ПРОГРАМНО-ТЕХНІЧНІ КОМПЛЕКСИ

В.П. ШИЛО, С.В. ЧУПОВ

УДК 519.687.4

ЕФЕКТИВНІ СПОСОБИ ОРГАНІЗАЦІЇ ПАРАЛЕЛЬНОЇ РОБОТИ ОПТИМІЗАЦІЙНИХ АЛГОРИТМІВ

Анотація. Наведено короткий огляд програмних та технічних засобів сучасної обчислювальної техніки, які дають змогу будувати ефективні системи паралельних обчислень. Представлено структурні схеми та детально описано роботу об'єднань таких паралельних оптимізаційних алгоритмів, як портфель і команда. Відзначено особливості організації роботи алгоритмів у цих об'єднаннях, пов'язані з синхронізацією паралельної роботи алгоритмів команди та з узгодженим обробленням отриманих алгоритмами даних.

Ключові слова: паралельні алгоритми, портфель алгоритмів, команда алгоритмів, синхронізація доступу до спільних даних.

ВСТУП

На сьогодні переважна більшість обчислювальних систем є багатопроцесорними. Тому розроблення нових паралельних алгоритмів пошуку розв'язків оптимізаційних задач і трансформація наявних послідовних алгоритмів у паралельні дасть змогу значно підвищити ефективність процесу розв'язання таких задач та є надзвичайно важливою проблемою. Основними об'єктами, на базі яких побудована вся робота сучасних операційних систем, є процеси (process) та потоки (thread). Загалом процесом можна назвати програму, що виконується процесором. За своєю суттю процеси — статичні об'єкти, які є лише контейнерами потоків. Саме потоки є виконавцями команд програми, розміщеної в адресному просторі процесу. Під час створення процесу один потік, який називають первинним, породжується автоматично. Довільний потік у разі потреби може створювати інші потоки. За логікою операційної системи всі потоки працюють паралельно. Це дає змогу реалізовувати різноманітні схеми паралельних обчислень.

Паралельний алгоритм — це схема розв'язання задачі, в якій деякі його кроки виконуються паралельно. Якщо вхідні дані для деяких кроків алгоритму не залежать від результатів роботи інших кроків, такі кроки можна виконувати паралельно. На сьогодні більшість оптимізаційних алгоритмів побудовано за послідовною схемою і доволі складно виокремити кроки, які б допускали їхню паралельну роботу. Але можна використовувати декілька різних послідовних алгоритмів або копій одного рандомізованого алгоритму [1] одночасно для розв'язання конкретної задачі. Очевидно, що за умови такого підходу ймовірність отримання якісного розв'язку буде значно вищою, ніж у випадку використання одного алгоритму. Якщо є декілька послідовних алгоритмів розв'язання однієї задачі, то найпростіший підхід до організації їхньої паралельної роботи полягає

© В.П. Шило, С.В. Чупов, 2019

у створенні окремого процесу для кожного алгоритму. Такі процеси працюватимуть паралельно над розв'язанням однієї задачі. Наприклад, якщо є n алгоритмів, то буде створено n процесів з одним потоком у кожному. З іншого боку, кожен алгоритм можна оформити у вигляді окремої підпрограми. Це дає змогу в одному процесі організувати паралельну роботу кількох послідовних алгоритмів. При цьому кожен алгоритм буде працювати у своєму власному потоці. Наприклад, для роботи n алгоритмів буде створено n потоків, які працюватимуть паралельно в одному процесі. Очевидно, повинна існувати можливість обміну результатами роботи між різними алгоритмами, які працюють паралельно над розв'язанням однієї задачі або, принаймні, фіксації отриманих результатів у одному загальнодоступному місці для подальшого оброблення.

Слід зауважити, що під час створення процесу операційна система виділяє для нього певний об'єм віртуальної пам'яті. Адресні простори всіх процесів ізольовані один від одного, тому прямий доступ до пам'яті одного процесу з іншого заборонений. Проте, є багато можливостей для організації паралельної роботи алгоритмів, які працюють в окремих процесах. Одна із найбільш розповсюджених технологій розпаралелювання обчислень — технологія MPI (Message Passing Interface) [2], яка є програмним інтерфейсом для передавання даних, що дає змогу здійснювати обмін повідомленнями між процесами. Цю універсальну технологію можна застосовувати в комп'ютерних системах будь-якої структури, як у системах глобальної мережі, так і у системах локальної мережі, та на окремих комп'ютерах. Для своєї роботи інтерфейс MPI використовує системний об'єкт низького рівня сокет (socket) з протоколом передавання даних UDP або TCP / IP. Якщо паралельну роботу алгоритмів організовують у межах локальної мережі або на окремому комп'ютері, то найбільш вдалим вибором для обміну даними є використання іменованих каналів (named pipes) [3]. Що стосується програмної реалізації робота з іменованим каналом нічим не відрізняється від роботи із звичайним двійковим файлом. При цьому швидкість обміну даними між процесами більша, ніж у разі використанні сокетів. І нарешті, якщо паралельну роботу алгоритмів планують здійснювати лише на одному комп'ютері, то найбільш ефективним буде використання файла, відображеного у пам'ять (file mapping) [3]. За цією технологією у віртуальному адресному просторі комп'ютера виділяється блок пам'яті і встановлюється взаємно однозначна відповідність між даними файла та комірками цього блоку пам'яті. Під час читання або модифікації певних комірок блоку автоматично здійснюється доступ до відповідних даних у файлі. Довільний процес може отримати доступ до файла, який відображено у пам'ять. При цьому кожен такий процес отримує адресу початку спільного віртуального блоку пам'яті й надалі всі операції з введення/виведення даних відбуваються на рівні роботи з оперативною пам'яттю. Цей спосіб є найшвидшим з-поміж усіх способів міжпроцесного обміну даними.

Теоретично всі потоки у системі працюють паралельно та незалежно один від одного. Але таку паралельність варто називати віртуальною, оскільки ступінь реального паралелізму залежить від технічних характеристик комп'ютера, зокрема, від кількості процесорів та ядер на одному процесорі. Для організації паралельної роботи потоків в операційній системі використовується механізм квантування часу. За цією технологію кожному потоку надається процесор тільки протягом невеликого проміжку часу, який називають квантом часу. У кожний момент часу кількість потоків, які реально працюють, дорівнює кількості фізичних процесорів та процесорних ядер у системі. Решта потоків призупиняються та очікують на звільнення процесорів після завершення чергового кванта часу. Загалом операційна система будь-якому потоку може призначати будь-який про-

цесор. Очевидно, що робота потоку сповільнюється, якщо після кожного кванта часу він призупиняється та переводиться у стан очікування. Тому чим більше послідовних квантів часу потік буде активним, тим більшою буде ефективність його роботи. Для збільшення неперервного часу активності потоку існує можливість визначати, які саме логічні процесори можуть використовуватися цим потоком чи процесом. Під логічним процесором розуміють або один фізичний процесор, або одне ядро фізичного процесора, або один фізичний потік ядра процесора. Наприклад, для систем Windows визначена команда *SetProcessAffinityMask* [4], яка дозволяє призначати бажані логічні процесори всім потокам конкретного процесу. Аналогічні дії для систем Linux здійснює команда *sched_ setaffinity* [5]. Крім того, існує можливість призначати для роботи окремого потоку один або декілька логічних процесорів. У системах Windows для цього використовується команда *SetThreadAffinityMask* [4], а в Linux — команда *pthread_ setaffinity_ np* [5].

Розглянуті вище програмні та технічні засоби дають змогу конструювати системи паралельних обчислень потрібної складності і ефективності, а також використовувати на повну потужність багатопроцесорну структуру сучасних комп'ютерів.

ОРГАНІЗАЦІЯ РОБОТИ ПОРТФЕЛЯ АЛГОРИТМІВ

Об'єднанням алгоритмів назвемо множину алгоритмів $A_1, A_2, \ldots, A_n$, які працюють паралельно над розв'язанням однієї задачі. Якщо в процесі роботи алгоритмів обмін результатами між ними не відбувається, тобто вони працюють незалежно один від одного, то таке об'єднання називають портфелем алгоритмів [6]. Після завершення роботи всіх алгоритмів $A_1, A_2, \ldots, A_n$ портфеля вибирається найкращий з отриманих розв'язків задачі.

Портфель алгоритмів можна представити у вигляді графу, в якого всі вершини ізольовані (нуль-граф). Нехай $P_i(A_j)$, $i = 0, 1, \ldots, n$, $j = 1, 2, \ldots, n$, — це *i*-й процес, в якому існує потік, що виконує *j*-й алгоритм. На рис. 1 зображено граф з чотирма ізольованими вершинами, який асоціюється з портфелем із чотирьох алгоритмів. Кожна вершина графу представляє собою окремий процес, в якому працює один алгоритм. Ізольованість вершин графу означає, що всі чотири алгоритми працюють незалежно один від одного.

Хоча алгоритми портфеля працюють незалежно один від одного, для зручного оброблення отримані ними результати варто зберігати в одному загальнодоступному місці. Це може бути адресний простір процесу, спеціально призначений для фіксації та подальшого оброблення результатів роботи кожного алгоритму портфеля. Таку ситуацію для портфеля з чотирьох алгоритмів представлено на рис. 2 у вигляді 1-регулярного графу. Зв'язані пари вершин $P_0(A_j) \leftrightarrow P_j(A_j)$, $j = 1, 2, \ldots, n$, вказують на те, що у процесі P_j алгоритм A_j виконує основні обчислення та передає результати своєї роботи до процесу P_0. У процесі P_0 результати роботи алгоритму A_j фіксуються та зберігаються для подальшого оброблення. Крім того, процес P_0 також може передавати потрібні алгоритму A_j дані до процесу P_j.

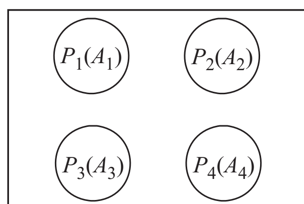

Рис. 1. Граф портфеля з чотирьох алгоритмів

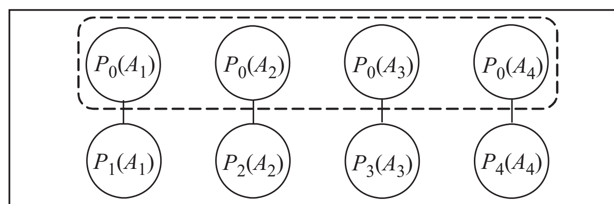

Рис. 2. Граф портфеля алгоритмів з фіксацією результатів

Скористаємося термінологією «клієнт-сервер» для опису роботи портфеля алгоритмів. Робота портфеля починається зі створення процесу, який отримує результати від усіх алгоритмів об'єднання та за потреби передає певні дані алгоритмам цього об'єднання. Такий процес назвемо головним процесом або сервером. На рис. 2 головний процес позначено P_0. Кожен алгоритм портфеля A_j, $j = 1, 2, \ldots, n$, працює у своєму власному процесі P_j. Такі процеси будемо називати клієнтами. Після створення головного процесу розпочинається цикл очікування для підключення до нього клієнтів портфеля. Як тільки черговий процес з одним із алгоритмів портфеля створено, на сервері автоматично створюється іменований канал для обміну даними між цим клієнтом та головним процесом. Разом із цим на сервері створюється потік $Thread_j$ для забезпечення зв'язку з клієнтом P_j (A_j). Цикл очікування продовжується до тих пір, поки усі n клієнтів не будуть підключені. При цьому, коли цикл очікування працює, усі потоки, як з боку сервера, так і з боку клієнтів, призупиняються. Коли всі клієнти приєднаються до сервера, робота всіх потоків синхронно поновлюється. У такий спосіб портфель алгоритмів розпочинає свою роботу. На рис. 3 у вигляді конструктивної схеми зображено взаємозв'язки між сервером та клієнтами, які виникають у процесі роботи портфеля алгоритмів.

Загалом робота кожного алгоритму портфеля A_j, $j = 1, 2, \ldots, n$, складається із однотипних кроків, які здійснюються до виконання умови завершення пошуку розв'язку задачі. На рис. 3 блок *Computing* у схемі роботи кожного клієнта відображає окремий крок алгоритму. Умова завершення пошуку розв'язку задачі схематично зображується як *Stop*?. Після завершення роботи чергового кроку алгоритму або, можливо, під час його роботи, отримані ним результати передаються на сервер. Як зазначено вище, з кожним клієнтом P_j (A_j), $j = 1, 2, \ldots, n$, пов'язаний свій власний потік $Thread_j$ на сервері. Основна функція цього потоку полягає в очікуванні повідомлень від свого клієнта, їхньому розшифруванні, обробленні та, за потреби, відправленні відповіді до працюючого алгоритму. На схемі рис. 3

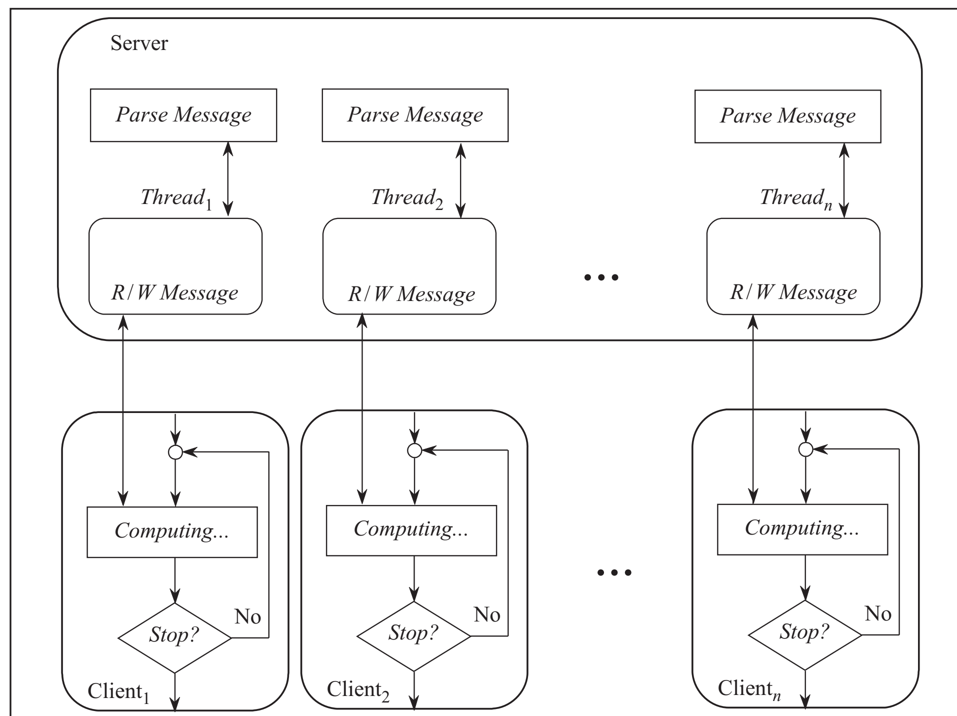

Рис. 3. Конструктивна схема роботи портфеля алгоритмів

такий обмін даними зображено як зв'язок $Computing \leftrightarrow R/W\ Message \leftrightarrow Parse\ Message$. При цьому блок $R/W\ Message$ у потоці $Thread_j$ на сервері відповідає за приймання та надсилання повідомлень клієнту, а блок *Parse Message* розшифровує повідомлення, обробляє дані, отримані від клієнта, та формує відповідь у вигляді повідомлення певної структури. Результати роботи всіх клієнтів фіксуються на сервері в окремих структурах даних. Доступ до результатів роботи алгоритму A_j, $j = 1, 2, \ldots, n$, дозволено тільки з потоку $Thread_j$. Отже, жодного узгодження доступу до даних між різними алгоритмами портфеля не потрібно. Після завершення роботи алгоритмів портфеля всі результати зафіксовано на сервері. Це дає змогу без жодних проблем визначити результат роботи портфеля — найкращий з отриманих алгоритмами портфеля A_j, $j = 1, 2, \ldots, n$, розв'язок.

ОРГАНІЗАЦІЯ РОБОТИ КОМАНДИ АЛГОРИТМІВ

Об'єднання алгоритмів називають командою алгоритмів [7], якщо у процесі їхнього паралельного виконання кожен алгоритм може здійснювати обмін результатами з будь-яким іншим алгоритмом цього об'єднання.

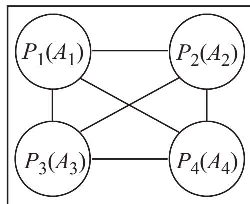

Рис. 4. Граф команди з чотирьох алгоритмів

Умовно команду алгоритмів A_j, $j = 1, 2, \ldots, n$, можна представити у вигляді графу, де кожна вершина зв'язана з усіма іншими, тобто повністю зв'язаного графу або $(n-1)$-регулярного графу. На рис. 4 зображено 3-регулярний граф з чотирма зв'язаними вершинами, який асоціюється з командою чотирьох алгоритмів. Кожна вершина графу представляє окремий процес $P_j(A_j)$, $j = 1, 2, 3, 4$, в якому працює один алгоритм. Зв'язність вершин графу означає, що всі чотири алгоритми у процесі роботи використовують дані один одного.

Хоча схема, представлена на рис. 4, добре відображає суть роботи команди алгоритмів, вона є не зовсім зручною у практичній реалізації. Для кожного алгоритму A_j, $j = 1, 2, \ldots, n$, команди потрібно встановлювати $(n-1)$ зв'язків з іншими алгоритмами. Крім того, всі результати роботи зберігаються в адресному просторі процесу, що містить сам алгоритм. А це, своєю чергою, робить дуже незручним остаточне оброблення результатів роботи всієї команди. Більш доцільним є створення окремого процесу, який виконує функцію контейнера для збереження результатів роботи всіх алгоритмів команди. При цьому будь-який алгоритм може звертатися до результатів роботи іншого алгоритму команди. На рис. 5 таку схему роботи команди представлено у вигляді дерева.

Коренева вершина дерева $P_0(A)$ визначає місце збереження результатів роботи всієї команди, доступне для будь-якого її алгоритму. Зв'язок $P_k(A_k) \leftrightarrow P_0(A) \leftrightarrow P_j(A_j)$, $k, j = 1, 2, \ldots, n$, вказує на те, що у процесах P_k та P_j алгоритми A_k та A_j виконують основні обчислення та передають результати до процесу P_0. У процесі P_0 результати роботи алгоритмів A_k та A_j фіксуються та зберігаються для подальшого оброблення. Крім того, через процес P_0 алгоритми A_k та A_j можуть отримувати результати роботи один одного. Іншими словами, між вершинами дерева $P_k(A_k)$ та $P_j(A_j)$ встановлено зв'язок $P_k(A_k) \leftrightarrow P_j(A_j)$.

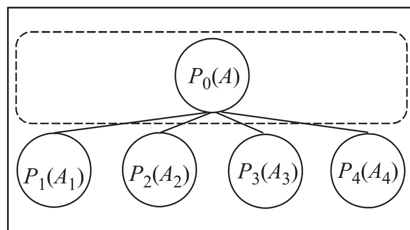

Рис. 5. Граф команди алгоритмів з можливістю обміну даними

Паралельну роботу команди алгоритмів організовано за технологією «клієнт-сервер» так само, як і для портфеля алгоритмів. На рис. 6 у вигляді конструктивної схеми

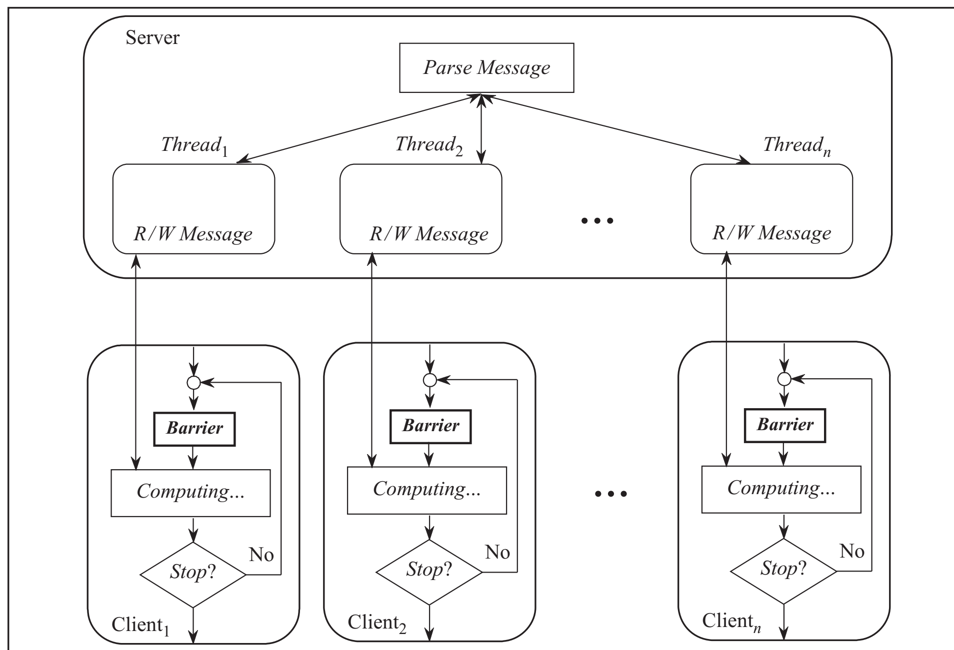

Рис. 6. Конструктивна схема роботи команди алгоритмів

зображено взаємозв'язки між сервером та клієнтами, які виникають у процесі роботи команди алгоритмів. Робота команди алгоритмів багато в чому схожа на роботу портфеля алгоритмів, детально описану вище. Тому розглянемо лише питання, що стосується використання спільних результатів, отриманих усіма алгоритмами команди. У процесі виконання чергового кроку кожним алгоритмом команди (блоки *Computing…*) отримані результати зберігаються на сервері. Крім того, алгоритми звертаються до сервера та з'ясовують, чи не отримано іншим алгоритмом результат, кращий за поточний для цього алгоритму. Якщо такий результат є, то він фіксується як кращий розв'язок для цього алгоритму. Робота поточного кроку переривається та починається новий крок обчислень з урахуванням отриманого кращого розв'язку.

У випадку такої організації роботи команди алгоритмів виникають дві проблеми. Перша — це узгодження або синхронізація доступу кожним алгоритмом команди до спільних результатів. Друга проблема — синхронізація виконання кроків алгоритмами команди. Для розв'язання цих проблем сучасні операційні системи надають так звані примітиви синхронізації.

Якщо два (або більше) клієнти одночасно потребують доступу до спільних результатів роботи команди, системі потрібний механізм синхронізації. Це необхідно для впевненості у тому, що у цей момент часу тільки один клієнт використовує ресурс. Наприклад, якщо у певний момент часу алгоритм A_j зберігає новий результат на сервері, а водночас алгоритму A_k потрібно зчитати останні результати алгоритму A_j, то може виникнути неузгодженість у тому сенсі, що алгоритм A_k отримає недостовірні дані. *Mutex* (*mutual exclusion*) [8] — це примітив синхронізації, який надає ексклюзивний доступ тільки одному потоку. Якщо потік $Thread_j$ отримує *mutex*, то другий потік $Thread_k$, який також хоче отримати цей *mutex*, призупиняється та очікує на його звільнення першим потоком. Після отримання потоком об'єкта *mutex* цей потік може безпечно обробити або отримати потрібні результати. У такий спосіб доступ до результатів роботи команди з будь-якого клієнта буде синхронізовано.

Зазвичай алгоритми команди виконують свої кроки асинхронно, тому що синхронізація роботи алгоритмів зазвичай призводить до втрат часу. Але у деяких випадках, наприклад під час виконання рестарту [1], необхідно синхронізувати початок рестарту для всіх алгоритмів команди. Для цього зручно використовувати об'єкт синхронізації *barrier* [9] — примітив синхронізації, який застосовується для того, щоб змусити множину потоків чекати, поки кожен з них не виконає певну функцію або не досягне певної точки в їхньому виконанні. Таким чином, використання примітиву *barrier* дозволить усім алгоритмам команди розпочинати свої кроки одночасно. Якщо якийсь алгоритм раніше завершить поточний крок, то перед початком наступного він зупиниться на бар'єрі та буде очікувати завершення кроків усіма іншими алгоритмами. Коли всі алгоритми команди досягнуть бар'єра, він буде знятий та алгоритми синхронно розпочнуть свої наступні кроки. У разі застосування такого підходу синхронізація роботи алгоритмів команди забезпечується, але загальна ефективність команди може знижуватись. У момент досягнення бар'єра робота алгоритму блокується, що призводить до даремного витрачання часу. Для уникнення цього можна організувати роботу команди алгоритмів у такий спосіб, щоб замість даремного очікування він продовжував свою роботу, здійснюючи деякі додаткові дії. Такі дії можуть бути пов'язані або з уточненням поточного розв'язку, або з підготовкою до виконання наступного кроку.

Про ефективність запропонованих схем паралельної роботи оптимізаційних алгоритмів свідчать результати численних експериментальних розрахунків з розв'язання конкретних задач дискретної оптимізації за допомогою портфелів і команд алгоритмів [6, 7, 10, 11].

ВИСНОВКИ

У статті представлено базові способи організації паралельної роботи об'єднання оптимізаційних алгоритмів — портфеля та команди алгоритмів. Завдяки використанню таких сучасних багатопроцесорних (багатоядерних) обчислювальних систем та новітніх програмних технологій, як Intel® Advanced Vector Extensions (AVX, AVX2) [12], ці способи показали високу ефективність розв'язання оптимізаційних задач, зокрема, задачі про максимальний розріз графу та квадратичної задачі про призначення. У подальших дослідженнях планується побудувати нові схеми організації паралельної роботи об'єднання алгоритмів, яким притаманні ознаки як портфеля, так і команди. Це так званий портфель команд алгоритмів, коли об'єднання розбито на команди алгоритмів, які не обмінюються між собою інформацією. На нашу думку, це — найбільш універсальне об'єднання, яке за умови оптимального розбиття на команди дасть змогу зберегти найкращі риси і портфеля, і команди алгоритмів. Таким чином, використання можливостей розпаралелювання обчислень відкриває надзвичайно широкі перспективи для побудови нових ефективних алгоритмів розв'язання різноманітних оптимізаційних задач.

СПИСОК ЛІТЕРАТУРИ

1. Сергиенко И.В., Шило В.П. Задачи дискретной оптимизации. Проблемы, методы решения, исследования. Киев: Наук. думка, 2003. 264 с.
2. Gropp W., Lusk E., Skjellum A. Using MPI: Portable parallel programming with the message-passing interface. 2nd ed. MIT Press, 1999. 371 p.
3. Interprocess communications. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx.

4. SetProcessAffinityMask. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686223 (v=vs.85).aspx.
5. Sched_setaffinity. URL: https://linux.die.net/man/2/sched_setaffinity.
6. Шило В.П., Рощин В.А., Шило П.В. Построение портфеля алгоритмов для распараллеливания процесса решения задачи о максимальном взвешенном разрезе графа. *Компьютерная математика.* 2014. № 2. С. 163–170.
7. Shylo V.P., Glover F., Sergienko I.V. Teams of global equilibrium search algorithms for solving weighted MAXIMUM CUT problem in parallel. *Кибернетика и системный анализ*. 2015. Т. 51, № 1. С. 20–29.
8. Mutex. URL: http://www.cplusplus.com/reference/mutex/mutex/.
9. Class barrier. URL: https://www.boost.org/doc/libs/1_33_1/doc/html/barrier.html.
10. Сергиенко И.В., Шило В.П. Технология ядра для решения задач дискретной оптимизации. *Кибернетика и системный анализ*. 2017. Т. 53, № 6. С. 73–83.
11. Shylo V.P., Shylo O.V. Algorithm portfolios and teams in parallel optimization. In: *Optimization Methods and Applications: In Honor of the 80th Birthday of Ivan V. Sergienko*. Butenko S., Pardalos P.M., Shylo V. (Eds.). New York; Heidelberg; Dordrecht; London: Springer, 2017. P. 481–493.
12. Intel® 64 and IA-32 Architectures Software Developer’s Manual. Vol. 1: Basic Architecture. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.

Надійшла до редакції 20.08.2018

В.П. Шило, С.В. Чупов

ЭФФЕКТИВНЫЕ СПОСОБЫ ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНОЙ РАБОТЫ ОПТИМИЗАЦИОННЫХ АЛГОРИТМОВ

Аннотация. Приведен краткий обзор программных и технических средств современной вычислительной техники, позволяющих строить эффективные системы параллельных вычислений. Представлены структурные схемы и детально описана работа таких объединений параллельных оптимизационных алгоритмов, как портфель и команда. Отмечены особенности организации работы алгоритмов в этих объединениях, связанные с синхронизацией параллельной работы алгоритмов команды и согласованной обработкой полученных алгоритмами данных.

Ключевые слова: параллельные алгоритмы, портфель алгоритмов, команда алгоритмов, синхронизация доступа к общим данным.

V.P. Shylo, S.V. Chupov

EFFICIENT METHODS FOR TO ORGNIZE PARALLEL OPERATION OF OPTIMIZATION ALGORITHMS

Abstract. The software and hardware of modern computers, which allow generating efficient systems of parallel computing are briefly overviewed. Structural schemes are presented and operation of combined parallel optimization algorithms such as a portfolio and a team is described in detail. The special features of the organization of the operation of algorithms in these unions related to both synchronization of parallel operation of the algorithms of the team and coherent processing of the data obtained by the algorithms are specified.

Keywords: parallel algorithms, portfolio of algorithms, command of algorithms, synchronization of access to common data.

Шило Володимир Петрович,
доктор фіз.-мат. наук, професор, провідний науковий співробітник Інституту кібернетики ім. В.М. Глушкова НАН України, Київ, e-mail: v.shylo@gmail.com.

Чупов Сергій Вікторович,
кандидат фіз.-мат. наук, доцент кафедри Вищого державного навчального закладу «Ужгородський національний університет», e-mail: serhii.chupov@uzhnu.edu.ua.