УДК 004.4'24 https://doi.org/10.15407/pp2020.02-03.126

ОПТИМІЗАЦІЯ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ З ВИКОРИСТАННЯМ МОДЕЛІ АКТОРІВ

А.Ю. Дорошенко, Є.М. Туліка

Запропоновані методи та інструментальні засоби для оптимізації блочно-рекурсивних алгоритмів із використанням моделі акторів. Формалізовано модель розподілення і координації задач в обчислювальному кластера у вигляді асинхронних реактивних процесів із обміном повідомленнями представлених моделлю акторів та хореографією акторів. Створено систему декларативного задання алгоритмів які трансформуються у систему акторів. На основі пріоритетизації операцій блочно-рекурсивних алгоритмів запропоновано схему розташування даних у кластері для зменшення часу очікування та зменшення кількості обмінів із підвищенням паралелізму при високій швидкодії процесора і зниженій пропускній здатності мережі. Створено підтримку адаптивної зміни схеми розміщення даних між вузлами кластеру під час виконання для підвищення ефективності в рамках поточного навантаження кластеру. Створено систему автотюнінгу схем розташування акторів у кластері, що враховує статистику попередніх запусків для оптимізації. Використання хореографії без центрального координуючого елементу дозволяє позбутися жорсткої залежності між вузлами кластера, дає гнучкість розташування даних, покращує надійність за відсутності єдиної точки відмови, та дає можливість самовідновлення.

Ключові слова: блочно-рекурсивні алгоритми, хмарні обчислення, модель акторів, хореографія акторів, автотюнинг хореографії акторів.

Предложены методы и инструментальные средства для оптимизации блочно-рекурсивных алгоритмов с использованием модели актеров Форматизирована модель распределения и координации задач в вычислительном кластере в виде асинхронных реактивных процессов с обменом сообщениями представленных моделью актеров и хореографией актеров. Создано систему декларативного задания алгоритмов, которые трансформируются в систему актеров. На основе приоритезации операций блочно-рекурсивных алгоритмов предложено схему расположения данных в кластере для уменьшения времени ожидания и уменьшения количества обменов с повышением параллелизма в ситуации высокой скорости процессора и сниженной пропускной способности сети. Создано систему поддержки адаптивной схемы размещения данных между узлами кластера во время исполнения для повышения эффективности в рамках текущей загрузки кластера. Создано систему автотьюнинга схем расположения актеров в кластере, которая использует статистику предыдущих запусков для оптимизации. Использование хореографии без центрального координирующего элемента позволяет избавится от жесткой зависимости между узлами кластера, дает гибкость расположения данных, улучшает надежность при отсутствии единой точки отказа, дает возможность самовосстановления.

Ключевые слова: блочно-рекурсивные алгоритмы, облачные вычисления, модель актеров, хореография, автотюнинг хореографии актеров.

Introduced methods and instrumentation tools for actor model applied to block recursive algorithms optimization. Created formal model of distribution and coordination of the tasks in computation cluster as asynchronous reactive processes with message-passing represented with an actor model and choreography of actors. Created declarative definitions of algorithms which compiles to the system of actors. Proposed scheme of data placement in a cluster using prioritization of block-recursive operations to reduce idling time, data movement, with increased parallelism in situation of high-speed processors and reduced network bandwidth. Implemented adaptive adjustment of the data placement in a cluster at run time to account for current cluster load. Created autotuning of the actor placement in a cluster which uses statistics of previous runs for optimization. Usage of choreography of actors allows to remove central coordinating element and to avoid hard dependencies between cluster nodes, which provides flexible data placement, improves fault tolerance with no single point of failure and allows to use self-healing.

Key words: block recursive algorithms, cloud computing, actor model, actor choreography, autotuning of actor choreography.

Вступ

Алгоритми паралелізації матричних операцій – це важливий компонент пришвидшення наукових обчислень. Оптимізація алгоритмів за часом виконання, кількістю операцій або обсягом переданих даних може суттєво зменшити бюджет обчислень, зменшити час на отримання результатів і дозволити вирішувати більші за обсягом задачі.

Механізми паралелізації на кшталт MPI та OpenMP надають примітиви для роботи із паралельними віддаленими викликами та координаційні механізми для синхронізації. Програміст, розробляючи розподілену систему, явно виражає паралелізм через засоби паралельного запуску, обміну та координацією очікування компонент; він управляє комунікацією компонент, обміном файлами в розподілених системах. Однак складність написання ефективного та стабільного розподіленого коду збільшується в залежності від врахованих обставин функціонування програми. Розподілені обчислення виконуються на обчислювальному кластері, який складається з множини вузлів кластера (обчислювальних пристроїв у межах одного обчислювального центру), поєднаних між собою високошвидкісною мережею.

У гетерогенному обчислювальному кластері кожен вузол кластеру може мати власні характеристики апаратного забезпечення. Під час роботи програми може змінюватись кількість вузлів кластеру, розміщення компонентів системи у кластері, наявність інших запущених задач на вузлах системи впливає на їх швидкодію, ємність пам’яті, та доступну пропускну здатність мережі. В будь який момент кожен вузол системи може вийти із ладу і всі його операції та дані мають бути перерозподілені між іншими вузлами. Такі фактори спільні для

всіх розподілених програм і важко програмуються у традиційному імперативному підході. Це мотивує віднайдення таких засобів програмування, в яких розробник фокусується на створенні ефективного розподіленого алгоритму, водночас як система підтримки розподілених програм візьме на себе всю складність координації компонент, обміну повідомленнями, управління ресурсами, топологією вузлів кластеру та їх прив'язки до даних.

Декларативне програмування – це один із підходів, який ставить за мету подолати цю складність. Ключова відмінність декларативного програмування полягає у тому, що розробник програм зосереджується на декларації поведінки системи, дозволяючи інфраструктурі підтримки декларацій знайти найефективніший спосіб їх інтерпретації та виконання. Оскільки система підтримки та виконання декларацій відповідальна за оптимізацію то декларації не можуть бути універсальні, вони часто розробляються для вирішення конкретного класу проблем. Розробка засобів декларації вимагає більше часу, однак забираючи із розробника відповідальність за запуск та ефективність розподілених програм натомість ми отримаємо можливість еволюціонувати технології підтримки декларацій, вдосконалюючи та оптимізуючи їх механізми. Декларативні способи з'являються там, де вже достатньо досвіду із розробки імперативних програм та вироблені механізми та розуміння предметної галузі.

Модель акторів – це формалізація досвіду із розробки ефективних асинхронних багатопроцесорних систем із розподіленою пам'яттю. Модель акторів розроблена як відповідь на складність розробки ефективного розподіленого коду, яка натомість надає декларативні інструменти опису бажаної поведінки розподіленої системи. Актор, як основний діючий елемент моделі формалізує поняття асинхронного реактивного, орієнтованого на обмін повідомленнями, процесу, який читає повідомлення із буферу повідомлень (поштової скриньки), виконує операції як реакцію на повідомлення та надсилає повідомлення як результат виконання операцій у поштові скриньки інших акторів. Декларативні засоби задання актора полягають у конфігурації можливих вхідних повідомлень актора, можливих станів актора, співвідношення між типом вхідного повідомлення та обробника повідомлення, а також множини вихідних повідомлень. Модель актора виявилась дуже ефективною в індустрії комунікацій, швидкісній обробці сигналів в "Інтернеті речей" (IoT) і в системах реального часу. В цій роботі розглядаються переваги використання акторів для блочно-рекурсивних алгоритмів над матрицями, що притаманні науковим обчисленням.

Класичні реалізації блочно-рекурсивних матричних алгоритмів обмежені жорсткою топологією розміщення акторів та характеризуються наперед заданим порядком виконання розподілених операцій. Як правило основний координуючий елемент обчислення розподіляє дані між вузлами системи та командує порядком запуску обрахунків блочних елементів. Такий підхід не враховує можливу гетерогенність обчислювального кластеру, де перебіг обчислень на кожному вузлі кластера може відбуватись із власною швидкістю, сповільнюючи обчислення до швидкості найповільнішого вузла. Еволюція таких підходів полягає у розбитті алгоритму на окремі задачі та створенні направленого впорядкованого графа задач, який буде використовуватись центральним *оркестраційним* компонентом для управління перебігом процесу [1]. В цій роботі ми пропонуємо новий погляд на такий підхід – використання *хореографії* компонент, в якому центральний координуючий елемент відсутній і натомість кожен компонент відповідає за координацію своєї роботи із пов'язаними компонентами. Це надає переваги у кращій адаптивності до мінливих умов, а також покращує відмовостійкість системи.

Основна мета роботи полягає у формалізації моделі розподілення і координації задач в обчислювальному кластера у вигляді асинхронних реактивних процесів із обміном повідомленнями представлених моделлю акторів та хореографією акторів. Практичною ціллю роботи є розробка інструментів для роботи із блочно-рекурсивними алгоритмами засобами моделі акторів, а саме:

1) створення адаптивної схеми розміщення блоків матриці між вузлами кластеру для підвищення ефективності в рамках конкретного кластеру та пристосування до гетерогенності апаратного забезпечення кластера в якому відбувається обчислення;

2) автотюнінг схем розташування акторів у кластері що враховуватиме статистику попередніх запусків для оптимізації в рамках конкретного кластеру;

3) використання хореографії для координації роботи компонент що дозволяє позбутися жорсткої залежності між конкретними вузлами кластеру та центральних координуючих елементів; разом це дає гнучкість у розташуванні підсистем у кластері, підвищення надійності оперування системи за відсутності єдиної точки відмови, дозволяє реалізувати здатність системи до самовідновлення;

4) оптимізація обчислення за рахунок зменшення сумарного часу очікування системи, використовуючи нові схеми розташування даних та пріоритетизацію операцій, зменшення кількість обмінів та підвищення паралелізму системи.

1. Пов'язані дослідження

Існує багато ефективних пакетів, що реалізують роботу із матрицями для систем із спільною пам'яттю. LAPACK надає набір інструментів для лінійної алгебри. ScaLAPACK – це підмножина підпрограм LAPACK, переписаних для розподіленої пам'яті та систем обміну повідомленнями. Вона реалізована з використанням

підходу Single-Program-Multiple-Data використовуючи явний обмін повідомленнями між процесорами. ScaLAPACK спроектована для гетерогенних систем та працює на будь-якому комп'ютері, що підтримує MPI [2]. На додаток багато із постачальників надають бібліотеки, оптимізовані для їх власного апаратного забезпечення, такі як Intel MKL, AMD ACML, IBM ESSL (PESSL), та Cray XT LibSci. Кожна із цих спеціальних бібліотек включають підпрограми LAPACK та ScaLAPACK.

Однак із збільшенням кількості процесорів стало зрозуміло, що підхід до паралелізму, запропонований в ScaLAPACK, в якому програма розгалужується, а потім очікує відповіді від кожної гілки, починає суттєво деградувати. Однією із причин є те, що далеко не всі гілки лежать на критичному шляху виконання програми. Натомість стали з'являтися системи із підходом, орієнтованим на паралелізм окремих задач.

Однією із найбільш розвинених систем такого типу є PaRSEC – система виконання, орієнтована на задачі для розподілених гетерогенних архітектур, яка здатна розгортати опис графа задач на множину ресурсів та перевіряти задовільність всі залежності за даними і ефективно розподіляти дані та просторі пам'яті між вузлами кластеру та планування задач на гетерогенних ресурсах [3, 4]. Предметно-орієнтовані мови в PaRSEC дозволяють експертам в предметній області фокусуватись тільки на предметі їх наукової галузі, приховуючи складність необхідних знань із комп'ютерних наук. Мова параметричних графів задач використовує компактне, параметризоване графове подання під назвою Потік Даних Задачі (Job Data Flow) для подання залежностей між задачами. Інша предметна мова під назвою Динамічний Пошук Задач (Dynamic Task Discovery) [5] надають альтернативні моделі програмування для задоволення більш загальних потреб для послідовного включення задач у виконання.

Схожий підхід, орієнтований на задачі, використовується у Spark, зокрема математичному пакету Spark ML. Spark ML реалізовує розподілені алгоритми, використовуючи представлення у вигляді ациклічного графу для відображення залежностей між даними. Центральний оркестратор Spark Master слідкує за перебігом виконання процесу та перезапускає задачі в разі виключних станів. Так само, як раніше у Hadoop, велика задача розбивається на маленькі підзадачі які упорядковуються у послідовність. Деякі із задач можуть виконуватись паралельно із даними, розміщеними серед кластеру. Із всіма цими задачами Spark створює потік задач, який представлено у вигляді направленого ациклічного графа, в якому вершинами є блоки даних, а ребрами – операції над блоками даних. Декомпозиція Холеські реалізована в пакеті Spark ML для однопроцессорних систем із спільною пам'яттю та використовує бібліотеку Breeze що є обгорткою навколо Netlib для мови Scala. Netlib-java в свою чергу є обгорткою навколо низькорівневих бібліотек BLAS, LAPACK та ARPACK і надає Java-інтерфейс до коду на C/Fortran. Однак Spark не надає розподіленої реалізації факторизації. Натомість існують дослідження [6, 7] про можливе використання Spark з метою масштабування та стійкості до відмов. Spark повністю управляє системою контрольних точок та відкатів до попереднього стану у разі виникнення виключних станів.

Обидві описані системи, PaRSEC та Spark, є прикладом систем оркестрації, де є центральний компонент, що задає порядок виконання задач, слідкує за перебігом процесів, контролює залежності за даними. Одним із недоліків систем оркестрації є те, що на центральний координуючий вузол припадає найбільше навантаження і його працездатність є ключовим фактором стабільності системи. Як правило контролюючий вузол потребує спеціалізованого апаратного забезпечення для компенсації завантаженості та забезпечення швидкості операцій координації.

Натомість в даній роботі розглядається механізм хореографії, де центральний контролюючий компонент відсутній, зате кожен компонент розподіленої системи відповідальний за координацію своєї роботи із рештою системи. Система підтримки хореографії реалізується як бібліотека, що включена до коду компонентів системи і прихована від програміста, а натомість надає точки розширення для підключення реалізації задач із предметної галузі.

2. Оптимізація розміщення даних у кластері

Блочне розташування даних матриці – це підхід, що використовується для підвищення продуктивності ієрархічної пам'яті розподіленої системи. В блочному розташуванні даних, матриця A розміру N розбивається на під-матриці (блоки) розміру $NB \times NB$. Початкова матриця A представлена матрицею AB що складається із блоків розміром $N/NB \times N/NB$. Операції над початковою матрицею A переписуються таким чином, щоб їх еквівалент міг виконуватись над матрицею AB. Елементи даних всередині блока розташовані послідовно у пам'яті для ефективного доступу під час виконання операції над блоками. У розподіленій системі, що складається із P вузлів, матриця AB буде розподілена між вузлами так, що кожний вузол відповідатиме за певну кількість блоків матриці AB, у найчастішому випадку – за $(AB \times AB)/P$ блоків, але ці параметри можуть варіюватись в залежності від реалізації.

Факторизація Холеські – це ітеративний процес розрахунку факторизації симетричної визначеної квадратної матриці розміру $N * N$ як добутку нижньої трикутної матриці L та її транспозиції, $A = LL^T$. Блочна форма факторизації Холеські індуктивно виводиться наступним чином. Якщо припустити що нижній трикутний фактор Холеські L_{11} матриці A_{11} відомий, то можемо вивести нерівності для блоків:

$$A_{11} \rightarrow L_{11}L_{11}^T,$$

$$L_{21} \leftarrow A_{21}(L_{11}^T)^{-1}, \quad (1)$$

$$\hat{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T. \quad (2)$$

Під час факторизацій Холеські почергово повторюються дві операції: факторизація панелі колонок, яка оновлює крайню ліву колекцію колонок, та оновлення решти матриці. Оновлення решти матриці використовує результат факторизації панелі. Для роботи із окремими блоками матриці *AB* в межах одного вузла із спільною пам'яттю ефективно використовується бібліотека BLAS. Бібліотека BLAS це набір матричних операцій реалізованих мою Фортран та С та оптимізованих під конкретне апаратне забезпечення. На даний момент BLAS є найшвидшою бібліотекою роботи із матрицями та надає інтерфейси до різних мов програмування. Більшість алгоритмів LAPACK та ScaLAPACK складаються із деякого числа фундаментальних операцій, які реалізовано із використанням бібліотеки BLAS Level-2 або Level-3. Продуктивність BLAS залежить від розміру матриці і найкраща продуктивність досягається з великими розмірами блоку. Використовуючи BLAS/LAPACK розрахунок кроків здійснюється використовуючи наступні операції:

- DPOTF2 вираховує факторизацію Холеські діагонального елемента A_{11};
- DTRSM обраховує колонку L_{21} за формулою (1);
- DSYRK, оновлює решту матриці A_{22} за формулою (2).

Оптимізація такого підходу, плиточна факторизація матриці [8], говорить про те, що панель колонок може в свою чергу бути розбито на під-матриці, кожна з яких може бути факторизована окремо. Замість оновлення решти матриці, оновиться лише відповідний рядок підматриць. Перевага плиточного розташування матриці полягає у тому що замість залежності по колонці, процес може продовжуватись як тільки верхні частини колонки факторизовано [9, 10].

Пересилання блоків матриці між фізичними обчислювальними пристроями є дорогою за часом операцією. В залежності від розміру блочної матриці час на пересилання даних може бути більшим від часу на факторизацію блочної матриці (рис. 1).

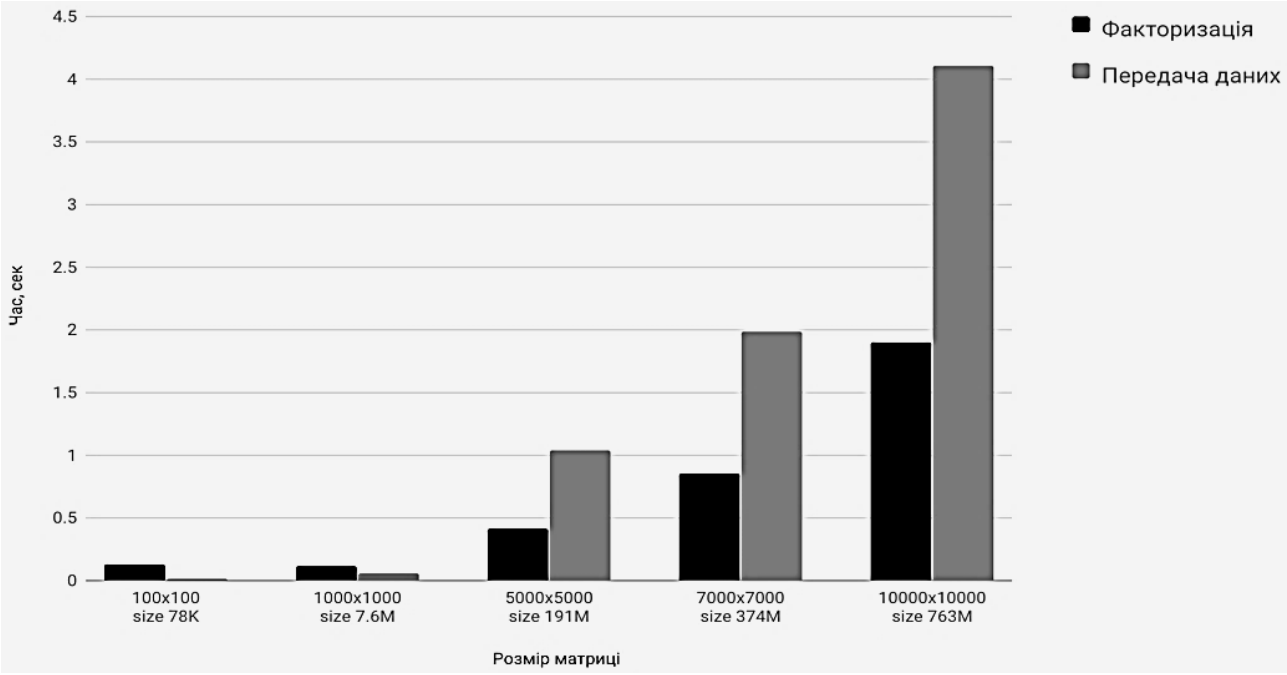

Рис. 1. Порівняння часу факторизації блочної матриці розміру N*N на 2.9 гГц Intel Core i7 16 Гб RAM до часу на пересилання даними через мережу із пропускною здатністю 25 гбіт/с, в залежності від розміру матриці

Водночас, обмін даними в межах одного сервера є відносно недорогою операцією, що потребує лише читання та декодування файлу в оперативну пам'ять актора. Тому через зміну розташування блоку матриці стосовно вузла кластера ми можемо контролювати та зменшувати кількість дорогих операцій обміну даними заміняючи їх дешевшими локальними обмінами. Задача оптимізація розміщення акторів полягає у створенні такої топології, в якої найменша кількість даних передаються мережею при найбільшому паралелізмі.

Для зручності пронумеруємо вузли кластеру від 0 до *P*–1 та матричні колонки та рядки від 1 до *N*. Одною з популярних схем розміщення блоків матриці серед множини обчислювальних пристроїв є одновимірний блочно-циклічний розподіл колонок. Обраний розмір блоку позначається як *NB*, колонки розбиваються у групи

розміру NB і розподіляються між процесорами циклічно. Це означає що колонка k зберігається на вузлі $[(k-1)/NB]\mathrm{mod}P$. Таке розміщення має той недолік, що факторизація $A_{k:N,k:k+b-1}$ здійснюватиметься на одному процесі, таким чином утворюючи послідовне вузьке місце.

Це послідовне вузьке місце зазвичай виправляється із наступною схемою розташування даних під назвою двовимірний циклічний розподіл блоків [11]. Тут вважаємо що P процесів впорядковані у квадратний масив $P_r * P_c$ процесів, індексованих двовимірним способом як (p_r, p_c), де $0 <= p_r < P_r$ та $0 <= p_c < P_c$. Всі процеси (p_r, p_c) із фіксованим p_c називаються колонкою процесів, всі процеси (p_r, p_c) із фіксованим p_r називаються рядком процесів. Таке розміщення дозволяє паралелізм у будь-якій колонці та використовує Level 2 BLAS та Level 3 BLAS на локальних матричних блоках. Також такий розподіл дає хороші властивості паралелізму. Схема двовимірного циклічного розподілу блоків використовується в ScaLAPACK [12].

В даній роботі для оцінки схеми розташування блоків матриці ми вводимо пріоритет операцій для мінімізації сумарного часу простою окремих елементів при виконанні операції над матрицею при найбільшому паралелізмі системи:

$$min(\sum_{n=0}^{numOperations(N*M)} waitTime(op_n)) \quad (3)$$

Очікування при виконанні розподілених операцій над матрицею виникає внаслідок залежності за даними між операціями над блоками матриці в рекурсивному розподіленому процесі. Критичний шлях – це всі операції які необхідно виконати щоб дістатись до наступного кроку рекурсії алгоритму. Всі кроки алгоритму які не знаходяться на критичному шляху рекурсії можуть виконуватись незалежно від просування алгоритму, а тому їх виконання може відбуватись паралельно із просуванням алгоритму і виконуватись на вузлах кластера не задіяних у критичному шляху. Наприклад, при факторизації Холеські критичним шляхом є всі операції які необхідні для того щоб дістатись факторизації наступного діагонального елемента (рис. 2). Факторизація діагонального елемента – це єдина блокуюча операція, від якої залежить решта операцій.

Факторизація Холеські має наступні залежності: $A_{(i,i)}$ може бути факторизовано, якщо всі оновлення виду (2) були застосовані до блоку $A_{(i,i)}$, це означає що наступні три умови виконуються:

- всі оновлення виду (2) були застосовані до блоку $A_{(\mathrm{i}-1,i)}$;
- діагональний елемент в стовпчику і $A_{(\mathrm{i}-1,i)}$ був факторизований;
- $L_{(\mathrm{i}-1,i)}$ була обрахована.

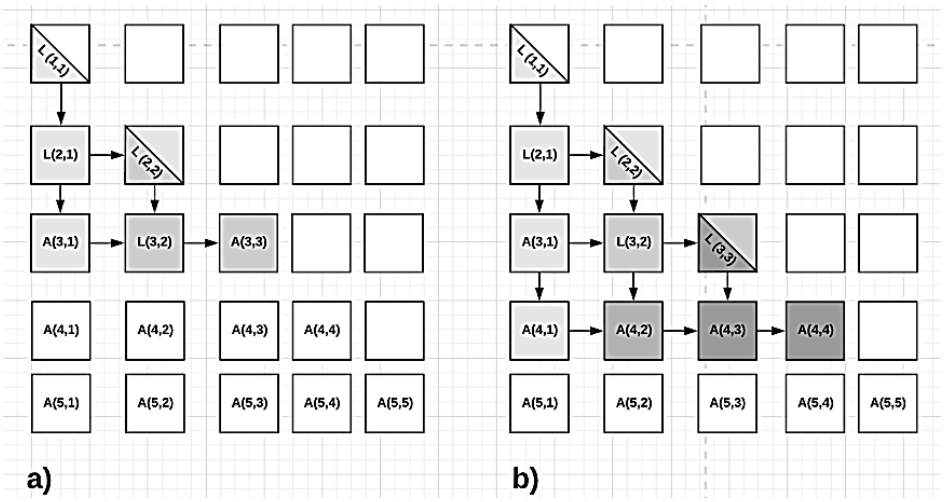

Рис. 2. Критичний шлях факторизації елемента $A_{(4,4)}$

При цьому оновлення в нижній частині матриці у рядках i+1, ... N не лежать на критичному шляху факторизації $A_{(i,i)}$. Водночас, факторизація $A_{(i,i)}$ має статися перед тим як діагональний елемент $A_{(i+1,\mathrm{i}+1)}$ і всі наступні будуть факторизовані. Затримка спричинена очікуванням від залежних даних при факторизації верхньої частини матриці, що сповільняє весь процес, в той час як затримка в факторизації нижньої частини буде непомітна. Ці обмеження дозволяють ввести наступні пріоритети операцій. Якщо поточний факторизований елемент матриці $i, i \in [0, N]$ то всі залежності для факторизації $A_{(i+1,\ i+1)}$ мають найвищий пріоритет. Це – операції оновлення елементів $A_{(1,i+1)} \ldots A_{(i,i+1)}$ та операція обрахунку $L_{(i,i+1)}$. Всі залежності для факторизації діагональних елементів у рядках наступних за i визначаються в спадаючому порядку по мірі віддаленості від i.

Використовуючи цей підхід спроектовано наступний варіант розміщення блоків матриці щоб скористатися локальністю даних для операцій із найвищим пріоритетом. Таке розміщення підвищує

ефективність процесу в ситуації, коли $t_1 >= t_2$, де t_1 – час на пересилання одного блоку матриці, t_2 – час факторизації одного блоку матриці (рис. 3).

В запропонованому розміщенні матриці для просування процесу факторизації після обчислення елементу A_{11} вузол кластеру p_1 має достатньо даних для обрахунку факторизаці $A_{21} \rightarrow L_{21}$, оновлення A_{22}, факторизація $A_{22} \rightarrow L_{22}L_{22}^T$, оновлення A_{32}, факторизації $A_{32} \rightarrow L_{32}$ та обрахунку $A_{33} \rightarrow L_{33}L_{33}^T$. В сумі, вісім перших операцій здійснюється в межах одного кластера. Якщо прийняти час на обмін даними вдвічі більший від часу на факторизацію елемента, тобто $t_2 = 2t_1$ то за час $8t_1$ перші 6 блоків матриці буде факторизовано. Тим часом за час $3t_1$ буде факторизовано блок A_{41} і передано дані із p_2 до p_1. Таким чином, p_1 матиме достатньо даних для факторизації та оновлення блоків A_{42}, A_{43} та A_{44} на критичному шляху до A_{44}, перш ніж вузол кластеру p_1 звільниться для їх обрахунку.

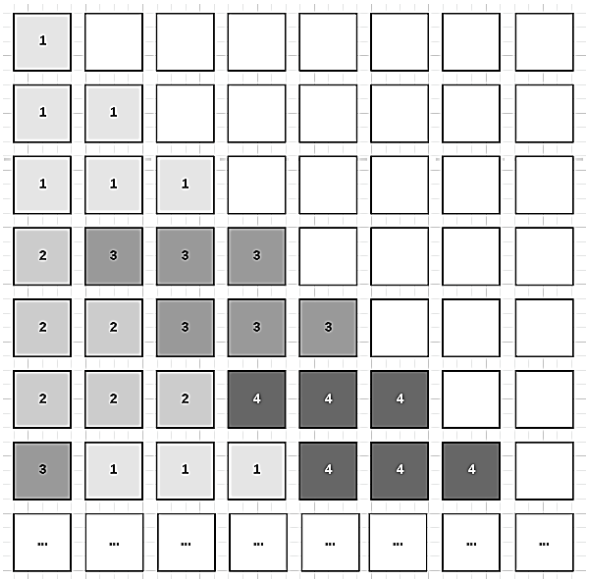

Рис. 3. Розміщення блоків матриці серед вузлів кластеру для підвищення локальності даних із пріоритезацією критичного шляху. Цифрами позначено номер вузла кластера відповідного за блок матриці

Основним правилом що використовується для розробки такої топології розміщення блоків матриці це мінімізація кількість передач між елементами у рядку, оскільки всі елементи у рядку, що знаходяться лівіше від діагонального елемента лежать на критичному шляху до факторизації діагонального елемента. У стовпчику тільки перший елемент під діагональним елементом лежить на критичному шляху при умові що всі елементи які йому передують вже були обраховані і їх дані передані. Всі елементи найближчі до діагоналі обраховуються в межах одного вузла кластера і при тому паралелізму достатньо щоб підготувати наступний набір елементів для наступної операції. Тому важливо знайти такий баланс між необхідною кількістю послідовних елементів у рядку та у стовпчику, щоб процес, який дістається до факторизації елемента завжди мав достатньо даних для його обрахунку з високою вірогідністю. Тестування показує, що такий підхід до розбиття матриці має переваги між традиційним блочно-циклічним розподілом для ситуацій із швидким обрахунком та довгою передачею даних. Однак, жорстка топологія розташування блоків матриці не відповідає на питання операційні обставин функціонування розподіленої системи. Далі представимо динамічну топологію.

3. Використання моделі акторів для роботи із блочно-рекурсивними алгоритмами

Модель акторів – це форма розподілених обчислень в яких кожен процес яких виконується в межах одного вузла із спільною пам'яттю зображено примітивом актору, а координація таких процесів в межах розподіленого процесу задана хореографією актора [13]. Модель акторів це спосіб представлення розподілених обчислень у декларативний спосіб, в якому вся складність управління розподіленим процесом прихована за конфігурацією актора та взаємодії акторів. Актор – це довготривалий процес, запущений на вузлі кластеру, що постійно очікує на вхідні повідомлення із вхідного каналу. В разі отримання повідомлення актор може:

1. Викликати локальну процедуру та змінити свій стан.
2. Надіслати скінченне число повідомлень іншим акторам, інформуючи про виконану операцію, свій стан та наявні дані.
3. Створити скінченну кількість інших акторів.

Формально, програма, реалізована із використанням моделі акторів, моделюється з асинхронною мережею, представленою у вигляді направленого графа $G = (V, E)$. V це множина вершин, в яких $v_i \in V$ відповідає за актора, E це множина направлених ребер де $e_{ij} \in E$ відповідає за канал обміну повідомленнями між v_i та v_j. В більшості реалізацій моделі акторів конкретний канал обміну повідомленнями являє собою чергу повідомлень із центральним брокером, в яку кожен із акторів може надіслати повідомлення і кожен із акторів може підписатись на повідомлення певного типу. Тому канал e_{ij} може бути уточнений як тип повідомлення T, де актор v_i може надіслати повідомлення актору v_j, що підписаний на отримання. Обмін повідомленнями асинхронний, тобто час t_1, коли повідомлення було надіслано актором a_1, може відрізнятись від часу t_2 , коли актор a_2 його прочитає, якщо порядок цих подій зберігається, $t_1 < t_2$.

Вся координація розподіленої системи переноситься на обмін повідомленнями. Якщо актор обробив повідомлення певного типу із певними даними, він сповіщає канал, надсилаючи нове повідомлення іншого типу із деталями здійсненої операції. Актори, зацікавлені у очікуванні результатів обробки, підписуються на повідомлення цього типу і прив'язують власну операцію як реакцію на повідомлення. У відповідь на отримане повідомлення актор виконує операцію та, в залежності від стану після виконання операції, може в свою чергу надіслати одне чи більше повідомлень. На таке повідомлення може бути підписаний інший актор, тощо. Тобто в результаті ми отримаємо ланцюжок повідомлень та акторів, які репрезентують гілку виконання розподіленого процесу. Певні типи повідомлень є термінальними, тобто, або ніякий інший актор не підписаний на таке повідомлення, або повідомлення може перевести актора у термінальний стан і ланцюжок виконання на цьому завершується. Відсутність термінального повідомлення означає, що програма буде працювати нескінченно. Аналіз ланцюгів повідомлення перед запуском програми дозволяє впевнитись що процес є скінченним.

Програмування розподіленого вирішення задачі із моделлю акторів являє собі конфігурацію множини акторів, кожен з яких характеризується наступними ключовими параметрами:

- типи повідомлень у вхідному каналі, на які актор реагує,
- обробник реакції на повідомлення – може бути зміна стану актора, виконання операції, запуск іншого актора,
- типи повідомлень які актор генерує у вихідний канал в залежності від стану актора.

Граф потоку керування, який представляв програму, трансформується у граф асинхронної мережі. Представлення архітектури програми у вигляді моделі акторів дає відповідь на два питання:

1) як виконувати незалежні секції програми на різних вузлах кластера;

2) як обмінюватись даними між ними.

Розглянемо підхід на прикладі блочно-рекурсивних матричних алгоритмів і, зокрема, факторизацію Холеські. Кожен блок матриці в блочному розбитті буде представлено окремим актором. Актор відповідає за всі операції над блоком матриці а також на обмін даними з результатами факторизації із іншими акторами. Початкові дані блоку матриці актора ініціалізуються під час запуску системи. Після цього актор завжди виконується на тому самому вузлі кластеру на якому зберігаються дані блоку матриці за які він відповідає.

В залежності від позиції блока всередині матриці, повідомлення, на які актор підписний та дії, які він може виконувати, різняться. Тому вводимо типізацію акторів в залежності від позиції блока, за який актор відповідає. Для факторизації Холеські основні значимі типи акторів – це Діагональний актор та Суб-діагональний актор. Для діагонального актора притаманні операції – це факторизація діагонального елемента та оновлення блоку. Для суб-діагонального актора притаманні операції – факторизиція елементу панелі колонки та оновлення блоку. Тип актора та його позиція визначає повідомлення, на які актор підписаний. Повідомлення ідентифікується темою (topic) та типом T. Тема повідомлення використовується для управління підписками, тип повідомлення визначає поведінку актора при отриманні повідомлення.

Блоки матриці об'єднуються у секцію. Кожна секція прив'язана до окремого вузла кластеру, тобто секція визначає фізичне розміщення файлів у кластері і потоки даних для їх обчислення. За прив'язку актора і разом із ним блоку матриці до вузла кластеру відповідає координатор секції. Початкове розміщення актора визначається конфігурацією секції. Впродовж функціонування програми актори можуть змінювати прив'язку до секції і переноситись на інші вузли кластеру. Тому координатор секції, зокрема, відповідає за динамічне відстеження належності блоків до вузлів кластеру. Стан секції актора є композитним станом всіх блоків які є частиною секції. Стан секції використовується для прийняття рішення щодо переносу акторів між різними секціями під час роботи системи.

Актори, що функціонують в межах однієї секції, мають доступ до швидкого обміну файлами через файлову систему або через обмін повідомленнями у пам'яті системи акторів. Однак, коли з'являється необхідність обміну даними між акторами, що належать до різних секцій, розпочинається процес обміну даними через мережу. Кожна секція має виділеного актора, що відповідає за обмін файлами між секціями. Передача файлів відбувається у три кроки:

- актор a_1, що належить до секції s_1, сповіщається про наявність даних у актора a_2, що належить до секції s_2, через повідомлення у вхідній скриньці актора із темою *topic_data_Y_ready*, на яку a_1 підписаний;
- актор a_1 надсилає повідомлення у канал повідомлень із темою *topic_file_transfer_2*, на яку підписаний актор f_2 секції s_2;
- актор f_2 надсилає файл актору a_1 через пряму peer-to-peer комунікацію акторів.

Кожен актор має вбудовану машину станів, яка використовується для постійного стеження за станом актора і його прогресом у процесі. Ці дані також використовуються для стеження за станом процесу в цілому. Стани актора змінюються в залежності від подій, що відбуваються із актором: отриманих та надісланих повідомлень, здійснених операцій. Актор має перелік станів, в які він може потрапити. Все починається із початкового стану INIT, закінчується станом COMPLETE.

Хореографія акторів – це декларативний спосіб опису залежності між акторами та порядку перебігу системи через задання конфігурації хореографії для кожного актора [14]. Для прикладу опису системи станів актора та опису хореографії візьмем актора, що належить до системи акторів факторизації Холеські та відповідає за блок у позиції $A_{(1,1)}$:

- це блок типу Діагональний,
- теми повідомлень, на які актор підписаний*: data_ready_A11,*
- стани актора: *INIT, A11_Received, A11_Processed, COMPLETE,*
- машина станів:
  - *“INIT, data_ready_A11 -> {Retrieve_data} -> A11_Received”,*
  - *“A11_Received -> {Factorize} -> 1_1_Processed”,*
  - *“A11_Processed -> {Send_message} -> COMPLETE”,*
- вихідні повідомлення: *data_ready_L11.*

Більш складний приклад опису актору, що відповідає за блок у позиції $A_{(3,4)}$:

- це блок типу Суб-діагональний,
- теми повідомлень, на які актор підписаний*: data_ready_A34, data_ready_L33, data_ready_L14, , data_ready_L24* ,
- стани актора: *INIT, A34_Received, [{L14_Received, L14_Processed}, {L24_Received, L24_Processed}], {L33_Received, L33_Processed}, COMPLETE,*
- машина станів:
  - *“.., data_ready_A34 -> {Retrieve_data} -> A34_Received”,*
  - *“.., data_ready_L33 -> {Retrieve_data} -> L33_Received”,*
  - *“.., data_ready_L14 -> {Retrieve_data} -> L14_Received”,*
  - *“.., data_ready_L24 -> {Retrieve_data} -> L24_Received”,*
  - *“(L33_Received , A34_Received, L14_Processed, L24_Processed, L33_Received) -> {ApplyL11} -> L33_Processed”,*
  - *“(A34_Received, L14_Received) -> { ApplyL21} -> L14_Processed”,*
  - *“(A34_Received, L24_Received) -> { ApplyL21} -> L24_Processed”,*
  - *“L33_Processed -> {Send_Message} -> COMPLETE”,*
- вихідні повідомлення: *data_ready_L34.*

Більшість із даних, що описують актора, повторюватимуться для акторів однакових типів та будуть різнитись лише деталями відповідними до належності актора до секції та позиції блока матриці, за який актор відповідає. Координатор секції відповідає зокрема за динамічну генерація акторів та хореографію за типами акторів та їх позицій у матриці. Далі розглянемо алгоритм зміни розміщення акторів серед вузлів кластера.

4. Адаптивне розміщення даних у кластері

Жорстка топологія розбивки матриці між вузлами кластеру передбачає що всі операції будуть виконуватись із приблизно однаковою швидкістю і обрахунок матриці просуватиметься рівномірно. Якщо процесор l відповідає за блоки A_x, A_y, A_z, а процесор m – за блоки A_a, A_b, A_c,то ці набори блоків будуть

обраховані за приблизно однаковий час. Якщо це припущення порушується, то повільний процесор, отримавши таку саму порцію даних, як і інші процесори, блокуватиме просування процесу і сповільнюватиме швидші процесори.

Така ситуація може виникнути в гетерогенному кластері. Кожен вузол кластера може бути обладнано своїм особливим процесором із вищою або нижчою тактовою частотою, кількістю ядер, вузол може мати відеографічні прискорювачі та оперативну пам'ять, вищу або нижчу пропускну здатність мережі. Навіть у простішому випадку, коли всі вузли мають однакові характеристики, температурні режими, процеси операційної системи, завантаженість мережі, інші фактори можуть негативно впливати на вузол кластеру таким чином, що він сповільнить сумарний час виконання. Функція оптимізації (3) зростатиме.

Для вирішення цієї проблеми розроблено процес адаптивного розміщення блоків матриці серед вузлів кластеру, який полягає у здатності системи змінювати топологію під час виконання. Процес використовує розподілене викрадення роботи (distributed work stealing). Процес розвивається у декілька кроків. На початковому етапі топологія ініціалізується певним жорстким розміщенням блоків. В цій роботі введено топологію розміщення блоків матриці серед вузлів кластеру для підвищення локальності даних із пріоритетизацією критичного шляху, яка найкраще підходить для систем із швидкою обробкою та повільною передачею даних. Далі, кожен вузол системи слідкує за двома ключовими метриками:

1. Тренд сумарної кількість тактів в стані очікування впродовж вікна моніторингу. За такт обирається одиниця часу рівна усередненому виконанню операції факторизиції на блоці обраного розміру. За вікно моніторингу обирається певна фіксована кількість послідовних тактів, впродовж яких обраховуватиметься рухоме середнє кількості простоїв.

2. Довжину черги необроблених операцій. Кожна незаблокована операція, тобто та, для якої вузол кластеру має достатньо даних для обрахунку, що очікує на обрахунок, додається в чергу операцій. Довжина черги визначає завантаженість вузла кластера.

При зростанні тренду тактів у стані очікування, вузол кластеру ініціює операцію запиту на перерозподіл топології. При зростанні черги необроблених операцій, вузол кластеру сповіщає інші вузли кластеру про те що він має дефіцит продуктивності. На критичному шляху факторизації можуть бути операції, які прив'язані до одного вузла кластеру і продуктивність цього вузла визначатиме час на проходження критичного шляху, а також час простою у стані очікування. При перерозподілі роботи критичний шлях враховується в першу чергу. Моніторинг критичного шляху розподіленої машини станів має найточнішу інформацію про те, які операції залишилося виконати та до яких вузлів кластеру вони належать.

Процес перерозподілу починається із узгодження двома секціями позиції актора кандидата на переніс. Узгодження проходить між секціями, які попередньо дійшли згоди про те що один вузол має довгий час простою, в той час як інший вузол має дефіцит продуктивності. Операція узгодження використовує розподілений протокол консенсусу. Процес переносу полягає у зупинці актора на одному вузлі кластеру та його запуск та ініціалізація його на іншому вузлі. Під час ініціалізації актор затребує останній стан блочної матриці, за яку він відповідає, а також чергу останніх повідомлень отриманих даним блоком.

Автотюнінг системи полягає у збереженні останньої використано топології розміщення блоків матриці із оперативною інформацією про час простою та перебіг процесу. Наступний раз отримана топологія використовуватиметься як вхідна для вирішення задачі. Якщо в результаті отримано кращий сукупний час обрахування задачі, то така топологія стає основною для заданих розмірів задачі і заданої системи.

Для перевірки підходу із хореографією акторів використаємо блочну факторизацію великої матриці. Тестове середовище розміщається на хмарній платформі Amazon AWS та складається із кластеру з 8 вузлів, кожен t2.medium EC2 (2 ЦПУ, 4Гб RAM, EBS). Експеримент складається із трьох частин:

- вимірювання масштабованості здійснюється на матриці розміром 12×12 блоків, розміром 1000×1000. Під час експерименту кількість вузлів кластеру змінюється. Зміна часу обчислення демонструє можливість системи задіяти додаткові ресурси;
- вимірювання часу необхідного програмі для факторизації матриці із зростаючим числом блоків n, але сталим розміром блоків. Характеризує складність синхронізації акторів без збільшення кількості обмінів даними;
- вимірювання часу необхідного програмі для факторизації матриці із зростаючою кількістю блоків *n* і сталим розміром блоку. Характеризує складність синхронізації акторів із збільшенням обміну даними між акторами.

Дані експерименту (рис. 4) демонструють, що модель актора може ефективно залучати додаткові обчислювальні ресурси та дозволяє лінійну масштабованість. До того ж стала кількість вузлів кластеру та зростаючий розмір задачі час на обрахунок задачі зростає лінійно із сталим розміром блоків матриці. Однак із зростанням кількості блоків, що обробляються одночасно, часова складність зростає майже квадратично, що підкреслює складність обміну даними між вузлами кластера.

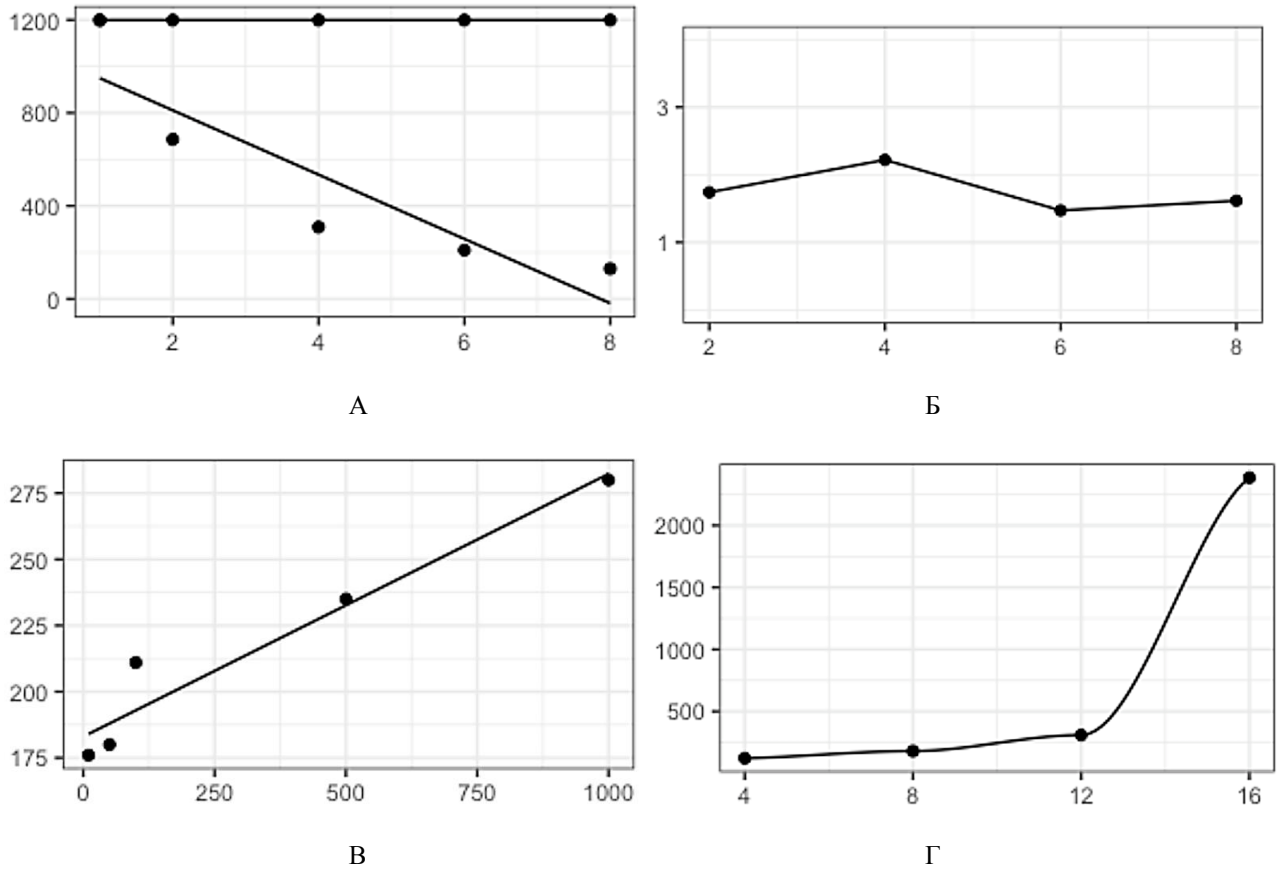

Рис. 4. **А –** час на факторизацію матриці 12x12 блоків, 1000x1000 елементів кожен із змінною кількістю вузлів кластера. Верхня лінія це послідовний процес, нижня – розподілений;
Б – фактор масштабування для матриці 12x12 блоків, 1000x1000 елементів кожен виражений як відношення між часом T_1 необхідний для кількості вузлів кластера r_1 та час отриманий із використанням вдвічі меншої кількості вузлів r_2. Показує сталий фактор масштабування ~2 рівний r_1/r_2;
В – час необхідний на факторизацію матриці 8x8 блоків із сталим числом вузлів кластера 4 і змінним розміром блока. Демонструє збільшення часу на розрахунок в зв'язку із збільшенням накладних витрат розрахунку;
Г – час необхідний для факторизації квадратної матриці із сталим розміром блоків 1000x1000 та змінною кількістю блоків при однаковому числі вузлів кластера. Демонструє затрати на комунікацію акторів

Висновки

Запропоновані методи та інструментальні засоби для оптимізації блочно-рекурсивних алгоритмів із використанням моделі акторів. Формалізовано модель розподілення і координації задач в обчислювальному кластера у вигляді асинхронних реактивних процесів із обміном повідомленнями, представлених моделлю акторів та хореографією акторів. Створено систему декларативного задання алгоритмів, які трансформуються у систему акторів. На основі пріоритетизації операцій блочно-рекурсивних алгоритмів запропоновано схему розташування даних у кластері для зменшення часу очікування та зменшення кількості обмінів із підвищенням паралелізму при високій швидкодії процесора і зниженій пропускній здатності мережі. Створено підтримку адаптивної зміни схеми розміщення даних між вузлами кластеру під час виконання для підвищення ефективності обчислень в рамках поточного навантаження кластеру. Створено систему автотюнінгу схем розташування акторів у кластері, що враховує статистику попередніх запусків для оптимізації. Використання хореографії без центрального координуючого елементу дозволяє позбутися жорсткої залежності між вузлами кластера, дає гнучкість розташування даних, покращує надійність за відсутності єдиної точки відмови та дає можливість самовідновлення.

Література

1. Song Fengguang, Asim YarKhan, and Jack Dongarra. "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems." *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009.

2. Dongarra J.J., Robert Van De Geijn, and David Walker. *LAPACK Working Note* 43: *A Look at Scalable Dense Linear Algebra Libraries*. University of Tennessee, Computer Science Department, 1992.
3. Seeger Matthias, et al. "Auto-differentiating linear algebra." *arXiv preprint arXiv:1710.08717* (2017).
4. Liu Jun, Yang Liang, and Nirwan Ansari. "Spark-based large-scale matrix inversion for big data processing." *IEEE Access* 4 (2016): 2166–2176.
5. Hoque Reazul, et al. "Dynamic task discovery in parsec: A data-flow task-based runtime." *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2017.
6. Misra Chandan, et al. "SPIN: A fast and scalable matrix inversion method in apache spark." *Proceedings of the 19th International Conference on Distributed Computing and Networking*. 2018.
7. Song Fengguang, et al. "Scalable tile communication-avoiding QR factorization on multicore cluster systems." *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010.
8. Buttari Alfredo, et al. "Parallel tiled QR factorization for multicore architectures." *Concurrency and Computation: Practice and Experience* 20.13 (2008): 1573–1590.
9. Cao Quinglei, et al. "Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools." *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 2019.
10. Zheng Weijian, et al. "Scaling Up Parallel Computation of Tiled QR Factorizations by a Distributed Scheduling Runtime System and Analytical Modeling." *Parallel Processing Letters* 28.01 (2018): 1850004.
11. Ballard Grey, et al. "Communication-optimal parallel and sequential Cholesky decomposition." *SIAM Journal on Scientific Computing* 32.6 (2010): 3495–3523.
12. Ltaief Hatem, et al. "A scalable high performant Cholesky factorization for multicore with GPU accelerators." *International Conference on High Performance Computing for Computational Science*. Springer, Berlin, Heidelberg, 2010.
13. Eugene Tulika, Anatoliy Doroshenko, Kostiantyn Zhereb, Adaptation of Legacy Fortran Applications to Cloud Computing. *Proceedings of the 12th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer*. Kyiv, Ukraine, June 21–24. 2016. P. 119–126.
14. Eugene Tulika, Anatoliy Doroshenko, and Kostiantyn Zhereb, Using Choreography of Actors and Rewriting Rules to Adapt Legacy Fortran Programs to Cloud Computing. In "Information and Communication Technologies in Education, Research, and Industrial Applications", A. Ginige et al. (Eds.): ICTERI 2016, CCIS 783, Springer International Publishing AG, 2017. P. 1–21.

References

1. Song Fengguang, Asim YarKhan, and Jack Dongarra. "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems." *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009.
2. Dongarra J.J., Robert Van De Geijn, and David Walker. *LAPACK Working Note* 43: *A Look at Scalable Dense Linear Algebra Libraries*. University of Tennessee, Computer Science Department, 1992.
3. Seeger Matthias, et al. "Auto-differentiating linear algebra." *arXiv preprint arXiv:1710.08717* (2017).
4. Liu Jun, Yang Liang, and Nirwan Ansari. "Spark-based large-scale matrix inversion for big data processing." *IEEE Access* 4 (2016): 2166–2176.
5. Hoque Reazul, et al. "Dynamic task discovery in parsec: A data-flow task-based runtime." *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2017.
6. Misra Chandan, et al. "SPIN: A fast and scalable matrix inversion method in apache spark." *Proceedings of the 19th International Conference on Distributed Computing and Networking*. 2018.
7. Song Fengguang, et al. "Scalable tile communication-avoiding QR factorization on multicore cluster systems." *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010.
8. Buttari Alfredo, et al. "Parallel tiled QR factorization for multicore architectures." *Concurrency and Computation: Practice and Experience* 20.13 (2008): 1573–1590.
9. Cao Quinglei, et al. "Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools." *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 2019.
10. Zheng Weijian, et al. "Scaling Up Parallel Computation of Tiled QR Factorizations by a Distributed Scheduling Runtime System and Analytical Modeling." *Parallel Processing Letters* 28.01 (2018): 1850004.
11. Ballard Grey, et al. "Communication-optimal parallel and sequential Cholesky decomposition." *SIAM Journal on Scientific Computing* 32.6 (2010): 3495–3523.
12. Ltaief Hatem, et al. "A scalable high performant Cholesky factorization for multicore with GPU accelerators." *International Conference on High Performance Computing for Computational Science*. Springer, Berlin, Heidelberg, 2010.
13. Eugene Tulika, Anatoliy Doroshenko, Kostiantyn Zhereb, Adaptation of Legacy Fortran Applications to Cloud Computing. *Proceedings of the 12th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer*. Kyiv, Ukraine, June 21–24. 2016. P. 119–126.
14. Eugene Tulika, Anatoliy Doroshenko, and Kostiantyn Zhereb, Using Choreography of Actors and Rewriting Rules to Adapt Legacy Fortran Programs to Cloud Computing. In "Information and Communication Technologies in Education, Research, and Industrial Applications", A. Ginige et al. (Eds.): ICTERI 2016, CCIS 783, Springer International Publishing AG, 2017. P. 1–21.

Одержано 10.03.2020

***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 60.
Індекс Гірша – 6.
http://orcid.org/0000-0002-8435-1451,

Туліка Євгеній Мирославович,
інженер програмних систем.
Кількість наукових публікацій в українських виданнях – 6.
Кількість наукових публікацій в зарубіжних виданнях – 2.
http://orcid.org/0000-0002-0153-0148.

Місце роботи авторів:

Інститут програмних систем НАН України,
03187, м. Київ-187, проспект Академіка Глушкова, 40.
Тел.: (044) 526 3559.
E-mail: doroshenkoanatoliy2@gmail.com,
eugene.tulika@gmail.com