УДК 004.4'24 https://doi.org/10.15407/pp2020.04.014

А.Ю. Дорошенко, О.С. Новак

ДО ПИТАННЯ ОПТИМІЗАЦІЇ ХМАРНИХ ОБЧИСЛЕНЬ З УРАХУВАННЯМ ЇХ ВАРТОСТІ

Запропоновано підхід для архітектурних налаштувань паралельних обчислень на хмарній платформі, що дозволяє в напівавтоматичному режимі здійснити оптимізацію паралельної програми з цільовою функцією мінімуму вартості обчислень. Для розв'язання оптимізаційної задачі використано лінійне програмування, що дозволяє підбирати значення архітектурних параметрів конфігурації програми, що істотно впливають на вартість обчислень. Проведено аналітичне випробування розробленого підходу на моделі хмарного мультипроцесорного кластеру, що показує можливість суттєвого скорочення вартості хмарних обчислень внаслідок проведених оптимізацій.

Ключові слова: хмарні платформи, паралельні обчислення, методи оптимізації, лінійне програмування, вартість обчислень.

Вступ

Хмарні обчислення на сьогодні стали однією з найпопулярніших розподілених обчислювальних парадигм. Вони виступають моделлю для забезпечення зручного доступу до мережі та надійного пулу настроюваних обчислювальних ресурсів [1, 2].

Постачальники хмарних обчислень пропонують широкий спектр послуг на різних рівнях: інфраструктурному, платформному та сервісному (відповідно, IaaS, PaaS та SaaS). Еластичність є важливою характеристикою, яка відрізняє хмарні обчислення від інших розподілених обчислювальних моделей. Функція еластичності дозволяє хмарним платформам ефективно обробляти різні навантаження, не порушуючи нормального стану програми. Вона означає не тільки те, що процеси підготовки та розподілу ресурсів можуть бути реалізовані автоматично та динамічно, але також і можливість гнучкої оцінки економічних показників роботи хмарних систем, таких як вартість обчислень [3].

Користувачі платять за хмарний сервіс, виходячи з потужності та часу використання послуг на базі інфраструктури постачальника. Оскільки хмарні сервіси стають все більш і більш популярними, це привертає величезну увагу лідерів постачальників хмарної інфраструктури, таких як Microsoft, Google і Amazon, спонукаючи їх до роботи із все більш централізованими центрами обробки даних. Сучасний хмарний центр обробки даних оснащений значною кількістю обчислювального обладнання, тому навіть розміщення віртуальних контейнерів по реальним серверам вже саме по собі являє собою дуже складну оптимізаційну задачу [4]. Крім того, все це обладнання споживає значну кількість електроенергії, що спричиняє різке зростання експлуатаційних витрат на хмарні дата-центри, які звичайно перекладаються на споживачів хмарних сервісів. Велике енергоспоживання дата-центрів веде також до збільшення викидів вуглекислого газу (CO_2) та погіршення якості навколишнього середовища. Таким чином, зниження енергоспоживання хмарними дата-центрами і, залом, оптимізація хмарних обчислень є актуальною задачею для розробників хмарних сервісів.

У попередніх роботах авторів [5–6] розроблявся метод самоналаштування (автотюнінгу) паралельних програм на цільову платформу, націлений на досягнення максимальної швидкодії конкретної прикладної програми на конкретну архітектуру обчислювальної системи.

У цій роботі запропоновано узагальнений підхід для самоналаштування паралельних обчислень на хмарній платформі, що дозволяє в напівавтоматичному режимі здійснити оптимізацію потоку паралельних завдань з цільовою функцією мінімуму вартості обчислень. Як буде показано далі, для розв'язання оптимізаційної задачі може бути використано лінійне програмування, що дозволяє у напівавто-

матичному режимі підбирати значення параметрів конфігурації програми які істотно впливають на вартість обчислень.

1. Проблема оптимізації вартості обчислень

Оптимізація вартості обчислень – складна задача, яка вимагає від експерта глибоких знань як про предметну область, так і про реальну систему, на якій задача буде виконуватись. А враховуючи те, що сучасна система може складатися з сотень серверів і досить широкого набору програмних засобів – дуже швидко кількість необхідних знань починає перевищувати реальний обсяг даних, яким може оперувати людина.

Навіть якщо таку систему було колись налаштовано – майже завжди цього не достатньо. Хоча в світі існують системи, які безперервно і майже без оновлень працювали десятками років – це рідше виключення, аніж правило. Швидкі темпи розвитку програмних та апаратних засобів призводять до того, що вже через короткий час може з'явитись необхідність оновити систему, щоб вона працювала і за меншу ціну. А крім того, відбувається розширення можливостей і збільшення навантаження на систему. І тоді система, оптимізована під існуюче колись навантаження, просто перестає справлятися з поточним навантаженням, що призводить до необхідності щоразу оптимізувати систему – вже під нові умови її роботи. Таким чином, для досягнення максимальної ефективності оптимізація системи повинна бути не одиничною задачею, а *постійним процесом* який має тривати впродовж всього циклу розвитку та супроводу системи.

Для вирішення проблеми постійного процесу вдосконалення налаштування системи на вхідний потік задач на сьогодні просувається ідея безсерверних (serverless) обчислень [7], тобто, ідея такої хмарної архітектури, яка передбачає використання абстрагованих обчислювальних потужностей без явного зазначення кількості та характеристик обладнання, що використовується. Такий підхід просувається як «срібляна куля», що здатна вирішити усі проблеми. Проте виявляється, що у реальному житті вона не працює, окрім відносно невеликих проектів, де ціна інфраструктури не йде у порівняння з ціною роботи експертів. І як тільки постає питання оптимізації ціни хмарної інфраструктури – постає питання як це зробити найкраще.

Окрім найбільш очевидного «менше обчислень – менша ціна» існує ще велика кількість проміжних варіантів того, як можна масштабувати систему, від використання можливостей тільки апаратних платформ (що частіше і відбувається у реальній практиці користувачів) до зменшення алгоритмічної складності програм, що реалізують функціональні специфікації прикладної задачі. Як приклади можна привести горизонтальне масштабування невеликих контейнерів ECS/EKS у Amazon (аналогічна технологія, але з іншою назвою, є і в інших постачальників хмарної інфраструктури), а також використання так званих точкових екземплярів (spot instances) чи переніс частини обчислень у хмарні функції (Azure functions чи Amazon Lambda) [7]. І хоча кожен з цих кроків може потребувати значної додаткової роботи, на великих масштабах економія від таких оптимізацій може буди досить значною. Але у цьому випадку складності додає той факт, що зазвичай відсутня можливість тонкого налаштування кожного з контейнерів, тому вибирати доводиться з набору типових конфігурацій. Наприклад, для алгоритму що потребує 4vCPU та 8 RAM, є можливість використовувати EC2 в AWS розміром a1.xlarge з ціною 57$/місяць, але вже при необхідності в 5 vCPU доведеться використовувати a1.2xlarge з надлишковою потужністю (бо це вже 8 vCPU та 16 RAM) і вартістю в 104$/місяць [8]. А враховуючи, що зазвичай потужність vCPU пропорційна розміру контейнера, ручний підбір конфігурації, яка за мінімальну вартість зможе обробляти необхідну кількість даних, – надто складний і тривалий процес.

Виходячи з вищенаведеного, можна зробити висновок, що для оптимізації системи недостатньо тільки оптимізувати алгоритм вирішення прикладної задачі. Для максимальної ефективності потрібно правильно підбирати інфраструктуру та оптимізувати алгоритм саме під цю інфраструк-

ктуру. І тут ми отримуємо класичну проблему – для того щоб оптимізувати алгоритм, нам спочатку потрібно зрозуміти, де і як він буде виконуватись, а для того щоб оптимізувати інфраструктуру, потрібно знати скільки саме ресурсів нам буде необхідно.

Саме для вирішення цієї проблеми ми можемо ефективно використовувати метод автотюнінгу. Як механізми оптимізації з необхідним параметром такі тюнери можуть модифікувати вихідний код алгоритму використовуючи правила переписування/підстановки [9]. А вже маючи інформацію про правила заміни які може робити система автотюнінгу (і як вони впливають на ефективність системи відносно іншої конфігурації) – виникає можливість підібрати оптимальну конфігурацію. За допомогою подальшого вдосконалення правил автотюнера можна покращувати результат самоналаштування.

2. Пропонований підхід

Налаштування мультисервісної системи на її оптимальну за вартістю роботу – надто складна наукова та інженерна проблема, щоб її вирішувати у всій її складності. Тому тут найчастіше застосовують евристичні методи, що на жаль не вирішують проблему і вимагають високої кваліфікації персоналу.

Натомість при спрощеному поданні системи у вигляді незалежних ланок (у випадку, коли це взагалі можливо) (див. рис. 1) вирішення цієї задачі можна звести до розв'язання модифікованої класичної задачі про рюкзаки за умови, що треба мінімізувати загальну вартість рюкзаків, необхідних для збору усіх речей. Це класична NP-повна задача з добре відомими підходами до вирішення. Зважаючи на відносно невелику кількість наявних варіантів, у даному випадку для вирішення цієї задачі є можливість навіть не будувати специфічний солвер (програма-розв’язувач задач), а використати лінійне програмування з використанням існуючого солвера. Хоча такі програмні продукти загалом і поступаються специфічним алгоритмам, але завдяки своїй універсальності вони допомагають досить швидко якщо і не вирішити задачу повністю, то хоча б отримати наближений результат [10,11]. У нашому випадку було використано саме python-mip солвер з використанням coin-or [12]. Він використовує підхід гілок і границь щоб отримати гарантовано оптимальне рішення для заданої моделі.

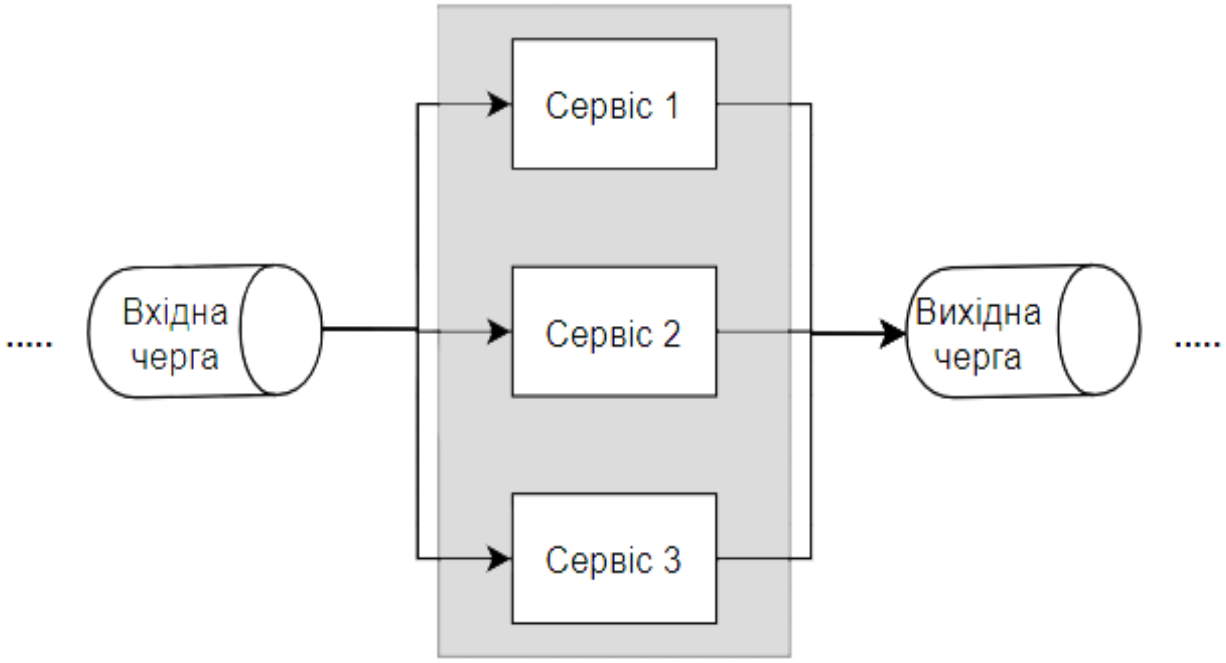

Рис. 1. Ланка архітектури для обробки даних мультисервісної платформи, здатна до горизонтального масштабування

Тобто, в нашому випадку задача зводиться до мінімізації виразу

$$\sum_{c \in C} S_c \times P_c$$

при обмеженні, що

$$\sum_{a \in A, c \in C} S_{(a,c)} \times T_{(a,c)} \leq L$$

де A – набір алгоритмів, C – набір можливих конфігурацій інфраструктурної одиниці, T – швидкодія, P – набір вартостей сервісів, L – мінімально необхідна швидкодія, S – кількість інфраструктурних одиниць.

Тепер, маючи набір можливих конфігурацій інфраструктури та декілька оптимізованих для запуску алгоритмів, можемо знайти оптимальну конфігурацію для навантаження системи.

3. Експеримент

Як експеримент розглянута ланка типового рішення, яка представляє набір з декількох сервісів поміж декількох черг. Розбивши велику задачу на декілька таких ланок ми отримуємо досить просту для моніторингу та масштабування систему, яка в свою чергу може витримувати значне навантаження. Тобто, навіть маючи десятки-сотні сервісів для обробки даних, ми можемо аналізувати пропускні можливості системи за допомогою динаміки кількості даних, що проходять через черги. Це особливо корисно коли потрібно порівняти загальну пропускну спроможність до та після якихось змін у налаштування окремих сервісів. А оскільки ланка сервіс-черга зазвичай не є вузьким місцем системи, то ми отримуємо можливість горизонтального масштабування системи без збільшення її складності.

В табл. 1 наведено список усіх доступних для експерименту контейнерів: де **ram**, **cpu** і **network –** величини що характеризують *відносну* кількість доступних контейнеру ресурсів. Як вже було сказано, у нас відсутня можливість точно налаштовувати розміри усіх характеристик контейнера, тому для порівняння ми використовуємо коефіцієнти відносно найменшого з доступних контейнерів. Тобто, cpu = 2 означає, що vCPU у цьому контейнері має приблизно у 2 рази більшу потужність, ніж у контейнері мінімального розміру. Саме по цій причині всі подальші розрахунки наведені в умовних одиницях.

Таблиця 1. Можливі конфігурації контейнерів.

ціна	назва	ram	cpu	network
1	d_0	1	0.8	1.2
2	d_1	2	1.8	2.2
3	d_2	3	2.8	3.2
4	d_3	4	3.8	4.2
5	d_4	5	4.8	5.2
1	r_0	1.2	0.72	1.2
2	r_1	2.3	2.72	2.2
3	r_2	3.4	4.72	3.2
1	c_0	0.9	0.96	1.2
2	c_1	2.9	2.06	2.2
3	c_2	4.9	3.16	3.2
1	n_0	0.9	0.72	1.68
2	n_1	1.8	1.62	3.68
3	n_2	2.7	2.52	5.68

Окремо слід уточнити причини використаного позначення контейнерів. Хоча й кількість варіантів розміру контейнера обмежена, але зазвичай компанії надають досить велику кількість можливих конфігурацій (у тому числі і специфічних для досить вузького кола задач). Тому склалася така практика, що формат назви являє собою поєднання префіксу, що характеризує тип контейнера, і суфіксу, що характеризує його розмір (як у вищенаведеному прикладі з AWS EC2 a1.xlarge та a1.2xlarge). Саме такий підхід до позначень використаний і тут. В нашому випадку маємо типи d (default), r (ram), c (cpu) та n (network) та декілька розмірів для кожного з них. Тобто, r_0 означає мінімальний розмір контейнера із збільшеною кількістю операційної пам’яті.

Тоді, маючи 2 алгоритми (a1 та a2) що вирішують нашу задачу різними способами (та з різним споживанням ресурсів) ми можемо запустити солвер і знайти оптимальну конфігурацію для навантаження 2500 одиниць (табл. 2). Як можна побачити – найдешевше необхідного результату можливо досягти використовуючи другий алгоритм на 833 контейнерах типу c_0 (мінімальній розмір, збільшена потужність cpu) та 1 контейнеру d_0 (звичайний контейнер мінімального розміру).

Зважаючи, що окрім оптимізації інфраструктури ми ще й використовуємо автотюнінг для оптимізації (прискорення) обчислювальних алгоритмів, ми не можемо виключати, що в нас ще є можливість оновити тюнер таким чином, щоб покращити один з параметрів ціною деякого погіршення іншого.

Тому є сенс додатково розглянути ще й можливість модифікацій характеристик алгоритму, коли погіршення значення одного параметру відбувається для покращення іншого. Тоді можна враховувати можливість автотюнера модифікувати алгоритм (використовуючи правила підстановки) при пошуку оптимальної конфігурації контейнерів. У цьому випадку, наша модель буде модифікована як

$$a_n = a_n^{'} + \Delta a_n ,$$

$$a_n^{'} = \left(cpu_n, ram_n, network_n\right),\ n \in N ,$$

$$\Delta a_n^{'} = \left(\Delta cpu_n,\ \Delta ram_n,\ \Delta network_n\right),\ n \in N ,$$

$$\sum_{e \in \Delta a_n} |e| \le D ,$$

$$\sum_{e \in \Delta a_n} e = 0 ,$$

де D – максимальна можлива величина модифікації (табл. 3).

Таблиця 2. Доступні конфігурації алгоритму

ціна	алг	ram	cpu	net	d_0	d_1	d_2	d_3	d_4	r_0	r_1	r_2	c_0	c_1	c_2	n_0	n_1	n_2
0	a1	0.1	0.2	0.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
834	a2	0.3	0.3	0.4	1	0	0	0	0	0	0	0	833	0	0	0	0	0

Таблиця 3. Приклад найвигіднішої модифікації

ціна	алг	ram	cpu	net	d_0	d_1	d_2	d_3	d_4	r_0	r_1	r_2	c_0	c_1	c_2	n_0	n_1	n_2
0	a1	0.2	0.1	0.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a2	0.1	0.1	0.8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a3	0.2	0.2	0.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a4	0.1	0.3	0.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a5	0.2	0.4	0.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a6	0.2	0.3	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a7	0.4	0.2	0.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a8	0.3	0.2	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	a9	0.4	0.3	0.3	0	0	0	0	0	0	0	0	0	1	1	0	0	0
699	a10	0.3	0.4	0.3	0	0	0	0	0	0	0	233	0	0	0	0	0	0

Як можна побачити, для досягнення мінімальної ціни нам необхідно одночасно використовувати декілька версій алгоритму, які будуть оптимізовані по різних параметрах. Тобто, використавши алгоритм a9 (зі зниженим на 25 % навантаженням на мережевий канал, та збільшенням необхідної пам'яті на 20 %) розгорнутий на двох контейнерах типу c_1 та c_2 одночасно з a10 (зі зниженим на 25 % навантаженням на мережевий канал та збільшеним на 20 % навантаженням на процесор) на 233 контейнерах типу r_2 ми зможемо досягти зменшення загальної вартості на 15.5 %. Як наслідок, можна зробити висновок про наявність вузького місця такої конфігурації і його локалізації. Наприклад, в даному випадку це необхідна ширина мережевого каналу.

Ще одним цікавим моментом є використання пріоритетів характеристик для подальшої оптимізації. Це дозволяє оцінити очікуваний результат від оптимізацій різного типу. По перше, це дозволяє в першу чергу оптимізувати ті частини системи, де результат оптимізацій буде найбільш помітним. По друге – це дозволяє оцінити доречність подальших оптимізацій взагалі. У цьому випадку наші обмеження на зміни будуть модифіковані до наступних:

$$\sum_{e \in \Delta a_n} e \geq D .$$

Тобто, при D = 0.1, ми отримаємо (табл. 4), що відповідає зменшенню загальної вартості на 18.2 % при умові зменшення навантаження мережу на 25 %.

Висновки

В роботі запропоновано підхід до архітектурних налаштувань паралельних обчислень на хмарній платформі, що дозволяє в напівавтоматичному режимі здійснити оптимізацію паралельної програми з цільовою функцією мінімуму вартості обчислень. Для розв’язання оптимізаційної задачі використано лінійне програмування і доступний програмний солвер, який за допомогою методу гілок і границь в напівавтоматичному режимі підбирає значення архітектурних параметрів конфігурації програми, що істотно впливають на вартість обчислень.

Таким чином узагальнено попередній метод автотюнінгу, що розроблявся авторами раніше, та поширено його на випадок комплексу сервісів, що виконуються на хмарній платформі.

Проведено аналітичне випробування розробленої системи на моделі хмарного мультипроцесорного кластеру, що показує можливість суттєвого скорочення вартості хмарних обчислень внаслідок проведених оптимізацій.

Таблиця 4. Приклад найвигіднішого напрямку оптимізації

ціна	алг	ram	cpu	net	d_0	d_1	d_2	d_3	d_4	r_0	r_1	r_2	c_0	c_1	c_2	n_0	n_1	n_2
0	a1	0.1	0.1	0.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a2	0.1	0.2	0.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a3	0.2	0.3	0.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a4	0.3	0.2	0.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
682	a5	0.3	0.3	0.4	0	0	0	0	0	0	0	341	0	0	0	0	0	0

Література

1. Duan Yucong, Fu Guohua, Zhou Nianjun, Sun Xiaobing, Narendra Nanjangud, Hu Bo (2015). "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". 2015 IEEE 8th International Conference on Cloud Computing. IEEE. P. 621–628. doi:10.1109/CLOUD.2015.88
2. Singh Jatinder, Powles Julia, Pasquier, Thomas, Bacon Jean (July 2015). "Data Flow Management and Compliance in Cloud Computing". *IEEE Cloud Computing*. 2(4). P. 24–32. doi:10.1109/MCC.2015.69
3. Zelenyuk V. (2013). "A scale elasticity measure for directional distance function and its dual: Theory and DEA estimation". *European Journal of Operational Research*. 228(3). P. 592–600. doi:10.1016/j.ejor.2013.01.012
4. Azizi S., Shojafar M., Abawajy J. and Buyya R. "GRVMP: A Greedy Randomized Algorithm for Virtual Machine Placement in Cloud Data Centers," in *IEEE Systems Journal*, doi: 10.1109/JSYST.2020.3002721.
5. Anatoliy Doroshenko, Pavlo Ivanenko, Oleksandr Novak, and Olena Yatsenko, A Mixed Method of Parallel Software Auto-Tuning Using Statistical Modeling and Machine Learning in: 14th International Conference, ICTERI 2018, Kyiv, Ukraine, May 14–16, 2018 (Vadim Ermolayev, Mari Carmen Suárez-Figueroa, Vitaliy Yakovyna, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky (Eds.)), Revised Selected Papers, Series: Communications in Computer and Information Science, Springer, Vol. 1007. 2019. https://doi.org/10.1007/978-3-030-13929-2_6.
6. Дорошенко А.Ю., Іваненко П.А., Новак О.С., Старушик А.М. Автотьюнінг паралельних програм з використанням системи аналізу даних IBM Watson Analytics. *Проблеми програмування*. 2018. № 1. С. 46–54.
7. Neil Savage. Going serverless. Communications of the ACM. 2018. Vol. 61(2). P. 15–16. doi:10.1145/3171583
8. https://calculator.aws/#/createCalculator
9. Андон Ф.И., Дорошенко А.Е., Жереб К.А., Шевченко Р.С., Яценко Е.А. Методы алгебраического программирования. *Формальные методы разработки параллельных программ.* Киев: Наукова думка. 2017. 440 с.
10. Gleixner Ambros, Hendel Gregor, Gamrath Gerald, Achterberg Tobias, Bastubbe Michael, Berthold Timo, Christophel Philipp M., Jarck Kati, Koch Thorsten, Linderoth Jeff, L'ubbecke Marco, Mittelmann Hans D., Ozyurt Derya, Ralphs Ted K., Salvagnin Domenico and Shinano Yuji. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. 2020 Mathematical Programming Computation. http://www.optimization-online.org/DB_FILE/2019/07/7285.pdf
11. Rimmi Anand, Divya Aggarwal & Vijay Kumar (2017). A comparative analysisof optimization solvers. *Journal of Statistics and Management Systems.* 20:4. P. 623–635. DOI:10.1080/09720510.2017.1395182
12. A Comparative Analysis of Optimization Solvers. Available from: https://www.researchgate.net/publication/314750497_A_Comparative_Analysis_of_Optimization_Solvers [accessed Nov 14 2020].

References

1. Duan Yucong, Fu Guohua, Zhou Nianjun, Sun Xiaobing, Narendra Nanjangud, Hu Bo (2015). "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". 2015 IEEE 8th International Conference on Cloud Computing. IEEE. P. 621–628. doi:10.1109/CLOUD.2015.88
2. Singh Jatinder, Powles Julia, Pasquier, Thomas, Bacon Jean (July 2015). "Data Flow Management and Compliance in Cloud Computing". IEEE Cloud Compu¬ting. 2(4). P. 24–32. doi:10.1109/MCC.2015.69
3. Zelenyuk V. (2013). "A scale elasticity measure for directional distance function and its dual: Theory and DEA estimation". European Journal of Operational Research. 228(3). P. 592–600. doi:10.1016/j.ejor.2013.01.012
4. Azizi S., Shojafar M., Abawajy J. and Buyya R. "GRVMP: A Greedy Randomized Algorithm for Virtual Machine Placement in Cloud Data Centers," in IEEE Systems Journal, doi: 10.1109/JSYST.2020.3002721.
5. Anatoliy Doroshenko, Pavlo Ivanenko, Oleksandr Novak, and Olena Yatsenko, A Mixed Method of Parallel Software Auto-Tuning Using Statistical Modeling and Machine Learning in: 14th International Conference, ICTERI 2018, Kyiv, Ukraine,

May 14–16, 2018 (Vadim Er¬molayev, Mari Carmen Suárez-Figueroa, Vitaliy Yakovyna, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky (Eds.)), Revised Selected Papers, Series: Communications in Computer and Information Science, Springer, Vol. 1007, 2019. https://doi.org/10.1007/978-3-030-13929-2_6

6. Doroshenko A., Ivanenko P., Novak O., Starushyk O. Autotuning of parallel programs using the data analysis system IBM Watson Analytics. *Problems of Programming*. 2018. N 1. P. 46–54.
7. Neil Savage. Going serverless. Communications of the ACM. 2018. Vol. 61(2). P. 15–16. doi:10.1145/3171583
8. https://calculator.aws/#/createCalculator
9. Andon P.I. et al. (2017). Methods of algebraic programming. Formal methods of parallel program development. Kyiv: Naukova dumka. (in Russian)
10. Gleixner Ambros, Hendel Gregor, Gamrath Gerald, Achterberg Tobias, Bastubbe Michael, Berthold Timo, Christophel Philipp M., Jarck Kati, Koch Thorsten, Linderoth Jeff, L'ubbecke Marco, Mittelmann Hans D., Ozyurt Derya, Ralphs Ted K., Salvagnin Domenico and Shinano Yuji. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. 2020 Mathematical Programming Computation. http://www.optimization-online.org/DB_FILE/2019/07/7285.pdf
11. Rimmi Anand, Divya Aggarwal & Vijay Kumar (2017). A comparative analysisof optimization solvers. *Journal of Statistics and Management Systems.* 20:4. P. 623–635. DOI:10.1080/09720510.2017.1395182
12. A Comparative Analysis of Optimization Solvers. Available from: https://www.researchgate.net/publication/314750497_A_Comparative_Analysis_of_Optimization_Solvers [accessed Nov 14 2020].

Одержано 15.11.2020

***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 імені Ігоря Сікорського".
Кількість наукових публікацій в
українських виданнях – понад 180.
Кількість наукових публікацій в
зарубіжних виданнях – понад 100.
Індекс Гірша – 6.
http://orcid.org/0000-0002-8435-1451,

Новак Олександр Сергійович,
молодший науковий співробітник.
Кількість наукових публікацій в
українських виданнях – 11.
Кількість наукових публікацій в
зарубіжних виданнях – 1.
https://orcid.org/0000-0002-1665-7360.

Місце роботи авторів:

Інститут програмних систем
НАН України,
03187, м. Київ-187,
проспект Академіка Глушкова, 40.

E-mail: doroshenkoanatoliy2@gmail.com