УДК 681.3

С.Д. ПОГОРІЛИЙ, М.І. ТРИБРАТ, Д.Ю. ВІТЕЛЬ

ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНИХ ВЕРСІЙ АЛГОРИТМУ ФЛОЙДА-УОРШАЛА ДЛЯ SMP- ТА MPP-АРХІТЕКТУР

Анотація. *Створено паралельні версії алгоритму Флойда-Уоршала для SMP- і MPP-архітектур та досліджено і проаналізовано їх часові характеристики. Визначено доцільність застосування певної архітектури в залежності від розмірності задачі.*
Ключові слова: *SMP-архітектури, MPP-архітектури, найкоротший шлях між джерелом даних та їх приймачем, алгоритм Флойда-Уоршала, механізми взаємодії IPC, синхронізація, спін-блокування, потік, процес, технологія MPI, кеш-пам'ять.*

Аннотация. *Созданы параллельные версии алгоритма Флойда-Уоршала для SMP- и MPP-архитектур, исследованы и проанализированы их временные характеристики. Определена целесообразность применения определенной архитектуры в зависимости от размерности задачи.*
Ключевые слова: *SMP-архитектуры, MPP-архитектуры, кратчайший путь между источником данных и их приемником, алгоритм Флойда-Уоршала, механизмы взаимодействия IPC, синхронизация, спин-блокировка, поток, процесс, технология MPI, кэш-память.*

Abstract. *Parallel versions of the Floyd-Warshall algorithm for SMP-and MPP-architectures were created. Their temporal characteristics were investigated and analyzed. It was shown that the expediency of particular architecture usage depends on the dimension of the task.*
Keywords: *SMP-architecture, MPP-architecture, the shortest path between a data source and the receiver, the Floyd-Warshall algorithm, mechanisms of interaction IPC, synchronization, spin locks, thread, process, MPI technology, cache memory.*

1. Вступ

Одним із головних факторів, що визначає ефективність функціонування сучасних комп'ютерних мереж, є алгоритм знаходження найкоротшого шляху між джерелом даних та їх приймачем. На практиці до цих алгоритмів висувається ряд вимог, які включають точність, надійність, стабільність, справедливість (по відношенню до вузлів, які обслуговуються) та оптимальність. Задовольняючи цим критеріям, алгоритм може бути покладений за основу протоколу маршрутизації, який повинен також визначати набір правил взаємодії вузлів, що приймають участь у процесі маршрутизації. Поряд з алгоритмами Белмана-Форда (протокол RIP) та Дейкстри (протокол OSPF), алгоритм Флойда-Уоршала дозволяє знаходити найкоротші шляхи між усіма парами вершин (складність – $O(n^3)$, де n – загальна кількість вузлів мережі). Проте він не може працювати із графами, що мають цикли від'ємної ваги (на відміну від алгоритму Белмана-Форда).

Алгоритм знаходить оптимальні шляхи, визначаючи на кожному кроці k вагову матрицю D^k (що складається з елементів d_{ij}^k) через вагову матрицю на попередньому кроці (D^{k-1}) [1]:

$$d_{ij}^k = \begin{cases} w_{ij} - \text{вага ребра } i \to j \text{ при } k=0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) \text{ при } k \neq 0 \end{cases}.$$

Кількість кроків дорівнює кількості вузлів графа n. Крім того, на кожній ітерації визначається матриця передування P^k, що на перетині i-го рядка та j-го стовпчика містить елемент π_{ij}^k – номер вершини, яка передує j-тій у найкоротшому шляху з i в j:

$$\pi_{ij}^{k} = \begin{cases} NIL \text{ при } i = j \text{ або } w_{ij} = \infty \\ i \text{ в іншому випадку} \end{cases},$$

$$\pi_{ij}^{k} = \begin{cases} \pi_{ij}^{k-1} \text{ при } d_{ij}^{k-1} \le (d_{ik}^{k-1} + d_{kj}^{k-1}) \\ \pi_{kj}^{k-1} \text{ в іншому випадку} \end{cases}.$$

Мета роботи полягає у дослідженні часу виконання паралельних реалізацій алгоритму Флойда-Уоршала в залежності від кількості вузлів (MPP-архітектури) та ядер мікропроцесора (SMP-архітектури).

2. Архітектура комп'ютерних систем зі спільною та розподіленою пам'яттю

З появою багатоядерних процесорів, парадигми паралельних кластерних обчислень з'явилась можливість оптимізувати роботу алгоритмів за критерієм часу, використовуючи технології розпаралелювання. Створені таким чином реалізації алгоритмів призначені для виконання в системах із розподіленою пам'яттю або в системах зі спільною пам'яттю.

До систем зі спільною пам'яттю належать багатоядерні комп'ютери та мультипроцесори. Їх особливість полягає у тому, що всі обчислювальні одиниці (ядра чи процесори) мають доступ до усієї наявної в системі пам'яті на відміну від систем із розподіленою пам'яттю (мультикомп'ютери, кластери), де кожен обчислювальний вузол контролює доступ до своєї ділянки. З одного боку, це приводить до того, що такі системи легше програмуються, ніж системи із розподіленою пам'яттю. Проте, з іншого боку, їх апаратна реалізація складніша. Ці системи поділяють на такі підкласи, як UMA, NUMA, COMA, різниця між якими визначається часом доступу до різних частин оперативної пам'яті та її організацією (розподілом між обчислювальними вузлами) [2]. Методи побудови застосувань для систем зазначених двох класів є суттєво різними.

3. Підходи до створення паралельних застосувань для систем зі спільною пам'яттю

Створення паралельних застосувань у системах зі спільною пам'яттю цілком залежить від цільової операційної системи (ОС). Остання, виконуючи роль інтерфейсу між застосуванням і апаратним забезпеченням, постачає ряд методів для доступу до ресурсів та створення паралельних потоків інструкцій на окремих вузлах системи. Більшість нових технологій грунтуються на цих ключових методах і призначені спростити процесс побудови паралельних застосувань, пропонуючи нові підходи.

Центральними сутностями сучасних ОС є потік та процесс [3, 4]. Основне їх призначення – ефективне планування використання ресурсу процесорного часу та захист застосувань від взаємного впливу. Саме по собі застосування є статичним набором команд, проте процес ОС – це набір ресурсів, даних та правил роботи ОС з ними. При створенні процес отримує такі ресурси, як віртуальний адресний простір, процесорний час, початковий потік. Будь-які два процеси в системі ізольовані і не можуть (без додаткових механізмів ОС) впливати один на одного. Одним із підходів до створення паралельних застосувань для систем зі спільною та розподіленою пам'яттю є використання паралельних процесів, які виконуються на обчислювальних вузлах системи. Інший підхід грунтується на використанні набору паралельних потоків у межах одного процессу. Потік фактично являє собою набір інструкцій, що послідовно обробляються одним вузлом системи. Проте з потоками також асоціюють додаткові ресурси – контекст (збережений набір регістрів CPU, що використовується ОС у процесах планування виконання потоків), стек (область віртуального адресного простору з динамічним виділенням пам'яті і LIFO-способом доступу), локальна пам'ять потоку (TLS – пам'ять, що доступна лише даному потоку). Різниця в зазначених методах формування паралельного застосування полягає в тому, що потоки в межах процесу не є ізольованими (окрім TLS-ресурсу) і тому фактично можуть

взаємодіяти між собою без додаткових механізмів системи. Процеси, в свою чергу, взаємодіють між собою за допомогою механізмів interprocess comunications (IPC) системи, до яких належать:

• Спільна пам'ять (в Unix-системах – shared memory objects, в Windows – file mapping). Підхід, при якому одна і та сама ділянка фізичної пам'яті відображається у віртуальні адресні простори декількох процесів.

• Гнізда (Berkley sockets). Обмін даними між процесами в мережі грунтується на цьому механізмі. Можна використовувати як для систем зі спільною, так із розподіленою пам'яттю.

• Конвейєри (pipes) пропонують підхід з використанням структур типу FIFO для обміну даними між процесами як в одній системі, так і в різних.

• Інші системно залежні механізми (Windows – mailbox, DDE, console buffer і т.д).

Створення каналу обміну даними зазначеними методами вимагає часового ресурсу. Далі наведені приклади системних механізмів ОС Windows, що дозволяють значно пришвидшити обмін даними між процесами, накладаючи при цьому ряд обмежень.

1. Використання процесів-відлагоджувачів дозволяє безпосередньо виконувати запис-зчитування в чужому адресному просторі, проте цей метод вимагає від таких процесів знання розподілу виділених сторінок пам'яті (це створює додаткові витрати часу). Крім того, ці процеси мають володіти відповідними правами доступу до чужого адресного простору.

2. Спільний сегмент пам’яті. При створенні процесу ОС виділяє фізичну пам’ять для набору інструкцій та даних. При цьому зміна процесом певної сторінки коду приведе до того, що в фізичній пам’яті з’явиться змінена копія. Оригінальні сторінки пам'яті також лишаються виділеними (механізм копіювання при записі). При застосуванні можна змінити дану поведінку копіювання сторінок. Це призведе до того, що декілька процесів зможуть модифікувати сторінки без копіювання. Але при цьому процеси мають бути породжені з одного виконуваного файла.

3. Буфер консолі. Батьківський і породжений процеси можуть мати спільну консоль і відповідні буфери вводу-виводу. Але процеси, що взаємодіють таким чином, мають належати одному дереву процесів з усієї ієрархії.

4. Особливості синхронізації для Windows-платформ

Синхронізація є одним із найважливіших способів взаємодії декількох потоків (одного чи різних процесів), оскільки забезпечує атомарний доступ до спільних даних. Більшість механізмів синхронізації Windows передбачає використання об’єктів ядра системи. Тому їх застосування ініціює виконання додаткового набору інструкцій, що переводять CPU з одного режиму (кільця захисту) в інший. Виконання цих додаткових команд може значно знизити продуктивність усього застосування.

Одним із широко застосованих шаблонів паралельного програмування є синхронізація типу бар’єр. Ідея такої взаємодії потоків полягає у тому, що в кожному з останніх існує група інструкцій, по досягненні якої подальше виконання даного потоку припиняється доти, доки всі інші потоки із групи не до-

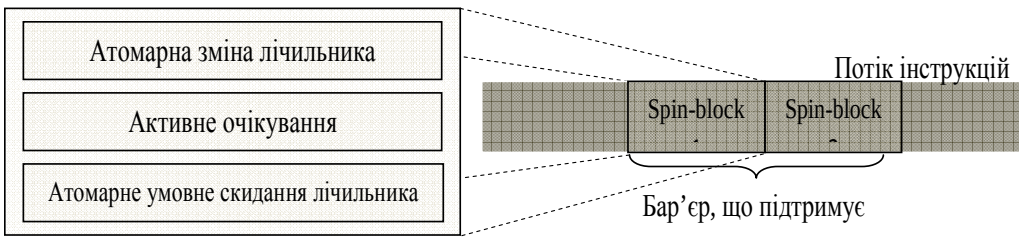

Рис. 1. Бар’єр, що використовує спін-блокування

сягнуть даного набору. Ці інструкції називають бар'єром. Побудувати такий спосіб взаємодії на існуючих примітивах ОС можна за допомогою механізму спін-блокування (spin-blocking). Схему реалізації бар'єра наведено на рис. 1.

Спін-блокування використовує один спільний для всіх потоків у групі лічильник, доступ до якого атомарний (на Windows-платформі реалізується за допомогою Interlocked-функцій, що не переводять CPU в інше кільце захисту). Проходячи набір інструкцій, потік змінює лічильник і входить в очікування, що реалізується за допомогою циклу. Це очікування активне і споживає ресурс процесорного часу. Для поліпшення часових характеристик спін-блокування можна скористатися іншим механізмом. Кожний системний потік має пріоритет, що надається йому системою. Для кожного пріоритету ОС містить чергу активних (готових отримати ресурс процесорного часу) потоків. Якщо черга з найвищим пріоритетом не порожня, то потоки з цієї черги обслуговуються CPU. В іншому випадку обслуговуються потоки з нижчим пріоритетом. Обслуговування полягає в тому, що система надає можливість потоку виконуватись на CPU (одному з ядер) певний квант часу (Windows XP – 10 ms). Після цього стан потоку зберігається в контексті, а сам потік розміщується в кінці своєї черги. Поведінкою планувальника частково можна керувати програмою. Windows надає можливість потоку за допомогою API-функції Sleep передати залишок свого кванта іншому потоку в черзі. Таким чином, активне очікування можна організувати ефективніше, ніж при використанні інших примітивів синхронізації. Вихід із очікування відбувається за умови проходження останнім потоком з групи атомарної зміни лічильника. Цих дій достатньо для організації бар'єра, який потоки проходять один раз за своє існування. Щоб створити бар'єр, який потоки проходять циклічно, слід додати ще одне спін-блокування з іншим лічильником. Крім того, після кожного активного очікування потоки повинні модифікувати вектор лічильників певним чином. Реалізація такого роду бар'єра була використана в паралельній багатопотоковій та багатопроцесній версіях алгоритму. Перевагами спін-блокування є гарна маштабованість, ефективність, мала ресурсоємність реалізації. Недоліком є активне очікування і складність реалізації для систем із розподіленою пам'яттю.

5. Взаємодія потоків у системах із розподіленою пам'яттю

Основний спосіб взаємодії потоків у розподілених системах описується шаблоном Message Passing, ідея якого полягає в тому, що для доступу до даних деякого потоку з групи реалізується певний інтерфейс чи протокол обміну. Фактично, використовуючи цей інтерфейс, один потік може запросити відповідні дані чи дію у іншого потоку. При цьому інший потік має реалізувати певну поведінку для обслуговування запитів до нього. В такій схемі окремо можна виділити синхронну та асинхронну взаємодії. Виділення в системі головного потоку (або арбітра) або певної їх сукупності дозволяє створити синхронізацію, позбавлену тупикових ситуацій.

Частковим випадком Message Passing взаємодії є клієнт-серверна взаємодія, яка реалізується в таких технологіях, як MPI [5], Windows Communication Foundation (WCF), Web Services, Win Services. Основою реалізації даного шаблону є механізм сокетів. Цей вид взаємодії інтегрований в деякі мови програмування (наприклад, F# (MailboxProcessor) [6], Scala (Actors), ErLang). Реалізація Message Passing шаблону (рис. 2) фактично є реалізацією скінченного автомата.

6. Реалізація послідовної версії алгоритму

На рис. 3 наведено схему обрахунку вагової матриці на k-ому кроці реалізації послідовної версії алгоритму. Така схема передбачає економію ресурсів оперативної пам'яті, оскільки кожен крок алгоритму не вимагає додаткових витрат (нове значення d_{ij}^{k} зберігається в то-

му місці, де зберігалось d_{ij}^{k-1}). Проте, якщо граф матиме від'ємні ребра при умові відсутності циклів від'ємної ваги, такий метод роботи із пам'яттю може призвести до невірних результатів. Так, якщо на k-ому кроці вага d_{ij}^{k} – від'ємна, алгоритм модифікує її, що призведе до невірних обрахунків інших значень ваги на цій ітерації. Для коректної роботи алгоритму в цьому випадку слід використовувати додаткові ресурси пам'яті.

Результати були отримані для графів з кількістю вузлів від 200 до 5000. Графік (рис. 3) демонструє поліноміальну залежність часу виконання від кількості вузлів (поліноміальну складність алгоритму).

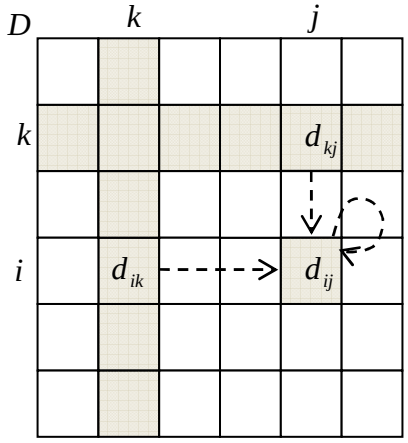

Рис. 2. Обрахунок матриці D на k-тій ітерації

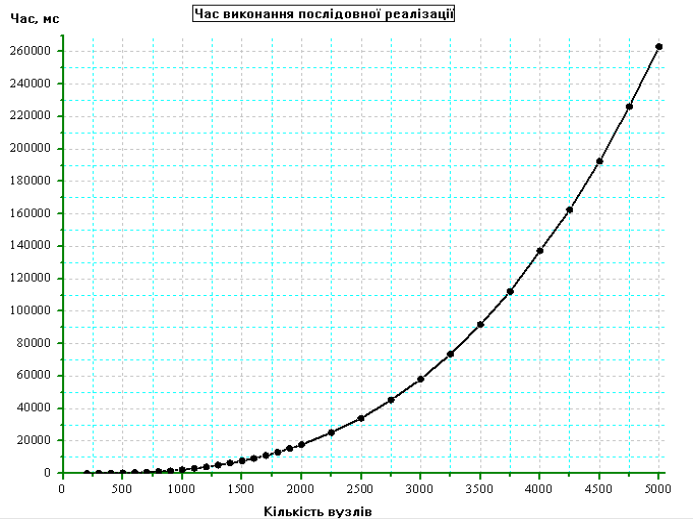

Рис. 3. Час роботи реалізації послідовної версії алгоритму в залежності від кількості вузлів графа

7. Концепція розпаралелювання алгоритму Флойда-Уоршала

На кожному кроці k алгоритм повністю обраховує нову матрицю D^k. При переході від послідовної версії алгоритму (1 потік) до паралельної (n потоків) слід зауважити, що кількість даних, яку повинен обробити один потік з групи, стає суттєво меншим. Таким чином, якщо потоки виконуються паралельно, то час виконання паралельної версії має бути приблизно в n разів меншим. Проте слід зазначити, що оскільки потоки мають на k-му кроці використовувати рядок і

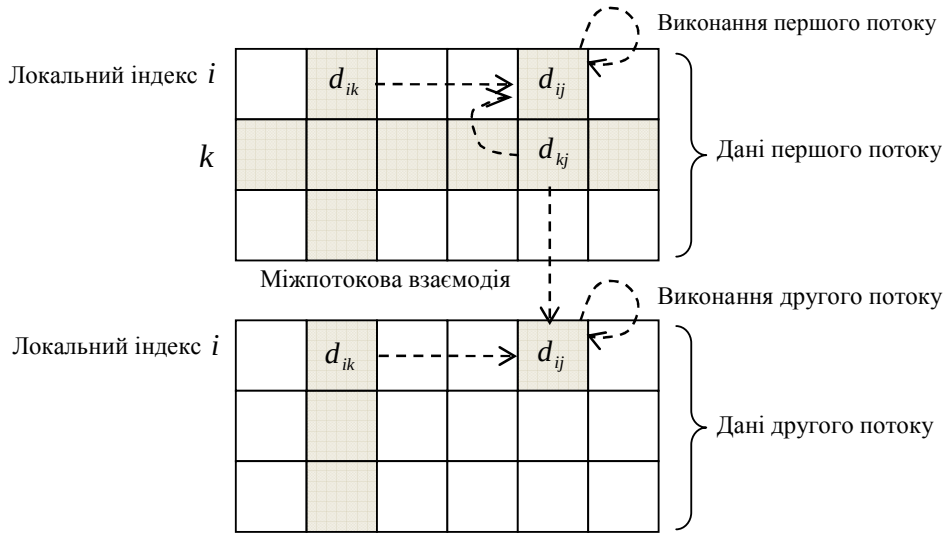

Рис. 4. Розбиття даних на множини, що паралельно обробляються

стовпчик з номером k, то ця область пам'яті виявляється спільною для кількох потоків. Якщо потоки виконуються в різних процесах, то існуватимуть затрати часу на обмін цими даними між ними, оскільки в цьому випадку матриця D^k розподілена по системі. Окрім того, на кожному кроці потоки слід синхронізувати за допомогою бар'єра, бо в іншому випадку існуватиме не нульова імовірність того, що з'явиться потік, який потрапить на наступну ітерацію раніше за інших і буде використовувати некоректні дані.

При створенні багатопотокової реалізації алгоритму вагова матриця D^k знаходиться у віртуальному адресному просторі одного процесу, кожен потік якого має безпосередній доступ до її комірок. Проте багатопроцесна реалізація для систем зі спільною пам'яттю вимагає створення спільної матриці D^k між усіма процесами. Зазначений механізм спільного сектора даних був використаний в реалізації для створення спільного вектора синхронізації. Для того, щоб зробити спільною для всіх процесів велику ділянку пам'яті, було використано механізм shared memory. В реалізації алгоритму для систем із розподіленою пам'яттю було використано технологію MPI (Microsoft реалізація, що входить до складу HPC Pack SDK). У цьому випадку матриця D^k була розподілена між вузлами мережі, і на кожному кроці певний вузол (що визначався динамічно під час виконання алгоритму) відсилав рядок з номером k іншим вузлам.

8. Інші особливості експериментальної реалізації

Програмні реалізації виконані за САА-схемами, наведеними в [6]. Основа алгоритму побудована за схемою 3, можливість використання якої зумовлена тим, що програмна реалізація моделює відсутність зв'язку у графі за допомогою деякої критичної величини ваги ребра (ця величина задається як вхідний параметр). Переваги подібного підходу такі:

• введення критичного значення ваги дає можливість оперувати з порожніми зв'язками графа так само, як із зв'язками, що мають скінченну вагу;

• зменшення кількості альфа-диз'юнкцій у вкладених альфа-ітераціях (зменшення числа перевірок на кожному кроці алгоритму).

Проте недоліком у цьому випадку є створене обмеження на максимально можливу вагу ребра вхідного графа.

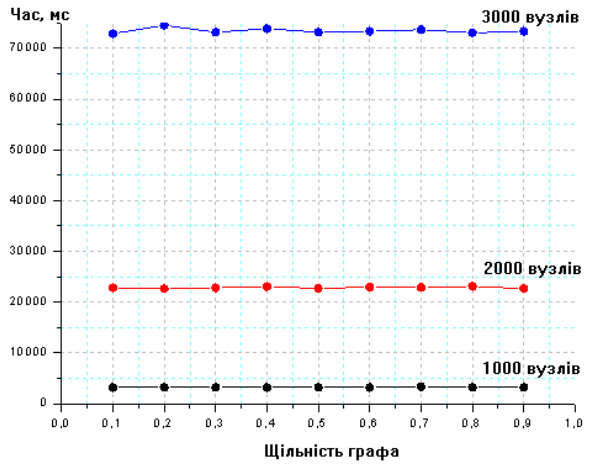

Рис. 5. Залежність часу роботи алгоритму від щільності графа. Приведені результати для графів з кількістю вузлів, рівною 1000, 2000, 3000

Наступні експериментальні результати були отримані для графів із фіксованою щільністю (відношенням кількості зв'язків з критичною вагою до загальної їх кількості), що дорівнює 0,5. Причиною цього є наслідки проведених попередніх досліджень залежності часу виконання алгоритму від щільності. Рис. 5 демонструє цю залежність для послідовної реалізації алгоритму для графів з кількістю вузлів, рівною 1000, 2000 та 3000. Аналогічні результати були отримані і для багатопотокової та багатопроцесної реалізацій, а отже можна зробити висновок про слабку залежність часу виконання алгоритму від щільності графів.

Як вже зазначалось, в утворених застосуваннях не було використано прив'язку потоків до ядер. З одного боку, це привело до суттєвого поліпшення часових характеристик реалізацій з використанням розглянутої синхронізації. Проте, з іншого боку, оскільки контекст потоку переміщується з одного ядра на інше неконтрольовано, розкид результатів,

що отримувались для одного графа при певній кількості потоків, збільшився. На зображених рисунках наводяться усереднені значення отриманих результатів. Велика відносна похибка спостерігалась при малих кількостях вузлів графа, що пояснюється малим часом виконання алгоритму і недостатньою точністю використаного таймера (було використано функцію GetSystemTime).

Запуск таймера для створених версій алгоритму відбувався безпосередньо перед циклом алгоритму, і тому отримані часові характеристики не включають час ініціалізації вагової матриці та матриці передування. Для багатопотокової та багатопроцесної реалізації врахований час створення паралельних потоків (процесів), проте для багатопроцесної реалізації не враховано час створення ділянки спільної пам'яті. Якщо ж враховувати повний час виконання реалізацій, то слід зазначити, що у випадку паралельних програм доцільніше було б організувати паралельне асинхронне зчитування потоками вхідного файла з ваговими коефіцієнтами.

Коректність отриманих застосувань було перевірено на заздалегідь обрахованих графах малої розмірності. Було встановлено відповідність між отримуваними результатами від різних реалізацій алгоритму.

9. Дослідження швидкодії реалізації алгоритму для систем зі спільною пам’яттю

Дослідження проводилися на графах з кількістю вузлів у діапазоні від 200 до 5000 (з кроком 100 для діапазону 200–2000 та 250 для діапазону 2000–5000) і кількістю потоків (процесів) від 2 до 20 (з кроком 2). Використана апаратна архітектура Intel Core Quad – 2,4 GHz – 64-розрядний, 4 ядра. Програмні реалізації створені мовою C з використанням Win-32 API (Win-64 API).

На рис. 6 наведено залежності часу виконання алгоритму від кількості вузлів графа та кількості паралельних потоків (процесів). Час виконання потокової реалізації виявляється меншим за час виконання реалізації, що використовує паралельні процеси, оскільки додатковий час витрачається на обмін даними між процесами при використанні IPC.

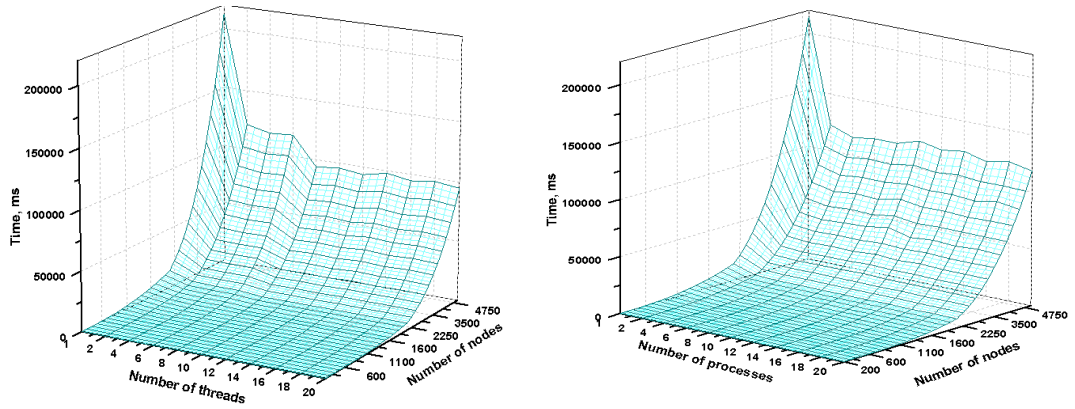

Рис. 6. Залежність часу виконання алгоритму від кількості паралельних потоків та кількості вузлів графа

Іншою особливістю результатів є характерні коливання залежності часу від кількості потоків (процесів). Локальні мінімуми цих коливань мають місце для кількості потоків, що кратна чотирьом. Це явище пояснюється тим, що дослідження проводилися на системі з чотирма ядрами. Тому при кількості потоків, що кратна 4, контекст кожного з них зберігається в L1 кеш-пам’яті певного ядра, а відповідний час на його завантаження не витрачається. Отже, в цьому випадку протягом виконання застосування кожне з ядер працює із своєю підмножиною потоків.

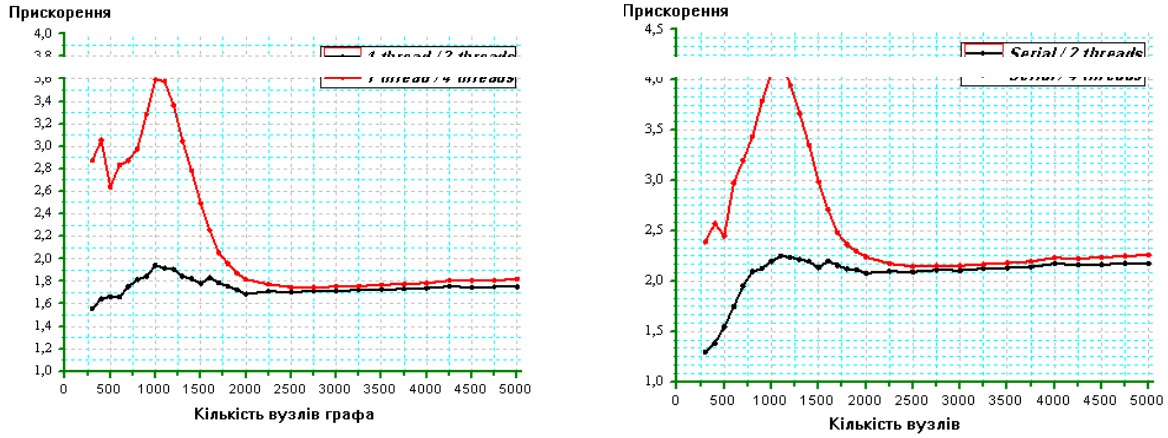

Рис. 7. Залежності відношення часів виконання багатопотокової реалізації з одним потоком (лівий рисунок) та послідовної реалізації (правий рисунок) до часу виконання багатопотокової реалізації для 2 та 4 потоків

На рис. 7 подано залежності відношення часу паралельної однопотокової та послідовної версій алгоритму до часу багатопотокової (2 та 4 потоки). З цих залежностей можна зробити такі висновки:

1. Відношення часу виконання послідовної версії алгоритму до паралельної з кількістю потоків, що рівна двом (чотирьом), більше за два (чотири відповідно) в певному діапазоні вузлів графа. Причиною, найбільш імовірно, є додаткова оптимізація паралельного коду компілятором (тобто, факт, що набір інструкцій, який виконує один потік у багатопотоковій реалізації над своїми даними, відрізняється від потоку інструкцій послідовної версії алгоритму). Тому більш коректно порівнювати часи виконання одного потоку паралельної реалізації з часами цієї ж реалізації, що використовує багато потоків. У цьому випадку максимальне отримане прискорення для двох потоків рівне приблизно 1,8, а для чотирьох – 3,6.

2. На отриманих рисунках чітко виділяються область піку прискорення алгоритму та область насичення. Причиною такої поведінки залежності прискорення від кількості вузлів графа може бути обмеженість ресурсу кеш-пам’яті L2 системи. При певній кількості вузлів графа починають активно виникати кеш-промахи, що призводить до латентної операції звертання до оперативної пам’яті. Проте канал взаємодії ядер з оперативною пам’яттю є спільним, і тому при обробці значного масиву даних паралельна обробка буде чергуватися із послідовним звертанням до оперативної пам’яті, що на залежності відповідає спаду до насичення. Оскільки кожне ядро процесора, на якому проводились дослідження, володіє власною кеш-пам’яттю, що менша за загальну кеш-пам'ять процесора, то використання групи паралельних потоків, які виконують алгоритм на графах з невеликою кількістю вершин (такою кількістю, що достатня для появи кеш-промахів), призведе до очікування частиною потоків даних з оперативної пам’яті. Отже, оскільки кількість даних, що обробляє кожен потік з набору, менша за кількість даних, що обробляється потоком в однопоточній версії, то границя, при якій активно проявляються кеш-промахи, зсувається в бік графів з великою кількістю вузлів. Це пояснює пік на отриманих характеристиках прискорення.

3. Отримані залежності для багатопроцесної реалізації (рис. 8) алгоритму подібні до багатопотокової реалізації, проте максимуми прискорення в даному випадку виявляються меншими. Це пояснюється тим, що операція створення паралельного потоку потребує менше ресурсів за операцію створення паралельного процесу.

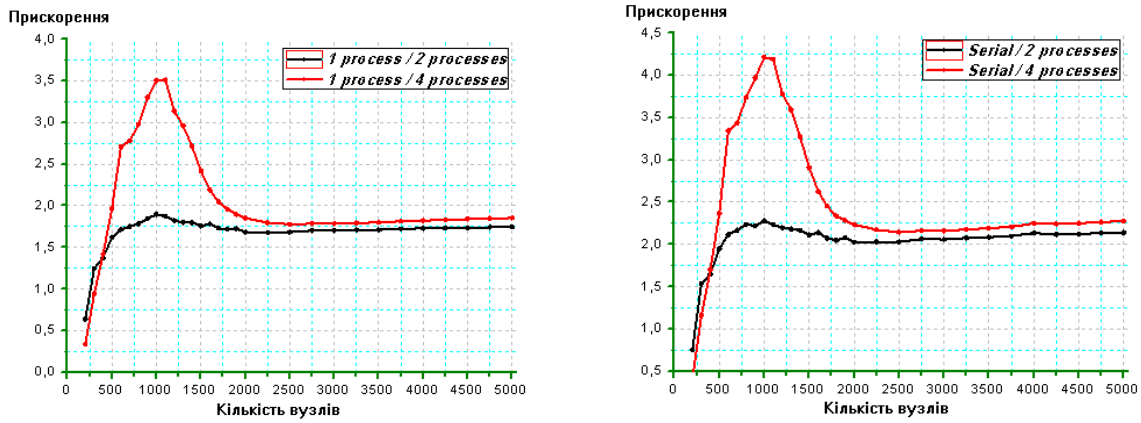

Рис. 8. Залежності відношення часів виконання багатопроцесної реалізації з одним процесом (лівий рисунок) та послідовної реалізації (правий рисунок) до часу виконання багатопроцесної реалізації для 2 та 4 процесів

Слід зазначити, що було проведено додаткові дослідження для підтвердження гіпотези про кеш-пам'ять. При цьому використано звичайний ітераційний алгоритм збільшення кожної комірки нульової матриці на одиницю паралельно та без використання синхронізації. Були отримані подібні характеристики. Також, при початкових дослідженнях неоптимізованої версії застосування отримані цікаві результати при заміні типів даних вхідних матриць з double та int на float та short. Для нових типів пік прискорення зміщувався в бік більшої кількості вузлів приблизно в 1,6 разів у порівнянні з початковою точкою максимуму. Якщо ж узяти до уваги обмеженість кеш-памяті, то теоретичне зміщення мало б бути рівним

$$\sqrt{\frac{sizeof(double)+sizeof(\text{ int })}{sizeof(float)+sizeof(short)}} = \sqrt{2} \approx 1{,}41\cdot$$

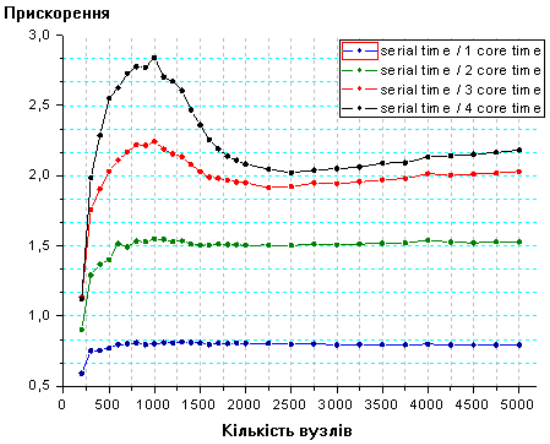

Рис. 9. Прискорення відносно послідовної версії алгоритму паралельної реалізації (1 вузол мережі, 1-4 використаних ядра) для систем із розподіленою пам'яттю

10. Дослідження швидкодії реалізації алгоритму для систем із розподіленою пам'яттю

При створенні застосування було використано технологію MPI, що надає можливість запускати паралельні процеси на різних вузлах мережі.

Перші дослідження були спрямовані на перевірку отриманих результатів попередніх етапів (паралельних реалізацій для систем зі спільною пам'яттю). Тому спочатку було використано один вузол мережі. Отримані залежності відношення часу послідовної версії алгоритму до часу побудованої паралельної реалізації для розподілених систем продемонстровані на рис. 9. Тут використано саме час послідовної версії для порівняння продуктивності паралельних реалізацій. Отже, ці залежності за поведінкою відповідають отриманим раніше залежностям прискорення. Це свідчить про те, що обраний спосіб синхронізації чи обміну даними між потоками слабко впливає на поведінку прискорення.

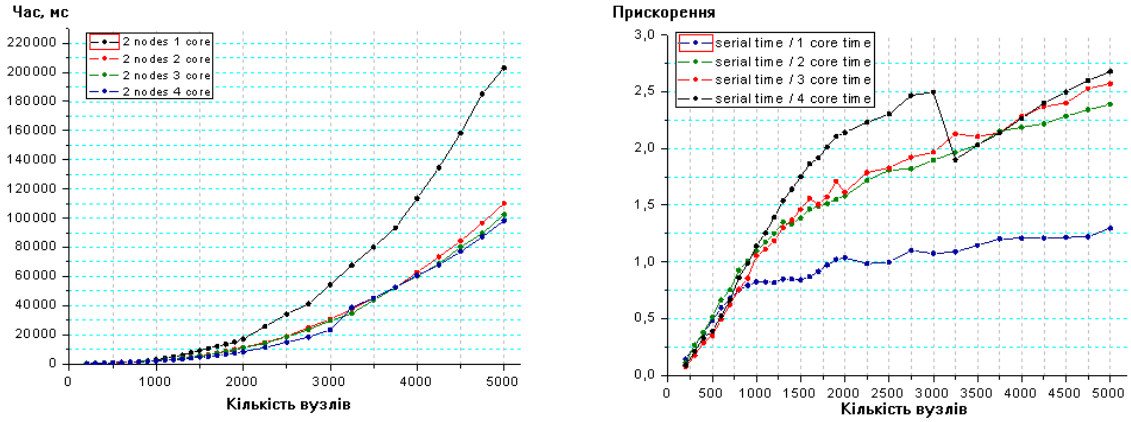

Рис. 10. Час виконання та прискорення відносно послідовної версії алгоритму паралельної реалізації (2 вузли мережі, 1-4 використаних ядра) для систем із розподіленою пам'яттю

Рис. 10 зображує характеристики застосування при використанні на двох комп'ютерах, з'єднаних між собою мережею. Отримані графіки прискорення мають ряд особливостей, що дозволяють зробити такі висновки.

1. На відміну від систем зі спільною пам'яттю, прискорення більше одиниці отримується для графів з великою кількістю вузлів. Порівняння прискорення алгоритму, що виконується на двох ядрах однієї машини, та алгоритму, що використовує одне ядро кожної з двох машин, приводить до висновку, що за певної критичної кількості вузлів графа друге застосування переважатиме за швидкодією перше. Це є перевагою систем із розподіленою пам'яттю. Проте для відносно малих кількостей вузлів графа продуктивність другого застосування гірша за продуктивність послідовної реалізації алгоритму.

2. При малій кількості вузлів графа алгоритм втрачає більше часу на обмін даними по мережі, ніж на корисну роботу. При великих кількостях даних – навпаки.

3. Викиди в залежності прискорення від кількості вузлів можна пояснити конкуренцією ядер за спільний канал обміну даними (це потребує додаткового дослідження).

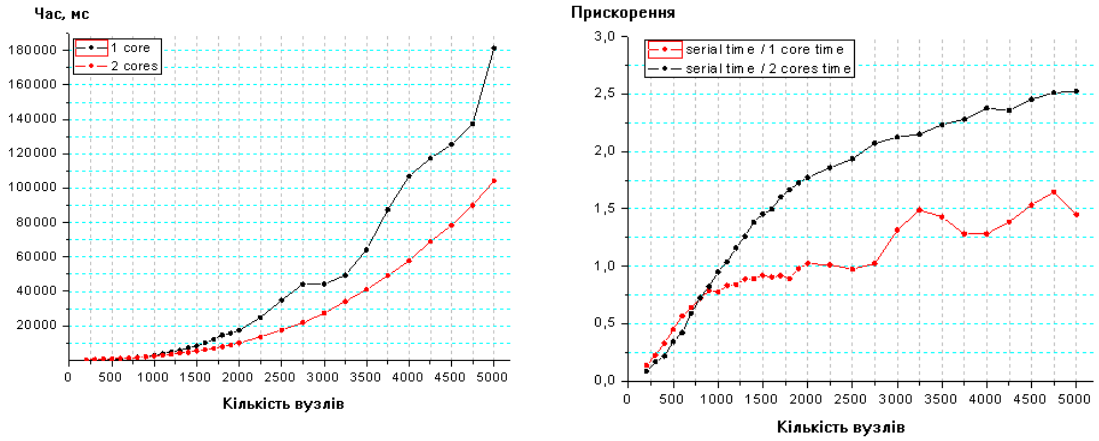

Рис. 11. Час виконання та прискорення відносно послідовної версії алгоритму паралельної реалізації (3 вузла мережі, 1-2 використаних ядра) для систем із розподіленою пам'яттю

Рис. 11 зображує отримані результати для трьох вузлів мережі. Відсутність поліноміальної залежності між часом виконання і кількістю вузлів графа пов'язана з мож-

ливими затримками передачі даних по мережі або з самою топологією мережі. Отже, можна зробити висновок, що додавання нового вузла обчислень не дає суттєвої переваги у швидкодії, проте додавання нового ядра на кожному вузлі значно підвищує продуктивність застосування.

11. Висновки

Дослідження роботи паралельних реалізацій алгоритму Флойда-Уоршала для систем зі спільною та розподіленою пам'яттю виявило ряд особливостей:

1. Прискорення низки потоків алгоритму (для SMP- та MPP-архітектур) доцільно рахувати по відношенню до часу виконання одного потоку на відповідній архітектурі.

2. Для систем SMP-архітектури (рис. 7, 8) передуючий викид і подальший спад прискорення паралельної реалізації пояснюються обмеженістю об'єму кеш-пам'яті системи.

3. Збільшення кількості вузлів понад два для систем MPP-архітектури (рис. 10, 11) не дає суттєвої переваги в часі виконання паралельної версії алгоритму.

4. Системи зі спільною пам'яттю доцільно використовувати для графів із кількістю вузлів, меншою певної величини (для даної серії експериментів приблизно 2500). При перевищенні цієї межі реалізація для систем із розподіленою пам'яттю буде давати кращі результати.

СПИСОК ЛІТЕРАТУРИ

1. Кормен Т. Алгоритмы, построение и анализ / Кормен Т., Лейзерсон Ч., Ривест Р.; пер. з англ. И.В. Красикова, Н.А. Ореховой, В.Н. Романова; под ред. И.В. Красикова. – М.: Издательский дом «Вильямс», 2005. – С. 719 – 725.
2. Таненбаум Э. Архитектура комп'ютера / Таненбаум Э. – [5-е изд.]. – СПб.: Питер, 2007. – С. 634 – 667.
3. Джеффри Р. Windows via C/C++. Программирование на языке Visual C++ / Рихтер Джеффри, Назар Кристоф; пер. с англ. – М.: Издательство "Русская Редакция"; СПб.: Питер, 2008. – 896 с.
4. Харт М.Дж. Системное программирование в среде Windows / Харт М. Дж.; пер. с англ. – [3-е изд.]. – М.: Издательский дом «Вильямс», 2005. – С. 200 – 309.
5. Богачёв К.Ю. Основы параллельного программирования / Богачёв К.Ю. – М.: БИНОМ. Лаборатория знаний, 2003. – С. 232 – 292.
6. Syme D. Expert F# / Syme D., Granicz A., Cisternino A. – Berkeley: Apress, 2007. – 639 p.
7. Погорілий С.Д. Формування узагальнених паралельних схем алгоритму Флойда-Уоршала / С.Д. Погорілий, В.А. Мар'яновський, Ю.В. Бойко [та ін.] // Системні дослідження та інформаційні технології. – 2010. – № 1. – С. 52 – 69.

Стаття надійшла до редакції 22.04.2011