхідних даних. Кожна із цих двох підпроцедур побудована як цикл за кількістю атомів із кроками обрахунків. Для кожного атома кроки обрахунку є незалежними один від одного за даними. При цьому підпроцедура Iterc залежить від результату виклику hcore за даними. Побудова Фокіану відбувається на основі інтегралів побудованих в hcore.

2. Еквівалентне перетворення коду для масштабованої паралелізації

Вибір методу паралелізації базуватиметься на структурних особливостях програми та особливостях платформи на якій буде запускатись програма. Для хмарних платформ основним способом паралелізації використовується масштабований паралелізм. Якщо цей тип паралелізму застосовний до коду програми, це означає можливість необмежено нарощувати кількість даних, яка програма може обробити. Наприклад, в прикладній задачі, що розглядається, при збільшенні кількості атомів, програма має залучати більше обчислювальних ресурсів.

Структура квантово хімічної задачі спрощено може бути змодельована наступним графом:

$$I \text{ -> } A([a_{1..n}]) \text{ -> } B([b_{1..n}]]) \text{ -> } O.$$

Вузли графу, I, A , B, O відображають послідовні кроки обрахунку задачі: I – введення вхідних даних і ініціалізація змінних, A – обрахунок інтегралів для кожного атому, B – обчислення Фокіану для кожного атому, O – кінцеві кроки та вивід даних. Дуги графу відповідають за передачу управління між вузлами програми, $a_{1..n}$ та $b_{1..n}$ відображають вхідні та вихідні дані підпроцедур. Метою на цьому кроці буде перетворити програму до вигляду:

$$I \text{ -> } A_{1..n}([a_{1..n}]) \text{ -> } B_{1..n}([b_{1..n}]]) \text{ -> } O.$$

Фрагмент коду A , що виконується послідовно для кожного атому, перетворюватиметься на $A_{1..n}$, який позначає виконання процедури на n паралельних процесах, де n це розрядність вхідних даних, аналогічно для фрагменту коду B. Виконуючи ручний аналіз залежності за даними вдалося встановити, що фрагменти A та B можуть бути виконані паралельно, оскільки відсутня залежностей між ітераціями циклу. Тобто процедура A , що перед цим відповідала за обрахунок всього набору даних, після трансформації до вигляду до A_i буде виконуватись ізольовано на кожному доступному процесорі і обраховувати лише один сегмент даних a_i. Оскільки початково B залежала на A за даними, то між паралельними процесами A_i та B_i необхідно ввести бар'єрну синхронізацію. Бар'єрна синхронізація для групи процесів в вихідному коді означає що кожен процес в групі має зупинитись до тих пір поки всі інші процеси в групі не досягнуть бар'єру. На нашому прикладі, процеси $B_{1..n}$ будуть чекати на завершення всіх процесів $A_{1..n}$ і почнуть обробку після завершення останнього A_i . Така структура паралельної програми описується за допомогою класичної моделі fork-join.

Визначившись із моделлю паралельного перетворення, фрагменти коду програми які мають бути виконані паралельно, мають гарантовано позбутись будь-яких сторонніх ефектів, які можуть спричинити

некоректну поведінку при паралельних запусках програми. В прикладній задачі, обмін даними між підпроцедурами відбувається за рахунок звернення до глобальних змінних Fortran COMMON BLOCK. Використання глобальних змінних означає, що кожна з підпроцедур має побічні ефекти – середа виконання змінюється під час виконання процедури. При переході до розподіленої архітектури, зміни в глобальному стані програми на одному обчислювальному пристрої, не гарантують що данні із глобальних змінних будуть доступні іншим ресурсам. В розширенні до Fortran 90 під назвою High Performance Fortran включена підтримка так званих "pure" (чистих) процедур, які згодом увійшли до стандарту мови Fortran 95. Чиста процедура або функція обмежена для того, щоб не мати сторонніх ефектів. Вона ніяк не впливає на стан програми крім дозволених ситуацій: функція повертає значення, підпроцедура модифікує параметри intent(out) та intent(inout). Глобальні змінні і об'єкти асоційовані з будь-якою частиною глобальних змінних заборонені для використання в чистих процедурах. Чисті процедури та функції не мають використовуватись в будь-якому контексті, який може змінити їх значення, наприклад бути об'єктом посилання. Також чисті процедури та функції не мають містити будь-яких зовнішніх операцій введення/виведення, таких як READ та WRITE, а також інструкцій PAUSE та STOP. В даній роботі використовуються ці рекомендації і кожен фрагмент коду, A_i та B_i, що виконується паралельно, має бути перетворено в чисту функцію.

3. Автоматизація перетворень з використанням системи TermWare

Для автоматичного перетворення коду програми, найкраще підходить система TermWare [9]. В попередній роботі [10] був описаний перехід від коду мовою Fortran до абстрактного синтаксичного дерева і побудова тріад TermWare, що трансформують елементи синтаксичного дерева в бажані цільові елементи. Такий самий підхід використовується в даній роботі, але фокусом перетворення в даному випадку є перехід до чистих функцій. Такий перехід описується декількома еквівалентними перетвореннями:

1. Створити FUNCTION замість SUBROUTINE.

2. Позбутись IMPLICIT тверджень – для цього код перетворення має знайти всі локальні змінні функції та явно їх задекларувати.

3. Позбутись COMMON BLOCK які є глобальними змінними – для цього відповідні глобальні змінні мають бути передані на вхід до функції, за копією, а також мають повертатись як результат функції. Тіло програми навколо функції має виконати читання та запис до глобальних змінних до та після виконання, Таке перетворення зберігає еквівалентність коду.

4. Позбутись операцій читання та запису – для цього всі операції читання та запису мають відбутись до та після виклику чистої функції, для підтримання еквівалентності перетворення.

Система TermWare дозволяє описувати перетворення у декларативному вигляді, що спрощує їх розробку та повторне використання. Правила Termware мають наступний загальний вигляд:

source [condition] -> destination [action].

Тут використовуються чотири терми:

- *source* – вхідний зразок;
- *destination* – вихідний зразок;
- *condition* – умова, що визначає застосовність правила;
- *action* – дія, виконувана при спрацьовуванні правила.

Виконувані дії і умови, що перевіряються є необов'язковими компонентами правила, які можуть викликати імперативний код. За рахунок цього базова модель переписування термів може бути розширена довільними додатковими можливостями. Як приклад реалізації перетворень, розглянемо систему правил для перетворення IMPLICIT тверджень на явні декларації змінних:

1. _MarkPure(Subroutine($name,$params,$return,$body))->
Function($name,$params,$return,_MkImp($body)).

2. _MkImp([$x:$y]) -> [_MkImp($x):_MkImp($y)].

3. _MkImp(NIL) -> NIL.

4. _MkImp(Declare($var,$type,$val) -> Declare($var,$type,$val) [check($var, $type)].

5. [_MkImp(Assign($var,$expr)) : $y] [isUnchecked($var)] -> [Declare_MARK($var, $type) :
[Assign($var,$expr) : $y]] [inferType($expr,$type)].

6. [$x : [Declare_MARK($var,$type) : $y]] -> [Declare_MARK($var,$type) : [$x :$y]].

7. Function($name,$params,$return,[Declare_MARK($var,$type) : $y]) ->
Function($name,$params,$return,[Declare($var,$type) : $y]).

Правило 1 запускає перетворення, відмічаючи тіло функції термом-міткою _MkImp. Правила 2 та 3 обходять тіло функції і розповсюджують мітку _MkImp по всім операціям. Правило 4 запам'ятовує ті змінні, для яких присутня явна декларація. Для цього використовується метод з БД фактів check($var, $type). Правило 5 знаходить змінні, яким присвоюється значення, але для яких не було явної декларації (з використанням методу isUnchecked($var)). Для таких змінних визначається тип (метод inferType($expr,$type)) і додається декларація з міткою Declare_MARK($var,$type). Правило 6 піднімає цю декларацію до початку тіла функції, після чого правило 7 прибирає вже непотрібну мітку з терму Declare_MARK. В результаті генерується терм Declare($var,$type), який потім перетворюється на декларацію змінної в коді.

Розглянемо дію описаних правил на прикладі простої процедури множення квадратних матриць. Початковий код має вигляд:

```
SUBROUTINE MATR_MULT(N, A, B, C)
INTEGER,INTENT(IN) :: N
REAL*8,INTENT(IN) :: A(N,N), B(N,N)
REAL*8,INTENT(OUT) :: C(N,N)

DO I=1,N
 DO J=1,N
       S = 0.0D+00
       DO K=1,N
             S=S+A(I,K)*B(K,J)
       END DO
       C(I,J)=S
 END DO
END DO
```

Деякі змінні використовуються без явної декларації. Після виконання правил фрагмент коду перетворюється до наступного вигляду:

```
FUNCTION MATR_MULT(N, A, B)
INTEGER,INTENT(IN) :: N
REAL*8,INTENT(IN) :: A(N,N), B(N,N)
REAL*8 :: MATR_MULT(N,N)

INTEGER :: I, J, K
REAL*8 :: S
DO I=1,N
 DO J=1,N
       S = 0.0D+00
       DO K=1,N
             S=S+A(I,K)*B(K,J)
       END DO
       MATR_MULT(I,J)=S
 END DO
END DO
```

Після перетворення всі змінні задекларовано. Також слід відмітити, що код процедури (SUBROUTINE) синтаксично дещо відрізняється від коду функції (FUNCTION). Але явно описувати такі зміни у вигляді правил не потрібно – достатньо лише замінити терм Subroutine на терм Function. На етапі генерації коду всі необхідні зміни будуть внесені автоматично [10]. Це демонструє одну з переваг використання системи TermWare порівняно з більш простими системами перетворення тексту програми.

4. Перехід до розподіленої програми що виконується у хмарних сервісах

Для переходу до розподілених обчислень, код програми в подальшому має бути трансформовано для підтримки викликів через мережу. Функції A_i та B_i мають бути конвертовані в веб-сервіси – окремі

програми з HTTP інтерфейсом, які можуть бути запущені на віддаленому вузлі через мережевий виклик. Основне тіло програми при цьому перетворюється на контролюючи вузол, який ініціює виклики до зовнішніх веб-сервісів та агрегує результати. Для підтримки роботи з HTTP із Fortran програми може використовуватись libcurl [11] із інтерфейсом на мові C , а перетворення функцій у окремі веб-сервіси здійснюється за допомогою надбудови на мові Java. Таким чином функції A_i та B_i перетворено на мережеву програму яка може бути запущена на вузлі хмарної платформи. Данні для виклику віддаленої програми формуються в контролюючій програмі. Вона має зібрати всі вхідні дані програми та сформувати пакет для виклику віддаленого сервісу, при цьому по мережі передаватимуться тільки ті дані, які згенеровані програмою до початку виконання A_i . Всі інші вхідні данні зберігатимуться локально до веб-сервісу, оскільки можуть бути прочитані при виконанні.

Операційна система хмарної платформи дозволяє конфігурацію автоматичного старту вузлів за заданими параметрами (auto-scaling). Ця можливість використовується контролюючою програмою: після зчитування вхідних даних і визначення кількості атомів, контролююча програма має здійснити виклик до API хмарної платформи та затребувати старту необхідної кількості вузлів необхідного типу. У найпростішому випадку для обробки N атомів знадобиться 2N вузлів системи – для по одному для кожного A_i та B_i . Однак кількість вузлів, час який вони працюють та кількість спожитої пам'яті впливають на вартість обрахунку. Для оптимізації вартості необхідно підібрати оптимальні параметри системи так, що вартість є мінімальною. В досліджені [12] розглядається підхід до оптимізації сервісно-орієнтованої програми за часом ви конання із допомогою оцінки навантаження. Схожий підхід може бути запропоновано з метою мінімізації вартості.

Такий підхід із контролюючою програмою, яка здійснює виклики веб-сервісів, у сервісно-орієнтованій архітектурі називається Оркестрацією. Проте використання Оркестрації має суттєві недоліки. В цій роботі [13] показано що використання окремого контролюючого вузла збільшує кількість викликів між процесами в більшості шаблонів обміну даними у розподілених системах. Кожен мережевий виклик додає час до накладних витрат при обрахунку даних.

5. Перехід до Хореографії

Для вирішення цієї проблеми використано Хореографію Сервісів. Хореографія – це форма композиції сервісів, в якій веб-сервіс сам приймає рішення про те коли і як він має бути введений у роботу. Для цього декларується протокол взаємодії між сервісами і кожен сервіс має знання про цей протокол, під час виконання, сервіс враховує стан процесу та вхідні повідомлення для визначення своєї позиції стосовно інших учасників взаємодії. Знаючи позицію і приймаючи до уваги протокол взаємодії, сервіс приймає рішення стосовно того, якими мають бути наступні кроки. На відміну від Оркестрації, в Хореографії немає єдиного процесу виконання, натомість вона приводиться в дію, коли кожен із учасників виконує свою роль.

З точки зору моделювання розподілених систем, Хореографія може бути змодельована за допомогою асинхронної мережі. Асинхронна мережа складається з множини процесів що коммунікують один із одним. Комунікація може включати прямий обмін повідомленнями із вузла до вузла, broadcast, при якому вузол посилає повідомлення всім вузлам, включаючи себе, та multicats в якому повідомлення посилається підмножині вузлів. При переході до Хореографії, контролююча програма елімінується а її обов'язки перерозподіляться.

Перш за все, має бути сформований протокол взаємодії. Протокол як правило створюється як код на предметно-орієнтованій мові програмування, із використанням синтаксису деякою мовою опису, наприклад json або xml, та описує сценарії взаємодії у форматі: "Якщо поточна роль процесу A_i , та на вхід прийшло повідомлення M_1 , потрібно виконати процедуру F, та передати повідомлення M_2 процесам $A_{2..m}$ ". Цей формат ідентичний опису кінцевого автомату. Такий протокол може взяти на себе відповідальність із запусків контролюючих циклів для A_i та B_i , підготовки повідомлень і виклику сервісів A_i та B_i та обробки результату сервісів. Для того, щоб протокол виконувався, кожен веб-сервіс у свої імплементації має мати контролюючий пристрій, який буде виконувати кінцевий автомат за описом протоколу. Контролюючий пристрій також написано на мові Java. Частина відповідальності за зчитування даних, переходить до самих веб-сервісів. Фрагмент I також здійснював початкову обробку вхідних даних – ця частина має перейти до нового сервісу I_1, який буде стартовим станом в описі автомату.

Виконання хореографії ініціюється початковою подією, з точки зору хмарного середовища це може бути завантаження файлу із вхідними даними на файлову систему. Процес I перебуваючи в стані очікування та отримавши повідомлення про цю подію від файлової системи, приведе в дію фрагмент фортран-коду, який відповідальний за початкову обробку даних. По завершенню цієї обробки сервіс I розсилає broadcast повідомлення із результатами виконання всім процесам асинхронної мережі. Для простоти скажемо, що це повідомлення отримають вузли $A_{1..n}$ та $B_{1..n}$, та I_1. Всі вузли $B_{1..n}$, та I_1 проігнорують це повідомлення оскільки для вузлів B ще недостатньо інформації для початку роботи, а вузол I свою роботу вже виконав. Тоді виконання почнуть процеси $A_{1..n}$.

Важливим моментом є імплементація бар'єрної синхронізації. В роботі [14] описано глобальний та локальний синхронізатори які можуть бути використані для синхронізації процесів асинхронної мережі. Якщо використовується глобальних синхронізатор то має бути задіяний окремий процес який слідкуватиме

за виконанням умови бар'єру. Локальний синхронізатор слідкує лише за сусідами і повідомлення про завершення роботи сервісів поступово розповсюджується по мережі. Це дозволяє зекономити на кількості обмінів повідомленнями. Імплементація синхронізатора задовільнить відповідальність (5). Виклик кінцевого фрагменту O що здійснює вивід результату, має взяти на себе окремий сервіс O_1, який після синхронізації всіх процесів $B_{1..n}$ виконає пост обробку даних та вивід результату.

Оптимізація ресурсів з використанням хореографії досягається за декількох чинників: по-перше кожен веб-сервіс має бути універсальним, вузол системи у хмарному середовищі не має перезавантажуватись для того, щоб запустити новий веб сервіс. Наприклад один вузол на початку може виконувати роль I, а потім взяти на себе роль A_i. Кожен із сервісів що виконували роль A_i можуть взяти на себе роль B_i. По-друге за рахунок відсутності глобального процесу Оркестрації, обмін повідомленнями відбувається лише між окремими процесами. Із використанням локального синхронізатора зовсім можна позбутись глобального стану системи.

6. Тестування підходу

Для перевірки запропонованого підходу обрано спрощену задачу із такою самою моделлю залежності даних – рішення рівняння методом Гаусса. Тестування проводитиметься на базі хмарної платформи Amazon і порівнюватиме дві різні конфігурації системи. Обидві конфігурації мають однакову сумарну кількість процесорів – 8, та оперативної пам'яті 32Gb. Перша конфігурація складається із одного сервера AWS m4.2xlarge (53.5 ECUs, 16 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 64 GiB memory, EBS only). Друга конфігурація складається із чотирьох серверів AWS m4.large (6.5 ECUs, 2 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 8 GiB memory, EBS only). На першій конфігурації запущено послідовну імплементацію програми, на другій – сервісно-орієнтовану імплементацію програми і вимір включає час обрахунку в залежності від розміру вхідних даних. Порівняння замірів показано на рисунку.

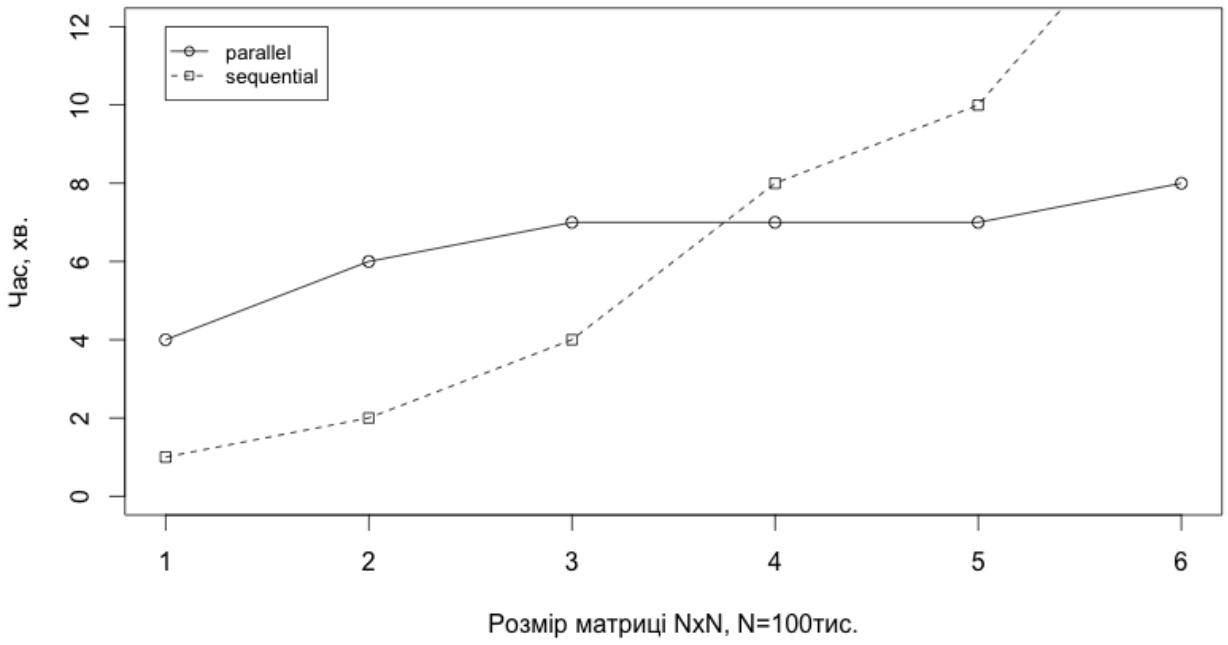

Рисунок. Порівняння часу виконання послідовної та сервісно-орієнтованої версії

Висновки

У роботі виконано аналіз масштабування успадкованого коду на мові фортран для запуску із застосуванням хмарних обчислень. Робота описує відмінності між хмарними обчисленнями та класичними підходами до масової паралельності. Запропоновано архітектуру системи із використанням хореографії веб-сервісів, яка дозволяє необмежене масштабування системи та зменшує накладні витрати із обміну повідомленнями. В якості експерименту використано прикладну програму із квантової хімії по обрахунку орбіталей атомів. Головним результатом роботи є виведення методології роботи із програмним кодом задля пристосування до хмарної інфраструктури. Продемонстровано аналіз коду програми та наведені покрокові дії необхідні для трансформації коду до розподіленої масштабованої архітектури.

1. *PGI* Compilers & Tools [Електронний ресурс]. – Режим доступу: http://www.pgroup.com/products/pvf.htm. – 24.02.2016 р.
2. *High* Performance Fortran [Електронний ресурс]. – Режим доступу: http://hpff.rice.edu/. – 24.02.2016 р.

3. *Fortran* is more popular than ever; Intel makes it fast [Електронний ресурс]. – Режим доступу: https://software.intel.com/en-us/blogs/2011/09/24/fortran-is-more-popular-than-ever-intel-makes-it-fast/. – 24.02.2016 р.
4. *Coarrays* in the next Fortran Standart [Електронний ресурс]. – Режим доступу: ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800/N1787.pdf/. – 24.02.2016 p.https://en.wikipedia.org/wiki/Coarray_Fortran
5. *Netlib* Repository [Електронний ресурс]. – Режим доступу: http://netlib.org/. – 24.02.2016 р.
6. *Дорошенко А.Е., Хаврюченко В.Д., Суслова Л.Н.* Моделирование результатов квантово-химических вычислений // УСиМ. – 2012. – № 5. – С. 83–87.
7. *Kai Li, Paul Hudak* Memory coherence in shared virtual memory systems // ACM Transactions on Computer Systems. – 1989. – Vol. 7, N 4. – P. 321–359.
8. *Bernstein A.J.* Analysis of programs for parallel processing // IEEE Transactions on Electronic Computers EC-15 (5). – 1966. – P. 757–763.
9. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. – 2006. – Vol. 72, N 1–3. – P. 95–108.
10. *Дорошенко А.Ю., Жереб К.А., Туліка Є.М.* Розпаралелювання програм на фортрані з використанням техніки переписувальних правил // Проблеми програмування. – 2012. – № 2-3. – P. 388–397.
11. *Libcurl* – the multiprotocol file transfer library [Електронний ресурс]. – Режим доступу: http://curl.haxx.se/libcurl/. – 24.02.2016 р.
12. *Туліка Є.М.* Підвищення ефективності систем із сервісно-орієнтованою архітектурою за рахунок оцінки і розподілу навантаження // Проблеми програмування. – 2010. – № 2-3 – С. 193–201.
13. *Barker, Adam, Jon B. Weissman, and Jano I. Van Hemert.* Reducing data transfer in service-oriented architectures: The circulate approach. Services Computing, IEEE Transactions on 5.3. – 2012. P. 437–449.
14. *Lynch, Nancy A.* Distributed algorithms. Morgan Kaufmann, 1996.

References

1. PGI Compilers & Tools [Online] Available from: http://www.pgroup.com/products/pvf.htm. [Accessed: 24th February 2016]
2. High Performance Fortran [Online] Available from: http://hpff.rice.edu/. [Accessed: 24th February 2016]
3. Fortran is more popular than ever; Intel makes it fast [Online] Available from: https://software.intel.com/en-us/blogs/2011/09/24/fortran-is-more-popular-than-ever-intel-makes-it-fast/. [Accessed: 24th February 2016]
4. Coarrays in the next Fortran Standart [Online] Available from: ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800/N1787.pdf/. [Accessed: 24th February 2016]
5. Netlib Repository [Online] Available from: http://netlib.org/. [Accessed: 24th February 2016]
6. Doroshenko, A., Khavryuchenko, V., Suslova, L. 2012. Modeling for quantum chemistry computations. Upravlencheskie sistemy i mashiny.- 2012, №5. – P. 83–87.
7. Li, K. and Hudak, P., 1989. Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems (TOCS), 7(4), P. 321–359.
8. Bernstein, A.J., 1966. Analysis of programs for parallel processing. Electronic Computers, IEEE Transactions on, (5), pp.757-763.
9. Doroshenko A., Shevchenko R. A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. – 2006.– Vol. 72, N 1–3.– P. 95–108.
10. Tulika, E., Zhereb, K., Doroshenko, A., 2012. Fortran Programs Parallelization Using Rewriting Rules Technique. Problems in Programming, Kyiv – v.2-3 2012 – P. 388–397.
11. Libcurl – the multiprotocol file transfer library [Online] Available from: http://curl.haxx.se/libcurl/. – [Accessed: 24th February 2016]
12. Tulika E., 2010. Performance Optimization in SOA Using Load Estimation and Load Balancing. Problems in Programming, Kyiv v.2-3 2010 – P. 193–201.
13. Barker, A., Weissman, J.B. and Van Hemert, J.I., 2012. Reducing data transfer in service-oriented architectures: The circulate approach. Services Computing, IEEE Transactions on, 5(3), P.437–449.
14. Lynch, N.A., 1996. Distributed algorithms. Morgan Kaufmann.

Про авторів:

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук, професор,
завідувач відділу теорії комп'ютерних обчислень Інституту програмних систем НАН України,
професор кафедри автоматики і управління в технічних системах НТУУ "КПІ".
Кількість наукових публікацій в українських виданнях – понад 180.
Кількість наукових публікацій в іноземних виданнях – понад 50.
Індекс Гірша – 3.
http://orcid.org/0000-0002-8435-1451.

Хаврюченко Володимир Дмитрович,
незалежний експерт, кандидат хімічних наук.
Кількість наукових публікацій в українських виданнях – 83.
http://orcid.org/0000-0002-0164-0149.

Туліка Євгеній Мирославович,
Інженер програмних систем.
Кількість наукових публікацій в українських виданнях – 3.
http://orcid.org/0000-0002-0153-0148.

Жереб Костянтин Анатолійович
кандидат фізико-математичних наук,
старший науковий співробітник відділу теорії комп'ютерних обчислень.
Кількість наукових публікацій в українських виданнях – понад 30.
Кількість наукових публікацій в іноземних виданнях – понад 10.
http://orcid.org/0000-0003-0881-2284

Місце роботи авторів:

Національний технічний університет України "КПІ",
кафедра автоматики і управління в технічних системах,
03056, м. Київ-056, вул. Політехнічна, 41, корпус 18.

Інститут програмних систем НАН України.
03187, Київ, Проспект Академіка Глушкова, 40.

Тел.: (044) 204 8610, (044) 204 9285.
E-mail: doroshenkoanatoliy2@gmail.com,
eugene.tulika@gmail.com.