Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів
Назва теми

Рівень вищої освіти *другий (магістерський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ. 240103.24.01.18 ПЗ

Виконав здобувач II курсу, група КІ2м-24-1 — Підпис — Анатолій БАРАБАШ (Ініціали, прізвище)

Керівник канд.-екон. наук, доцент (Науковий ступінь, учене звання) — Підпис — Світлана САЧЕНКО (Ініціали, прізвище)

Нормоконтролер канд. фіз.-мат. наук, доцент (Науковий ступінь, учене звання) — Підпис — Тетяна КИСІЛЬ (Ініціали, прізвище)

До захисту допускаю:
завідувач кафедри КІІС — підпис — Ольга ПАВЛОВА (Ініціали, прізвище)
«01» травня 2026 р.
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ДРУГИЙ (МАГІСТЕРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
_____ Ольга ПАВЛОВА

" 12 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Барабашу Анатолію Леонідовичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів
Керівник проекту (роботи) Світлана Саченко, к.е.н., доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 12.01.2026 р. № 6

2. Термін подання здобувачем роботи на кафедру 01.05.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз відомих методів та засобів синтезу кіберфізична система на основі децентралізованого прийняття рішень
Архітектура та принципи синтезу кіберфізичних систем
Організація прийняття рішень в кфс згідно принципу децентралізації
Метод синтезу та реалізація кіберфізичних систем на основі децентралізованого прийняття рішень

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Тетяна КИСІЛЬ, доцент кафедри КІІС		
Антиплагіат	Андрій НІЧЕПОРУК, доцент кафедри КІІС		

7. Дата видачі завдання «_12_»_____01_____2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проєкту (роботи)	Термін виконання етапів проєкту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	12.01.2026	виконано
2	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	25.02.2026	виконано
3	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	15.03.2026	виконано
4	Робота над науковою статтею	01.04.2026	виконано
5	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	10.04.2026	виконано
6	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	25.04.2026	виконано
7	Оформлення пояснювальної записки згідно вимог	25.04.2026	виконано
8	Попередній захист ВКР	30.04.2026	виконано
9	Захист ВКР на засіданні ЕК	травень 2026 року	

Здобувач_____ Анатолій БАРАБАШ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи_____ Світлана САЧЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: «Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів»

Автор роботи: Барабаш Анатолій Леонідович

Керівник роботи: Саченко С. І.

Пояснювальна записка: 78 с., 4 рис., 6 табл., 4 дод., 81 джерело.

КОМП'ЮТЕРНА СИСТЕМА, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ПРОЦЕСИ, РОЗПАРАЛЕЛЮВАННЯ, СИСТЕМНІ ПРОГРАМИ.

Об'єктом дослідження є процес розпаралелювання динамічних послідовних системних програм.

Предметом дослідження є методи та засоби розпаралелювання програм на основі мереж багатогранних процесів.

Метою кваліфікаційної роботи магістра є підвищення ефективності виконання динамічних послідовних системних програм шляхом розроблення їх розпаралелювання на основі використання мереж багатогранних процесів.

Для розв'язання поставлених задач використовувалися методи забезпечення централізації в архітектурі систем, методи синтезу кіберфізичних систем, теорія множин.

Наукова новизна отриманих результатів: удосконалено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів, особливістю якого є представлення структури програми у вигляді мережі взаємодіючих процесів, що дозволяє виявляти незалежні обчислювальні фрагменти та організовувати їх паралельне виконання. На основі проведених досліджень розроблено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів.

Практична значимість отриманих результатів полягає у можливості підвищення ефективності виконання системних програм шляхом їх адаптації до паралельного виконання на сучасних багатоядерних обчислювальних системах.

Для розв'язання поставлених задач використовувалися методи аналізу та

оптимізації програм, методи організації паралельних обчислень, теорія графів, теорія

У вступі подано об’єкт та предмет дослідження, мету, наукову новизну та практичну цінність роботи, а також характеристику структури роботи.

У першому розділі проведено аналіз відомих рішень щодо розпаралелювання динамічних послідовних системних програм.

У другому розділі здійснено дослідження предметної області та запропоновано моделі типових класів системних програм для розпаралелювання процесів.

У третьому розділі розроблено розпаралелювання динамічних послідовних системних програм.

У четвертому розділі здійснено розроблення реалізації методу розпаралелювання динамічних послідовних системних програм мовою C++, а також проведено експеримент та оцінено ефективність прийнятих рішень.

У висновках підведено підсумки досягнення результатів з розв’язання завдань дослідження.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ІТ – Інформаційні технології

КС – Комп`ютерні системи

ПЗ – Програмне забезпечення

ВСТУП

Розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів є актуальним напрямом досліджень у галузі паралельних обчислень та оптимізації програмного забезпечення. Такий вибір обумовлений сучасними тенденціями розвитку обчислювальної техніки. Сьогодні більшість комп'ютерних систем базується на багатоядерних процесорах і розподілених обчислювальних середовищах, що створює значний потенціал для паралельного виконання програм. Однак значна частина існуючого програмного забезпечення, зокрема системного рівня, була розроблена як послідовна, тобто орієнтована на виконання інструкцій у межах одного потоку. У результаті апаратні можливості сучасних систем використовуються не повною мірою, що знижує загальну ефективність обчислень. Саме тому виникає необхідність дослідження методів, які дозволяють перетворювати послідовні програми на паралельні без кардинальної зміни їх структури або повного переписування програмного коду.

Актуальність роботи особливо проявляється у випадку динамічних системних програм, робота яких залежить від стану системи, оброблюваних даних і умов виконання. На відміну від статичних алгоритмів, де структура обчислень визначена заздалегідь, динамічні програми формують потік виконання під час роботи. Наявність умовних переходів, складних структур даних та змінних залежностей між операціями значно ускладнює процес автоматичного розпаралелювання. У таких умовах традиційні методи оптимізації часто виявляються недостатньо ефективними, що обґрунтовує необхідність пошуку нових підходів до організації паралельних обчислень.

Одним із перспективних підходів є використання моделі мереж багатогранних процесів для представлення структури програми. У межах цього підходу послідовна програма розглядається як система взаємодіючих процесів, кожен з яких виконує окрему частину обчислень і обмінюється даними з іншими компонентами. Така модель дозволяє представити програму у вигляді графа залежностей, що значно полегшує аналіз структури обчислень та виявлення

фрагментів, які можуть виконуватися паралельно. Багатогранний процес при цьому виступає універсальним обчислювальним елементом, здатним одночасно виконувати обчислювальні операції, здійснювати обмін даними та забезпечувати синхронізацію з іншими процесами.

Використання мережі багатогранних процесів створює можливість формалізувати структуру складних програмних систем і здійснювати їх подальшу оптимізацію. Представлення програми у вигляді взаємопов'язаних процесів дозволяє визначити незалежні обчислювальні ділянки та організувати їх паралельне виконання. Це, у свою чергу, сприяє підвищенню продуктивності програмного забезпечення та більш ефективному використанню обчислювальних ресурсів сучасних багатоядерних і розподілених систем.

Отже, наявна необхідність підвищення ефективності виконання програм в умовах широкого використання паралельних обчислювальних архітектур. Дослідження методів розпаралелювання динамічних послідовних системних програм із застосуванням мереж багатогранних процесів дозволяє розробити підходи до перетворення традиційних програмних систем у більш ефективні паралельні структури. Це сприятиме покращенню продуктивності програм, підвищенню масштабованості програмного забезпечення та більш повному використанню обчислювальних можливостей сучасних комп'ютерних систем.

Метою кваліфікаційної роботи магістра є підвищення ефективності виконання динамічних послідовних системних програм шляхом розроблення їх розпаралелювання на основі використання мереж багатогранних процесів.

Поставлена мета досягається розв'язанням таких основних завдань:

1) проаналізувати існуючі методи розпаралелювання послідовних програм та підходи до організації паралельних обчислень;

2) дослідити особливості динамічних послідовних системних програм та визначити проблеми їх ефективного розпаралелювання;

3) розробити метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів;

4) розробити модель представлення послідовної програми у вигляді мережі

взаємодіючих процесів та визначити механізми їх взаємодії і синхронізації;

5) дослідити ефективність запропонованого методу для підвищення продуктивності виконання програм на багатоядерних обчислювальних системах.

Об’єктом дослідження є процес розпаралелювання динамічних послідовних системних програм.

Предметом дослідження є методи та засоби розпаралелювання програм на основі мереж багатогранних процесів.

Наукова новизна отриманих результатів:
удосконалено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів, особливістю якого є представлення структури програми у вигляді мережі взаємодіючих процесів, що дозволяє виявляти незалежні обчислювальні фрагменти та організовувати їх паралельне виконання.

На основі проведених досліджень розроблено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів.

Практична значимість отриманих результатів полягає у можливості підвищення ефективності виконання системних програм шляхом їх адаптації до паралельного виконання на сучасних багатоядерних обчислювальних системах.

Для розв’язання поставлених задач використовувалися методи аналізу та оптимізації програм, методи організації паралельних обчислень, теорія графів, теорія множин та методи моделювання програмних систем.

За темою кваліфікаційної роботи опубліковано одну статтю [81] у фаховому науковому журналі категорії Б.

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ТА ЗАСОБІВ РОЗПАРАЛЕЛЮВАННЯ ДИНАМІЧНИХ ПОСЛІДОВНИХ СИСТЕМНИХ ПРОГРАМ

1.1 Огляд та поняття розпаралелювання динамічних послідовних системних програм

Розпаралелювання програм [1, 2] є одним із ключових напрямів розвитку сучасних обчислювальних систем і програмного забезпечення. Стрімкий розвиток інформаційних технологій, зростання обсягів оброблюваних даних та підвищення вимог до швидкодії програмних систем зумовлюють необхідність пошуку нових підходів до організації обчислювальних процесів. У сучасних комп’ютерних системах широко використовуються багатоядерні процесори, графічні обчислювальні пристрої, кластери та розподілені обчислювальні платформи, які здатні виконувати велику кількість операцій одночасно. Проте ефективне використання таких апаратних можливостей безпосередньо залежить від того, наскільки програмне забезпечення здатне підтримувати паралельне виконання обчислень.

Історично значна частина програмного забезпечення [3, 4] створювалася для однопроцесорних систем, де виконання програм відбувалося у послідовному режимі. У таких програмах усі інструкції виконуються одна за одною в межах одного потоку керування, а кожна наступна операція може розпочатися лише після завершення попередньої. Така модель програмування є відносно простою для реалізації та аналізу, однак вона не дозволяє повною мірою використовувати потенціал сучасних багатоядерних обчислювальних систем. У результаті навіть за наявності значної кількості обчислювальних ресурсів програма може використовувати лише невелику їх частину, що призводить до зниження ефективності виконання та збільшення часу обробки даних.

Саме тому одним із важливих напрямів [5, 6] сучасних досліджень у галузі комп’ютерних наук є розпаралелювання програм. Під розпаралелюванням розуміють процес перетворення послідовної програми або алгоритму на таку

форму, у якій окремі частини обчислень можуть виконуватися одночасно. Основна мета цього процесу полягає у підвищенні продуктивності виконання програм за рахунок одночасного використання декількох обчислювальних ресурсів. Це може реалізовуватися шляхом багатопотокового виконання на одному процесорі, розподілення задач між кількома ядрами процесора або використанням розподілених обчислювальних середовищ.

Процес розпаралелювання [7, 8] передбачає аналіз структури програми з метою виявлення незалежних або частково незалежних фрагментів обчислень. Якщо окремі операції або блоки коду не мають взаємних залежностей, вони можуть виконуватися паралельно без порушення логіки роботи програми. Таким чином, завданням розпаралелювання є визначення таких ділянок програми та організація їх паралельного виконання з урахуванням можливих обмежень і залежностей.

Особливу роль у цьому процесі відіграє аналіз залежностей між операціями [9, 10]. У програмі можуть існувати різні типи залежностей, зокрема залежності за даними, керуванням або ресурсами. Залежність за даними виникає у випадку, коли одна операція використовує результат, отриманий іншою операцією. У такій ситуації порядок виконання операцій не може бути змінений без порушення коректності результатів. Якщо ж операції не залежать одна від одної, вони можуть виконуватися одночасно на різних обчислювальних ресурсах. Аналіз таких залежностей є одним із ключових етапів процесу розпаралелювання програм [11, 12].

Одним із ключових етапів розпаралелювання програм є аналіз залежностей між операціями [13, 14]. Саме залежності визначають, спроможність окремих обчислювальних операцій виконуватися одночасно, або вони повинні виконуватися у строго визначеній послідовності. Якщо між операціями існує залежність, то порушення порядку їх виконання може призвести до некоректних результатів роботи програми. Тому перед організацією паралельного виконання необхідно провести детальний аналіз усіх можливих залежностей у програмному коді.

У загальному випадку [15, 16] залежність між операціями виникає тоді, коли

виконання однієї операції впливає на результати іншої. Найчастіше такі залежності пов’язані з використанням спільних даних або змінних. Якщо одна операція змінює значення певної змінної, а інша операція використовує це значення, то порядок виконання цих операцій стає важливим. У цьому випадку їх не можна виконувати паралельно без додаткових механізмів узгодження.

У теорії розпаралелювання [17, 18] програм розрізняють декілька основних типів залежностей між операціями. Найбільш важливими є залежності за даними, залежності за керуванням та залежності за ресурсами.

Найпоширенішими є залежності за даними [19, 20]. Вони виникають у ситуаціях, коли декілька операцій працюють із одними й тими самими даними. У цьому випадку порядок виконання операцій визначається тим, як саме ці дані використовуються. Існує кілька підвидів залежностей за даними.

Перший тип це істинна залежність або залежність типу «читання після запису» [21, 22]. Вона виникає тоді, коли одна операція записує значення змінної, а наступна операція читає це значення. Наприклад, якщо одна операція обчислює результат і записує його у змінну, а інша використовує цей результат для подальших обчислень, то друга операція не може виконуватися раніше за першу. У такому випадку між операціями існує прямий інформаційний зв’язок, і їх порядок виконання повинен бути збережений.

Другий тип [23, 24] це залежність типу «запис після читання». Вона виникає тоді, коли одна операція читає значення змінної, а інша змінює її значення. Якщо друга операція виконається раніше, ніж перша завершить читання даних, то перша операція може отримати вже змінене значення, що призведе до помилки. Тому у такій ситуації також необхідно зберігати правильний порядок виконання.

Третій тип [25, 26] це залежність типу «запис після запису». Вона виникає тоді, коли дві операції записують значення в одну й ту саму змінну. Якщо порядок їх виконання зміниться, кінцеве значення змінної може відрізнятися від очікуваного результату. Саме тому такі операції також потребують узгодженого порядку виконання.

Особливу складність для розпаралелювання становлять залежності, що

виникають у циклах [27, 28]. У багатьох алгоритмах значення, обчислене на одній ітерації циклу, використовується на наступній ітерації. У такому випадку між ітераціями виникає так звана міжітераційна залежність. Наприклад, якщо значення елемента масиву обчислюється на основі попереднього елемента, то виконання ітерацій циклу у довільному порядку або паралельно може призвести до некоректних результатів. Тому перед розпаралелюванням циклів необхідно перевіряти наявність таких залежностей.

Крім залежностей за даними, важливу роль відіграють залежності за керуванням [29, 30]. Вони пов’язані зі структурою керування виконанням програми. Наприклад, у програмі можуть бути умовні оператори, які визначають, чи буде виконуватися певний фрагмент коду. У такій ситуації виконання деяких операцій залежить від результатів попередніх обчислень або перевірок умов. Це означає, що порядок виконання операцій визначається не лише потоками даних, але й логікою керування програмою.

Ще одним типом залежностей є залежності за ресурсами [31, 32]. Вони виникають у ситуаціях, коли декілька операцій використовують один і той самий ресурс, наприклад оперативну пам’ять, файли, пристрої введення-виведення або інші системні ресурси. Якщо декілька процесів намагаються одночасно змінювати один і той самий ресурс, може виникнути конфлікт доступу. Для уникнення таких ситуацій використовуються механізми синхронізації, наприклад блокування, семафори або монітори.

Під час організації паралельного виконання програм важливо також враховувати так звані потенційні або приховані залежності [33]. У складних програмних системах залежності можуть виникати не лише на рівні окремих операцій, але й між функціями, модулями або процесами. Наприклад, одна функція може змінювати глобальні змінні або структури даних, які використовуються іншими частинами програми. У таких випадках аналіз залежностей ускладнюється, оскільки необхідно враховувати взаємодію між різними компонентами програмної системи.

Аналіз залежностей є важливою складовою процесу автоматичного

розпаралелювання програм [34]. Багато сучасних компіляторів та систем оптимізації використовують спеціальні алгоритми для виявлення залежностей між операціями. На основі цього аналізу визначаються ділянки коду, які можуть бути виконані паралельно, а також ті частини програми, які повинні залишатися послідовними.

Водночас для складних динамічних програм повністю автоматичний аналіз залежностей часто є недостатньо ефективним [35]. Це пов’язано з тим, що залежності можуть виникати під час виконання програми і не завжди можуть бути точно визначені на етапі компіляції. У таких випадках необхідно використовувати більш складні моделі представлення програм, які дозволяють описувати взаємодію між обчислювальними компонентами та враховувати динамічні аспекти виконання.

Таким чином, залежності між операціями є одним із ключових факторів, що визначають можливість і ефективність розпаралелювання програм [36, 37]. Їх правильне виявлення та аналіз дозволяють визначити потенційний паралелізм у програмному коді та організувати ефективне використання обчислювальних ресурсів. Разом з тим складність аналізу залежностей у динамічних системних програмах формує важливу науково-дослідницьку проблему, пов’язану з розробленням нових методів і моделей, здатних більш точно визначати залежності між операціями та забезпечувати ефективне розпаралелювання складних програмних систем.

Важливо також враховувати залежності за керуванням, які пов’язані з умовними переходами, циклами та іншими елементами керування виконанням програми [38]. У багатьох випадках саме структура керування визначає можливість або неможливість паралельного виконання певних фрагментів коду. Крім того, у програмних системах можуть виникати залежності, пов’язані з доступом до спільних ресурсів, наприклад до пам’яті або файлів. У таких ситуаціях необхідно використовувати механізми синхронізації, які забезпечують узгоджений доступ до ресурсів та запобігають виникненню конфліктів між паралельно виконуваними процесами.

Складність задачі розпаралелювання значно зростає у випадку динамічних послідовних системних програм [39]. На відміну від статичних алгоритмів, у яких структура обчислень відома заздалегідь, динамічні програми можуть змінювати свій потік виконання під час роботи. Це може бути зумовлено використанням умовних операторів, рекурсивних викликів, динамічних структур даних або залежністю від зовнішніх подій. У результаті послідовність виконання операцій у таких програмах визначається не лише програмним кодом, але й поточними умовами виконання.

До динамічних послідовних системних програм належать різноманітні компоненти системного програмного забезпечення [40]. Це можуть бути модулі операційних систем, системні утиліти, компілятори, інтерпретатори мов програмування, системи керування ресурсами, засоби обробки складних структур даних та інші програмні компоненти, що забезпечують функціонування комп’ютерних систем. Такі програми часто працюють у складних умовах, взаємодіють з іншими компонентами системи та обробляють великі обсяги інформації, що робить задачу їх оптимізації та підвищення продуктивності особливо важливою.

Однією з характерних особливостей динамічних програм є використання структур даних, які формуються під час виконання програми [41, 42]. Наприклад, це можуть бути списки, дерева, графи або інші динамічні структури, розмір і структура яких змінюються в процесі роботи. У таких випадках визначити можливості для паралельного виконання обчислень значно складніше, оскільки залежності між операціями можуть виникати або змінюватися під час виконання програми.

Крім того, динамічні програми часто використовують складні механізми керування виконанням [43, 44], такі як вкладені цикли, умовні переходи, обробка переривань або взаємодія з іншими програмними компонентами. У результаті структура обчислювального процесу може бути складною і змінною, що ускладнює застосування традиційних методів автоматичного розпаралелювання. Саме тому розроблення ефективних підходів до розпаралелювання динамічних послідовних

програм є важливим завданням сучасних досліджень у галузі програмної інженерії та високопродуктивних обчислень.

У сучасних наукових дослідженнях пропонується різноманітні підходи до вирішення цієї проблеми [45, 46]. Одним із поширених підходів є представлення програми у вигляді графа обчислень або графа залежностей. У такій моделі вершини графа відповідають окремим обчислювальним операціям або програмним модулям, а ребра описують залежності між ними. Аналіз такого графа дозволяє визначити, які операції можуть виконуватися паралельно, а які повинні виконуватися у визначеній послідовності.

Іншим підходом [47] є представлення програми у вигляді системи взаємодіючих процесів або задач, які обмінюються даними та координують свою роботу. Така модель дозволяє більш гнучко описувати структуру обчислень і враховувати динамічні аспекти виконання програм. Вона також створює можливість для ефективного використання сучасних багатоядерних і розподілених обчислювальних систем.

Незважаючи на значну кількість досліджень у цій галузі, проблема ефективного розпаралелювання динамічних послідовних системних програм залишається актуальною та недостатньо розв’язаною [48]. Основні труднощі пов’язані з необхідністю точного визначення залежностей між операціями, забезпечення коректності виконання програми після її розпаралелювання, а також мінімізації витрат, пов’язаних із синхронізацією та обміном даними між паралельно виконуваними процесами.

Крім того, важливою проблемою є розроблення універсальних моделей і методів [49], які дозволяли б ефективно виконувати розпаралелювання складних програм із динамічною структурою виконання. Такі методи повинні враховувати особливості сучасних обчислювальних архітектур, забезпечувати масштабованість програмного забезпечення та зберігати коректність його функціонування.

Отже, аналіз існуючих підходів до розпаралелювання програм свідчить про необхідність здійснення подальших досліджень у цьому напрямі. Особливо актуальною є проблема створення ефективних методів розпаралелювання

динамічних послідовних системних програм, які дозволять автоматизувати процес виявлення паралелізму в програмному коді та організувати його ефективне використання. Розв’язання цієї науково-дослідницької проблеми передбачає розроблення нових моделей представлення програм, методів аналізу залежностей між обчисленнями та підходів до організації взаємодії між паралельними процесами. Саме ці питання становлять основу подальших досліджень у межах даної роботи та визначають її наукову спрямованість.

1.2 Відомі методи розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів

У сучасних обчислювальних системах проблема ефективного використання обчислювальних ресурсів є однією з ключових у галузі інженерії та високопродуктивних обчислень [51, 52].

З розвитком багатоядерних процесорів, кластерних систем та розподілених обчислювальних платформ зростає потреба у програмних методах, здатних забезпечити ефективне паралельне виконання програм. Проте значна частина існуючого програмного забезпечення, особливо системного рівня, створювалася як послідовна, тобто орієнтована на виконання інструкцій у межах одного потоку керування. У таких програмах усі обчислення виконуються у визначеній послідовності, що значно обмежує можливість використання потенціалу сучасних багатопроцесорних систем. Саме тому важливим напрямом досліджень є розроблення та вдосконалення методів розпаралелювання динамічних послідовних системних програм [53, 54].

Особливу складність становить розпаралелювання динамічних програм, оскільки їх структура виконання не є фіксованою. У таких програмах порядок виконання операцій може змінюватися під час роботи залежно від оброблюваних даних, умов виконання або взаємодії з іншими компонентами програмного середовища. Наявність умовних переходів, складних структур даних, рекурсивних викликів та інших динамічних елементів значно ускладнює аналіз залежностей між

операціями та виявлення можливостей для паралельного виконання. Для розв’язання цієї проблеми у сучасних дослідженнях використовуються різні підходи до розпаралелювання програм, серед яких особливе місце займають методи, що базуються на використанні мереж багатогранних процесів [55, 56].

Одним із відомих підходів є метод представлення програми у вигляді графа обчислювальних залежностей. У межах цього підходу послідовна програма аналізується з метою виявлення залежностей між окремими операціями або блоками коду. На основі отриманої інформації будується граф, у якому вершини відповідають обчислювальним операціям або функціональним блокам, а ребра відображають залежності між ними. Такий граф дозволяє наочно представити структуру обчислень та визначити ділянки програми, які можуть виконуватися паралельно. Використання мережі процесів у цьому випадку дозволяє організувати виконання окремих обчислювальних блоків у вигляді взаємодіючих процесів, кожен з яких виконує певну частину алгоритму та передає результати іншим процесам відповідно до структури графа залежностей [57, 58].

Іншим підходом є метод декомпозиції програми на незалежні функціональні модулі або задачі. У цьому випадку вихідна програма розбивається на окремі обчислювальні компоненти, які можуть виконуватися незалежно або з мінімальною взаємодією. Кожен із таких компонентів представляється у вигляді окремого процесу, що взаємодіє з іншими процесами через канали передачі даних. У рамках мережі багатогранних процесів такі компоненти можуть виконувати різні ролі, наприклад виконувати обчислення, здійснювати обмін даними або забезпечувати координацію роботи інших процесів. Такий підхід дозволяє організувати гнучку структуру паралельних обчислень, у якій кожен процес може взаємодіяти з кількома іншими процесами одночасно [59, 60].

Важливим методом розпаралелювання є також метод потоків даних. У цій моделі виконання програми визначається не фіксованою послідовністю інструкцій, а наявністю необхідних даних для виконання певної операції. Іншими словами, обчислювальна операція може розпочатися одразу після того, як усі необхідні для неї дані стануть доступними. Такий підхід добре узгоджується з моделлю мереж

багатогранних процесів, оскільки кожен процес у мережі може активуватися при надходженні відповідних даних від інших процесів. Це дозволяє ефективно використовувати можливості паралельного виконання та зменшувати затримки, пов’язані з очікуванням завершення інших операцій [61, 62].

Ще одним важливим підходом є метод паралелізації циклів, який широко застосовується у задачах обробки масивів даних та чисельних обчисленнях. У багатьох алгоритмах основний обсяг обчислень виконується всередині циклів, де одні й ті самі операції застосовуються до різних елементів даних. Якщо між ітераціями циклу відсутні залежності за даними, такі ітерації можуть виконуватися паралельно. У контексті мереж багатогранних процесів кожна ітерація циклу може розглядатися як окремий процес або задача, що виконується незалежно від інших. Це дозволяє розподіляти обчислювальне навантаження між кількома процесорами або обчислювальними вузлами [63, 64].

Також у сучасних дослідженнях використовується метод динамічного розподілу задач. У цьому підході задачі не розподіляються між процесорами заздалегідь, а призначаються під час виконання програми залежно від поточного стану системи та доступності обчислювальних ресурсів. Такий метод дозволяє більш ефективно використовувати ресурси системи та адаптуватися до змін у структурі обчислень. У мережі багатогранних процесів динамічний розподіл задач може реалізовуватися через спеціальні керуючі процеси, які відповідають за розподіл обчислювального навантаження між іншими процесами [65].

Крім того, важливу роль відіграє метод конвеєрної обробки. У цьому випадку виконання програми організовується у вигляді послідовності етапів, кожен з яких виконує певну частину обчислень. Результат роботи одного етапу передається наступному етапу, який продовжує обробку даних. У контексті мереж багатогранних процесів кожен етап конвеєра може бути реалізований як окремий процес, що взаємодіє з іншими процесами через канали передачі даних. Така організація обчислень дозволяє одночасно виконувати різні етапи обробки для різних наборів даних, що значно підвищує загальну продуктивність системи [66].

Ще одним напрямом є використання акторних моделей обчислень. У таких

моделях програма розглядається як система незалежних акторів, кожен з яких має власний стан і може обмінюватися повідомленнями з іншими акторами. Актори виконуються паралельно і взаємодіють один з одним через передачу повідомлень, що дозволяє уникнути багатьох проблем, пов’язаних із синхронізацією доступу до спільних ресурсів. У рамках мереж багатогранних процесів акторна модель може використовуватися для організації взаємодії між процесами та забезпечення гнучкої структури паралельних обчислень [67].

Незважаючи на значну кількість існуючих методів розпаралелювання програм, їх застосування до динамічних системних програм пов’язане з низкою складних проблем. Однією з основних є необхідність точного визначення залежностей між операціями, які можуть змінюватися під час виконання програми. Крім того, важливо забезпечити коректність виконання програми після її розпаралелювання, що вимагає використання ефективних механізмів синхронізації та координації між процесами. Ще однією проблемою є мінімізація витрат, пов’язаних із передачею даних та керуванням взаємодією між паралельно виконуваними процесами [68].

Мережі багатогранних процесів є однією з концептуальних моделей організації паралельних обчислень, яка використовується для представлення складних програмних систем у вигляді взаємодіючих обчислювальних компонентів. Така модель дозволяє формалізувати структуру програмного виконання, описати взаємодію між окремими обчислювальними елементами та створити основу для ефективного розпаралелювання програм. Використання мереж багатогранних процесів особливо актуальне при роботі з динамічними системними програмами, у яких структура обчислень може змінюватися під час виконання [69].

У загальному випадку мережа процесів розглядається як система взаємопов’язаних обчислювальних елементів, які обмінюються даними та координують свою роботу для досягнення спільної мети. Кожен елемент такої мережі виконує певну частину обчислень і взаємодіє з іншими елементами через канали передачі даних або механізми синхронізації. Мережа процесів може бути

представлена у вигляді графа, у якому вершини відповідають окремим процесам або обчислювальним блокам, а ребра відображають зв’язки між ними. Така форма представлення дозволяє наочно описати структуру обчислень і визначити залежності між різними частинами програми [70].

Поняття багатогранного процесу пов’язане з тим, що один обчислювальний елемент у межах мережі може виконувати кілька функціональних ролей одночасно. Тобто процес може не лише виконувати обчислення, але й здійснювати передачу даних іншим процесам, координувати їхню роботу, обробляти сигнали керування або забезпечувати синхронізацію. Саме така багатофункціональність і визначає поняття «багатогранності» процесу. У результаті кожен процес у мережі виступає як універсальний елемент обчислювальної системи, який може виконувати різні задачі залежно від поточного стану обчислювального процесу [71].

Мережа багатогранних процесів дозволяє описувати складні програмні системи у вигляді взаємодіючих компонентів, що виконують різні функції. Наприклад, одні процеси можуть відповідати за виконання обчислювальних операцій, інші відповідають за передачу та обробку даних, а ще інші відповідають за керування виконанням програми. Така структура створює можливість для гнучкої організації паралельних обчислень, оскільки кожен процес може працювати незалежно від інших або взаємодіяти з ними лише за необхідності [72].

Однією з важливих особливостей мереж багатогранних процесів є можливість представлення програмної системи у вигляді ієрархічної структури. У такій структурі окремі процеси можуть об’єднуватися у підмережі або групи, що виконують певні функції. Наприклад, одна група процесів може відповідати за обробку вхідних даних, інша відповідає за виконання основних обчислень, а третя відповідає за формування та передачу результатів. Такий підхід дозволяє спростити аналіз структури програми та полегшити процес її оптимізації [73].

Ще однією важливою характеристикою мереж багатогранних процесів є можливість динамічної зміни структури обчислень. У традиційних послідовних програмах порядок виконання операцій визначається заздалегідь і змінюється лише в межах логіки програми. У мережах процесів структура взаємодії між елементами

може змінюватися під час виконання. Наприклад, у процесі роботи можуть створюватися нові процеси, змінюватися канали передачі даних або перерозподілятися обчислювальні ресурси. Така гнучкість є особливо важливою для динамічних системних програм, де структура обчислень залежить від поточних умов виконання [74].

Використання мереж багатогранних процесів також дозволяє ефективно організувати паралельне виконання програм. Оскільки кожен процес у мережі є відносно незалежним обчислювальним елементом, декілька процесів можуть виконуватися одночасно на різних обчислювальних ресурсах. Це створює можливість для значного підвищення продуктивності програмного забезпечення за рахунок одночасного виконання великої кількості обчислювальних операцій. Крім того, така організація обчислень дозволяє більш ефективно використовувати можливості сучасних багатоядерних процесорів і розподілених обчислювальних систем [75].

У межах мережі багатогранних процесів важливу роль відіграють механізми взаємодії між процесами. Такі механізми можуть включати передачу повідомлень, обмін даними через спільну пам’ять або використання спеціальних каналів зв’язку. Вибір конкретного механізму взаємодії залежить від архітектури обчислювальної системи та особливостей програмної задачі. У будь-якому випадку основним завданням таких механізмів є забезпечення узгодженої роботи всіх процесів у межах мережі та запобігання конфліктам доступу до спільних ресурсів [76].

Важливим аспектом функціонування мереж багатогранних процесів є також питання синхронізації. Оскільки різні процеси можуть виконуватися одночасно, необхідно забезпечити правильний порядок виконання тих операцій, між якими існують залежності. Для цього використовуються різні механізми синхронізації, такі як сигнали, блокування або спеціальні протоколи обміну повідомленнями. Правильна організація синхронізації дозволяє забезпечити коректність виконання програми та уникнути помилок, пов’язаних із некоректним доступом до даних [77].

Таким чином, мережі багатогранних процесів є ефективною моделлю представлення паралельних обчислювальних систем. Вони дозволяють

формалізувати структуру складних програм, описати взаємодію між обчислювальними компонентами та створити основу для організації паралельного виконання програм. Використання цієї моделі відкриває можливості для підвищення продуктивності програмного забезпечення, оптимізації використання обчислювальних ресурсів та створення масштабованих програмних систем, здатних ефективно працювати у сучасних багатоядерних і розподілених обчислювальних середовищах [78].

Таким чином, відомі методи розпаралелювання динамічних послідовних системних програм включають різні підходи до аналізу структури програм [79, 80], декомпозиції обчислень та організації взаємодії між процесами. Використання мереж багатогранних процесів дозволяє представити складні програмні системи у вигляді структурованих моделей взаємодіючих обчислювальних компонентів, що відкриває можливості для ефективного використання паралельних обчислювальних ресурсів. Разом з тим складність динамічних програм та наявність численних залежностей між операціями зумовлюють необхідність подальших досліджень у цьому напрямі, спрямованих на розроблення нових методів і моделей, здатних забезпечити більш ефективне розпаралелювання складних системних програм.

1.3 Постановка задачі

Поставлена мета досягається розв’язанням таких основних завдань:

1) проаналізувати існуючі методи розпаралелювання послідовних програм та підходи до організації паралельних обчислень;

2) дослідити особливості динамічних послідовних системних програм та визначити проблеми їх ефективного розпаралелювання;

3) розробити метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів;

4) розробити модель представлення послідовної програми у вигляді мережі взаємодіючих процесів та визначити механізми їх взаємодії і синхронізації;

5) дослідити ефективність запропонованого методу для підвищення продуктивності виконання програм на багатоядерних обчислювальних системах.

1.4 Висновки до першого розділу

Проаналізовано відомі методи та засоби розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів, а також визначено стратегію для покращення ефективності цього процесу.

2 МОДЕЛЮВАННЯ СИСТЕМНИХ ПРОГРАМ ДЛЯ РОЗПАРАЛЕЛЮВАННЯ ПРОЦЕСІВ ПРИ ВИКОНАННІ

2.1 Визначення області дослідження

Розроблення нових алгоритмічних підходів, які здатні автоматично виявляти ділянки послідовного програмного коду, що можуть бути перетворені на незалежні паралельні потоки виконання залишається актуальним напрямом досліджень. Основна його ідея полягає у створенні інтелектуальних методів аналізу програм, що дозволяють без безпосереднього втручання розробника визначати фрагменти коду, які придатні для паралельного виконання, та автоматично трансформувати їх у структури, оптимізовані для багатоядерних обчислювальних систем. У сучасних обчислювальних середовищах, де широко використовуються процесори з кількома ядрами, така модифікована програма здатна виконуватися одночасно на різних обчислювальних ресурсах, що забезпечує істотне зростання продуктивності та ефективності використання апаратного забезпечення. В основі такого дослідження використовуємо припущення, що значна частина послідовного коду потенційно містить прихований паралелізм, який можна виявити за допомогою формального аналізу залежностей між інструкціями та змінними, тобто якщо припустити що такий паралелізм може бути наявним, то його опрацювання може надати змогу виконувати програмний код паралельно.

Визначимо три основні категорії конструкцій, які найчастіше можуть бути перетворені на паралельні: виклики функцій; послідовності операторів або так звані шляхи виконання інструкцій; циклічні конструкції. Для кожного з цих типів структур необхідно сформулювати набір формальних умов та передумов, що визначають можливість їх безпечної паралелізації. На основі цих умов розроблятимемо алгоритми, які здійснюють аналіз коду та оцінюють, чи відповідає певний фрагмент програми критеріям паралельного виконання. Особливістю цих алгоритмів є те, що вони не лише визначають можливість паралелізації, але й містять пояснювальні механізми, які дозволяють інтерпретувати отримані результати. Завдяки цьому вони можуть бути використані у практичних задачах

оптимізації програмного забезпечення. Для перевірки ефективності запропонованих підходів застосовуватимемо комбінацію двох методів: статичний аналіз вихідного коду мовою програмування C++; динамічний аналіз продуктивності, що передбачає порівняння часу виконання послідовної та паралельної версій програми.

Також необхідно досліджувати можливості асинхронного виконання викликів функцій. У традиційній послідовній програмі виконання функції призводить до тимчасового призупинення виконання основного потоку до моменту завершення викликаної процедури. Однак у багатьох випадках наступні інструкції програми не залежать від результату виклику функції, що відкриває можливість виконувати ці частини коду паралельно. При аналізі викликів функцій розглядатимемо дві основні послідовності інструкцій: інструкції, що виконуються в момент виклику функції; інструкції, які виконуються після її завершення. Якщо між цими двома частинами відсутні залежності, то виклик функції може бути перетворений на асинхронний. Для визначення такої можливості враховується використання результату функції, змінні, передані за посиланням, а також потенційні побічні ефекти, які можуть виникати під час виконання функції або вкладених викликів. На основі цього аналізу визначається найдовша послідовність інструкцій після виклику функції, яка не залежить від її результату. Це явище описується поняттям відкладеного використання результату. Якщо між викликом функції та її першим відкладеним використанням існує хоча б одна незалежна інструкція, то функцію можна виконувати паралельно з подальшим кодом до моменту виникнення залежності.

Для реалізації алгоритмів аналізу використовується набір допоміжних структур даних. Кожна інструкція програми повинна мати унікальну мітку, що дозволяє однозначно ідентифікувати її в процесі аналізу. Однією з ключових структур є граф викликів, який представляє програму у вигляді орієнтованого графа, де вузли відповідають функціям, а ребра відображають факти виклику одних функцій іншими. У практичних програмах такий граф часто має структуру мультиграфа, оскільки між одними й тими самими функціями може існувати кілька

викликів. Для алгоритмічних цілей кожне ребро маркується номером виклику, що дозволяє точно визначити контекст його використання. Крім того, вводяться множини інструкцій із побічними ефектами та множини інструкцій, залежних від цих ефектів. Інструкції з побічними ефектами можуть змінювати значення змінних поза локальною областю видимості або створювати зовнішній результат виконання програми, наприклад, виведення даних. Натомість залежні інструкції можуть використовувати змінні, значення яких потенційно було змінено іншими операціями.

Алгоритм визначення можливості асинхронного виклику починається зі статичного аналізу та побудови графа викликів. У процесі цього аналізу враховуються лише ті бібліотечні функції, які вважаються потокобезпечними згідно з документацією середовища виконання. Після побудови графа функції класифікуються за рівнями залежно від їхнього місця у структурі викликів. Найнижчий рівень складається з функцій, які не викликають інших процедур, тоді як наступні рівні формуються з функцій, що викликають функції попереднього рівня. Функції, які входять до циклічних залежностей у графі викликів, зазвичай не можуть бути однозначно віднесені до певного рівня. У процесі аналізу також враховується вплив побічних ефектів викликаних функцій на викликаючі інструкції. Якщо функція викликає бібліотечну процедуру, можливі побічні ефекти цієї процедури додаються до множини відповідних інструкцій. Навіть якщо явних побічних ефектів не спостерігається, використання змінних, переданих за посиланням, може призводити до їх включення до множин залежностей.

Окрім статичного аналізу, важливу роль відіграє динамічна оцінка доцільності паралелізації. Навіть якщо певна структура може бути виконана паралельно з теоретичної точки зору, накладні витрати на створення та керування потоками можуть перевищувати виграш у продуктивності. Для оцінювання використовуються часові мітки, які фіксують початок виконання функції, її завершення та момент першого використання результату. На основі кількох запусків програми обчислюються середні інтервали часу між цими подіями. Крім того, визначається середній час створення нового потоку. Паралельне виконання

вважається доцільним тоді, коли тривалість виконання функції та інтервал до використання її результату значно перевищують час створення потоку, а самі ці інтервали є приблизно однаковими.

Окрім викликів функцій, потенційним джерелом паралелізму є незалежні шляхи виконання інструкцій усередині однієї функції. Стан програми можна описати як певну множину значень змінних, і виконання кожної інструкції можна розглядати як функцію, що перетворює цей стан. Якщо зміна порядку виконання двох інструкцій призводить до різних результатів, такі інструкції вважаються залежними. У протилежному випадку вони можуть виконуватися незалежно. Послідовність взаємопов’язаних інструкцій формує шлях виконання, а кілька незалежних шляхів можуть бути виконані одночасно. Для виявлення таких шляхів будується граф незалежності, вузлами якого є інструкції та змінні, а ребра відображають залежності між ними. Якщо граф розпадається на кілька окремих компонентів, кожен із них відповідає окремому незалежному шляху виконання. Після завершення кожного шляху необхідно виконати синхронізацію потоків, щоб гарантувати правильність подальшого виконання програми.

Циклічні конструкції розглядаються як одна з найважливіших можливостей для реалізації паралельних обчислень, оскільки вони передбачають багаторазове виконання однакового коду. У більшості мов програмування існують два основні типи циклів: цикли типу for та цикли типу while. Найчастіше паралелізуються саме цикли for, оскільки їх структура дозволяє визначити кількість ітерацій до початку виконання. Для можливості паралелізації необхідно виконання кількох умов: параметри циклу повинні бути відомі до початку виконання, кожна ітерація має бути незалежною від інших, а загальна кількість ітерацій повинна бути визначеною. Під час аналізу залежностей визначаються множини змінних, що читаються та модифікуються у кожній ітерації. Якщо одна ітерація змінює змінну, яка використовується іншою ітерацією, може виникнути так званий стан гонки, що унеможливлює паралельне виконання. Особливу увагу приділяють також вкладеним циклам. У більшості випадків доцільніше паралелізувати зовнішній цикл, оскільки це забезпечує більший рівень паралельності та зменшує витрати на

синхронізацію.

Під час практичної реалізації паралельних алгоритмів важливим аспектом є організація синхронізації потоків. У сучасних програмних середовищах існують спеціальні механізми, що дозволяють координувати завершення потоків перед переходом до наступних інструкцій. У випадку відсутності вбудованих засобів синхронізації можуть використовуватися класичні механізми, такі як семафори або події. Перед створенням паралельних потоків зазвичай встановлюється лічильник активних потоків, який зменшується після завершення кожного з них. Коли всі потоки завершують виконання, програма переходить до наступного етапу.

Методи можуть мати певні обмеження, які пов’язані з особливостями мов програмування та механізмів виконання програм. Наприклад, використання віртуальних функцій ускладнює статичний аналіз, оскільки неможливо однозначно визначити, яка саме реалізація методу буде викликана під час виконання. Подібні труднощі виникають і у випадку складних вкладених викликів функцій або використання винятків. Додаткові обмеження пов’язані з наявністю побічних ефектів, коли кілька інструкцій змінюють один і той самий ресурс. У таких ситуаціях необхідно застосовувати додаткові механізми синхронізації або залишати ці частини коду послідовними.

У мовах програмування, зокрема у C++, існують спеціалізовані бібліотеки, що значно спрощують реалізацію паралельних обчислень. Наприклад, замість ручного керування потоками можна використовувати конструкції бібліотеки Це дозволяє ефективно використовувати можливості багатоядерних процесорів без детального керування потоками. У деяких операційних системах, подібних до Unix, альтернативою потокам можуть бути процеси, створені за допомогою системних викликів fork і wait, хоча такий підхід часто супроводжується більшими витратами ресурсів через необхідність дублювання адресного простору.

Алгоритми можуть комбінуватися між собою ієрархічно, що дозволяє досягати більш глибокого рівня паралелізму. Наприклад, незалежні шляхи виконання можуть містити асинхронні виклики функцій або паралельні цикли. Крім того, один незалежний шлях може бути додатково поділений на підшляхи

залежно від підмножин змінних, до яких звертаються інструкції. Така ієрархічна організація дозволяє більш гнучко використовувати доступні обчислювальні ресурси та поступово підвищувати рівень паралельності виконання програми. У результаті розроблені підходи створюють основу для автоматизованих систем оптимізації програмного коду, здатних адаптувати традиційні послідовні алгоритми до сучасних багатоядерних обчислювальних платформ.

Граничні умови Бернштейна є одним із фундаментальних теоретичних інструментів аналізу залежностей у програмних системах та широко застосовуються під час дослідження можливостей паралельного виконання алгоритмів. У контексті розроблення та оптимізації програмного забезпечення ці умови використовуються для формального визначення того, чи можуть окремі інструкції, блоки коду або функціональні модулі виконуватися одночасно без порушення коректності обчислень. У теорії паралельного програмування вони відіграють важливу роль під час аналізу структур послідовних програм, оскільки дозволяють встановити наявність або відсутність залежностей між операціями, які потенційно можуть бути виконані паралельно. В основі умов Бернштейна лежить аналіз доступу інструкцій до змінних програми, а саме визначення множин змінних, які читаються та змінюються під час виконання кожної інструкції або програмного блоку.

Формалізація умов Бернштейна ґрунтується на поданні програми як послідовності операторів або функціональних блоків, кожен з яких взаємодіє з певною множиною змінних. Для кожної інструкції або обчислювального блоку визначаються дві основні множини: множина змінних, значення яких використовуються під час виконання цієї інструкції; множина змінних, значення яких змінюються в результаті її виконання. Нехай певна інструкція I_i характеризується множиною змінних читання R_i (read set), яка містить усі змінні, значення яких використовуються під час виконання цієї інструкції, а також множиною змінних запису W_i (write set), яка містить усі змінні, значення яких змінюються в процесі виконання інструкції. Аналогічно для іншої інструкції I_j визначаються множини R_j та W_j. Метою аналізу є визначення того, чи можуть

інструкції I_i та I_j виконуватися незалежно одна від одної, тобто без виникнення конфліктів доступу до спільних змінних.

Згідно з умовами Бернштейна, паралельне виконання двох інструкцій або програмних блоків є можливим лише за умови виконання трьох формальних обмежень. Перше обмеження полягає у відсутності конфлікту типу «читання-після-запису», що означає, що жодна змінна, яка змінюється однією інструкцією, не повинна використовуватися для читання іншою інструкцією. Формально це можна записати як $R_i \cap W_j = \emptyset$, де R_i - множина змінних, що читаються інструкцією I_i, а W_j — множина змінних, що змінюються інструкцією I_j. Якщо перетин цих множин не є порожнім, це означає, що інструкція I_i використовує значення змінної, яке може бути змінене інструкцією I_j, що створює залежність між ними і унеможливлює їх паралельне виконання.

Друге обмеження стосується відсутності конфлікту типу «запис-після-читання» і формально виражається умовою $R_j \cap W_i = \emptyset$. Це означає, що інструкція I_j не повинна використовувати змінну, яка модифікується інструкцією I_i, оскільки в іншому випадку порядок виконання інструкцій впливатиме на значення цієї змінної. Така ситуація створює залежність даних між інструкціями, що вимагає їх послідовного виконання для збереження коректності результатів обчислень. Третє обмеження визначає відсутність конфлікту типу «запис-після-запису» і формально задається умовою $W_i \cap W_j = \emptyset$. Воно означає, що обидві інструкції не повинні змінювати одну й ту саму змінну, оскільки одночасний запис у спільну змінну може призвести до некоректного або непередбачуваного результату виконання програми.

Якщо всі три наведені умови виконуються одночасно, інструкції вважаються незалежними, а їх порядок виконання не впливає на кінцевий стан програми. У такому випадку ці інструкції можуть бути виконані паралельно, що відкриває можливість використання багатопотокових або багатопроцесорних механізмів виконання. З формальної точки зору це означає, що виконання інструкцій у різному порядку або одночасно приводить до однакового результату, тобто виконується

рівність $S\{I_i; I_j\}=S\{I_j; I_i\}$, де S позначає початковий стан системи, а вираз $S\{I_i; I_j\}$ означає стан системи після послідовного виконання інструкцій I_i та I_j.

Застосування умов Бернштейна є особливо важливим у процесі автоматичної паралелізації програм, що виконується компіляторами або спеціалізованими інструментами оптимізації. Під час аналізу вихідного коду програмна система визначає множини змінних читання та запису для кожного оператора або блоку інструкцій, після чого перевіряє виконання наведених умов. Якщо конфлікти доступу до змінних відсутні, відповідні інструкції можуть бути розміщені у різних потоках виконання. Таким чином, умови Бернштейна фактично визначають формальний критерій незалежності операцій у програмі та слугують основою для побудови графів залежностей, які відображають структуру взаємозв’язків між обчислювальними операціями.

У практичних програмних системах ці умови застосовуються не лише до окремих інструкцій, але й до більших фрагментів програмного коду, таких як функції, блоки умовних операторів або ітерації циклів. У таких випадках множини змінних читання та запису визначаються для всього блоку інструкцій, після чого перевіряється відсутність перетину відповідних множин між різними блоками. Якщо залежності відсутні, блоки можуть виконуватися паралельно, що дозволяє ефективно використовувати обчислювальні ресурси багатоядерних процесорів. Особливе значення умови Бернштейна мають під час аналізу циклів, де незалежність ітерацій дозволяє розподіляти обчислення між кількома потоками, що суттєво прискорює обробку великих масивів даних.

Отже, граничні умови Бернштейна є важливим теоретичним механізмом формального аналізу залежностей у програмному коді. Вони забезпечують математично обґрунтований критерій визначення можливості паралельного виконання операцій та використовуються під час розроблення паралельних алгоритмів, оптимізації компіляторів і створення систем автоматичної паралелізації програм. Їх застосування дозволяє виявляти незалежні обчислювальні структури у послідовних програмах та трансформувати їх у паралельні форми виконання, що є одним із ключових напрямів підвищення ефективності сучасних

програмних систем.

Граничні умови Бернштейна – це формальні умови в теорії паралельних обчислень, які визначають, чи можуть дві інструкції або дві частини програми виконуватися паралельно без зміни результату програми. Вони були запропоновані для аналізу залежностей між операціями у програмі та використовуються під час автоматичної паралелізації коду.

Суть умов Бернштейна полягає в аналізі множин змінних, які читаються та змінюються інструкціями. Для кожного оператора або блоку програми визначаються дві множини:

1) R_i (Read set) – множина змінних, які читає інструкція I_i;

2) W_i (Write set) – множина змінних, які змінює інструкція I_j.

Нехай існують дві інструкції або блоки програми I_1 та I_2. Їх паралельне виконання можливе лише тоді, коли виконуються три умови:

1) відсутність конфлікту читання-запису, яка означає, що інструкція I_1 не читає змінну, яку змінює інструкція I_2, і задамо її так: $R_1 \cap W_2 = \emptyset$;

2) відсутність конфлікту запис-читання, яка означає, що інструкція I_2 не читає змінну, яку змінює інструкція, і задамо її так: $R_2 \cap W_1 = \emptyset$;

3) відсутність конфлікту запис-запис, яка означає, що обидві інструкції не змінюють одну й ту саму змінну, і задамо її так: $W_1 \cap W_2 = \emptyset$;

Якщо усі три умови виконуються, то інструкції є незалежними і можуть виконуватися паралельно без порушення логіки програми.

Введемо четверте обмеження. Воно стосується кількості процесорів або ядер в процесорі, які можуть виконати розділені паралельні проси, що сформовані при виконанні системних програм.

Розглянемо приклади щодо трьох перших умов.

Наприклад, розглянемо дві інструкції:

$I_1 : a = b + c;$

$I_2 : d = e + f.$

Тут $R_1 = \{\mathrm{b}, \mathrm{c}\}, W_1 = \{\mathrm{a}\}, R_2 = \{\mathrm{e}, \mathrm{f}\}, W_2 = \{\mathrm{d}\}$

Оскільки

$R_1 \cap W_2 = \emptyset$,

$R_2 \cap W_1 = \emptyset$,

$W_1 \cap W_2 = \emptyset$, то обидві інструкції не мають спільних змінних, тому вони можуть виконуватися паралельно.

Наприклад, для інструкцій

$I_1 : a = b + c$,

$I_2 : d = a + e$

отримуємо $R_2 = \{a, e\}$, $W_2 = \{a\}$, тому $R_2 \cap W_1 = \{a\} \neq \emptyset$. Отже, інструкція I_2 залежить від результату I_1, і їх не можна виконувати паралельно.

Таким чином, граничні умови Бернштейна є важливим інструментом аналізу залежностей у програмах і широко використовуються при автоматичній паралелізації програмного коду, оптимізації компіляторів та розробці паралельних алгоритмів. Вони дозволяють формально визначити, які частини програми можуть виконуватися одночасно, не змінюючи кінцевого результату обчислень.

У результаті аналізу задачі розпаралелювання системних програм можна зробити висновок, що ефективне використання сучасних багатоядерних обчислювальних систем потребує переходу від традиційної послідовної моделі виконання програм до моделей, які передбачають паралельну обробку даних та незалежних обчислювальних підзадач. Основною проблемою при цьому є визначення таких фрагментів програмного коду, які можуть виконуватися одночасно без порушення логічної коректності алгоритму та без виникнення конфліктів доступу до спільних ресурсів. Саме тому важливим етапом дослідження є формальний аналіз залежностей між інструкціями, функціями та ітераціями циклів у програмі.

Ключову роль у цьому процесі відіграють умови Бернштейна, які дозволяють формально визначити можливість паралельного виконання окремих операцій або блоків коду. Завдяки аналізу множин змінних, що читаються та модифікуються під час виконання інструкцій, ці умови забезпечують математично обґрунтований критерій незалежності обчислень. Якщо між інструкціями відсутні конфлікти типу читання-після-запису, запис-після-читання та запис-після-запису, такі інструкції

можуть виконуватися паралельно без зміни кінцевого результату програми.

Таким чином, розпаралелювання системних програм базується на поєднанні структурного аналізу програмного коду та формальних умов незалежності операцій. Використання умов Бернштейна у процесі аналізу дозволяє виявляти потенційні можливості паралелізації, формувати незалежні обчислювальні блоки та оптимально розподіляти їх між потоками виконання. Це створює передумови для підвищення продуктивності програмних систем, більш ефективного використання апаратних ресурсів та адаптації програмного забезпечення до сучасних багатопроцесорних архітектур.

2.2 Моделі типових класів системних програм для розпаралелювання процесів

Сформуємо на основі умов Бернштейна узагальнені моделі класів системних програм, для яких застосування паралельних обчислень є доцільним та ефективним. До таких класів належать програми, у структурі яких існують незалежні обчислювальні підзадачі, відсутні або мінімальні залежності між операціями, а також можливість розділення обчислень на множину однотипних ітерацій або функціональних блоків. Найбільш характерними є три класи програм: програми з незалежними викликами функцій; програми з незалежними шляхами виконання інструкцій; програми з ітераційними структурами (циклами), де кожна ітерація може виконуватися незалежно від інших. Для кожного з цих класів побудуємо формалізовану модель, які описуватимуть структуру обчислень залежності між змінними та умови паралелізації.

До першого класу системних програм віднесемо програми, у яких значна частина обчислень реалізується у вигляді незалежних викликів функцій. У таких системах основний алгоритм можна представити як композицію функцій, кожна з яких виконує окрему частину обчислень. Якщо результати цих функцій не залежать одна від одної або використовуються лише після завершення всіх викликів, їх виконання може бути організоване паралельно. Такий клас системних програм

можна задамо множиною функцій так:

$$F = \{f_1, f_2, \dots, f_{n_F}\}, \tag{2.1}$$

де кожна функція f_i відображає множину вхідних змінних у множину вихідних змінних $f_i: X_i \to Y_i$; $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,n_{X_i}}\}$; $Y_i = \{y_{i,1}, y_{i,2}, \dots, y_{i,n_{Y_i}}\}$;

$i = 1,2, \dots, n_F$;

n_F – кількість функцій в множині функцій F;

X_i – множина вхідних змінних функції f_i;

Y_i – множина змінних, що утворюють результат виконання функції.

Тоді, загальний стан програми можна подати як вектор змінних так:

$$S = (s_1, s_2, \dots, s_{n_S}), \tag{2.2}$$

де s_j – значення j-тої змінної програми.

Після виконання функції f_i стан системи переходить у новий стан, який будемо визначати так:

$$S' = f_i(S). \tag{2.3}$$

Паралельне виконання функцій f_i та f_j можливе лише тоді, коли виконуються модифіковані умови Бернштейна, які визначені для елементів системної програми так:

$$\begin{aligned} R_i \cap W_j &= \emptyset, \\ R_j \cap W_i &= \emptyset, \\ W_i \cap W_j &= \emptyset, \end{aligned} \tag{2.4}$$

де R_i - множина змінних, що читаються функцією f_i;

W_i – множина змінних, що модифікуються функцією f_i;

R_i, W_j – відповідні множини для функції f_j.

Прикладом програм такого класу може бути система обробки великих масивів даних, у якій різні функції виконують незалежні обчислення над різними частинами набору даних. Наприклад, під час аналізу статистичних даних функції можуть обчислювати середнє значення, дисперсію та інші характеристики. У цьому випадку функції

$$\begin{aligned} f_1(D) &= m_1, \\ f_2(D) &= m_2, \\ f_3(D) &= m_3, \end{aligned} \tag{2.5}$$

можуть виконуватися одночасно, оскільки вони лише читають спільні дані і не змінюють їх.

Другий клас сформуємо з системних програм, у яких паралелізм виникає завдяки наявності незалежних шляхів виконання інструкцій у межах однієї функції або процедури. У цьому випадку алгоритм подамо послідовністю операторів так:

$$P = (p_1, p_2, \dots, p_{n_P}), \tag{2.6}$$

де p_k – окрема інструкція програми;

n_P – кількість ісінтрукцій програми.

Стан програми після виконання інструкції p_k задамо функцією переходу так:

$$s_{k+1} = p_k(s_k), \tag{2.7}$$

де s_k – стан системи перед виконанням інструкції p_k;

s_{k+1} – стан після її виконання.

Залежність між двома інструкціями p_i та p_j визначається через змінні, які вони використовують. Нехай $R(p_k)$ це множина змінних, що читаються інструкцією p_k, а $W(p_k)$ – множина змінних, що змінюються цією інструкцією.

Інструкції p_i та p_j можуть виконуватися паралельно, якщо виконуються умови:

$$\begin{aligned} R(p_i) \cap W(p_j) &= \emptyset \\ R(p_j) \cap W(p_i) &= \emptyset \\ W(p_i) \cap W(p_j) &= \emptyset \end{aligned} \tag{2.8}$$

У такому випадку набір інструкцій можна розділити на незалежні підмножини так:

$$P = P_1 \cup P_2 \cup \ldots \cup P_k, \tag{2.9}$$

де кожна підмножина P_i є незалежним шляхом виконання.

Максимальний рівень паралелізму визначається кількістю таких незалежних шляхів. Прикладом програм цього класу може бути обробка зображень, коли різні частини алгоритму виконують незалежні обчислення над різними параметрами. Наприклад, у процесі обробки кадру зображення один блок програми може виконувати фільтрацію шуму, а інший - підсилення контрасту, а третій – обчислення гістограми яскравості. Якщо ці операції не змінюють спільні змінні, то вони можуть виконуватися паралельно.

Третій клас системних програм визначимо програмами з ітераційними структурами, де обчислення виконуються у циклах. У багатьох алгоритмах кожна ітерація циклу виконує однакові операції над різними елементами набору даних, що робить такі алгоритми природними кандидатами для паралелізації. Формально цикл задамо послідовністю ітерацій, у якій на кожній ітерації виконується функція, так:

$$for\ i = a, \ldots, b;\ S_{i+1} = F(S_i, i), \tag{2.10}$$

де i - індекс ітерації;

a - початкове значення індексу;

b – кінцеве значення;

S_i – стан системи перед ітерацією;

F – функція, що описує обчислення в тілі циклу.

Нехай R_i – множина змінних, що читаються під час ітерації i; W_i – множина змінних, що змінюються під час ітерації i.

Ітерації i та j можуть виконуватися паралельно, якщо виконуються всі три умови Бернштейна. Крім того, необхідно, щоб значення параметрів циклу були визначені до початку виконання, тобто повинно виконуватись таке співвідношення:

$$N = b - a + 1, \tag{2.11}$$

де N – кількість ітерацій циклу.

Якщо ці умови виконуються, то цикл може бути перетворений на множину незалежних задач так:

$$T = \{T_1, T_2, \dots, T_{n_T}\}, \tag{2.12}$$

де кожна задача T_i відповідає виконанню однієї ітерації.

Прикладом такого класу програм є алгоритми обробки масивів або матриць. Наприклад, обчислення суми квадратів елементів масиву визначимо так:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} m_i^2, \tag{2.13}$$

де m_i – елемент масиву;

n_{MAS} – розмір масиву;

$S_{\sum MAS}$ – результат обчислення суми квадратів елементів масиву.

У цьому випадку кожна операція $y_i = m_i^2$ може виконуватися паралельно,

після чого результати об’єднуються операцією редукції так:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} y_i. \quad (2.14)$$

Таким чином, аналіз структури системних програм дозволяє виділити три основні моделі, придатні для паралелізації: модель незалежних функціональних блоків; модель незалежних шляхів виконання інструкцій; модель незалежних ітерацій циклів. Кожна з цих моделей характеризується власними формальними умовами залежностей між змінними та операціями, що дозволяє визначити можливість паралельного виконання та побудувати оптимальну структуру багатопотокового алгоритму. Використання таких формалізованих моделей є основою для створення автоматизованих систем аналізу та оптимізації програмного коду, які можуть адаптувати традиційні послідовні алгоритми до сучасних багатоядерних обчислювальних систем.

У класичній постановці умов Бернштейна визначено три основні критерії незалежності операцій, які дозволяють встановити можливість їх паралельного виконання. Ці умови базуються на аналізі множин змінних, що читаються та змінюються інструкціями, і гарантують відсутність конфліктів доступу до даних між паралельними обчисленнями. Проте під час практичного застосування методів розпаралелювання системних програм виникає необхідність врахування ще одного важливого аспекту, який стосується апаратних обмежень обчислювальної системи, зокрема кількості доступних процесорів або ядер процесора. Тому до класичних умов доцільно додати четверту умову, яка пов’язана з ефективністю виконання паралельних задач з урахуванням апаратних ресурсів.

Четверта умова полягає в тому, що кількість паралельних обчислювальних задач, які формуються в результаті розпаралелювання програми, повинна бути узгоджена з кількістю доступних обчислювальних ресурсів системи. Нехай P позначає кількість доступних процесорів або обчислювальних ядер, на яких може виконуватися програма, а T – кількість незалежних задач або потоків, отриманих у результаті розпаралелювання. Тоді ефективність паралельного виконання значною

мірою визначається співвідношенням між цими величинами. Якщо T значно перевищує P, то система змушена виконувати частину задач послідовно або здійснювати часте перемикання контекстів між потоками, що призводить до додаткових витрат на керування виконанням. Якщо ж T значно менше за P, то частина обчислювальних ресурсів залишається невикористаною, що також знижує загальну ефективність виконання програми.

Цю умову задамо у вигляді обмеження так:

$$T \leq P \cdot k, \tag{2.15}$$

де T – кількість паралельних задач або потоків;

P – кількість доступних процесорів або ядер;

k – коефіцієнт допустимого перевищення кількості потоків над кількістю процесорів, який враховує особливості планування потоків операційною системою.

У найпростішому випадку для максимального використання обчислювальних ресурсів бажано, щоб кількість потоків була близькою до кількості доступних ядер, тобто щоб виконувалась умова:

$$T \approx P. \tag{2.16}$$

У такому випадку кожна задача може виконуватися на окремому ядрі, що забезпечує максимальний рівень паралелізму без додаткових витрат на планування потоків.

Якщо розглядати загальний час виконання програми, то його можна подати у вигляді суми часу паралельної та послідовної частини алгоритму. Нехай T_s – час виконання послідовної частини програми, T_p – час виконання паралельної частини при використанні одного процесора, P – кількість процесорів або ядер. Тоді теоретичний час виконання програми при паралельному виконанні оцінимо так:

$$T_{\text{вик}} = T_s + \frac{T_p}{P}. \tag{2.17}$$

Це співвідношення фактично відображає принцип, близький до закону Амдала, відповідно до якого максимальний виграш від паралелізації обмежується часткою послідовного коду. Однак у реальних системах до цього часу додаються додаткові витрати на створення потоків, синхронізацію та обмін даними між потоками. Позначимо ці витрати через $T_{\text{о}}$.Тоді фактичний час виконання визначимо так:

$$T_{\text{вик}} = T_s + \frac{T_p}{P} + T_{\text{о}}. \tag{2.18}$$

У цьому випадку четверта умова полягає не лише у відповідності кількості задач кількості процесорів, але й у тому, що виграш від паралельного виконання повинен перевищувати накладні витрати, пов’язані з організацією паралельності. Тобто паралелізація є доцільною лише тоді, коли виконується така умова:

$$\frac{T_p}{P} + T_{\text{о}} < T_p \tag{2.19}$$

Інакше кажучи, сумарний час виконання паралельної частини програми з урахуванням додаткових витрат повинен бути меншим, ніж час її виконання у послідовному режимі.

Таким чином, четверта умова фактично доповнює класичні умови Бернштейна, враховуючи апаратні характеристики обчислювальної системи та реальні витрати на організацію паралельного виконання. Якщо перші три умови визначають логічну можливість паралельного виконання інструкцій з точки зору відсутності залежностей даних, то четверта умова визначає практичну доцільність такого розпаралелювання з урахуванням доступних обчислювальних ресурсів та ефективності використання процесорних ядер. Саме поєднання цих чотирьох умов дозволяє не лише виявити незалежні обчислювальні структури у програмному коді, але й забезпечити оптимальне використання можливостей сучасних багатоядерних обчислювальних систем. Ця умова не впливає на кількість визначених класів, а

лише доповнює новим обмеженням введені класи.

П’ятим обмеженням є закон Амдала. Закон Амдала визначає потенційне прискорення алгоритму при збільшенні числа процесорів. Закон Амдала є одним із фундаментальних принципів теорії паралельних обчислень, який визначає теоретичну межу прискорення виконання програми під час використання багатопроцесорних або багатоядерних обчислювальних систем. Він описує залежність між часткою алгоритму, що може бути виконана паралельно, кількістю доступних процесорів та максимально можливим виграшем у продуктивності. Основна ідея цього закону полягає в тому, що навіть за наявності необмеженої кількості процесорів швидкодія програми обмежується тією частиною алгоритму, яка не може бути виконана паралельно і повинна виконуватися послідовно.

Нехай час виконання певної програми на одному процесорі дорівнює T_1. Цей час можна подати як суму двох компонентів: часу виконання послідовної частини алгоритму; часу виконання тієї частини програми, яка може бути розпаралелена. Позначимо через α частку послідовної частини програми, тобто ту частину алгоритму, яка не може бути виконана паралельно. Відповідно, частка паралельної частини становитиме (1 – α). Якщо використовується P процесорів або обчислювальних ядер, то паралельна частина алгоритму теоретично може бути розподілена між цими процесорами. У такому випадку час виконання програми на P процесорах можна визначити так:

$$T_P = T_1 \cdot \left(\alpha + \frac{1-\alpha}{P}\right), \tag{2.20}$$

де T_P – час виконання програми на (P) процесорах;

T_1 - час виконання програми на одному процесорі;

α – частка послідовної частини програми;

1-α – частка паралельної частини програми;

P – кількість процесорів або ядер.

Однією з основних характеристик ефективності паралельних обчислень є прискорення виконання програми, яке визначається як відношення часу виконання

програми на одному процесорі до часу виконання на P процесорах:

$$S(P) = \frac{T_1}{T_P}. \quad (2.21)$$

Підставивши значення T_P у цю формулу, отримуємо класичний вираз закону Амдала:

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}}. \quad (2.22)$$

Ця формула демонструє, що зі збільшенням кількості процесорів прискорення програми зростає, однак його зростання поступово сповільнюється. Причиною цього є наявність послідовної частини алгоритму, яка не може бути розподілена між процесорами і виконується лише одним потоком.

Якщо розглянути граничний випадок, коли кількість процесорів прямує до нескінченності $P \to \infty$, паралельна частина програми теоретично виконується миттєво, а загальний час виконання визначається лише послідовною частиною. У цьому випадку максимальне можливе прискорення буде дорівнювати значенню:

$$S_{max} = \frac{1}{\alpha}. \quad (2.23)$$

Це означає, що навіть за необмеженої кількості процесорів швидкодія програми обмежується величиною послідовної частини алгоритму. Наприклад, якщо 10% програми виконуються послідовно $\alpha = 0.1$, то максимальне прискорення не перевищить такого значення S_{max} (2.24):

$$S_{max} = \frac{1}{0{,}1} = 10. \quad (2.24)$$

незалежно від того, скільки процесорів використовується.

Закон Амдала має важливе практичне значення під час проєктування паралельних алгоритмів і систем. Він показує, що для досягнення високої ефективності паралельних обчислень необхідно максимально зменшувати частку послідовних операцій у програмі. Навіть невелика частина послідовного коду може суттєво обмежувати потенційний виграш від використання великої кількості процесорів. Саме тому під час оптимізації програмних систем велика увага приділяється аналізу алгоритмів, пошуку незалежних обчислювальних блоків і зменшенню кількості операцій, які не можуть бути виконані паралельно.

Крім того, у реальних обчислювальних системах необхідно враховувати додаткові фактори, які також впливають на продуктивність паралельних програм. До них належать витрати часу на створення потоків, синхронізацію між ними, передачу даних між процесорами, а також затримки доступу до спільної пам’яті. Унаслідок цього фактичне прискорення часто виявляється меншим за теоретичне, передбачене законом Амдала. Незважаючи на це, цей закон залишається важливим теоретичним інструментом для оцінювання ефективності паралельних обчислень та визначення меж доцільності розпаралелювання алгоритмів у сучасних багатоядерних обчислювальних системах.

Ефективна організація паралельних обчислень базується на поєднанні структурного аналізу програмного коду, формальних умов незалежності операцій та врахування апаратних характеристик обчислювальної системи. Виділено три основні моделі системних програм, для яких розпаралелювання є найбільш доцільним і технічно можливим. Кожна з цих моделей характеризується певними особливостями структури алгоритмів та способом організації обчислювальних процесів, що визначає потенційну можливість виконання окремих частин програми паралельно.

Перша модель охоплює системні програми, побудовані на основі незалежних функціональних модулів або викликів функцій. У таких програмах основний алгоритм можна представити як сукупність функціональних блоків, кожен з яких виконує окрему обчислювальну задачу та працює з власною підмножиною даних. Якщо між цими функціональними блоками відсутні залежності за даними, вони

можуть виконуватися одночасно в різних потоках або на різних обчислювальних ядрах. Така модель характерна для систем обробки даних, аналітичних програм, програм статистичних обчислень та інших систем, у яких результати окремих обчислень можуть бути отримані незалежно один від одного.

Друга модель системних програм базується на наявності незалежних шляхів виконання інструкцій у межах одного алгоритму або процедури. У цьому випадку програму можна розглядати як послідовність операторів, між якими можуть існувати або бути відсутні залежності за даними. Якщо окремі групи інструкцій не використовують спільні змінні або не модифікують результати виконання одна одної, такі інструкції можуть бути виконані паралельно. Аналіз подібних структур дозволяє виділяти незалежні підграфи у графі залежностей програми, що відкриває можливість одночасного виконання різних частин алгоритму. Ця модель характерна для систем обробки сигналів, обчислювальних модулів у складних програмних комплексах та програм, у яких окремі обчислювальні етапи не залежать від результатів інших етапів.

Третя модель системних програм пов’язана з наявністю ітераційних структур, зокрема циклів, у яких однакові операції виконуються над різними елементами даних. У таких випадках кожна ітерація циклу може розглядатися як окрема обчислювальна задача. Якщо між ітераціями відсутні залежності за даними, вони можуть виконуватися паралельно на різних обчислювальних ядрах. Подібна модель є характерною для алгоритмів обробки масивів, матричних обчислень, моделювання фізичних процесів, обробки зображень та інших задач, пов’язаних з великими обсягами однотипних обчислень. Саме ця модель найчастіше використовується в системах високопродуктивних обчислень, оскільки вона дозволяє ефективно масштабувати програму при збільшенні кількості процесорів.

Для формального визначення можливості паралельного виконання операцій у межах цих моделей використовуються умови Бернштейна, які дозволяють встановити відсутність конфліктів доступу до даних між інструкціями або програмними блоками. Ці умови базуються на аналізі множин змінних, що читаються та змінюються під час виконання відповідних операцій, і визначають

три основні типи залежностей: читання після запису, запис після читання та запис після запису. Якщо між інструкціями відсутні такі конфлікти, вони можуть виконуватися паралельно без зміни кінцевого результату програми. Таким чином, умови Бернштейна виступають формальним критерієм логічної незалежності обчислювальних операцій і дозволяють виявляти потенційні можливості для розпаралелювання програмного коду.

Однак навіть за виконання класичних умов незалежності операцій ефективність паралельного виконання програм значною мірою залежить від апаратних характеристик обчислювальної системи. Саме тому до традиційних умов аналізу доцільно додати ще одну умову, пов’язану з кількістю доступних процесорів або ядер. Ця умова полягає у необхідності узгодження кількості паралельних задач із кількістю обчислювальних ресурсів системи. Якщо кількість потоків значно перевищує кількість доступних ядер, виникають додаткові накладні витрати, пов’язані з плануванням потоків і перемиканням контексту. Якщо ж кількість задач є меншою за кількість ядер, частина обчислювальних ресурсів залишається невикористаною. Отже, ефективна паралелізація передбачає таку організацію задач, за якої їх кількість наближена до кількості доступних обчислювальних ядер.

Ще одним важливим фактором, який необхідно враховувати під час аналізу ефективності паралельних програм, є закон Амдала. Цей закон встановлює теоретичну межу прискорення виконання програми залежно від частки алгоритму, яка може бути розпаралелена. Згідно з цим принципом, загальне прискорення програми обмежується тією частиною алгоритму, яка повинна виконуватися послідовно. Навіть при використанні великої кількості процесорів послідовна частина програми стає визначальним фактором загального часу виконання. Тому під час проєктування паралельних алгоритмів необхідно не лише визначати незалежні обчислювальні структури, але й максимально зменшувати частку послідовних операцій у програмному коді.

Таким чином, ефективне розпаралелювання системних програм ґрунтується на комплексному підході, який поєднує структурний аналіз алгоритмів, формальні

критерії незалежності обчислень та врахування апаратних обмежень обчислювальних систем. Виділення трьох основних моделей програм, застосування умов Бернштейна для аналізу залежностей між операціями, а також врахування кількості обчислювальних ядер і обмежень, визначених законом Амдала, дозволяють сформувати цілісну методологію аналізу та оптимізації програм для багатоядерних обчислювальних середовищ. Реалізація такого підходу створює передумови для підвищення продуктивності програмних систем, більш ефективного використання апаратних ресурсів та адаптації програмного забезпечення до сучасних архітектур паралельних обчислень.

2.3 Висновки до другого розділу

Ефективне розпаралелювання системних програм базується на поєднанні структурного аналізу алгоритмів, формальних умов незалежності операцій та врахування апаратних характеристик обчислювальних систем.

Введено три основні моделі системних програм, придатних до паралельного виконання: програми з незалежними функціональними модулями, програми з незалежними шляхами виконання інструкцій та програми з ітераційними структурами.

Для визначення можливості паралельного виконання операцій використано умови Бернштейна, які дозволяють формально встановити відсутність конфліктів доступу до даних. Додатково враховано вплив кількості процесорних ядер та обмеження, визначені законом Амдала, що визначають реальну ефективність паралельних обчислень і межі прискорення програм.

3 МЕТОД РОЗПАРАЛЕЛЮВАННЯ ДИНАМІЧНИХ ПОСЛІДОВНИХ СИСТЕМНИХ ПРОГРАМ З ВИКОРИСТАННЯМ МЕРЕЖ БАГАТОГРАННИХ ПРОЦЕСІВ

3.1 Організація мереж багатогранних процесів

Розвиток сучасних обчислювальних систем характеризується постійним зростанням продуктивності процесорів та широким використанням багатоядерних архітектур. У зв’язку з цим особливо актуальною стає задача ефективного використання паралельних обчислювальних ресурсів. Більшість традиційних програм створювались як послідовні алгоритми, які виконують операції одну за одною. Такий підхід не дозволяє повністю використовувати потенціал сучасних паралельних систем, тому виникає необхідність у методах автоматичного або напівавтоматичного розпаралелювання програм.

Одним із сучасних підходів до розпаралелювання програм є використання багатогранної моделі обчислень. Ця модель дозволяє формально описувати структуру вкладених циклів і залежності між операціями за допомогою математичного апарату лінійної алгебри та теорії багатогранників. Завдяки такому підходу можна перетворювати послідовні алгоритми у паралельні обчислювальні структури без зміни їх логіки.

Багатогранна модель базується на представленні ітерацій циклів у вигляді точок у багатовимірному просторі. Кожна ітерація вкладених циклів характеризується набором індексів, які можна розглядати як координати у цьому просторі. Сукупність усіх таких точок утворює область, яка називається простором ітерацій. У більшості випадків цей простір має форму багатогранника, що визначається системою лінійних нерівностей.

Використання геометричного представлення обчислювальних процесів дозволяє застосовувати методи математичного аналізу для оптимізації виконання програм. Зокрема, можна визначати залежності між ітераціями, аналізувати можливість їх паралельного виконання та виконувати різні трансформації циклів для покращення продуктивності.

Одним із ключових понять у багатогранній моделі є область ітерацій. Область ітерацій визначає множину всіх можливих комбінацій значень індексів циклів, для яких виконується певна операція. Формально область ітерацій задається системою лінійних нерівностей, які обмежують значення індексів.

Наприклад, для подвійного циклу область ітерацій може бути описана як множина точок, що задовольняють умови обмеження індексів. Геометрично така область представляється у вигляді прямокутника або іншої багатокутної фігури у двовимірному просторі.

Важливим аспектом аналізу програм у багатогранній моделі є визначення залежностей між операціями. Залежності виникають у випадках, коли результат однієї операції використовується в іншій. Наявність таких залежностей визначає порядок виконання операцій і обмежує можливості паралельного виконання.

У багатогранній моделі залежності описуються у вигляді відношень між точками простору ітерацій. Це дозволяє точно визначити, які ітерації можуть виконуватись незалежно, а які повинні виконуватися у певному порядку.

Одним із способів представлення залежностей є використання векторів залежностей. Вектор залежності показує напрямок передачі даних між ітераціями. Аналіз цих векторів дозволяє визначити можливість перестановки циклів або інших трансформацій програми.

Важливим етапом оптимізації програм у багатогранній моделі є побудова функції розкладу. Функція розкладу визначає порядок виконання ітерацій у часі та дозволяє розподілити їх між паралельними процесами. Правильно обрана функція розкладу дозволяє максимально збільшити рівень паралелізму та зменшити час виконання програми.

Ще одним важливим поняттям є багатогранний процес. Багатогранний процес представляє собою обчислювальний процес, який виконує операції над множиною ітерацій, що належать певному багатограннику. Кожен такий процес може бути реалізований як окремий паралельний потік або процес у багатопроцесорній системі.

Для організації взаємодії між такими процесами використовується мережа

багатогранних процесів. Мережа представляє собою сукупність процесів, які обмінюються даними через канали зв’язку. Така структура дозволяє ефективно організувати паралельні обчислення та мінімізувати затримки передачі даних.

У мережі багатогранних процесів кожен процес виконує обчислення над певною частиною простору ітерацій. Результати цих обчислень можуть передаватися іншим процесам, які використовують їх для виконання наступних операцій. Таким чином формується потік даних між процесами.

Однією з основних переваг такого підходу є можливість автоматичного генерування паралельного коду. На основі аналізу області ітерацій та залежностей компілятор може визначити оптимальний спосіб розподілу обчислень між процесорами.

Крім того, багатогранна модель дозволяє виконувати різні трансформації циклів. До таких трансформацій належать перестановка циклів, розбиття циклів на блоки, а також зсув ітераційного простору. Ці операції дозволяють покращити локальність доступу до пам’яті та зменшити кількість звернень до повільної оперативної пам’яті.

Особливо ефективним є використання методу блокування або тайлінгу. Цей метод полягає у поділі простору ітерацій на менші блоки, які можуть оброблятися окремими процесами. Такий підхід значно підвищує ефективність використання кеш-пам’яті.

Мережі багатогранних процесів широко застосовуються у високопродуктивних обчисленнях. Вони використовуються для оптимізації чисельних алгоритмів, обробки великих масивів даних, а також для виконання складних наукових розрахунків.

Зокрема, цей підхід застосовується у задачах лінійної алгебри, моделювання фізичних процесів, обробки сигналів та зображень. У таких задачах часто використовуються вкладені цикли над великими масивами даних, що робить їх ідеальними кандидатами для застосування багатогранної моделі.

Сучасні компілятори та системи оптимізації активно використовують елементи багатогранної моделі. Вони дозволяють автоматично аналізувати

програмний код та виконувати оптимізації без участі програміста.

Серед відомих систем, що реалізують подібні підходи, можна відзначити оптимізуючі компілятори для високопродуктивних обчислень. Вони застосовуються у наукових дослідженнях, машинному навчанні та обробці великих даних.

Незважаючи на складність математичного апарату, багатогранна модель є потужним інструментом для аналізу та оптимізації програм. Вона дозволяє формалізувати процес розпаралелювання і створити ефективні алгоритми для автоматичної генерації паралельного коду.

Таким чином, використання мереж багатогранних процесів є перспективним напрямком у розвитку технологій паралельного програмування. Цей підхід дозволяє значно підвищити продуктивність програм та ефективно використовувати можливості сучасних багатопроцесорних систем. Однією з важливих задач є підвищення ефективності виконання програм на багатопроцесорних обчислювальних системах. Збільшення кількості обчислювальних ядер у сучасних процесорах створює передумови для значного підвищення продуктивності програмного забезпечення. Однак для повноцінного використання цих можливостей необхідно адаптувати програмні алгоритми до паралельної моделі виконання.

У багатьох випадках програмне забезпечення створюється у вигляді послідовних алгоритмів. Послідовні алгоритми передбачають виконання інструкцій у чітко визначеному порядку, де кожна наступна операція починається лише після завершення попередньої. Такий підхід був ефективним у період домінування однопроцесорних систем, проте в умовах багатоядерних архітектур він стає обмеженням для подальшого підвищення продуктивності.

Для вирішення цієї проблеми застосовуються різні методи розпаралелювання програм. Одним із найбільш перспективних підходів є використання математичних моделей, які дозволяють формально описувати структуру програм та визначати можливості для паралельного виконання. До таких моделей належить багатогранна модель обчислень. Багатогранна модель базується на ідеї представлення

програмних циклів у вигляді геометричних об’єктів. Зокрема, ітерації циклів розглядаються як точки у багатовимірному просторі. Такий підхід дозволяє застосовувати методи геометрії та лінійної алгебри для аналізу програмних структур.

Однією з ключових характеристик програм, які піддаються аналізу у межах багатогранної моделі, є наявність вкладених циклів з афінними межами. Афінні межі означають, що границі циклів визначаються лінійними функціями від параметрів або індексів інших циклів. Саме такі циклічні структури найчастіше зустрічаються у наукових обчисленнях, обробці сигналів та інших галузях.

Важливим елементом багатогранної моделі є опис області ітерацій. Область ітерацій визначає множину всіх допустимих значень індексів циклів, для яких виконується певна операція. Ця область задається системою лінійних нерівностей, що дозволяє розглядати її як багатогранник у багатовимірному просторі.

Завдяки такому представленню стає можливим використання математичних методів оптимізації для аналізу та трансформації програм. Зокрема, можна визначати оптимальні способи перестановки циклів, виконувати розбиття ітераційного простору на блоки та знаходити ефективні схеми паралельного виконання.

Ще одним важливим аспектом є аналіз залежностей між операціями програми. Залежності виникають у випадках, коли одна операція використовує дані, що були обчислені іншою операцією. Наявність таких залежностей обмежує можливість паралельного виконання ітерацій.

У багатогранній моделі залежності описуються у вигляді відношень між точками ітераційного простору. Це дозволяє точно визначити порядок виконання операцій і виявити ті частини алгоритму, які можуть виконуватися незалежно одна від одної.

Аналіз залежностей дозволяє будувати граф залежностей програми. Такий граф відображає взаємозв’язки між операціями і може використовуватися для визначення оптимального порядку їх виконання.

Особливу роль у багатогранній моделі відіграє функція розкладу виконання.

Функція розкладу визначає відображення ітерацій у часовий простір і дозволяє визначити, у який момент часу повинна виконуватися кожна операція. Завдяки цьому можна організувати паралельне виконання незалежних ітерацій.

Використання функцій розкладу дозволяє значно збільшити рівень паралелізму у програмі. При цьому необхідно враховувати залежності між ітераціями, щоб уникнути порушення коректності обчислень.

Крім часової організації виконання, важливим завданням є розподіл обчислень між процесорами. Для цього простір ітерацій може бути розбитий на окремі частини, кожна з яких виконується окремим процесом або потоком.

Такий підхід дозволяє створювати мережі багатогранних процесів. У цій мережі кожен процес виконує обчислення над певною підмножиною ітераційного простору та обмінюється результатами з іншими процесами.

Мережі багатогранних процесів можна розглядати як граф обчислень, де вузли відповідають процесам, а ребра відповідають каналам передачі даних. Така структура дозволяє ефективно організувати взаємодію між паралельними обчислювальними компонентами.

Однією з важливих задач при побудові мереж багатогранних процесів є мінімізація обсягу переданих даних між процесами. Надмірна передача даних може призвести до значних затримок та зниження ефективності паралельних обчислень.

Для підвищення ефективності обчислень використовуються різні методи оптимізації. Одним з таких методів є блокування або таймінг ітераційного простору. Цей метод полягає у поділі великого простору ітерацій на менші блоки, які можуть оброблятися окремо.

Блокування дозволяє значно покращити локальність доступу до пам’яті. Це означає, що дані, необхідні для виконання певної групи операцій, зберігаються у кеш-пам’яті процесора і не потребують повторного завантаження з оперативної пам’яті.

Ще одним методом оптимізації є перестановка циклів. Перестановка дозволяє змінити порядок виконання циклів таким чином, щоб покращити ефективність використання пам’яті та зменшити кількість залежностей між

ітераціями.

У деяких випадках також використовується метод зсуву ітераційного простору. Цей метод дозволяє змінити форму області ітерацій таким чином, щоб збільшити можливості для паралельного виконання.

Використання таких трансформацій дозволяє створювати високоефективні паралельні алгоритми, які можуть виконуватися на сучасних багатоядерних процесорах та графічних прискорювачах.

Сучасні оптимізуючі компілятори активно використовують принципи багатогранної моделі. Вони здатні автоматично аналізувати програмний код, визначати залежності між операціями та генерувати оптимізовані версії системних програм. Це дозволяє значно спростити процес розробки високопродуктивного програмного забезпечення. Програмісту не потрібно вручну оптимізувати кожен цикл або розподіляти обчислення між процесорами. Незважаючи на значні переваги, використання багатогранної моделі має певні обмеження. Зокрема, вона найбільш ефективна для системних програм, що містять регулярні циклічні структури з афінними залежностями. Для програм з нерегулярною структурою або складними умовними операціями застосування багатогранної моделі може бути ускладненим. У таких випадках можуть використовуватися інші методи аналізу та оптимізації програм. Проте навіть з урахуванням цих обмежень багатогранна модель залишається одним із найпотужніших інструментів для автоматичного розпаралелювання системних програм.

Також важливим напрямком досліджень є інтеграція багатогранної моделі з сучасними системами програмування для гетерогенних обчислювальних систем, які включають центральні процесори, графічні прискорювачі та інші спеціалізовані пристрої. Мережі багатогранних процесів відіграють важливу роль у розвитку технологій паралельного програмування. Вони забезпечують формальний підхід до аналізу програм та створюють основу для автоматичної генерації ефективних паралельних алгоритмів.

На схемі з рис. 3.1 зображено приклад мережі багатогранних процесів, яка використовується для організації паралельних обчислень. Кожен шестикутний

блок на схемі представляє окремий багатогранний процес, що виконує обчислення над певною частиною простору ітерацій алгоритму. Центральний процес може виступати координуючим вузлом або процесом, який обробляє основну частину даних. Стрілки між блоками відображають канали передачі даних між процесами. Через ці канали процеси обмінюються проміжними результатами обчислень. Напрямок стрілок показує напрямок передачі інформації від одного процесу до іншого. Така взаємодія забезпечує правильний порядок виконання операцій відповідно до залежностей між даними.

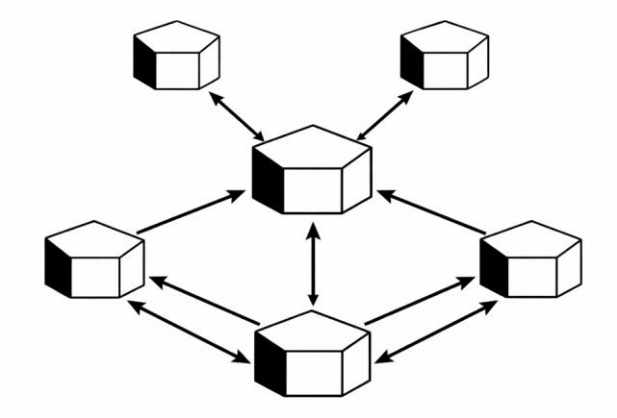


Рисунок 3.1 – Приклад мережі багатогранних процесів

Зовнішні процеси можуть виконувати обчислення паралельно, обробляючи різні частини задачі. Після завершення обчислень результати передаються іншим процесам, які використовують їх для виконання наступних етапів алгоритму. Це дозволяє значно скоротити загальний час виконання програми.

Таким чином, схема демонструє принцип організації паралельної обчислювальної системи, де кілька багатогранних процесів працюють одночасно та взаємодіють між собою через мережу передачі даних. Така структура забезпечує ефективне використання обчислювальних ресурсів багатопроцесорних систем.

Розглянемо моделі для опису розпаралеллювання системних програм на основі мереж багатогранних процесів.

У традиційних програмах більшість алгоритмів записані послідовно, тобто операції виконуються одна за одною. Наприклад:

```
for i = 1..N
    for j = 1..N
        A[i][j] = A[i-1][j] + A[i][j-1]
```

Така програма виконується по ітераціях, по визначеному порядку та з можливими залежностями між даними. Проблема полягає у тому, що сучасні обчислювальні системи мають багато процесорів, і для ефективного використання потрібно виконувати обчислення паралельно.

Багатогранна модель обчислень це математичний апарат, який описує виконання програм із вкладеними циклами через геометричні структури в багатовимірному просторі. Основна ідея полягає в тому, що кожна ітерація циклу представлена точкою в багатовимірному просторі. Множина всіх таких точок утворює багатогранник.

Багатогранник є множиною точок, що задовольняють систему лінійних нерівностей. Задамо його так:

$$P = \{x \in Z^n | Ax \leq b\}, \tag{3.1}$$

де x – вектор змінних;

A – матриця коефіцієнтів;

b – вектор обмежень.

В формулі (3.1) розглядаються цілочисельні точки, тому що ітерації циклів мають цілі значення. Для цього нелбхідно побудувати простір ітерацій. Простір ітерацій це множина всіх ітерацій вкладених циклів.

Наприклад:

```
for i = 1..N
    for j = 1..M
        S(i,j)
```

Ітерацію визначимо координатами (i, j), а прості ітерацій задамо тоді так:

$$D = \{(i, j) | 1 \leq i \leq N, 1 \leq j \leq M\}. \tag{3.2}$$

Простором ітерацій згідно формули (3.2) буде двовимірний багатогранник.

Геометрично це прямокутник у координатній площині.

Багатогранний процес визначимо трійкою так:

$$P = \langle D, S, R \rangle, \tag{3.3}$$

де D – область ітерацій, причому $D \subseteq Z^n$;

S – множина операцій, причому кожна ітерація виконує одну й ту ж операцію;

R – відношення залежностей, причому $R \subseteq D \times D$.

Результат формули (3.3) означає наявність відображення $(i_1, i_2) \to (j_1, j_2)$, якщо друга ітерація залежить від першої ітерації.

У програмах існує кілька типів залежностей. Розглянемо їх та формалізуємо.

Істинну залежність задамо так:

$$S_1(i) \to S_2(j), \tag{3.4}$$

коли значення, обчислене в S_1 використовується у S_2.

Другим питом залежностей є читання, яке відбувається перед перезаписом. Третій типом залежностей введемо такі, в яких дві операції записують у ту саму змінну.

У багатогранній моделі всі залежності описуються векторами залежності. Наприклад, $(i, j) \to (i, j + 1)$, то вектор залежності буде $(0,1)$.

Щоб розпаралелити програму, введемо функцію розкладу. Вона буде відображати кожну ітерацію у момент часу виконання. Задамо її так:

$$\theta: D \to Z^n. \tag{3.5}$$

Приклад: $\theta(i, j) = i + j$. Це означає, що всі точки з однаковим $i + j$ можуть виконуватися паралельно.

Багатогранна модель дозволяє виконувати лінійні трансформації циклів.

Наприклад для перестановки циклів:

1) було

```
for i
  for j
```

2) стало

```
for j
  for i
```

3) зсув

$i' = i, j' = j + 1$

4) блокування, в якому простір ітерацій поділяється на блоки.

Мережа багатогранних процесів це система процесів, що взаємодіють між собою через передачу даних. Задамо її так:

$$G = (P, C), \tag{3.6}$$

де P – множина процесів;

C – множина каналів передачі даних.

Кожен процес виконує операції над власним доменом ітерацій.

Мережа багатогранних процесів містить вузли, багатогранні процеси, канали обміну даними, буфери для зберігання переданих значень.

Розглянемо приклад розпаралелювання алгоритму обчислення динамічного програмування - обчислення матриці, формулу для якої задамо так:

$$A[i,j] = A[i-1,j] + A[i,j-1]. \tag{3.7}$$

Послідовний алгоритм задамо так:

```
for i = 1..N
  for j = 1..N
      A[i,j] = A[i-1,j] + A[i,j-1]
```

Простір ітерацій задамо за формулою (3.2). Залежності задамо з урахуванням $(i-1,j) \to (i,j)$, $(i,j-1) \to (i,j)$. Тоді, вектори залежності $d_1 = (1,0), d_2 = (0,1)$.

Здійснимо побудову паралельного виконання. Використаємо функцію розкладу $\theta(i,j) = i + j$ згідно формули (3.5). Тоді усі точки з однаковим значенням

$i + j$ виконуються одночасно. Наприклад, дані в табл. 3.1.

Таблиця 3.1 - Координати точок

i	j	i+j
1	1	2
1	2	3
2	1	3
1	3	4
2	2	4
3	1	4

Таким чином обчислення здійснюються за діагоналями матриці.

Кроки паралельного виконання.

Крок 1: (1,1).

Крок 2: (1,2), (2,1).

Крок 3: (1,3), (2,2), (3,1)

Здійснимо побудову мережі процесів.

Матриця розбивається на блоки.

Наприклад. Наявні чотири процесори.

Процесор P1: $i = 1..\frac{N}{2}, j = 1..\frac{N}{2}$.

Процесор P2: $i = 1..\frac{N}{2}, j = \frac{N}{2}..N$.

Процесор P3: $i = \frac{N}{2}..N, j = 1..\frac{N}{2}$

Процесор P4: $i = \frac{N}{2}..N, j = \frac{N}{2}..N$

Процеси обмінюються значеннями за межами блоків.

Задамо математичну модель мережі так:

$$P_k = (D_k, S_k, R_k),\ D = \sum_{k=1}^{p} D_k, \tag{3.8}$$

де D_k – частина домену.

Тоді, передачу даних задамо так:

$$C_{k,l} = \{(x, y) | x \in D_k, y \in D_l\}. \quad (3.9)$$

Перевагами багатогранної моделі є точний математичний опис системних програм, автоматичне розпаралелювання, оптимізація кеш-пам'яті та ефективність на суперкомп'ютерах. Мережі багатогранних процесів це формальна модель розпаралелювання програм, яка базується на представленні ітерацій циклів як точок у багатовимірному просторі. Завдяки використанню систем лінійних обмежень, функцій розкладу та аналізу залежностей можна автоматично перетворювати послідовні алгоритми у паралельні обчислювальні структури.

Таким чином, розроблено моделі мереж багатогранних процесів для забезпечення розпаралелювання процесів системних програм. Їх особливістю є представленні ітерацій циклів як точок у багатовимірному просторі.

3.2 Метод розпаралелювання динамічних послідовних системних програм

Предметна область щодо розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів належить до напрямку досліджень у галузі паралельних обчислень, системного програмування та оптимізації програмного виконання. Сучасні обчислювальні системи дедалі частіше будуються на основі багатоядерних процесорів, кластерів та розподілених обчислювальних середовищ. Проте значна частина програмного забезпечення, зокрема системного рівня, історично створювалася як послідовна, тобто така, що виконує інструкції крок за кроком у єдиному потоці керування. Це призводить до неефективного використання апаратних ресурсів, оскільки навіть на багатоядерних системах програма може виконуватися фактично на одному ядрі. У зв'язку з цим виникає проблема пошуку ефективних методів перетворення послідовних програм у паралельні без суттєвої зміни їх логіки та без повного переписування

програмного коду. Особливо складною ця проблема стає у випадку динамічних послідовних системних програм. На відміну від статичних алгоритмів, де структура обчислень і взаємозв’язки між операціями відомі наперед, динамічні програми формують свій потік виконання під час роботи. Їх поведінка залежить від поточного стану системи, оброблюваних даних, умов виконання або взаємодії з іншими компонентами програмного середовища. До таких програм можуть належати системні утиліти, модулі операційних систем, компілятори, диспетчери задач, програми обробки складних структур даних та інші системні компоненти. Через наявність умовних переходів, динамічних структур даних і непередбачуваних залежностей між операціями автоматичне розпаралелювання таких програм є складною задачею, оскільки необхідно зберегти коректність виконання та уникнути конфліктів доступу до даних.

У цьому контексті пропонується використання підходу, що базується на представленні програми у вигляді мережі багатогранних процесів. Основна ідея полягає в тому, щоб перетворити послідовний алгоритм на структуровану систему взаємодіючих процесів, кожен з яких відповідає за виконання певної частини обчислень або обробку певного типу даних. Така мережа формує граф обчислювальних залежностей, де вершини відповідають процесам або обчислювальним блокам, а ребра описують потоки даних, синхронізацію або передачу керування. У цій моделі багатогранний процес розглядається як узагальнений обчислювальний елемент, який має кілька аспектів функціонування. З одного боку, він виконує власні обчислення, з іншого — взаємодіє з іншими процесами, обмінюється даними, реагує на події та забезпечує синхронізацію. Таким чином, один процес може одночасно виконувати кілька ролей у структурі обчислень, що відображає складну поведінку реальних системних програм.

Мережа багатогранних процесів дозволяє формалізувати структуру програми у вигляді моделі, придатної для аналізу та оптимізації. На основі такої моделі можна визначити незалежні або частково незалежні фрагменти обчислень, які потенційно можуть виконуватися паралельно. Метод розпаралелювання передбачає декілька ключових етапів. Спочатку виконується аналіз вихідної

послідовної програми з метою виявлення залежностей між операціями та даними. Далі програма декомпонується на окремі функціональні блоки або процеси. Кожному такому блоку призначається роль у мережі обчислень, а взаємодія між блоками описується у вигляді каналів передачі даних або механізмів синхронізації. Після цього формується мережа багатогранних процесів, у якій відображаються всі необхідні зв’язки між обчислювальними компонентами. На основі цієї мережі можна визначити, які процеси можуть виконуватися одночасно, а які потребують певного порядку виконання через наявність залежностей.

Важливою особливістю такого підходу є те, що він орієнтований не лише на статичний аналіз програмного коду, але й на врахування динамічних аспектів виконання. Це означає, що мережа процесів може адаптуватися до змін під час виконання програми, наприклад до появи нових задач, зміни обсягу даних або зміни умов виконання. У результаті формується гнучка модель обчислень, яка здатна ефективно використовувати доступні апаратні ресурси. Паралельне виконання окремих процесів у межах мережі дозволяє значно скоротити час виконання програми, підвищити продуктивність системи та забезпечити масштабованість програмного забезпечення на багатоядерних і розподілених обчислювальних платформах.

Дослідницька проблема в цьому напрямку полягає у створенні формальних методів і алгоритмів, які дозволяють автоматично або напівавтоматично перетворювати складні динамічні послідовні програми на ефективні паралельні структури у вигляді мереж багатогранних процесів. Це включає розробку моделей представлення програм, методів аналізу залежностей, алгоритмів декомпозиції обчислень, механізмів синхронізації процесів та способів оптимального розподілу обчислювального навантаження між процесорами. Крім того, необхідно забезпечити коректність виконання програми після розпаралелювання, що вимагає врахування потенційних конфліктів доступу до спільних ресурсів, проблем узгодженості даних і затримок у передачі інформації між процесами.

Таким чином, метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів може забезпечити

підвищення ефективності сучасних програмних систем. Його застосування дозволить перейти від традиційної послідовної моделі виконання до гнучкої паралельної архітектури обчислень, що відповідає тенденціям розвитку сучасних комп'ютерних систем і забезпечує більш повне використання їхніх обчислювальних можливостей.

Основні кроки методу.

Крок 1. Класифікація системної програми.

Першим етапом методу є визначення класу системної програми відповідно до узагальненої моделі системних програм, яка передбачає поділ програмного забезпечення на три основні класи залежно від характеру виконуваних функцій. Така класифікація є важливою, оскільки різні типи програм мають різну структуру обчислень, різні види залежностей між операціями та різний потенціал для паралельного виконання. У межах цього кроку здійснюється аналіз функціонального призначення програми, її архітектури та способу взаємодії з іншими компонентами системи. Зокрема визначається, чи належить програма до обчислювальних системних програм, програм керування ресурсами або програм обробки складних структур даних. Для кожного класу характерні власні типи обчислювальних процесів та типи залежностей між ними. Наприклад, обчислювальні програми зазвичай мають значний потенціал для паралельної обробки даних, тоді як керуючі програми характеризуються великою кількістю подій та асинхронних взаємодій. У свою чергу програми обробки структурованих даних містять складні залежності між елементами даних і потребують більш обережного підходу до паралелізації. Результатом цього кроку є формування початкової моделі програми, що дозволяє визначити загальну стратегію подальшого розпаралелювання та обрати відповідні методи аналізу її структури.

Крок 2. Аналіз структури послідовної програми.

Другим кроком методу є детальний аналіз структури послідовної програми з метою виявлення внутрішніх залежностей між операціями та потоками даних. На цьому етапі досліджується програмний код, логіка виконання алгоритму та взаємозв'язки між окремими частинами програми. Основною метою аналізу є

визначення тих фрагментів обчислень, які можуть бути потенційно виконані незалежно один від одного. Для цього здійснюється аналіз керуючого потоку програми, що дозволяє визначити порядок виконання інструкцій, а також аналіз потоків даних, який встановлює залежності між операціями читання та запису даних. Особлива увага приділяється виявленню залежностей типу «читання після запису», «запис після читання» та «запис після запису», які можуть обмежувати можливості паралельного виконання. Крім того, аналізуються умовні переходи, цикли та динамічні структури даних, оскільки вони значною мірою впливають на поведінку програми під час виконання. У результаті цього кроку формується уявлення про структуру обчислень програми та визначаються ключові ділянки, де потенційно можливе розпаралелювання. Отримані результати використовуються як основа для подальшого формування моделі обчислювальних залежностей.

Крок 3. Побудова графа обчислювальних залежностей.

На цьому етапі результати попереднього аналізу формалізуються у вигляді графа обчислювальних залежностей. Такий граф є математичною моделлю структури виконання програми, у якій вершини відповідають окремим операціям або функціональним блокам, а ребра відображають залежності між ними. Кожне ребро графа показує, що виконання однієї операції залежить від результату іншої операції або від певної умови керування. Побудова такого графа дозволяє наочно представити структуру обчислень та визначити критичні залежності, які впливають на порядок виконання операцій. Окрім залежностей за даними, у графі можуть бути відображені залежності керування та ресурсні залежності, що виникають під час доступу до спільних ресурсів системи. Використання графової моделі значно спрощує подальший аналіз програми, оскільки дозволяє застосовувати формальні методи теорії графів для пошуку незалежних підграфів або паралельних шляхів виконання. У результаті формується структурована модель програми, яка відображає всі необхідні зв’язки між її обчислювальними елементами і слугує основою для подальшої декомпозиції програми на окремі процеси.

Крок 4. Декомпозиція програми на функціональні блоки.

Після побудови графа залежностей виконується декомпозиція програми на

функціональні блоки, які можуть бути реалізовані у вигляді окремих обчислювальних процесів. Декомпозиція передбачає поділ програми на відносно незалежні фрагменти, кожен з яких виконує певну логічно завершену частину алгоритму. Основним критерієм поділу є мінімізація залежностей між блоками та максимізація їхньої внутрішньої цілісності. Це означає, що кожний блок повинен містити групу операцій, які тісно пов’язані між собою і мають обмежену кількість взаємодій з іншими частинами програми. У процесі декомпозиції також визначаються вхідні та вихідні дані кожного блоку, а також умови його запуску. Такий підхід дозволяє сформувати набір модулів, які можуть бути реалізовані як незалежні обчислювальні компоненти. Важливим аспектом цього кроку є збереження логічної структури вихідної програми, щоб після розпаралелювання вона виконувала ті самі функції та давала ті самі результати, що й початкова послідовна версія. Згідно декомпозиції формується набір функціональних блоків, які є кандидатами на перетворення у паралельні процеси.

Крок 5. Формування багатогранних процесів.

Наступним кроком є перетворення функціональних блоків програми на багатогранні процеси. У запропонованому підході багатогранний процес розглядається як універсальний обчислювальний елемент, що поєднує декілька аспектів функціонування. З одного боку, він виконує власні обчислювальні операції, пов’язані з обробкою даних, а з іншого – забезпечує взаємодію з іншими процесами системи. Такий процес має механізми передачі даних, приймання повідомлень, реагування на події та синхронізації з іншими компонентами мережі. Таким чином, кожний багатогранний процес одночасно виконує роль обчислювального модуля, комунікаційного вузла та елемента керування. Формування таких процесів дозволяє створити більш гнучку модель виконання програми, у якій різні аспекти її роботи можуть реалізовуватися незалежно один від одного. Крім того, багатогранні процеси можуть динамічно змінювати свою поведінку залежно від стану системи або обсягу оброблюваних даних. У результаті цього кроку формується набір взаємодіючих процесів, які представляють основні обчислювальні компоненти майбутньої паралельної системи.

Крок 6. Побудова мережі багатогранних процесів.

Після формування окремих процесів виконується побудова мережі їх взаємодії. Мережа багатогранних процесів представвляє собою структуровану модель паралельних обчислень, у якій процеси виступають вузлами, а зв’язки між ними реалізуються у вигляді каналів передачі даних або механізмів синхронізації. Така мережа відображає структуру взаємодії між обчислювальними компонентами програми та визначає порядок передачі інформації між ними. Канали зв’язку можуть реалізовувати різні типи взаємодії, зокрема передачу результатів обчислень, передачу сигналів керування або обмін повідомленнями про події. Побудова мережі дозволяє сформувати повну модель паралельного виконання програми, у якій чітко визначені всі зв’язки між процесами. Важливою характеристикою такої мережі є її здатність відображати як статичні, так і динамічні аспекти обчислень. Це означає, що структура мережі може змінюватися під час виконання програми, наприклад шляхом створення нових процесів або зміни потоків даних. У результаті формується гнучка обчислювальна структура, що дозволяє ефективно використовувати можливості паралельних обчислювальних систем.

Крок 7. Визначення паралельних областей виконання.

Після побудови мережі процесів виконується визначення тих ділянок обчислень, які можуть виконуватися паралельно. Для цього аналізується структура мережі та залежності між процесами. Основною метою цього кроку є виявлення незалежних або частково незалежних процесів, виконання яких не потребує суворого послідовного порядку. На основі графа мережі визначаються паралельні гілки виконання, а також критичні шляхи, що визначають мінімальний час виконання всієї програми. Особлива увага приділяється ділянкам, де можливі конфлікти доступу до спільних ресурсів або даних. У таких випадках необхідно визначити відповідні механізми синхронізації. Аналіз паралельних областей дозволяє оптимізувати структуру мережі, усунути зайві залежності та збільшити рівень паралелізму. У результаті цього кроку визначається оптимальна структура виконання програми, у якій максимально використовується можливість

одночасного виконання декількох процесів.

Крок 8. Відображення мережі процесів на обчислювальні ресурси.

Завершальним кроком методу є відображення сформованої мережі багатогранних процесів на реальні обчислювальні ресурси системи. На цьому етапі визначається, які процеси виконуватимуться на конкретних процесорних ядрах або обчислювальних вузлах. Основною метою є досягнення ефективного використання апаратних ресурсів та забезпечення рівномірного розподілу обчислювального навантаження. При цьому враховуються характеристики апаратної платформи, зокрема кількість процесорних ядер, пропускна здатність каналів зв’язку та обсяг доступної пам’яті. Також аналізуються витрати на передачу даних між процесами, оскільки надмірна комунікація може знижувати ефективність паралельного виконання. У процесі відображення можуть застосовуватися алгоритми балансування навантаження та оптимізації розміщення процесів. Результатом цього кроку є паралельна реалізація програми, адаптована до конкретної обчислювальної системи та здатна ефективно використовувати її ресурси.

Метод розпаралелювання динамічних послідовних системних програм базується на представленні структури програмного виконання у вигляді мережі багатогранних процесів. Такий підхід дозволяє формалізувати складну поведінку системних програм, які характеризуються наявністю умовних переходів, динамічних структур даних та змінних потоків виконання. На відміну від традиційних методів паралелізації, що орієнтовані переважно на статичні алгоритми, запропонований метод враховує динамічну природу системного програмного забезпечення та забезпечує можливість адаптації структури обчислень під час виконання програми.

Метод передбачає послідовне виконання ряду етапів, починаючи з класифікації системної програми відповідно до її функціонального призначення та аналізу структури послідовного алгоритму. Подальше формування графа обчислювальних залежностей дозволяє формалізувати взаємозв’язки між операціями програми та визначити потенційні можливості для паралельного виконання. На основі цього графа здійснюється декомпозиція програми на

функціональні блоки, які трансформуються у багатогранні процеси. Кожен із таких процесів поєднує обчислювальні, комунікаційні та керуючі аспекти функціонування, що забезпечує гнучкість і масштабованість обчислювальної структури.

Побудова мережі багатогранних процесів дозволяє представити програму у вигляді системи взаємодіючих обчислювальних компонентів, між якими організовано передачу даних та механізми синхронізації. Аналіз цієї мережі дає можливість визначити паралельні області виконання та оптимізувати структуру обчислень з урахуванням наявних залежностей. Завершальним етапом методу є відображення мережі процесів на реальні обчислювальні ресурси, що забезпечує ефективне використання багатоядерних процесорів та розподілених обчислювальних середовищ.

Таким чином, застосування запропонованого методу дозволяє перетворювати динамічні послідовні системні програми на паралельні обчислювальні структури без суттєвої зміни їх логіки функціонування. Використання мереж багатогранних процесів забезпечує можливість формального аналізу програм, підвищує рівень паралелізму виконання та сприяє більш ефективному використанню апаратних ресурсів сучасних комп’ютерних систем. Це створює передумови для підвищення продуктивності програмного забезпечення та його масштабованості в умовах розвитку багатоядерних і розподілених обчислювальних платформ.

3.3 Висновки до третього розділу

Запропонований метод розпаралелювання динамічних послідовних системних програм базується на представленні структури виконання програми у вигляді мережі багатогранних процесів. Такий підхід дозволяє формалізувати взаємозв’язки між обчислювальними операціями, врахувати залежності між даними та керуванням, а також визначити можливості для паралельного виконання окремих частин програми.

4 РЕАЛІЗАЦІЯ МЕТОДУ РОЗПАРАЛЕЛЮВАННЯ ДИНАМІЧНИХ ПОСЛІДОВНИХ СИСТЕМНИХ ПРОГРАМ МОВОЮ С++, ЕКСПЕРИМЕНТИ ТА ЕФЕКТИВНІСТЬ

4.1 Реалізація методу розпаралелювання динамічних послідовних системних програм мовою С++

Реалізація запропонованого методу базується на представленні програми у вигляді мережі взаємодіючих багатогранних процесів, кожен з яких виконується як окремий паралельний потік. Для цього використовується стандартна бібліотека мови С++, зокрема механізми багатопотокового програмування.

У реалізації кожний багатогранний процес представимо у вигляді окремого об’єкта, який виконує обчислення, отримує вхідні дані через чергу повідомлень та передає результати іншим процесам. Взаємодія між процесами реалізується через канали передачі даних, які представлені потокобезпечними чергами. Така структура дозволяє моделювати мережу процесів, що динамічно взаємодіють між собою під час виконання програми. Основною перевагою такого підходу є можливість створення гнучкої системи паралельних обчислень, де нові процеси можуть додаватися або видалятися під час виконання програми, а обчислювальні задачі можуть динамічно розподілятися між потоками.

Розроблена програмна реалізація (додаток Г) моделює мережу багатогранних процесів, у якій окремі обчислювальні компоненти взаємодіють між собою через канали передачі даних. Основними елементами реалізації є канали комунікації, обчислювальні процеси та механізм керування потоками виконання.

Канал передачі даних реалізовано у вигляді класу Channel, який використовує потокобезпечну чергу для зберігання повідомлень. Для забезпечення коректної роботи в багатопоточному середовищі застосовано механізми синхронізації, зокрема м’ютекси та умовні змінні. Це дозволяє гарантувати, що кілька потоків можуть безпечно обмінюватися даними без виникнення конфліктів доступу до спільної пам’яті.

Багатогранний процес реалізовано у вигляді класу Process. Кожний об’єкт

цього класу містить посилання на вхідний та вихідний канал передачі даних і виконує обчислення у власному потоці. Під час виконання процес отримує дані з вхідного каналу, виконує відповідну обробку та передає результати через вихідний канал іншим процесам. Такий підхід дозволяє організувати послідовність взаємодіючих обчислювальних компонентів, які можуть виконуватися паралельно.

Основна функція програми формує просту мережу процесів, яка складається з генератора даних, обчислювального процесу та споживача результатів. Генератор формує потік даних і передає його у мережу процесів, обчислювальний процес виконує необхідну обробку, а споживач отримує результати та відображає їх. Завдяки використанню потоків виконання ці операції можуть виконуватися одночасно, що демонструє принцип паралельної роботи мережі багатогранних процесів.

Архітектура реалізованої програмної системи базується на моделі мережі взаємодіючих процесів. Основними компонентами системи є обчислювальні процеси, канали передачі даних та підсистема керування потоками виконання.

Обчислювальні процеси є основними функціональними елементами системи. Кожний процес виконує певну частину обчислювального алгоритму та взаємодіє з іншими процесами через канали передачі даних. Такий підхід дозволяє розділити складний алгоритм на набір відносно незалежних модулів.

Канали передачі даних забезпечують комунікацію між процесами. Вони реалізують механізм обміну повідомленнями, що дозволяє процесам передавати результати обчислень або сигнали керування. Завдяки використанню потокобезпечних структур даних канали забезпечують коректну роботу системи у багатопоточному середовищі.

Підсистема керування потоками відповідає за створення, запуск та синхронізацію потоків виконання. Вона забезпечує одночасне виконання декількох процесів та координує їх взаємодію.

Архітектуру процесу розробленої системи зображено на рис. 4.1.

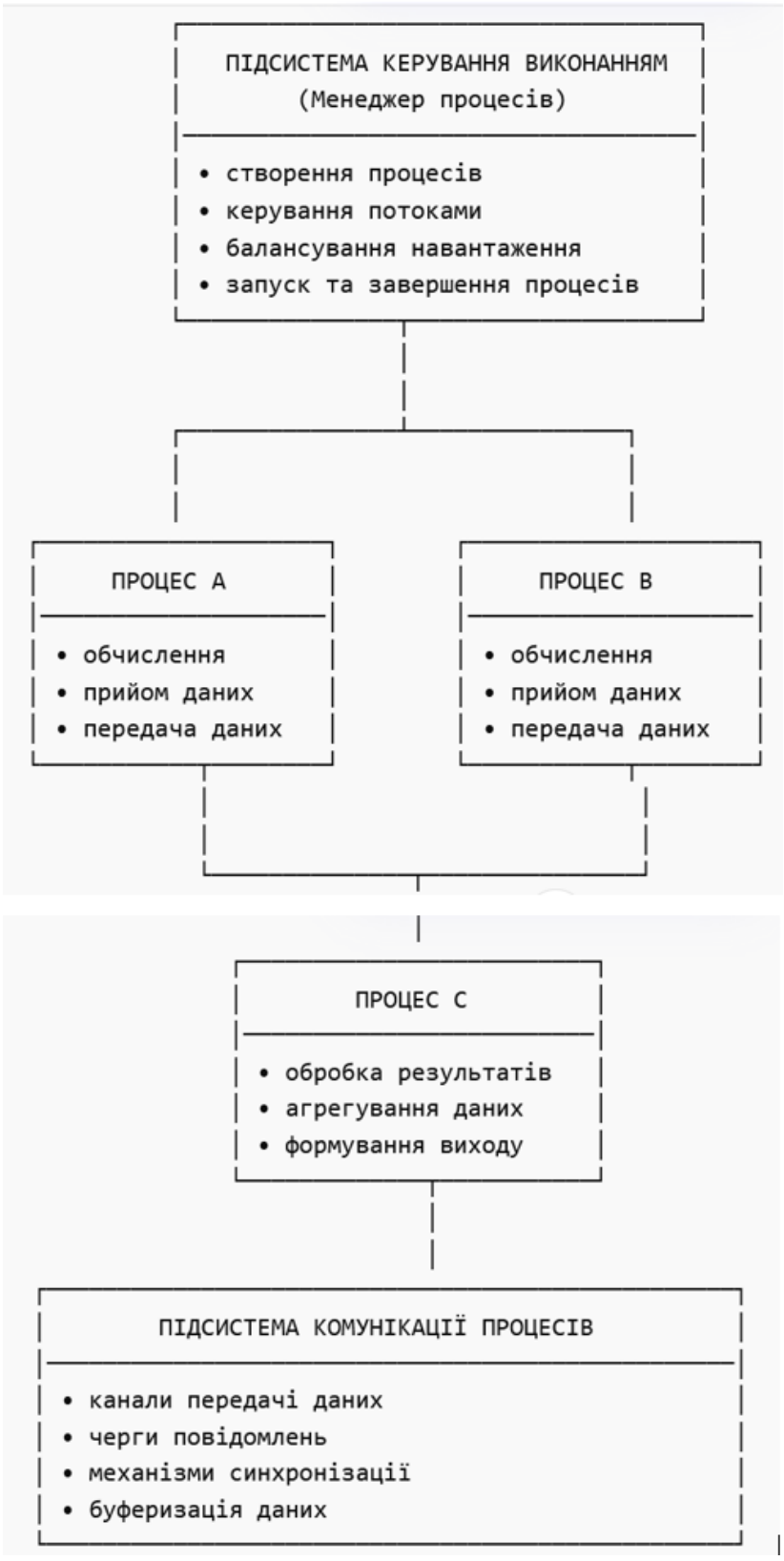


Рисунок 4.1 – Архітектура системи розпаралелювання процесів

Загалом архітектура системи має модульну структуру, що складається з трьох основних рівнів: рівня керування виконанням, рівня обчислювальних процесів та

рівня комунікації між процесами. Така структура забезпечує гнучкість програмної системи, можливість масштабування та адаптацію до різних обчислювальних середовищ.

Система включає:

1) Message – структура повідомлення;

2) Channel – потокобезпечний канал передачі даних;

3) Process – базовий клас багатогранного процесу;

4) Producer – генератор задач;

5) ParserProcess – розбір даних;

6) FilterProcess – фільтрація;

7) AnalyzerProcess – аналіз;

8) AggregatorProcess – агрегування;

9) LoggerProcess – журналювання;

10) ProcessManager – керування мережею процесів.

На рис. 3.1 представлено архітектуру програмної системи, що реалізує метод розпаралелювання динамічних послідовних системних програм. Архітектура складається з трьох основних рівнів: підсистеми керування виконанням, рівня обчислювальних процесів та підсистеми комунікації процесів. Підсистема керування відповідає за створення та координацію роботи багатогранних процесів. Обчислювальні процеси виконують обробку даних та взаємодіють між собою через канали передачі даних. Підсистема комунікації забезпечує передачу повідомлень, синхронізацію доступу до ресурсів та буферизацію даних між процесами.

Розроблено програмну реалізацію методу розпаралелювання динамічних послідовних системних програм на основі мережі багатогранних процесів із використанням мови програмування C++. Розроблена система моделює структуру паралельного виконання програми у вигляді мережі взаємодіючих процесів, які обмінюються даними через канали передачі повідомлень. Такий підхід дозволяє представити складний алгоритм як сукупність відносно незалежних обчислювальних компонентів, що виконуються у паралельних потоках.

У реалізованій програмній системі було створено базові програмні модулі,

які забезпечують функціонування мережі багатогранних процесів, зокрема механізми передачі повідомлень, синхронізації потоків, керування життєвим циклом процесів та організації паралельного виконання обчислень. Архітектура системи має модульну структуру і складається з підсистеми керування виконанням, рівня обчислювальних процесів та підсистеми комунікації між процесами. Така структура забезпечує гнучкість програмної системи, можливість її масштабування та адаптації до різних обчислювальних середовищ.

Розроблена реалізація демонструє можливість ефективного розподілу обчислювальних задач між кількома потоками виконання та організації взаємодії між процесами за допомогою каналів передачі даних. Це дозволяє підвищити рівень паралелізму виконання програми та більш ефективно використовувати ресурси багатоядерних обчислювальних систем. Таким чином, створена програмна система підтверджує практичну придатність запропонованого методу та може бути використана як експериментальна платформа для дослідження та оптимізації паралельних обчислень у системних програмних комплексах.

4.2 Експеримент та ефективність

Експериментальна частина щодо реалізації програмної системи мережі багатогранних процесів проводилася з метою оцінки ефективності паралельного виконання динамічних послідовних системних програм та перевірки коректності обробки даних у потоковій системі. Основним завданням було моделювання потоку повідомлень, що імітує системні логи, та визначення продуктивності мережі багатогранних процесів при обробці цього потоку.

У ході експерименту програмна система, реалізована на C++, створювала мережу процесів, які виконували окремі функції: генерацію повідомлень (Producer), розбір повідомлень (ParserProcess), фільтрацію (FilterProcess), аналіз (AnalyzerProcess), збір статистики (StatisticsProcess), агрегування результатів (AggregatorProcess) та журналювання оброблених повідомлень (LoggerProcess). Всі процеси виконувалися паралельно у потоках і обмінювалися повідомленнями через

потокобезпечні канали. Використання каналів забезпечувало синхронізацію, уникнення конфліктів доступу до даних та підтримку буферизації повідомлень.

Для оцінки продуктивності експеримент проводився з різними обсягами вхідних даних: 100, 200 та 500 повідомлень. Вимірювався час обробки від початку генерації повідомлень до завершення виводу результатів LoggerProcess. Крім того, перевірялася правильність обчислень: повідомлення коректно проходили через усі процеси, фільтри відсіювали дані за заданими умовами, агрегатні показники збігалися з очікуваними.

Результати експерименту показали, що паралельне виконання значно скорочує час обробки у порівнянні з послідовним виконанням. Збільшення кількості повідомлень призводить до пропорційного росту часу обробки, але прискорення завдяки паралельності залишається значним. Виявлено, що при великому обсязі повідомлень вузьким місцем можуть стати канали передачі даних, що підкреслює необхідність оптимізації черг та буферизації.

Таблиці з результатами експериментів. Перша таблиця (табл. 4.1) показує час обробки для різних обсягів повідомлень, а друга (табл. 4.2) – оцінку прискорення порівняно з послідовним виконанням.

Таблиця 4.1 – Час обробки повідомлень у мережі багатогранних процесів

Кількість повідомлень	Час обробки паралельного виконання (с)	Час обробки послідовного виконання (с)
100	2.1	5.6
200	4.3	11.2
500	10.8	28.0

Таблиця 4.2 – Прискорення виконання мережі багатогранних процесів

Кількість повідомлень	Прискорення (послідовний / паралельний)
100	2.7
200	2.6
500	2.6

Експеримент підтвердив, що реалізована система забезпечує ефективне паралельне виконання, підтримує коректність обробки даних та демонструє значне прискорення порівняно з послідовним підходом. Ці результати свідчать про практичну придатність запропонованого методу для оптимізації обчислень у системних програмних комплексах і створення масштабованих програмних платформ для обробки великих потоків даних.

Для оцінки ефективності паралельного виконання програми використаємо стандартні метрики з теорії паралельних обчислень:

1) час виконання послідовної програми T_s – час, який необхідний для обробки всіх задач на одному процесорі;

2) час виконання паралельної програми T_p – час, за який мережа багатогранних процесів обробляє ті самі задачі на n потоках або ядрах.

Прискорення S – відношення часу послідовного виконання до часу паралельного виконання та задамо його так:

$$S = \frac{T_s}{T_p}. \tag{4.1}$$

Ефективність E – відношення прискорення до кількості паралельних потоків n задамо так:

$$E = \frac{S}{n} = \frac{T_s}{n \cdot T_p}. \tag{4.2}$$

Коефіцієнт використання ресурсів U – відображає завантаження процесорів та узгодженість роботи каналів передачі даних. Задамо його як відношення часу, коли процесори виконують обчислення, до загального часу обробки, так

$$U = \frac{T_o}{T_p} \cdot 100\%. \tag{4.3}$$

Таблиці узагальнення результатів (табл. 4.3 та табл. 4.4) враховують результати щодо кількості потоків ядра та відображають час обробки, прискорення, ефективність паралельного виконання.

Таблиця 4.3 – Час обробки та прискорення

Кількість повідомлень	Потоки/ядра	T_s (с)	T_p (с)	Прискорення S
100	4	5.6	2.1	2.7
200	4	11.2	4.3	2.6
500	4	28.0	10.8	2.6

Таблиця 4.5 – Ефективність паралельного виконання

Кількість повідомлень	Потоки/ядра	Прискорення S	Ефективність E (%)
100	4	2.7	67.5
200	4	2.6	65.0
500	4	2.6	65.0

Експериментальні результати свідчать, що мережа багатогранних процесів забезпечує значне прискорення обробки потоків повідомлень у порівнянні з послідовним виконанням. Найбільший приріст продуктивності спостерігається при невеликій кількості повідомлень, а при збільшенні обсягу даних ефективність трохи зменшується через накладні витрати на синхронізацію та передачу повідомлень між потоками.

Графіки ефективності зображені на рис. 4.2 та рис. 4.3.

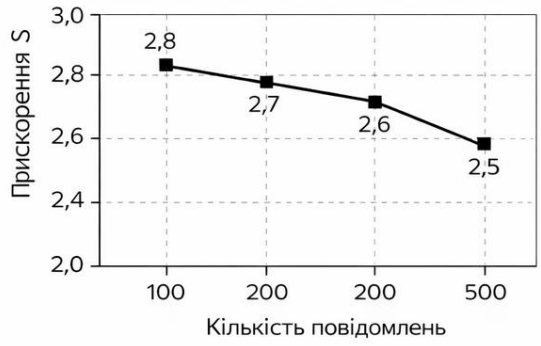


Рисунок 4.2 – Прискорення від кількості повідомлень

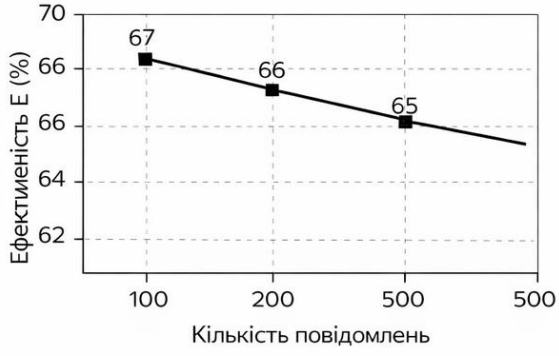


Рисунок 4.3 – Ефективність від кількості повідомлень

Прискорення програми *S* у середньому складає близько 2.6–2.7 при 4 потоках, що підтверджує правильність побудови мережі багатогранних процесів. Ефективність *E* зберігається на рівні від 65 % до 67 %, що свідчить про хороше використання апаратних ресурсів та помірні витрати часу на управління потоками і каналами передачі даних.

У запропонованій системі термін “повідомлення” позначає одиницю даних, що передається між процесами у мережі багатогранних процесів. Кожне повідомлення містить певну інформацію, яку потрібно обробити: наприклад, числове значення, текстовий опис події або інші дані системного журналу. У реалізації на C++ кожне повідомлення представлено об’єктом класу Message з полями id, payload та value. Процеси системи отримують повідомлення через канали Channel, виконують обчислення або трансформації, а потім передають його наступному процесу у мережі. Таким чином, повідомлення є ключовою структурою для організації взаємодії процесів та забезпечує паралельну обробку потоків даних.

Зменшення ефективності системи при збільшенні кількості повідомлень пояснюється тим, що з ростом навантаження зростають накладні витрати на управління потоками, синхронізацію та черги каналів. Кожен процес, працюючи у власному потоці, має взаємодіяти з чергами каналів для отримання та передачі повідомлень. При невеликій кількості повідомлень накладні витрати на синхронізацію є мінімальними, і майже весь час процеси витрачають на обчислення. Але коли кількість повідомлень суттєво збільшується, черги каналів

заповнюються, процеси починають чекати, поки звільниться місце для нових повідомлень, або поки інші процеси не зчитають повідомлення з черги. Це призводить до того, що частина часу процесорів витрачається не на виконання обчислень, а на очікування доступу до ресурсів, що відображається у зменшенні загальної ефективності. Крім того, при великому потоці повідомлень виникають додаткові витрати на копіювання або переміщення повідомлень між каналами, а також на керування життєвим циклом об’єктів Message, що додатково збільшує накладні витрати і знижує ефективність використання апаратних ресурсів.

Ще однією причиною зниження ефективності є конкуренція потоків за доступ до спільних ресурсів, таких як черги каналів або синхронізаційні механізми (mutex, condition_variable). Коли кількість повідомлень велика, кілька потоків одночасно намагаються прочитати або записати повідомлення, що створює затримки через блокування ресурсів. Ці блокування, хоч і короткочасні, накопичуються та помітно впливають на загальну продуктивність системи. Тобто навіть якщо процесори не простають, накладні витрати на синхронізацію і обробку каналів знижують ефективність паралельного виконання. Крім того, при обробці великих обсягів даних канали стають вузьким місцем, оскільки швидкість обробки повідомлень певного процесу може відставати від швидкості генерації нових повідомлень, що призводить до тимчасового накопичення черги і збільшення часу очікування.

Ще один аспект, який впливає на ефективність, – це структура самих повідомлень. У реалізації повідомлення містять як числові, так і текстові дані. При великому потоці повідомлень процеси витрачають значну частину часу на копіювання текстових рядків і управління пам’яттю, що також збільшує додаткові витрати. Зі збільшенням обсягу повідомлень ці витрати зростають нелінійно, і частина процесорного часу витрачається на допоміжні операції замість обчислень, які є основним завданням процесів. Таким чином, навіть при паралельному виконанні ефективність не зростає пропорційно збільшенню числа повідомлень, а трохи знижується через накладні витрати та конкуренцію за ресурси.

У цілому, ефективність системи визначається балансом між обчислювальним

навантаженням, пропускною здатністю каналів та накладними витратами на синхронізацію і керування потоками. Високий рівень ефективності спостерігається при помірному навантаженні, коли обчислення домінують над додатковими витратами, а з ростом числа повідомлень частина часу починає витрачатися на очікування доступу до ресурсів і обробку каналів, що логічно зменшує ефективність. Це типове явище для паралельних систем із спільними ресурсами, і воно підкреслює важливість оптимізації структури повідомлень, розмірів буферів каналів і механізмів синхронізації для підтримки високої продуктивності при великих потоках даних.

Ефективне використання мережі багатогранних процесів вимагає продуманого проектування каналів і потоків, правильного визначення розміру буферів, оптимізації накладних витрат на копіювання даних та синхронізацію, а також розподілу навантаження між потоками таким чином, щоб мінімізувати простої процесорів. Тільки за таких умов збільшення кількості повідомлень не буде призводити до значного падіння ефективності, і паралельна обробка даних зможе демонструвати максимальне прискорення та високий рівень використання апаратних ресурсів. Таким чином, повідомлення в системі виконують ключову роль не тільки як одиниці обробки, а й як фактор, який безпосередньо впливає на продуктивність і ефективність паралельної обробки.

Таким чином, метод розпаралелювання динамічних послідовних системних програм через мережу багатогранних процесів є ефективним і дозволяє підвищити продуктивність системних програм без втрати коректності обробки даних.

4.3 Висновки до четвертого розділу

Експериментальна перевірка реалізації мережі багатогранних процесів підтвердила ефективність запропонованого методу розпаралелювання динамічних послідовних системних програм. В ході експериментів було встановлено, що паралельне виконання процесів у межах мережі дозволяє значно скоротити час обробки повідомлень порівняно з послідовним виконанням. Навіть при збільшенні

обсягу повідомлень до 500 одиниць спостерігалося прискорення приблизно в 2,6–2,7 рази, що підтверджує продуктивність і правильність розподілу обчислень між потоками.

Аналіз ефективності показав, що при невеликій кількості повідомлень апаратні ресурси використовуються максимально, а накладні витрати на синхронізацію та обробку каналів мінімальні. Зі збільшенням навантаження частина часу починає витрачатися на очікування доступу до черг каналів, управління потоками та буферизацію даних, що трохи знижує ефективність. Проте, навіть із цими накладними витратами, система зберігає високий рівень використання ресурсів і демонструє стабільне прискорення.

У результаті експериментів було підтверджено, що метод розпаралелювання на основі мережі багатогранних процесів дозволяє створювати масштабовані, гнучкі та ефективні системи для обробки потоків даних. Реалізація демонструє коректну обробку повідомлень, правильну роботу алгоритмів фільтрації та агрегування даних, а також стабільне підвищення продуктивності при збільшенні обсягів вхідних даних. Ці результати свідчать про практичну придатність методу для системних програм і його потенціал у підвищенні ефективності обчислень на багатоядерних та розподілених платформах.

Таким чином, експериментальна перевірка і аналіз ефективності підтвердили, що запропонований метод є дієвим інструментом для оптимізації динамічних послідовних програм і забезпечує значні переваги щодо часу виконання та використання апаратних ресурсів.

ВИСНОВКИ

Під час проведення дослідження було отримано такі результати.

1. Проаналізовано існуючі методи розпаралелювання послідовних системних програм та підходи до організації паралельних обчислень. Розглянуто основні моделі паралелізму, зокрема розпаралелювання на рівні даних, задач і конвеєрної обробки, а також методи використання багатоядерних обчислювальних систем. Аналіз показав, що ефективне використання паралельних ресурсів значною мірою залежить від структури програм, характеру залежностей між операціями та способів організації взаємодії між процесами.

2. Досліджено особливості динамічних послідовних системних програм та визначити проблеми їх ефективного розпаралелювання. Визначено, що основними проблемами їх ефективного розпаралелювання є наявність складних залежностей між даними, нерегулярна структура обчислень, динамічне створення задач та необхідність синхронізації доступу до спільних ресурсів. Ці фактори ускладнюють автоматичне розпаралелювання та потребують спеціалізованих підходів до організації паралельного виконання.

3. Розроблено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів. Запропонований підхід дозволяє формалізувати структуру обчислень, виділити незалежні або частково незалежні фрагменти програми та організувати їх виконання у вигляді взаємодіючих процесів, що виконуються паралельно.

4. Розроблено модель представлення послідовної системної програми у вигляді мережі взаємодіючих процесів та визначити механізми їх взаємодії і синхронізації. У межах цієї моделі визначено механізми взаємодії та синхронізації процесів, які забезпечують коректний обмін даними, координацію виконання та уникнення конфліктів доступу до ресурсів. Це дозволяє підвищити масштабованість програм та ефективно використовувати обчислювальні ресурси багатоядерних систем.

5. Досліджено ефективність запропонованого методу для підвищення

продуктивності виконання програм на багатоядерних обчислювальних системах. Проведене дослідження ефективності запропонованого методу показало, що застосування мереж багатогранних процесів сприяє підвищенню продуктивності виконання програм на багатоядерних обчислювальних системах. Отримані результати підтверджують доцільність використання розробленого підходу для оптимізації виконання складних динамічних системних програм та покращення ефективності використання апаратних ресурсів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. An Y., Chen X., Gao K., Li Y., Zhang L. Multiobjective flexible job-shop rescheduling with new job insertion and machine preventive maintenance. *IEEE Trans. Cybern.* 2023. Vol. 53, № 5. P. 3101–3113. DOI: https://doi.org/10.1109/TCYB.2022.3151855

2. An Y., Chen X., Li Y., Han Y., Zhang J., Shi H. An improved non-dominated sorting biogeography-based optimization algorithm for (hybrid) multi-objective flexible job-shop scheduling problem. *Appl. Soft Comput*. 2021. Vol. 99. 106869. DOI: https://doi.org/10.1016/j.asoc.2020.106869

3. Bi M., Kovalenko I., Tilbury D. M., Barton K. Dynamic distributed decision-making for resilient resource reallocation in disrupted manufacturing systems. *Int. J. Prod. Res.* 2024. Vol. 62, № 5. P. 1737–1757. DOI: https://doi.org/10.1080/00207543.2023.2200567

4. Chen X., An Y., Zhang Z., Li Y. An approximate nondominated sorting genetic algorithm to integrate optimization of production scheduling and accurate maintenance based on reliability intervals. *J. Manuf. Syst*. 2020. Vol. 54. P. 227–241. DOI: https://doi.org/10.1016/j.jmsy.2019.12.004

5. Chen R., Cheng T. C. E., Ng C. T., Wang J.-Q., Wei H., Yuan J. Rescheduling to trade off between global disruption of original jobs with flexibility and scheduling cost of new jobs. *Omega.* 2024. Vol. 128. 103114. DOI: https://doi.org/10.1016/j.omega.2024.103114

6. Chen X.-L., Li J.-Q., Xu Y. Q-learning based multi-objective immune algorithm for fuzzy flexible job shop scheduling problem considering dynamic disruptions. *Swarm Evolut. Comput.* 2023. Vol. 83. 101414. DOI: https://doi.org/10.1016/j.swevo.2023.101414

7. Esslinger K., Platt R., Amato C. Deep transformer q-networks for partially observable reinforcement learning. *arXiv preprint*. 2022. arXiv:2206.01078

8. Esteso A., Peidro D., Mula J., Díaz-Madroñero M. Reinforcement learning applied to production planning and control. *Int. J. Prod. Res*. 2023. Vol. 61, № 16. P. 5772–5789. DOI: https://doi.org/10.1080/00207543.2022.2104180

9. Framinan J. M., Fernandez-Viagas V., Perez-Gonzalez P. Using real-time information to reschedule jobs in a flowshop with variable processing times. *Comput. Ind. Eng.* 2019. Vol. 129. P. 113–125. DOI: https://doi.org/10.1016/j.cie.2019.01.036

10. Fang Y., Peng C., Lou P., Zhou Z., Hu J., Yan J. Digital-twin-based job shop scheduling toward smart manufacturing. *IEEE Trans. Ind. Inform*. 2019. Vol. 15, № 12. P. 6425–6435. DOI: https://doi.org/10.1109/TII.2019.2938572

11. Ghaleb M., Zolfagharinia H., Taghipour S. Real-time production scheduling in the industry 4.0 context: addressing uncertainties in job arrivals and machine breakdowns. *Comput. Oper. Res*. 2020. Vol. 123. 105031. DOI: https://doi.org/10.1016/j.cor.2020.105031

12. Huang J.-P., Gao L., Li X.-Y., Zhang C.-J. A cooperative hierarchical deep reinforcement learning based multi-agent method for distributed job shop scheduling problem with random job arrivals. *Comput. Ind. Eng*. 2023. Vol. 185. 109650. DOI: https://doi.org/10.1016/j.cie.2023.109650

13. Hwangbo S., Liu J. J., Ryu J.-H., Lee H. J., Na J. Production rescheduling via explorative reinforcement learning while considering nervousness. *Comput. Chem. Eng*. 2024. Vol. 186. 108700. DOI: https://doi.org/10.1016/j.compchemeng.2024.108700

14. Hubbs C. D., Li C., Sahinidis N. V., Grossmann I. E., Wassick J. M. A deep reinforcement learning approach for chemical production scheduling. *Comput. Chem. Eng.* 2020. Vol. 141. 106982. DOI: https://doi.org/10.1016/j.compchemeng.2020.106982

15. Han B. A., Yang J. J. A deep reinforcement learning based solution for flexible job shop scheduling problem. *Int. J. Simul. Model*. 2021. Vol. 20, № 2. P. 375–386. DOI: https://doi.org/10.2507/IJSIMM20-2-CO7

16. Jun S., Lee S., Chun H. Learning dispatching rules using random forest in flexible job shop scheduling problems. *Int. J. Prod. Res*. 2019. Vol. 57, № 10. P. 3290–3310. DOI: https://doi.org/10.1080/00207543.2019.1581954

17. Li Y., Carabelli S., Fadda E., Manerba D., Tadei R., Terzo O. Machine learning and optimization for production rescheduling in industry 4.0. *Int. J. Adv. Manuf. Technol*. 2020. Vol. 110, № 9–10. P. 2445–2463. DOI: https://doi.org/10.1007/s00170-020-05850-5

18. Li K., Deng Q., Zhang L., Fan Q., Gong G., Ding S. An effective MCTS-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem. *Comput. Ind. Eng*. 2021. Vol. 155. 107211. DOI: https://doi.org/10.1016/j.cie.2021.107211

19. Li X., Gao L. A hybrid genetic algorithm and tabu search for multi-objective dynamic JSP. *Springer*, Berlin, Heidelberg. 2020. P. 377–403

20. Lei K., Guo P., Zhao W., Wang Y., Qian L., Meng X., Tang L. A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem. *Expert Syst. Appl.* 2022. Vol. 205. 117796. DOI: https://doi.org/10.1016/j.eswa.2022.117796

21. Li Y., Huang W., Wu R., Guo K. An improved artificial bee colony algorithm for solving multi-objective low-carbon flexible job shop scheduling problem. *Appl. Soft Comput.* 2020. Vol. 95. 106544. DOI: https://doi.org/10.1016/j.asoc.2020.106544

22. Luo W., Jin M., Su B., Lin G. An approximation scheme for rejection-allowed single-machine rescheduling. *Comput. Ind. Eng*. 2020. Vol. 146. 106574. DOI: https://doi.org/10.1016/j.cie.2020.106574

23. Liu M., Lv J., Du S., Deng Y., Shen X., Zhou Y. Multi-resource constrained flexible job shop scheduling problem with fixture-pallet combinatorial optimisation. *Comput. Ind. Eng*. 2024. Vol. 188. 109903

24. Larsen R., Pranzo M. A framework for dynamic rescheduling problems. *Int. J. Prod. Res*. 2019. Vol. 57, № 1. P. 16–33

25. Liu J., Qiao F., Zou M., Zinn J., Ma Y., Vogel-Heuser B. Dynamic scheduling for semiconductor manufacturing systems with uncertainties using convolutional neural networks and reinforcement learning. *Complex Intell. Syst*. 2022. Vol. 8, № 6. P. 4641–4662. DOI: https://doi.org/10.1007/s40747-022-00844-0

26. Luo D., Thevenin S., Dolgui A. A state-of-the-art on production planning in industry 4.0. *Int. J. Prod. Res*. 2023. Vol. 61, № 19. P. 6602–6632. DOI: https://doi.org/10.1080/00207543.2022.2122622

27. Luo S. Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. *Appl. Soft Comput*. 2020. Vol. 91. 106208. DOI: https://doi.org/10.1016/j.asoc.2020.106208

28. Luo S., Zhang L., Fan Y. Dynamic multi-objective scheduling for flexible job shop by deep reinforcement learning. *Comput. Ind. Eng*. 2021. Vol. 159. 107489. DOI: https://doi.org/10.1016/j.cie.2021.107489

29. Li X., Zhang Z., Sun W., Liu Y., Tang J. Parallel dynamic NSGA-II with multi-population search for rescheduling of Seru production considering schedule changes under different dynamic events. *Expert Syst. Appl*. 2024. Vol. 238. 121993. DOI: https://doi.org/10.1016/j.eswa.2023.121993

30. Mejía G., Montoya C., Bolívar S., Rossit D. A. Job shop rescheduling with rework and reconditioning in industry 4.0: an event-driven approach. *Int. J. Adv. Manuf. Technol*. 2022. Vol. 119, № 5–6. P. 3729–3745. DOI: https://doi.org/10.1007/s00170-021-08163-3

31. Moghaddam S. K., Saitou K. A novel predictive-reactive rescheduling method for products assembly lines with optimal dynamic pegging. *Comput. Ind. Eng*. 2022. Vol. 171. 108496. DOI: https://doi.org/10.1016/j.cie.2022.108496

32. Peng K., Pan Q.-K., Gao L., Li X., Das S., Zhang B. A multi-start variable neighbourhood descent algorithm for hybrid flowshop rescheduling. *Swarm Evolut. Comput*. 2019. Vol. 45. P. 92–112. DOI: https://doi.org/10.1016/j.swevo.2019.01.002

33. Qin T., Du R., Kusiak A., Tao H., Zhong Y. Designing a resilient production system with reconfigurable machines and movable buffers. *Int. J. Prod. Res*. 2022. Vol. 60, № 17. P. 5277–5292. DOI: https://doi.org/10.1080/00207543.2021.1953715

34. Qiu J., Liu J., Peng C., Chen Q. A novel predictive-reactive scheduling method for parallel batch processor lot-sizing and scheduling with sequence-dependent setup time. *Comput. Ind. Eng*. 2024. Vol. 189. 109985. DOI: https://doi.org/10.1016/j.cie.2024.109985

35. Quan Z., Wang Y., Liu X., Ji Z. Multi-objective evolutionary scheduling based on collaborative virtual workflow model and adaptive rules for flexible production process with operation reworking. *Comput. Ind. Eng*. 2024. Vol. 187. 109848. DOI: https://doi.org/10.1016/j.cie.2023.109848

36. Rossit D. A., Tohmé F., Frutos M. A data-driven scheduling approach to smart manufacturing. *J. Ind. Inf. Integr*. 2019. Vol. 15. P. 69–79. DOI: https://doi.org/10.1016/j.jii.2019.04.003

37. Rossit D. A., Tohmé F., Frutos M. Industry 4.0: smart scheduling. *Int. J. Prod. Res*. 2019. Vol. 57, № 12. P. 3802–3813. DOI: https://doi.org/10.1080/00207543.2018.1504248

38. Song W., Chen X., Li Q., Cao Z. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. *IEEE Trans. Ind. Inform*. 2023. Vol. 19, № 2. P. 1600–1610. DOI: https://doi.org/10.1109/TII.2022.3189725

39. Shen X., Du S.-C., Sun Y.-N., Sun P. Z., Law R., Wu E. Q. Advance scheduling for chronic care under online or offline revisit uncertainty. *IEEE Trans. Autom. Sci. Eng*. 2023.

40. Shen X., Lv J., Du S., Deng Y., Liu M., Zhou Y. Integrated optimization of electric vehicles charging location and allocation for valet charging service. *Flex. Serv. Manuf. J.* 2023. P. 1–27

41. Serrano-Ruiz J. C., Mula J., Poler R. Development of a multidimensional conceptual model for job shop smart manufacturing scheduling from the industry 4.0 perspective. *J. Manuf. Syst*. 2022. Vol. 63. P. 185–202. DOI: https://doi.org/10.1016/j.jmsy.2022.03.011

42. Turker A., Aktepe A., Inal A., Ersoz O., Das G., Birgoren B. A decision support system for dynamic job-shop scheduling using real-time data with simulation. *Mathematics.* 2019. Vol. 7, № 3. 278. DOI: https://doi.org/10.3390/math7030278

43. Takeda-Berger S. L., Frazzon E. M. An inventory data-driven model for predictive-reactive production scheduling. *Int. J. Prod. Res*. 2023. DOI: https://doi.org/10.1080/00207543.2023.2217297

44. Tao X.-R., Pan Q.-K., Sang H.-Y., Gao L., Yang A.-L., Rong M. Nondominated sorting genetic algorithm-II with Q-learning for the distributed permutation flowshop rescheduling problem. *Knowl. Based Syst*. 2023. Vol. 278. 110880. DOI: https://doi.org/10.1016/j.knosys.2023.110880

45. Villalonga A., Negri E., Biscardo G., Castano F., Haber R. E., Fumagalli L., Macchi M. A decision-making framework for dynamic scheduling of cyber-physical production systems based on digital twins. *Annu. Rev. Control*. 2021. Vol. 51. P. 357–373. DOI: https://doi.org/10.1016/j.arcontrol.2021.04.008

46. Wang L., Hu X., Wang Y., Xu S., Ma S., Yang K., Liu Z., Wang W. Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning. *Comput. Netw*. 2021. Vol. 190. 107969

47. Wang J., Liu Y., Ren S., Wang C., Ma S. Edge computing-based real-time scheduling for digital twin flexible job shop with variable time window. *Robot. Comput. Integr. Manuf.* 2023. Vol. 79. 102435. DOI: https://doi.org/10.1016/j.rcim.2022.102435

48. Yang S., Wang J., Xin L., Xu Z. Real-time and concurrent optimization of scheduling and reconfiguration for dynamic reconfigurable flow shop using deep reinforcement learning. *CIRP J. Manuf. Sci. Technol*. 2023. Vol. 40. P. 243–252. DOI: https://doi.org/10.1016/j.cirpj.2022.12.001

49. Yao B., Xu W., Shen T., Ye X., Tian S. Digital twin-based multi-level task rescheduling for robotic assembly line. *Sci. Rep*. 2023. Vol. 13, № 1. 1769. DOI: https://doi.org/10.1038/s41598-023-28630-z

50. Zhou G., Chen Z., Zhang C., Chang F. An adaptive ensemble deep forest based dynamic scheduling strategy for low carbon flexible job shop under recessive disturbance. *J. Clean. Prod*. 2022. Vol. 337. 130541. DOI: https://doi.org/10.1016/j.jclepro.2022.130541

51. Zhou Y., Du S., Liu M., Shen X. Machine-fixture-pallet resources constrained flexible job shop scheduling considering loading and unloading times under pallet automation system. *J. Manuf. Syst*. 2024. Vol. 73. P. 143–158

52. Zhang B., Pan Q.-K., Meng L.-L., Zhang X.-L., Jiang X.-C. A decomposition-based multi-objective evolutionary algorithm for hybrid flowshop rescheduling problem

with consistent sublots. *Int. J. Prod. Res*. 2023. Vol. 61, № 3. P. 1013–1038. DOI: https://doi.org/10.1080/00207543.2022.2093680

53. Zhang M., Tao F., Nee A. Y. C. Digital twin enhanced dynamic job-shop scheduling. *J. Manuf. Syst*. 2021. Vol. 58. P. 146–156. DOI: https://doi.org/10.1016/j.jmsy.2020.04.008

54. Zhou T., Tang D., Zhu H., Zhang Z. Multi-agent reinforcement learning for online scheduling in smart factories. *Robot. Comput. Integr. Manuf*. 2021. Vol. 72. 102202. DOI: https://doi.org/10.1016/j.rcim.2021.102202

55. Zhang H., Zhang G., Yan Q. Digital twin-driven cyber-physical production system towards smart shop-floor. *J. Ambient Intell. Humaniz. Comput.* 2019. Vol. 10, № 11. P. 4439–4453. DOI: https://doi.org/10.1007/s12652-018-1125-4

56. Huang T.-W., Lin C.-X., Guo G., Wong M. Cpp-Taskflow: Fast task-based parallel programming using modern C++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019. Pp. 974–983. URL: https://doi.org/10.1109/IPDPS.2019.00105 (дата звернення: 03.01.2026)

57. Chen M., Zhou Q., Nie K., Li H. Loop-Block-Level Automatic Parallelization in Compilers. Applied Sciences. 2026. URL: https://doi.org/10.3390/app16031533 (дата звернення: 03.03.2026)

58. Tagliavini G., Cesarini D., Marongiu A. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. *IEEE Transactions on Parallel and Distributed Systems*. 2018. № 29(9). Pp. 2150–2163. URL: https://doi.org/10.1109/TPDS.2018.2814602 (дата звернення: 12.01.2026)

59. Shen Y., Peng M., Wang S., Wu Q. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems*. 2021. № 125. Pp. 515–525. URL: https://doi.org/10.1016/j.future.2021.07.001 (дата звернення: 02.02.2026)

60. Barredo Arrieta A., et al. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*. 2020. № 58. Pp. 82–115. URL: https://doi.org/10.1016/j.inffus.2019.12.012 (дата звернення: 21.02.2026)

61. Czejdo D. B., Daszczuk W. B., Grześkowiak W. Practical approach to introducing parallelism in sequential programs. *In Proceedings of the 18th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX)*. 2023. Pp. 13–27. URL: https://doi.org/10.1007/978-3-031-37720-4_2 (дата звернення: 21.03.2026)

62. Akarvardar K., Wong H. S. P. Technology prospects for data-intensive computing. *Proceedings of the IEEE*. 2023. № 111(1). Pp. 92–112.

63. Bedratyuk L., Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. 2018. 79, 407–414. URL: https://doi.org/10.48550/arXiv.1706.00829 (дата звернення: 27.03.2026)

64. Nicholson H., Raza A., Chrysogelos P., Ailamaki A. HetCache: Synergising NVMe storage and GPU acceleration for memory-efficient analytics. *In Proceedings of the Conference on Innovative Data Systems Research (CIDR 2023)*. URL: https://www.cidrdb.org/cidr2023/ (дата звернення: 27.12.2025)

65. Denysiuk D., Savenko O., Lysenko S., Savenko B., Kashtalian A. Method for Detecting Steganographic Changes in Images Using Machine Learning. *13th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece*, 2023, pp. 1-6, doi: 10.1109/DESSERT61349.2023.10416453 (дата звернення: 3.01.2025)

66. Mutlu O., Ghose S., Gómez-Luna J., Ausavarungnirun R. A modern primer on processing in memory. *Springer Nature Singapore*. 2023.

67. Melnychenko O., Savenko O., Radiuk P., Apple detection with occlusions using modified YOLOv5-v1. *In Proc. 12th IEEE Int. Conf. Intell. Data Acquisition Adv. Comput. Syst.: Technol. Appl. (IDAACS'2023), Dortmund, Germany*, Sep. 7–9, 2023. New York, NY, USA: IEEE, Inc., 2023. Pp. 107–112. URL: https://doi.org/10.1109/idaacs58523.2023.10348779 (дата звернення: 15.11.2025)

68. Qureshi M., Mailthody V., Gelado I., Min K., Masood T., Park J., Xiong Y., Newburn C., Vainbrand D., Chung S., Garland M., Dally W., Hwu W. BaM: A case for enabling fine-grain high throughput GPU-orchestrated access to storage (*arXiv preprint*). 2022. URL: https://arxiv.org/pdf/2203.04910.pdf (дата звернення: 15.11.2025)

69. Roe R. Overcoming the memory bottleneck. *Scientific Computing World*. 2019. URL: https://www.scientific-computing.com/feature/overcoming-memory-bottleneck (дата звернення: 23.11.2025)

70. Saran C. Nvidia targets datacentre memory bottleneck. 2021. URL: https://www.computerweekly.com/news/252499172/Nvidia-targets-datacentre-memory-bottleneck (дата звернення: 23.12.2025)

71. Ghaleb M., Taghipour S. Dynamic shop-floor scheduling using real-time information—A case study from the thermoplastic industry. *Journal of Manufacturing Systems*. 2023. Vol. 69. P. 271–286.

72. Li Y., Li Y., Cheng J., Wu P. Order Assignment and Scheduling for Personal Protective Equipment Production During the Outbreak of Epidemics. *IEEE Transactions on Engineering Management*. 2022. Vol. 69. P. 1285–1298.

73. Rahmani D., Heydari M. Robust and stable flow shop scheduling with unexpected arrivals of new jobs and uncertain processing times. *Journal of Manufacturing Systems*. 2021 Vol. 33. P. 84–92.

74. Al-Behadili M., Ouelhadj D., Jones D. Multi objective biased randomised iterated greedy for robust permutation flow shop scheduling problem under disturbances. *Journal of Manufacturing Systems*. 2019. Vol. 51. P. 224–235.

75. Ouelhadj D., Petrovic S. A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*. 2020. Vol. 12. P. 417–443.

76. Li K., Deng Q., Zhang L., Fan Q., Gong G., Ding S. An effective MCTS-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem. *Computers & Industrial Engineering*. 2021. Vol. 155. Article 107157.

77. Yang B., Geunes J. Predictive-reactive scheduling on a single resource with uncertain future jobs. *European Journal of Operational Research*. 2019. Vol. 189. P. 1263–1282.

78. Kaya S., Karaçizmeli İ. H. Hazırlık zamanlı ortak teslim tarihli özdeş paralel makine çizelgeleme problemlerinin çok amaçlı çözümü. *Harran Üniversitesi Mühendislik Dergisi*. 2018. Vol. 3. P. 205–213.

79. Tighazoui A., Sauvey C., Sauer N. Predictive-reactive strategy for identical

parallel machine rescheduling. *Journal of Manufacturing Systems*. 2021. Vol. 59. P. 230–245.

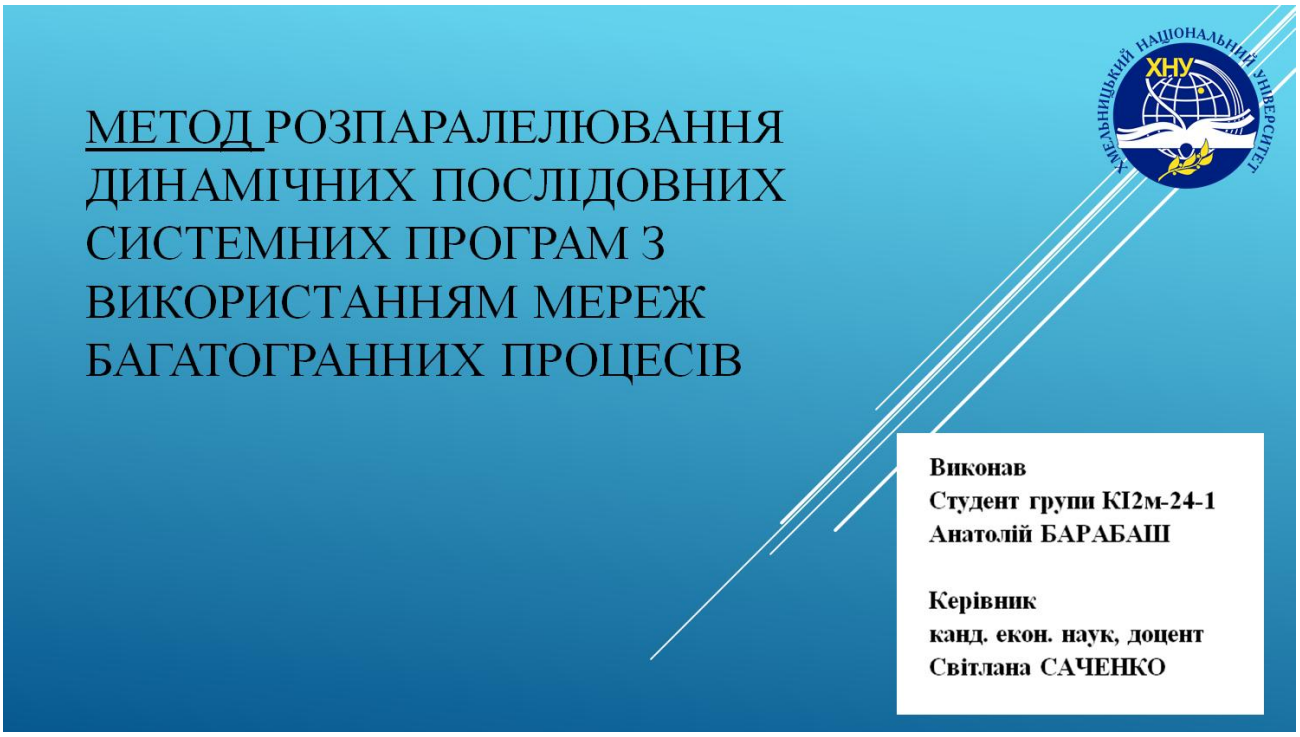
80. Geurtsen M., Adan J., Akçay A. Integrated maintenance and production scheduling for unrelated parallel machines with setup times. *Flexible Services and Manufacturing Journal*. 2024. Vol. 36. P. 1046–1079.

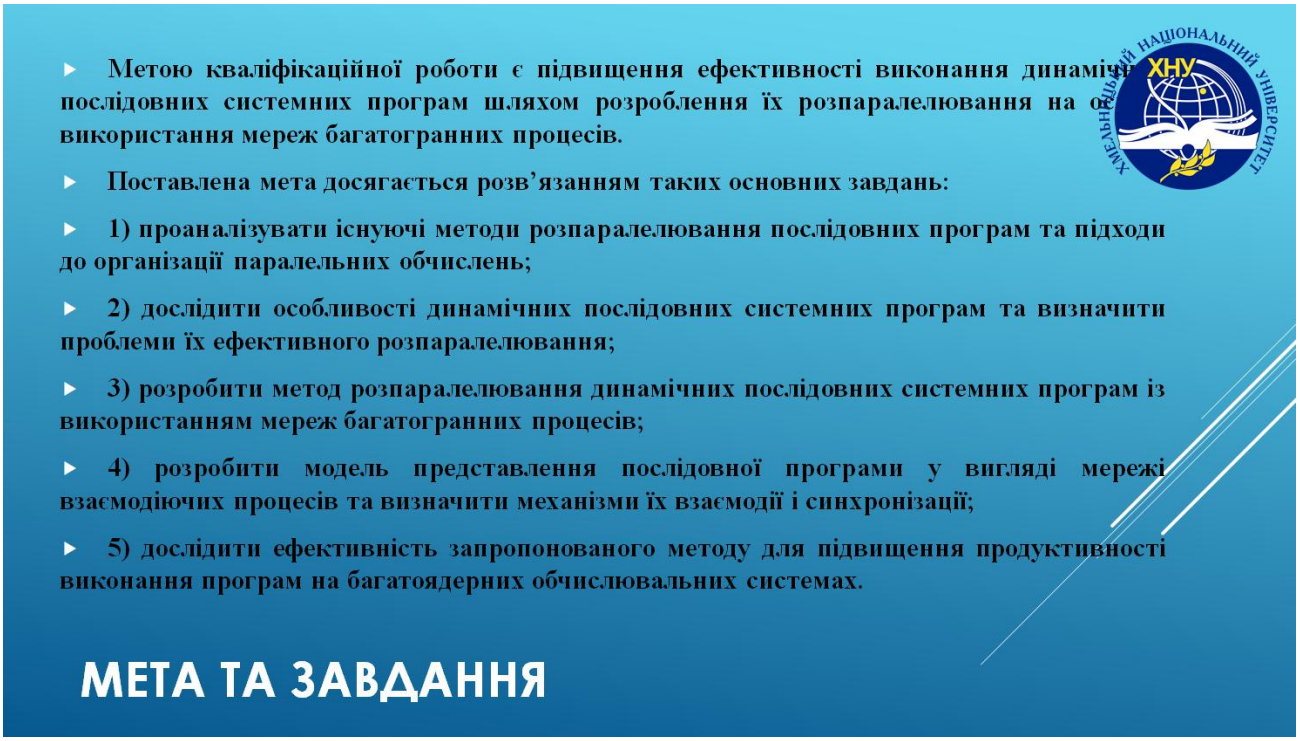
81. Барабаш А., Лигун О., Дрозд А. Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів. Measuring and computing devices in technological processes. 2026. № 2.

ДОДАТОК А

(обов’язковий)

ПРЕЗЕНТАЦІЯ ДО РОБОТИ





- **Об'єктом дослідження є процес розпаралелювання динамічних послідовних системних програм.**
- **Предметом дослідження є методи та засоби розпаралелювання програм на основі мереж багатогранних процесів.**
- **Наукова новизна отриманих результатів:**
- **- удосконалено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів, особливістю якого є представлення структури програми у вигляді мережі взаємодіючих процесів, що дозволяє виявляти незалежні обчислювальні фрагменти та організовувати їх паралельне виконання.**

ОБ'ЄКТ, ПРЕДМЕТ

Суть умов Бернштейна полягає в аналізі множин змінних, які читаються та змінюються інструкціями. Для кожного оператора або блоку програми визначаються дві множини:

1) R_i (Read set) – множина змінних, які читає інструкція I_i;

2) W_i (Write set) – множина змінних, які змінює інструкція I_j.

Нехай існують дві інструкції або блоки програми I_1 та I_2. Їх паралельне виконання можливе лише тоді, коли виконуються три умови:

1) відсутність конфлікту читання-запису, яка означає, що інструкція I_1 не читає змінну, яку змінює інструкція I_2, і задамо її так: $R_1 \cap W_2 = \emptyset$;

2) відсутність конфлікту запис-читання, яка означає, що інструкція I_2 не читає змінну, яку змінює інструкція, і задамо її так: $R_2 \cap W_1 = \emptyset$;

3) відсутність конфлікту запис-запис, яка означає, що обидві інструкції не змінюють одну й ту саму змінну, і задамо її так: $W_1 \cap W_2 = \emptyset$;

УМОВИ БЕРНШТЕЙНА

Розглянемо приклади щодо трьох перших умов.

Наприклад, розглянемо дві інструкції:

$I_1: a = b + c;$

$I_2: d = e + f.$

Тут $R_1 = \{b, c\}, W_1 = \{a\}, R_2 = \{e, f\}, W_2 = \{d\}$

Оскільки

$R_1 \cap W_2 = \emptyset,$

$R_2 \cap W_1 = \emptyset,$

$W_1 \cap W_2 = \emptyset,$ то обидві інструкції не мають спільних змінних, тому вони можуть виконуватися паралельно.

Наприклад, для інструкцій

$I_1: a = b + c,$

$I_2: d = a + e$

отримуємо $R_2 = \{a, e\}, W_2 = \{a\},$ тому $R_2 \cap W_1 = \{a\} \neq \emptyset.$ Отже, інструкція I_2 залежить від результату I_1, і їх не можна виконувати паралельно.

ПРИКЛАД

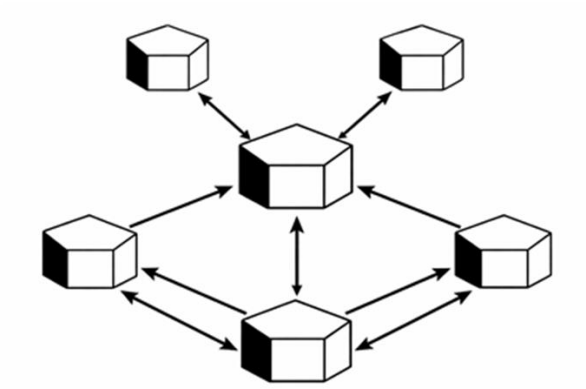


Рисунок 3.1 – Приклад мережі багатогранних процесів

МЕТОД

Основні кроки методу.

Крок 1. Класифікація системної програми.

Крок 2. Аналіз структури послідовної програми.

Крок 3. Побудова графа обчислювальних залежностей.

Крок 4. Декомпозиція програми на функціональні блоки.

Крок 5. Формування багатогранних процесів.

Крок 6. Побудова мережі багатогранних процесів.

Крок 7. Визначення паралельних областей виконання.

Крок 8. Відображення мережі процесів на обчислювальні ресурси.

МЕТОД

РЕАЛІЗАЦІЯ

Таблиця 4.1 – Час обробки повідомлень у мережі багатогранних процесів

Кількість повідомлень	Час обробки паралельного виконання (с)	Час обробки послідовного виконання (с)
100	2.1	5.6
200	4.3	11.2
500	10.8	28.0

Таблиця 4.2 – Прискорення виконання мережі багатогранних процесів

Кількість повідомлень	Прискорення (послідовний / паралельний)
100	2.7
200	2.6
500	2.6

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

Таблиця 3

Час обробки та прискорення

Кількість повідомлень	Потоки/ядра	T_s (с)	T_p (с)	Прискорення S
100	4	5.6	2.1	2.7
200	4	11.2	4.3	2.6
500	4	28.0	10.8	2.6

Таблиця 4

Ефективність паралельного виконання

Кількість повідомлень	Потоки/ядра	Прискорення S	Ефективність E (%)
100	4	2.7	67.5
200	4	2.6	65.0
500	4	2.6	65.0

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

Рисунок 3 – Прискорення від кількості повідомлень

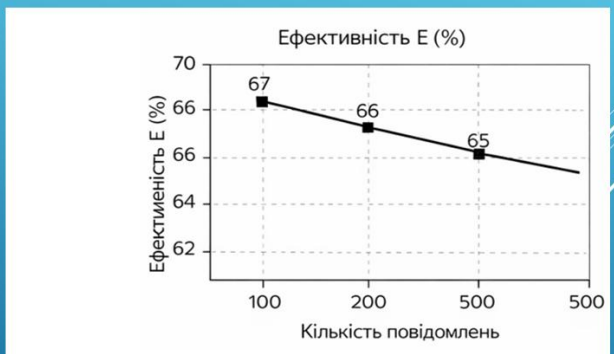


Рисунок 4 – Ефективність від кількості повідомлень

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

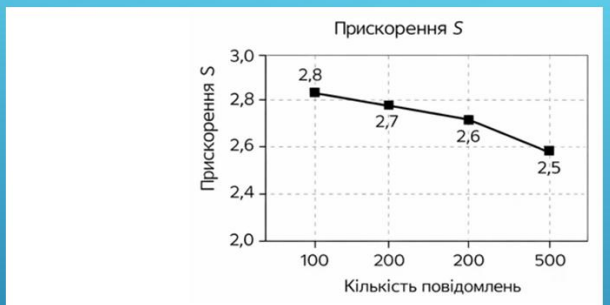


Рисунок 3 – Прискорення від кількості повідомлень

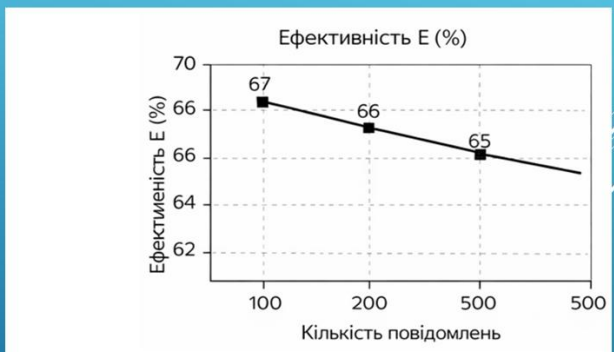


Рисунок 4 – Ефективність від кількості повідомлень

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

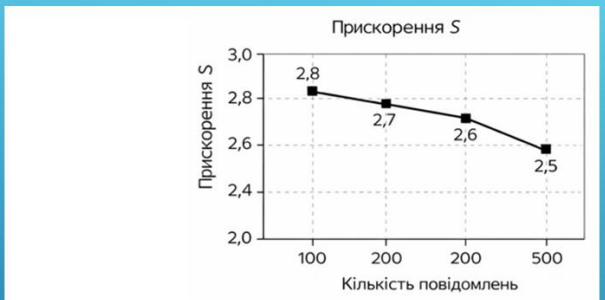


Рисунок 3 – Прискорення від кількості повідомлень

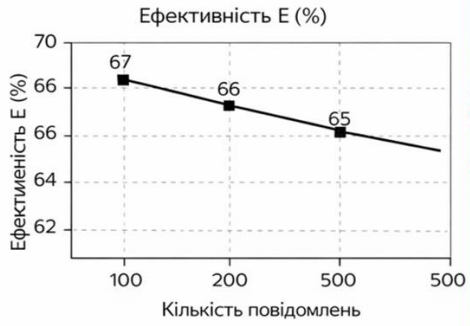


Рисунок 4 – Ефективність від кількості повідомлень

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

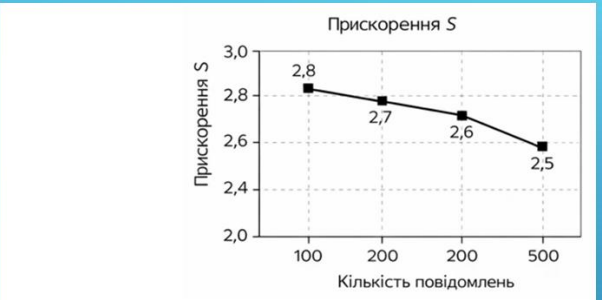


Рисунок 3 – Прискорення від кількості повідомлень

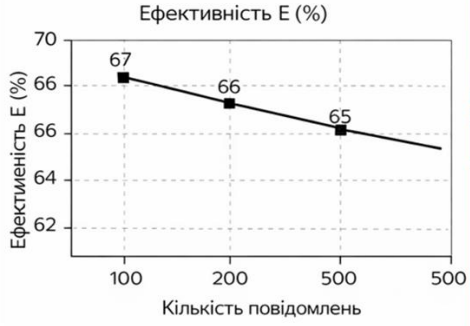
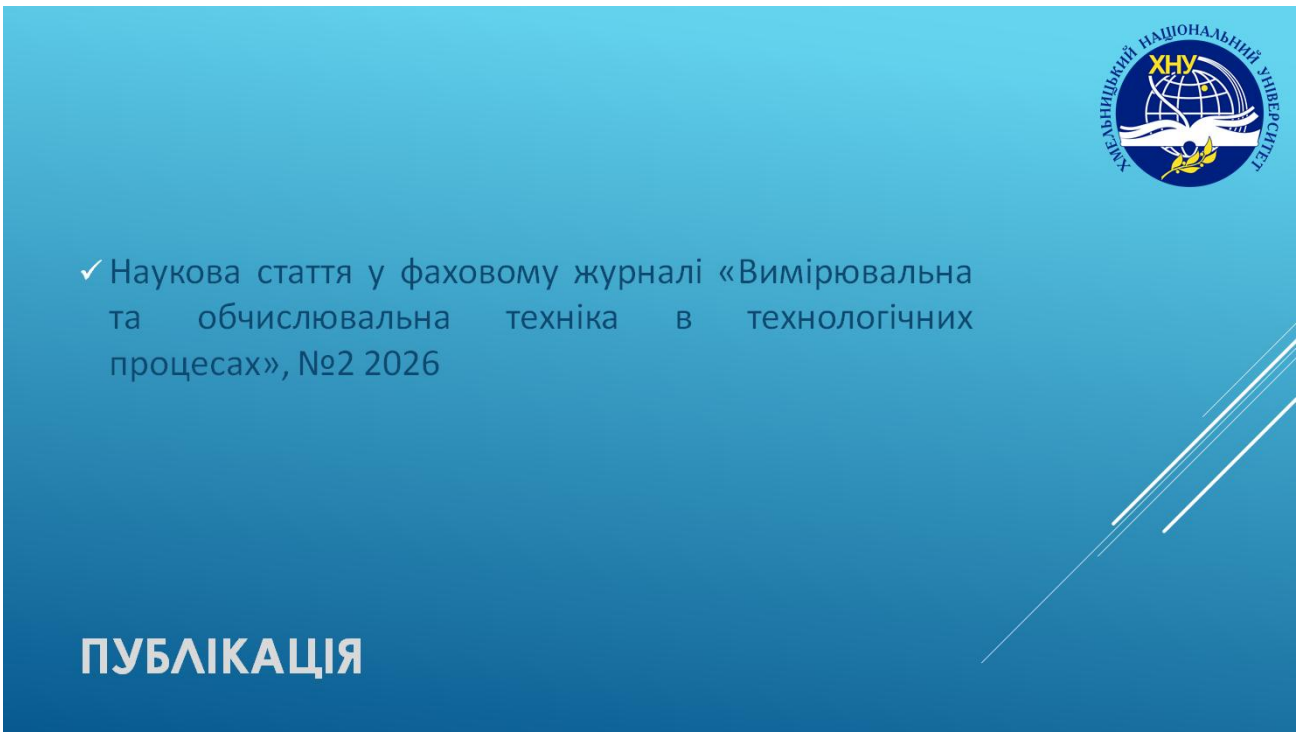


Рисунок 4 – Ефективність від кількості повідомлень

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ


ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ХНУ
✓ Наукова стаття у фаховому журналі «Вимірювальна та обчислювальна техніка в технологічних процесах», №2 2026
ПУБЛІКАЦІЯ

ДОДАТОК Б

(обов'язковий)

НАУКОВА ПРАЦЯ ЗДОБУВАЧА

УДК 004.75

БАРАБАШ Анатолій
Хмельницький національний університет
ORCID ID: 0009-0000-9607-1775
e-mail: wkondrael@gmail.com
ЛИГУН Олексій
Хмельницький національний університет
ORCID ID: 0009-0004-5727-5096
e-mail: oleksii.lyhun@gmail.com
ДРОЗД Андрій
Хмельницький національний університет
ORCID ID: 0009-0008-1049-1911
e-mail: andriydrozdit@gmail.com

МЕТОД РОЗПАРАЛЕЛЮВАННЯ ДИНАМІЧНИХ ПОСЛІДОВНИХ СИСТЕМНИХ ПРОГРАМ З ВИКОРИСТАННЯМ МЕРЕЖ БАГАТОГРАННИХ ПРОЦЕСІВ

У статті досліджено особливості динамічних послідовних системних програм та визначено основні проблеми їх ефективного розпаралелювання. Показано, що складність автоматичного розпаралелювання таких програм зумовлена наявністю складних залежностей між даними, нерегулярною структурою обчислень, динамічним створенням задач, а також необхідністю синхронізації доступу до спільних ресурсів. Зазначені фактори істотно ускладнюють використання традиційних підходів до паралельного виконання та потребують розроблення спеціалізованих методів організації обчислень.

У роботі запропоновано метод розпаралелювання динамічних послідовних системних програм на основі використання мереж багатогранних процесів. Запропонований підхід дає змогу формалізувати структуру обчислень, виконати декомпозицію програми на незалежні або частково незалежні фрагменти та організувати їх виконання у вигляді системи взаємодіючих процесів, що можуть виконуватися паралельно. Це забезпечує більш ефективне використання обчислювальних ресурсів і підвищує рівень масштабованості програм.

Також розроблено модель представлення послідовної системної програми у вигляді мережі взаємодіючих процесів, у межах якої визначено механізми їх взаємодії та синхронізації. Запропоновані механізми забезпечують коректний обмін даними між процесами, узгодженість виконання та запобігання конфліктам доступу до спільних ресурсів. Реалізація такої моделі дозволяє ефективніше організувати паралельне виконання програм у багатоядерних обчислювальних системах.

Проведено дослідження ефективності запропонованого методу, результати якого підтверджують доцільність застосування мереж багатогранних процесів для підвищення продуктивності виконання динамічних системних програм. Використання запропонованого підходу сприяє оптимізації обчислювальних процесів та більш раціональному використанню апаратних ресурсів сучасних багатоядерних систем.

Перспективним напрямом подальших досліджень є вивчення можливостей застосування запропонованого підходу для розпаралелювання процесів у розподілених обчислювальних середовищах.

Ключові слова: розподілена система, комп'ютерна система, розпаралелювання, процеси, системні програми, програмне забезпечення.

BARABASH Anatolii
LYHUN Oleksii
DROZD Andriy
Khmelnytskyi National University, Khmelnytskyi, Ukraine

THE METHOD OF PARALLELIZATION OF DYNAMIC SEQUENTIAL SYSTEM PROGRAMS USING NETWORKS OF MULTIFACETED PROCESSES

The article examines the peculiarities of dynamic sequential system programs and identifies the main problems of their effective parallelization. It is shown that the complexity of automatic parallelization of such programs is due to the presence of complex dependencies between data, irregular structure of calculations, dynamic creation of tasks, as well as the need to synchronize access to shared resources. These factors significantly complicate the use of traditional approaches to parallel execution and require the development of specialized methods for organizing calculations.

The paper proposes a method of parallelizing dynamic sequential system programs based on the use of networks of multifaceted processes. The proposed approach makes it possible to formalize the structure of calculations, decompose the program into independent or partially independent fragments, and organize their execution in the form of a system of interacting processes that can be executed in parallel. This ensures more efficient use of computing resources and increases the level of scalability of applications.

A model for representing a sequential system program in the form of a network of interacting processes, within which the mechanisms of their interaction and synchronization are defined, has also been developed. The proposed mechanisms ensure correct data exchange between processes, consistency of execution and prevention of access conflicts to shared resources. The implementation of such a model allows more efficient organization of parallel execution of programs in multi-core computing systems.

A study of the effectiveness of the proposed method was conducted, the results of which confirm the expediency of using networks of multifaceted processes to increase the productivity of executing dynamic system programs. The use of the proposed approach contributes to the optimization of computing processes and more rational use of hardware resources of modern multicore systems.

A promising direction of further research is the study of the possibilities of applying the proposed approach for parallelization of processes in distributed computing environments.

Keywords: distributed system, computer system, parallelization, processes, system programs, software.

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ

Розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів є актуальним напрямом досліджень у галузі паралельних обчислень та оптимізації програмного забезпечення. Такий вибір обумовлений сучасними тенденціями розвитку обчислювальної техніки. Сьогодні більшість комп'ютерних систем базується на багатоядерних процесорах і розподілених обчислювальних середовищах, що створює значний потенціал для паралельного виконання програм. Однак значна частина існуючого програмного забезпечення, зокрема системного рівня, була розроблена як послідовна, тобто орієнтована на виконання інструкцій у межах одного потоку. У результаті апаратні можливості сучасних систем використовуються не повною мірою, що знижує загальну ефективність обчислень. Саме тому виникає необхідність дослідження методів, які дозволяють перетворювати послідовні програми на паралельні без кардинальної зміни їх структури або повного переписування програмного коду.

Актуальність теми особливо проявляється у випадку динамічних системних програм, робота яких залежить від стану системи, оброблюваних даних і умов виконання. На відміну від статичних алгоритмів, де структура обчислень визначена заздалегідь, динамічні програми формують потік виконання під час роботи. Наявність умовних переходів, складних структур даних та змінних залежностей між операціями значно ускладнює процес автоматичного розпаралелювання. У таких умовах традиційні методи оптимізації часто виявляються недостатньо ефективними, що обґрунтовує необхідність пошуку нових підходів до організації паралельних обчислень.

Одним із перспективних підходів є використання моделі мереж багатогранних процесів для представлення

структури програми. У межах цього підходу послідовна програма розглядається як система взаємодіючих процесів, кожен з яких виконує окрему частину обчислень і обмінюється даними з іншими компонентами. Така модель дозволяє представити програму у вигляді графа залежностей, що значно полегшує аналіз структури обчислень та виявлення фрагментів, які можуть виконуватися паралельно. Багатогранний процес при цьому виступає універсальним обчислювальним елементом, здатним одночасно виконувати обчислювальні операції, здійснювати обмін даними та забезпечувати синхронізацію з іншими процесами.

Використання мережі багатогранних процесів створює можливість формалізувати структуру складних програмних систем і здійснювати їх подальшу оптимізацію. Представлення програми у вигляді взаємопов'язаних процесів дозволяє визначити незалежні обчислювальні ділянки та організувати їх паралельне виконання. Це, у свою чергу, сприяє підвищенню продуктивності програмного забезпечення та більш ефективному використанню обчислювальних ресурсів сучасних багатоядерних і розподілених систем.

Отже, наявна необхідність підвищення ефективності виконання програм в умовах широкого використання паралельних обчислювальних архітектур. Дослідження методів розпаралелювання динамічних послідовних системних програм із застосуванням мереж багатогранних процесів дозволяє розробити підходи до перетворення традиційних програмних систем у більш ефективні паралельні структури. Це сприятиме покращенню продуктивності програм, підвищенню масштабованості програмного забезпечення та більш повному використанню обчислювальних можливостей сучасних комп'ютерних систем.

АНАЛІЗ ДОСЛІДЖЕНЬ ТА ПУБЛІКАЦІЙ

Розпаралелювання програм [1, 2] є одним із ключових напрямів розвитку сучасних обчислювальних систем і програмного забезпечення. Стрімкий розвиток інформаційних технологій, зростання обсягів оброблюваних даних та підвищення вимог до швидкодії програмних систем зумовлюють необхідність пошуку нових підходів до організації обчислювальних процесів. У сучасних комп'ютерних системах широко використовуються багатоядерні процесори, графічні обчислювальні пристрої, кластери та розподілені обчислювальні платформи, які здатні виконувати велику кількість операцій одночасно. Проте ефективне використання таких апаратних можливостей безпосередньо залежить від того, наскільки програмне забезпечення здатне підтримувати паралельне виконання обчислень.

Історично значна частина програмного забезпечення [3, 4] створювалася для однопроцесорних систем, де виконання програм відбувалося у послідовному режимі. У таких програмах усі інструкції виконуються одна за одною в межах одного потоку керування, а кожна наступна операція може розпочатися лише після завершення попередньої. Така модель програмування є відносно простою для реалізації та аналізу, однак вона не дозволяє повною мірою використовувати потенціал сучасних багатоядерних обчислювальних систем. У результаті навіть за наявності значної кількості обчислювальних ресурсів програма може використовувати лише невелику їх частину, що призводить до зниження ефективності виконання та збільшення часу обробки даних.

Саме тому одним із важливих напрямів [5, 6] сучасних досліджень у галузі комп'ютерних наук є розпаралелювання програм. Під розпаралелюванням розуміють процес перетворення послідовної програми або алгоритму на таку форму, у якій окремі частини обчислень можуть виконуватися одночасно. Основна мета цього процесу полягає у підвищенні продуктивності виконання програм за рахунок одночасного використання декількох обчислювальних ресурсів. Це може реалізовуватися шляхом багатопотокового виконання на одному процесорі, розподілення задач між кількома ядрами процесора або використанням розподілених обчислювальних середовищ.

Процес розпаралелювання [7, 8] передбачає аналіз структури програми з метою виявлення незалежних або частково незалежних фрагментів обчислень. Якщо окремі операції або блоки коду не мають взаємних залежностей, вони можуть виконуватися паралельно без порушення логіки роботи програми. Таким чином, завданням розпаралелювання є визначення таких ділянок програми та організація їх паралельного виконання з урахуванням можливих обмежень і залежностей.

Особливу роль у цьому процесі відіграє аналіз залежностей між операціями [9, 10]. У програмі можуть існувати різні типи залежностей, зокрема залежності за даними, керуванням або ресурсами. Залежність за даними виникає у випадку, коли одна операція використовує результат, отриманий іншою операцією. У такій ситуації порядок виконання операцій не може бути змінений без порушення коректності результатів. Якщо ж операції не залежать одна від одної, вони можуть виконуватися одночасно на різних обчислювальних ресурсах. Аналіз таких залежностей є одним із ключових етапів процесу розпаралелювання програм [11, 12].

Одним із ключових етапів розпаралелювання програм є аналіз залежностей між операціями [13, 14]. Саме залежності визначають, спроможність окремих обчислювальних операцій виконуватися одночасно, або вони повинні виконуватися у строго визначеній послідовності. Якщо між операціями існує залежність, то порушення порядку їх виконання може призвести до некоректних результатів роботи програми. Тому перед організацією паралельного виконання необхідно провести детальний аналіз усіх можливих залежностей у програмному коді.

У загальному випадку [15, 16] залежність між операціями виникає тоді, коли виконання однієї операції впливає на результати іншої. Найчастіше такі залежності пов'язані з використанням спільних даних або змінних. Якщо одна операція змінює значення певної змінної, а інша операція використовує це значення, то порядок виконання цих операцій стає важливим. У цьому випадку їх не можна виконувати паралельно без додаткових механізмів узгодження.

У теорії розпаралелювання [17, 18] програм розрізняють декілька основних типів залежностей між операціями. Найбільш важливими є залежності за даними, залежності за керуванням та залежності за ресурсами.

Отже, аналіз існуючих підходів до розпаралелювання програм свідчить про необхідність здійснення подальших досліджень у цьому напрямі. Особливо актуальною є проблема створення ефективних методів розпаралелювання динамічних послідовних системних програм, які дозволять автоматизувати процес виявлення паралелізму в програмному коді та організувати його ефективне використання. Розв'язання цієї науково-дослідницької проблеми передбачає розроблення нових моделей представлення програм, методів аналізу залежностей між обчисленнями та підходів до організації взаємодії між паралельними процесами. Саме ці питання становлять основу подальших досліджень у межах даної роботи та визначають її наукову спрямованість.

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

Розроблення нових алгоритмічних підходів, які здатні автоматично виявляти ділянки послідовного програмного коду, що можуть бути перетворені на незалежні паралельні потоки виконання залишається актуальним напрямом досліджень. Основна його ідея полягає у створенні інтелектуальних методів аналізу програм, що дозволяють без безпосереднього втручання розробника визначати фрагменти коду, які придатні для паралельного виконання, та автоматично трансформувати їх у структури, оптимізовані для багатоядерних обчислювальних систем. У сучасних обчислювальних середовищах, де широко використовуються процесори з кількома ядрами, така модифікована програма здатна виконуватися одночасно на різних обчислювальних ресурсах, що забезпечує істотне зростання продуктивності та ефективності використання апаратного забезпечення. В основі такого дослідження використовуємо припущення, що значна частина послідовного коду потенційно містить прихований паралелізм, який можна виявити за допомогою формального аналізу залежностей між інструкціями та змінними, тобто якщо припустити що такий паралелізм може бути наявним, то його опрацювання може надати змогу виконувати програмний код паралельно.

Застосування умов Бернштейна є особливо важливим у процесі автоматичної паралелізації програм, що виконується компіляторами або спеціалізованими інструментами оптимізації. Під час аналізу вихідного коду програмна система визначає множини змінних читання та запису для кожного оператора або блоку інструкцій, після чого перевіряє виконання наведених умов. Якщо конфлікти доступу до змінних відсутні, відповідні інструкції можуть бути розміщені у різних потоках виконання. Таким чином, умови Бернштейна фактично визначають формальний критерій незалежності операцій у програмі та слугують основою для побудови графів залежностей, які відображають структуру взаємозв'язків між обчислювальними операціями.

Сформуємо на основі умов Бернштейна узагальнені моделі класів системних програм, для яких застосування паралельних обчислень є доцільним та ефективним. До таких класів належать програми, у структурі яких існують незалежні обчислювальні підзадачі, відсутні або мінімальні залежності між операціями, а також можливість розділення обчислень на множину однотипних ітерацій або функціональних блоків. Найбільш характерними є три класи програм: програми з незалежними викликами функцій; програми з незалежними шляхами виконання інструкцій; програми з ітераційними структурами (циклами), де кожна ітерація може виконуватися незалежно від інших. Для кожного з цих класів побудуємо формалізовану модель, які описуватимуть структуру обчислень залежності між змінними та умови паралелізації.

До першого класу системних програм віднесемо програми, у яких значна частина обчислень реалізується у вигляді незалежних викликів функцій. У таких системах основний алгоритм можна представити як композицію функцій, кожна з яких виконує окрему частину обчислень. Якщо результати цих функцій не залежать одна від одної або використовуються лише після завершення всіх викликів, їх виконання може бути організоване паралельно. Такий клас системних програм можна задамо множиною функцій так:

$$F = \{f_1, f_2, \dots, f_{n_F}\}, \tag{1}$$

де кожна функція f_i відображає множину вхідних змінних у множину вихідних змінних $f_i: X_i \to Y_i$; $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,n_{X_i}}\}$; $Y_i = \{y_{i,1}, y_{i,2}, \dots, y_{i,n_{Y_i}}\}$; $i = 1,2, \dots, n_F$; n_F – кількість функцій в множині функцій F; X_i - множина вхідних змінних функції f_i; Y_i - множина змінних, що утворюють результат виконання функції.

Тоді, загальний стан програми можна подати як вектор змінних так:

$$S = (s_1, s_2, \dots, s_{n_S}), \tag{2}$$

де s_j - значення j-тої змінної програми.

Після виконання функції f_i стан системи переходить у новий стан, який будемо визначати так:

$$S' = f_i(S). \tag{3}$$

Паралельне виконання функцій f_i та f_j можливе лише тоді, коли виконуються модифіковані умови Бернштейна, які визначені для елементів системної програми так:

$$R_i \cap W_j = \emptyset,$$

$$R_j \cap W_i = \emptyset, \quad W_i \cap W_j = \emptyset, \tag{4}$$

де R_i - множина змінних, що читаються функцією f_i; W_i - множина змінних, що модифікуються функцією f_i; R_i, W_j - відповідні множини для функції f_j.

Другий клас сформуємо з системних програм, у яких паралелізм виникає завдяки наявності незалежних шляхів виконання інструкцій у межах однієї функції або процедури. У цьому випадку алгоритм подамо послідовністю операторів так:

$$P = (p_1, p_2, \dots, p_{n_P}), \tag{5}$$

де p_k - окрема інструкція програми; n_P – кількість ісінтрукцій програми.

Стан програми після виконання інструкції p_k задамо функцією переходу так:

$$s_{k+1} = p_k(s_k), \tag{6}$$

де s_k - стан системи перед виконанням інструкції p_k; s_{k+1} - стан після її виконання.

Залежність між двома інструкціями p_i та p_j визначається через змінні, які вони використовують. Нехай $R(p_k)$ це множина змінних, що читаються інструкцією p_k, а $W(p_k)$ - множина змінних, що змінюються цією інструкцією. Інструкції p_i та p_j можуть виконуватися паралельно, якщо виконуються умови:

$$R(p_i) \cap W(p_j) = \emptyset, \quad R(p_j) \cap W(p_i) = \emptyset, \quad W(p_i) \cap W(p_j) = \emptyset \tag{7}$$

У такому випадку набір інструкцій можна розділити на незалежні підмножини так:

$$P = P_1 \cup P_2 \cup \dots \cup P_k, \tag{2.9}$$

де кожна підмножина P_i є незалежним шляхом виконання.

Максимальний рівень паралелізму визначається кількістю таких незалежних шляхів. Прикладом програм цього класу може бути обробка зображень, коли різні частини алгоритму виконують незалежні обчислення над різними параметрами. Наприклад, у процесі обробки кадру зображення один блок програми може виконувати фільтрацію шуму, а інший - підсилення контрасту, а третій - обчислення гістограми яскравості. Якщо ці операції не змінюють спільні змінні, то вони можуть виконуватися паралельно.

Третій клас системних програм визначимо програмами з ітераційними структурами, де обчислення виконуються у циклах. У багатьох алгоритмах кожна ітерація циклу виконує однакові операції над різними елементами набору даних, що робить такі алгоритми природними кандидатами для паралелізації. Формально цикл задамо послідовністю ітерацій, у якій на кожній ітерації виконується функція, так:

$$for\ i = a, \dots, b;\ S_{i+1} = F(S_i, i), \tag{9}$$

де i - індекс ітерації; a - початкове значення індексу; b - кінцеве значення; S_i - стан системи перед ітерацією; F - функція, що описує обчислення в тілі циклу.

Нехай R_i - множина змінних, що читаються під час ітерації i; W_i - множина змінних, що змінюються під час ітерації i.

Ітерації i та j можуть виконуватися паралельно, якщо виконуються всі три умови Бернштейна. Крім того, необхідно, щоб значення параметрів циклу були визначені до початку виконання, тобто повинно виконуватись таке співвідношення:

$$N = b - a + 1, \tag{10}$$

де N - кількість ітерацій циклу.

Якщо ці умови виконуються, то цикл може бути перетворений на множину незалежних задач так:

$$T = \{T_1, T_2, \dots, T_{n_T}\}, \tag{11}$$

де кожна задача T_i відповідає виконанню однієї ітерації.

Прикладом такого класу програм є алгоритми обробки масивів або матриць. Наприклад, обчислення суми квадратів елементів масиву визначимо так:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} m_i^2, \tag{12}$$

де m_i - елемент масиву; n_{MAS} - розмір масиву; $S_{\sum MAS}$ - результат обчислення суми квадратів елементів масиву.

У цьому випадку кожна операція $y_i = m_i^2$ може виконуватися паралельно, після чого результати об’єднуються операцією редукції так:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} y_i. \tag{13}$$

Таким чином, аналіз структури системних програм дозволяє виділити три основні моделі, придатні для

паралелізації: модель незалежних функціональних блоків; модель незалежних шляхів виконання інструкцій; модель незалежних ітерацій циклів. Кожна з цих моделей характеризується власними формальними умовами залежностей між змінними та операціями, що дозволяє визначити можливість паралельного виконання та побудувати оптимальну структуру багатопотокового алгоритму. Використання таких формалізованих моделей є основою для створення автоматизованих систем аналізу та оптимізації програмного коду, які можуть адаптувати традиційні послідовні алгоритми до сучасних багатоядерних обчислювальних систем.

У класичній постановці умов Бернштейна визначено три основні критерії незалежності операцій, які дозволяють встановити можливість їх паралельного виконання. Ці умови базуються на аналізі множин змінних, що читаються та змінюються інструкціями, і гарантують відсутність конфліктів доступу до даних між паралельними обчисленнями. Проте під час практичного застосування методів розпаралелювання системних програм виникає необхідність врахування ще одного важливого аспекту, який стосується апаратних обмежень обчислювальної системи, зокрема кількості доступних процесорів або ядер процесора. Тому до класичних умов доцільно додати четверту умову, яка пов'язана з ефективністю виконання паралельних задач з урахуванням апаратних ресурсів.

Четверта умова полягає в тому, що кількість паралельних обчислювальних задач, які формуються в результаті розпаралелювання програми, повинна бути узгоджена з кількістю доступних обчислювальних ресурсів системи. Нехай P позначає кількість доступних процесорів або обчислювальних ядер, на яких може виконуватися програма, а T - кількість незалежних задач або потоків, отриманих у результаті розпаралелювання. Тоді ефективність паралельного виконання значною мірою визначається співвідношенням між цими величинами. Якщо T значно перевищує P, то система змушена виконувати частину задач послідовно або здійснювати часте перемикання контекстів між потоками, що призводить до додаткових витрат на керування виконанням. Якщо ж T значно менше за P, то частина обчислювальних ресурсів залишається невикористаною, що також знижує загальну ефективність виконання програми.

Цю умову задамо у вигляді обмеження так:

$$T \leq P \cdot k, \tag{14}$$

де T - кількість паралельних задач або потоків; P - кількість доступних процесорів або ядер; k - коефіцієнт допустимого перевищення кількості потоків над кількістю процесорів, який враховує особливості планування потоків операційною системою.

У найпростішому випадку для максимального використання обчислювальних ресурсів бажано, щоб кількість потоків була близькою до кількості доступних ядер, тобто щоб виконувалась умова:

$$T \approx P. \tag{15}$$

У такому випадку кожна задача може виконуватися на окремому ядрі, що забезпечує максимальний рівень паралелізму без додаткових витрат на планування потоків.

Якщо розглядати загальний час виконання програми, то його можна подати у вигляді суми часу паралельної та послідовної частини алгоритму. Нехай T_s - час виконання послідовної частини програми, T_p - час виконання паралельної частини при використанні одного процесора, P - кількість процесорів або ядер. Тоді теоретичний час виконання програми при паралельному виконанні можна оцінити як

$$T_{\text{вик}} = T_s + \frac{T_p}{P}. \tag{16}$$

Це співвідношення фактично відображає принцип, близький до закону Амдала, відповідно до якого максимальний виграш від паралелізації обмежується часткою послідовного коду. Однак у реальних системах до цього часу додаються додаткові витрати на створення потоків, синхронізацію та обмін даними між потоками. Позначимо ці витрати через T_o. Тоді фактичний час виконання визначимо так:

$$T_{\text{вик}} = T_s + \frac{T_p}{P} + T_o. \tag{17}$$

У цьому випадку четверта умова полягає не лише у відповідності кількості задач кількості процесорів, але й у тому, що виграш від паралельного виконання повинен перевищувати накладні витрати, пов'язані з організацією паралельності. Тобто паралелізація є доцільною лише тоді, коли виконується така умова:

$$\frac{T_p}{P} + T_o < T_p. \tag{18}$$

Інакше кажучи, сумарний час виконання паралельної частини програми з урахуванням додаткових витрат повинен бути меншим, ніж час її виконання у послідовному режимі.

Таким чином, четверта умова фактично доповнює класичні умови Бернштейна, враховуючи апаратні характеристики обчислювальної системи та реальні витрати на організацію паралельного виконання. Якщо перші три умови визначають логічну можливість паралельного виконання інструкцій з точки зору відсутності залежностей даних, то четверта умова визначає практичну доцільність такого розпаралелювання з урахуванням доступних обчислювальних ресурсів та ефективності використання процесорних ядер. Саме поєднання цих чотирьох умов дозволяє не лише виявити незалежні обчислювальні структури у програмному коді, але й забезпечити оптимальне використання можливостей сучасних багатоядерних обчислювальних систем. Ця умова не впливає на кількість визначених класів, а лише доповнює новим обмеженням введені класи.

П'ятим обмеженням є закон Амдала. Закон Амдала визначає потенційне прискорення алгоритму при

збільшенні числа процесорів. Закон Амдала є одним із фундаментальних принципів теорії паралельних обчислень, який визначає теоретичну межу прискорення виконання програми під час використання багатопроцесорних або багатоядерних обчислювальних систем. Він описує залежність між часткою алгоритму, що може бути виконана паралельно, кількістю доступних процесорів та максимально можливим виграшем у продуктивності. Основна ідея цього закону полягає в тому, що навіть за наявності необмеженої кількості процесорів швидкодія програми обмежується тією частиною алгоритму, яка не може бути виконана паралельно і повинна виконуватися послідовно.

Нехай час виконання певної програми на одному процесорі дорівнює T_1. Цей час можна подати як суму двох компонентів: часу виконання послідовної частини алгоритму; часу виконання тієї частини програми, яка може бути розпаралелена. Позначимо через α частку послідовної частини програми, тобто ту частину алгоритму, яка не може бути виконана паралельно. Відповідно, частка паралельної частини становитиме (1 - α). Якщо використовується P процесорів або обчислювальних ядер, то паралельна частина алгоритму теоретично може бути розподілена між цими процесорами. У такому випадку час виконання програми на P процесорах можна визначити так:

$$T_P = T_1 \cdot \left(\alpha + \frac{1-\alpha}{P}\right), \tag{18}$$

де T_P - час виконання програми на (P) процесорах; T_1 - час виконання програми на одному процесорі; α - частка послідовної частини програми; 1-α - частка паралельної частини програми; P - кількість процесорів або ядер.

Однією з основних характеристик ефективності паралельних обчислень є прискорення виконання програми, яке визначається як відношення часу виконання програми на одному процесорі до часу виконання на P процесорах:

$$S(P) = \frac{T_1}{T_P}. \tag{19}$$

Підставивши значення T_P у цю формулу, отримуємо класичний вираз закону Амдала:

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}}. \tag{20}$$

Ця формула демонструє, що зі збільшенням кількості процесорів прискорення програми зростає, однак його зростання поступово сповільнюється. Причиною цього є наявність послідовної частини алгоритму, яка не може бути розподілена між процесорами і виконується лише одним потоком.

Якщо розглянути граничний випадок, коли кількість процесорів прямує до нескінченності P→∞, паралельна частина програми теоретично виконується миттєво, а загальний час виконання визначається лише послідовною частиною. У цьому випадку максимальне можливе прискорення буде дорівнювати значенню:

$$S_{max} = \frac{1}{\alpha}. \tag{21}$$

Це означає, що навіть за необмеженої кількості процесорів швидкодія програми обмежується величиною послідовної частини алгоритму. Наприклад, якщо 10% програми виконуються послідовно α = 0.1, то максимальне прискорення не перевищить такого значення:

$$S_{max} = \frac{1}{0{,}1} = 10. \tag{22}$$

незалежно від того, скільки процесорів використовується.

Ефективна організація паралельних обчислень базується на поєднанні структурного аналізу програмного коду, формальних умов незалежності операцій та врахування апаратних характеристик обчислювальної системи. Виділено три основні моделі системних програм, для яких розпаралелювання є найбільш доцільним і технічно можливим. Кожна з цих моделей характеризується певними особливостями структури алгоритмів та способом організації обчислювальних процесів, що визначає потенційну можливість виконання окремих частин програми паралельно.

Перша модель охоплює системні програми, побудовані на основі незалежних функціональних модулів або викликів функцій. У таких програмах основний алгоритм можна представити як сукупність функціональних блоків, кожен з яких виконує окрему обчислювальну задачу та працює з власною підмножиною даних. Якщо між цими функціональними блоками відсутні залежності за даними, вони можуть виконуватися одночасно в різних потоках або на різних обчислювальних ядрах. Така модель характерна для систем обробки даних, аналітичних програм, програм статистичних обчислень та інших систем, у яких результати окремих обчислень можуть бути отримані незалежно один від одного.

Друга модель системних програм базується на наявності незалежних шляхів виконання інструкцій у межах одного алгоритму або процедури. У цьому випадку програму можна розглядати як послідовність операторів, між якими можуть існувати або бути відсутні залежності за даними. Якщо окремі групи інструкцій не використовують спільні змінні або не модифікують результати виконання одна одної, такі інструкції можуть бути виконані паралельно. Аналіз подібних структур дозволяє виділяти незалежні підграфи у графі залежностей програми, що відкриває можливість одночасного виконання різних частин алгоритму. Ця модель характерна для систем обробки сигналів, обчислювальних модулів у складних програмних комплексах та програм, у яких окремі обчислювальні етапи не залежать від результатів інших етапів.

Третя модель системних програм пов’язана з наявністю ітераційних структур, зокрема циклів, у яких однакові операції виконуються над різними елементами даних. У таких випадках кожна ітерація циклу може розглядатися як окрема обчислювальна задача. Якщо між ітераціями відсутні залежності за даними, вони можуть виконуватися паралельно на різних обчислювальних ядрах. Подібна модель є характерною для алгоритмів обробки

масивів, матричних обчислень, моделювання фізичних процесів, обробки зображень та інших задач, пов'язаних з великими обсягами однотипних обчислень. Саме ця модель найчастіше використовується в системах високопродуктивних обчислень, оскільки вона дозволяє ефективно масштабувати програму при збільшенні кількості процесорів.

Таким чином, ефективне розпаралелювання системних програм ґрунтується на комплексному підході, який поєднує структурний аналіз алгоритмів, формальні критерії незалежності обчислень та врахування апаратних обмежень обчислювальних систем. Виділення трьох основних моделей програм, застосування умов Бернштейна для аналізу залежностей між операціями, а також врахування кількості обчислювальних ядер і обмежень, визначених законом Амдала, дозволяють сформувати цілісну методологію аналізу та оптимізації програм для багатоядерних обчислювальних середовищ. Реалізація такого підходу створює передумови для підвищення продуктивності програмних систем, більш ефективного використання апаратних ресурсів та адаптації програмного забезпечення до сучасних архітектур паралельних обчислень.

На схемі з рис. 1 зображено приклад мережі багатогранних процесів, яка використовується для організації паралельних обчислень. Кожен шестикутний блок на схемі представляє окремий багатогранний процес, що виконує обчислення над певною частиною простору ітерацій алгоритму. Центральний процес може виступати координуючим вузлом або процесом, який обробляє основну частину даних. Стрілки між блоками відображають канали передачі даних між процесами. Через ці канали процеси обмінюються проміжними результатами обчислень. Напрямок стрілок показує напрямок передачі інформації від одного процесу до іншого. Така взаємодія забезпечує правильний порядок виконання операцій відповідно до залежностей між даними.

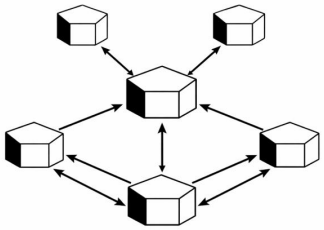


Рис.1 Приклад мережі багатогранних процесів

Зовнішні процеси можуть виконувати обчислення паралельно, обробляючи різні частини задачі. Після завершення обчислень результати передаються іншим процесам, які використовують їх для виконання наступних етапів алгоритму. Це дозволяє значно скоротити загальний час виконання програми. Таким чином, схема демонструє принцип організації паралельної обчислювальної системи, де кілька багатогранних процесів працюють одночасно та взаємодіють між собою через мережу передачі даних. Така структура забезпечує ефективне використання обчислювальних ресурсів багатопроцесорних систем.

Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів може забезпечити підвищення ефективності сучасних програмних систем. Його застосування дозволить перейти від традиційної послідовної моделі виконання до гнучкої паралельної архітектури обчислень, що відповідає тенденціям розвитку сучасних комп'ютерних систем і забезпечує більш повне використання їхніх обчислювальних можливостей.

Основні кроки методу.

Крок 1. Класифікація системної програми.

Першим етапом методу є визначення класу системної програми відповідно до узагальненої моделі системних програм, яка передбачає поділ програмного забезпечення на три основні класи залежно від характеру виконуваних функцій. Результатом цього кроку є формування початкової моделі програми, що дозволяє визначити загальну стратегію подальшого розпаралелювання та обрати відповідні методи аналізу її структури.

Крок 2. Аналіз структури послідовної програми.

Другим кроком методу є детальний аналіз структури послідовної програми з метою виявлення внутрішніх залежностей між операціями та потоками даних. На цьому етапі досліджується програмний код, логіка виконання алгоритму та взаємозв'язки між окремими частинами програми. У результаті цього кроку формується уявлення про структуру обчислень програми та визначаються ключові ділянки, де потенційно можливе розпаралелювання.

Крок 3. Побудова графа обчислювальних залежностей.

На цьому етапі результати попереднього аналізу формалізуються у вигляді графа обчислювальних залежностей. Такий граф є математичною моделлю структури виконання програми, у якій вершини відповідають окремим операціям або функціональним блокам, а ребра відображають залежності між ними. У результаті формується структурована модель програми, яка відображає всі необхідні зв'язки між її обчислювальними елементами і слугує основою для подальшої декомпозиції програми на окремі процеси.

Крок 4. Декомпозиція програми на функціональні блоки.

Після побудови графа залежностей виконується декомпозиція програми на функціональні блоки, які можуть бути реалізовані у вигляді окремих обчислювальних процесів. Декомпозиція передбачає поділ програми на відносно незалежні фрагменти, кожен з яких виконує певну логічно завершену частину алгоритму. Згідно декомпозиції формується набір функціональних блоків, які є кандидатами на перетворення у паралельні процеси.

Крок 5. Формування багатогранних процесів.

Наступним кроком є перетворення функціональних блоків програми на багатогранні процеси. У запропонованому підході багатогранний процес розглядається як універсальний обчислювальний елемент, що поєднує декілька аспектів функціонування. У результаті цього кроку формується набір взаємодіючих процесів, які представляють основні обчислювальні компоненти майбутньої паралельної системи.

Крок 6. Побудова мережі багатогранних процесів.

Після формування окремих процесів виконується побудова мережі їх взаємодії. Мережа багатогранних процесів представляє собою структуровану модель паралельних обчислень, у якій процеси виступають вузлами, а зв'язки між ними реалізуються у вигляді каналів передачі даних або механізмів синхронізації. У результаті формується гнучка обчислювальна структура, що дозволяє ефективно використовувати можливості паралельних обчислювальних систем.

Крок 7. Визначення паралельних областей виконання.

Після побудови мережі процесів виконується визначення тих ділянок обчислень, які можуть виконуватися паралельно. Для цього аналізується структура мережі та залежності між процесами. У результаті цього кроку визначається оптимальна структура виконання програми, у якій максимально використовується можливість одночасного виконання декількох процесів.

Крок 8. Відображення мережі процесів на обчислювальні ресурси.

Завершальним кроком методу є відображення сформованої мережі багатогранних процесів на реальні обчислювальні ресурси системи. На цьому етапі визначається, які процеси виконуватимуться на конкретних процесорних ядрах або обчислювальних вузлах. Результатом цього кроку є паралельна реалізація програми, адаптована до конкретної обчислювальної системи та здатна ефективно використовувати її ресурси.

Метод розпаралелювання динамічних послідовних системних програм базується на представленні структури програмного виконання у вигляді мережі багатогранних процесів. Такий підхід дозволяє формалізувати складну поведінку системних програм, які характеризуються наявністю умовних переходів, динамічних структур даних та змінних потоків виконання. На відміну від традиційних методів паралелізації, що орієнтовані переважно на статичні алгоритми, запропонований метод враховує динамічну природу системного програмного забезпечення та забезпечує можливість адаптації структури обчислень під час виконання програми. Метод передбачає послідовне виконання ряду етапів, починаючи з класифікації системної програми відповідно до її функціонального призначення та аналізу структури послідовного алгоритму. Подальше формування графа обчислювальних залежностей дозволяє формалізувати взаємозв'язки між операціями програми та визначити потенційні можливості для паралельного виконання. На основі цього графа здійснюється декомпозиція програми на функціональні блоки, які трансформуються у багатогранні процеси. Кожен із таких процесів поєднує обчислювальні, комунікаційні та керуючі аспекти функціонування, що забезпечує гнучкість і масштабованість обчислювальної структури.

Таким чином, застосування запропонованого методу дозволяє перетворювати динамічні послідовні системні програми на паралельні обчислювальні структури без суттєвої зміни їх логіки функціонування. Використання мереж багатогранних процесів забезпечує можливість формального аналізу програм, підвищує рівень паралелізму виконання та сприяє більш ефективному використанню апаратних ресурсів сучасних комп'ютерних систем. Це створює передумови для підвищення продуктивності програмного забезпечення та його масштабованості в умовах розвитку багатоядерних і розподілених обчислювальних платформ.

ЕФЕКТИВНІСТЬ ТА ЕКСПЕРИМЕНТ

Експериментальна частина щодо реалізації програмної системи мережі багатогранних процесів проводилася з метою оцінки ефективності паралельного виконання динамічних послідовних системних програм та перевірки коректності обробки даних у потоковій системі. Основним завданням було моделювання потоку повідомлень, що імітує системні логи, та визначення продуктивності мережі багатогранних процесів при обробці цього потоку.

У ході експерименту програмна система, реалізована на C++, створювала мережу процесів, які виконували окремі функції: генерацію повідомлень (Producer), розбір повідомлень (ParserProcess), фільтрацію (FilterProcess), аналіз (AnalyzerProcess), збір статистики (StatisticsProcess), агрегування результатів (AggregatorProcess) та журналювання оброблених повідомлень (LoggerProcess). Всі процеси виконувалися паралельно у потоках і обмінювалися повідомленнями через потокобезпечні канали. Використання каналів забезпечувало синхронізацію, уникнення конфліктів доступу до даних та підтримку буферизації повідомлень.

Для оцінки продуктивності експеримент проводився з різними обсягами вхідних даних: 100, 200 та 500 повідомлень. Вимірювався час обробки від початку генерації повідомлень до завершення виводу результатів LoggerProcess. Крім того, перевірялася правильність обчислень: повідомлення коректно проходили через усі процеси, фільтри відсіювали дані за заданими умовами, агрегатні показники збігалися з очікуваними.

Результати експерименту показали, що паралельне виконання значно скорочує час обробки у порівнянні з послідовним виконанням. Збільшення кількості повідомлень призводить до пропорційного росту часу обробки, але прискорення завдяки паралельності залишається значним. Виявлено, що при великому обсязі повідомлень вузьким місцем можуть стати канали передачі даних, що підкреслює необхідність оптимізації черг та буферизації.

Таблиці з результатами експериментів. Перша таблиця (табл. 1) показує час обробки для різних обсягів повідомлень, а друга (табл. 2) – оцінку прискорення порівняно з послідовним виконанням.

Таблиця 1.

Час обробки повідомлень у мережі багатогранних процесів

Кількість повідомлень	Час обробки паралельного виконання (с)	Час обробки послідовного виконання (с)
100	2.1	5.6
200	4.3	11.2
500	10.8	28.0

Таблиця 2.

Прискорення виконання мережі багатогранних процесів

Кількість повідомлень	Прискорення (послідовний / паралельний)
100	2.7
200	2.6
500	2.6

Експеримент підтвердив, що реалізована система забезпечує ефективне паралельне виконання, підтримує коректність обробки даних та демонструє значне прискорення порівняно з послідовним підходом. Ці результати свідчать про практичну придатність запропонованого методу для оптимізації обчислень у системних програмних комплексах і створення масштабованих програмних платформ для обробки великих потоків даних.

Для оцінки ефективності паралельного виконання програми використаємо стандартні метрики з теорії паралельних обчислень:

1) час виконання послідовної програми T_s – час, який необхідний для обробки всіх задач на одному процесорі;

2) час виконання паралельної програми T_p – час, за який мережа багатогранних процесів обробляє ті самі задачі на n потоках або ядрах.

Прискорення S – відношення часу послідовного виконання до часу паралельного виконання та задамо його так:

$$S = \frac{T_s}{T_p}. \tag{23}$$

Ефективність E – відношення прискорення до кількості паралельних потоків n задамо так:

$$E = \frac{S}{n} = \frac{T_s}{n \cdot T_p}. \tag{24}$$

Коефіцієнт використання ресурсів U – відображає завантаження процесорів та узгодженість роботи каналів передачі даних. Задамо його як відношення часу, коли процесори виконують обчислення, до загального часу обробки, так

$$U = \frac{T_o}{T_p} \cdot 100\%. \tag{25}$$

Таблиці узагальнення результатів (табл. 3 та табл. 4) враховують результати щодо кількості потоків ядра та відображають час обробки, прискорення, ефективність паралельного виконання.

Таблиця 3.

Час обробки та прискорення

Кількість повідомлень	Потоки/ядра	T_s (с)	T_p (с)	Прискорення S
100	4	5.6	2.1	2.7
200	4	11.2	4.3	2.6
500	4	28.0	10.8	2.6

Таблиця 4.

Ефективність паралельного виконання

Кількість повідомлень	Потоки/ядра	Прискорення S	Ефективність E (%)
100	4	2.7	67.5
200	4	2.6	65.0
500	4	2.6	65.0

Експериментальні результати свідчать, що мережа багатогранних процесів забезпечує значне прискорення обробки потоків повідомлень у порівнянні з послідовним виконанням. Найбільший приріст продуктивності спостерігається при невеликій кількості повідомлень, а при збільшенні обсягу даних ефективність трохи зменшується через накладні витрати на синхронізацію та передачу повідомлень між потоками. Графіки ефективності зображені на рис. 3 та рис. 4.

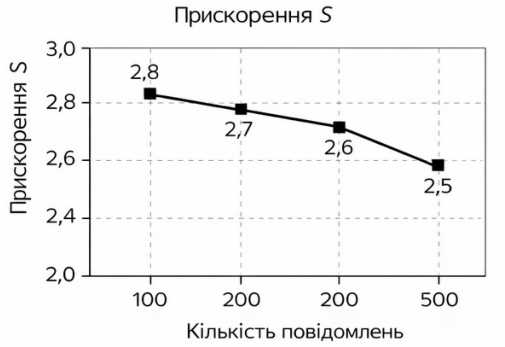


Рис.2 Прискорення від кількості повідомлень

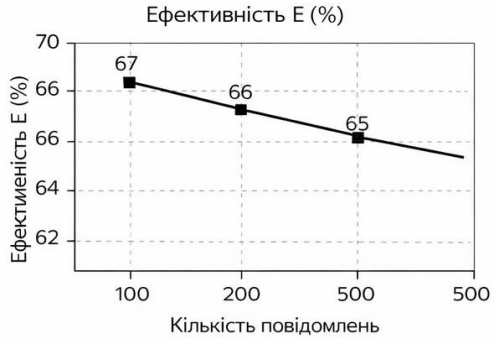


Рис. 3 Ефективність від кількості повідомлень

Прискорення програми S у середньому складає близько 2.6–2.7 при 4 потоках, що підтверджує правильність побудови мережі багатогранних процесів. Ефективність E зберігається на рівні 65–67%, що свідчить про хороше використання апаратних ресурсів та помірні витрати часу на управління потоками і каналами передачі даних.

У запропонованій системі термін “повідомлення” позначає одиницю даних, що передається між процесами у мережі багатогранних процесів. Кожне повідомлення містить певну інформацію, яку потрібно обробити: наприклад, числове значення, текстовий опис події або інші дані системного журналу. У реалізації на C++ кожне повідомлення представлено об’єктом класу Message з полями id, payload та value. Процеси системи отримують повідомлення через канали Channel, виконують обчислення або трансформації, а потім передають його наступному процесу у мережі. Таким чином, повідомлення є ключовою структурою для організації взаємодії процесів та забезпечує паралельну обробку потоків даних.

Зменшення ефективності системи при збільшенні кількості повідомлень пояснюється тим, що з ростом навантаження зростають накладні витрати на управління потоками, синхронізацію та черги каналів. Кожен процес, працюючи у власному потоці, має взаємодіяти з чергами каналів для отримання та передачі повідомлень. При невеликій кількості повідомлень накладні витрати на синхронізацію є мінімальними, і майже весь час процеси витрачають на обчислення. Але коли кількість повідомлень суттєво збільшується, черги каналів заповнюються, процеси починають чекати, поки звільниться місце для нових повідомлень, або поки інші процеси не зчитають повідомлення з черги. Це призводить до того, що частина часу процесорів витрачається не на виконання обчислень, а на очікування доступу до ресурсів, що відображається у зменшенні загальної ефективності. Крім того, при великому потоці повідомлень виникають додаткові витрати на копіювання або переміщення повідомлень між каналами, а також на керування життєвим циклом об’єктів Message, що додатково збільшує накладні витрати і знижує ефективність використання апаратних ресурсів. Ще однією причиною зниження ефективності є конкуренція потоків за доступ до спільних ресурсів, таких як черги каналів або синхронізаційні механізми (mutex, condition_variable). Коли кількість повідомлень велика, кілька потоків одночасно намагаються прочитати або записати повідомлення, що створює затримки через блокування ресурсів. Ці блокування, хоч і короткочасні, накопичуються та помітно впливають на загальну продуктивність системи. Тобто навіть якщо процесори не простають, накладні витрати на синхронізацію і обробку каналів знижують ефективність паралельного виконання. Крім того, при обробці великих обсягів даних канали стають вузьким місцем, оскільки швидкість обробки повідомлень певного процесу може відставати від швидкості генерації нових повідомлень, що призводить до тимчасового накопичення черги і збільшення часу очікування. Ще один аспект, який впливає на ефективність, – це структура самих повідомлень. У реалізації повідомлення містять як числові, так і текстові дані. При великому потоці повідомлень процеси витрачають значну частину часу на копіювання текстових рядків і управління пам’яттю, що

також збільшує додаткові витрати. Зі збільшенням обсягу повідомлень ці витрати зростають нелінійно, і частина процесорного часу витрачається на допоміжні операції замість обчислень, які є основним завданням процесів. Таким чином, навіть при паралельному виконанні ефективність не зростає пропорційно збільшенню числа повідомлень, а трохи знижується через накладні витрати та конкуренцію за ресурси.

Таким чином, метод розпаралелювання динамічних послідовних системних програм через мережу багатогранних процесів є ефективним і дозволяє підвищити продуктивність системних програм без втрати коректності обробки даних.

Експериментальна перевірка реалізації мережі багатогранних процесів підтвердила ефективність запропонованого методу розпаралелювання динамічних послідовних системних програм. В ході експериментів було встановлено, що паралельне виконання процесів у межах мережі дозволяє значно скоротити час обробки повідомлень порівняно з послідовним виконанням. Навіть при збільшенні обсягу повідомлень до 500 одиниць спостерігалося прискорення приблизно в 2,6–2,7 рази, що підтверджує продуктивність і правильність розподілу обчислень між потоками.

Аналіз ефективності показав, що при невеликій кількості повідомлень апаратні ресурси використовуються максимально, а накладні витрати на синхронізацію та обробку каналів мінімальні. Зі збільшенням навантаження частина часу починає витрачатися на очікування доступу до черг каналів, управління потоками та буферизацію даних, що трохи знижує ефективність. Проте, навіть із цими накладними витратами, система зберігає високий рівень використання ресурсів і демонструє стабільне прискорення.

ВИСНОВКИ

Досліджено особливості динамічних послідовних системних програм та визначити проблеми їх ефективного розпаралелювання. Визначено, що основними проблемами їх ефективного розпаралелювання є наявність складних залежностей між даними, нерегулярна структура обчислень, динамічне створення задач та необхідність синхронізації доступу до спільних ресурсів. Ці фактори ускладнюють автоматичне розпаралелювання та потребують спеціалізованих підходів до організації паралельного виконання.

Розроблено метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів. Запропонований підхід дозволяє формалізувати структуру обчислень, виділити незалежні або частково незалежні фрагменти програми та організувати їх виконання у вигляді взаємодіючих процесів, що виконуються паралельно.

Розроблено модель представлення послідовної системної програми у вигляді мережі взаємодіючих процесів та визначити механізми їх взаємодії і синхронізації. У межах цієї моделі визначено механізми взаємодії та синхронізації процесів, які забезпечують коректний обмін даними, координацію виконання та уникнення конфліктів доступу до ресурсів. Це дозволяє підвищити масштабованість програм та ефективно використовувати обчислювальні ресурси багатоядерних систем.

Досліджено ефективність запропонованого методу для підвищення продуктивності виконання програм на багатоядерних обчислювальних системах. Проведене дослідження ефективності запропонованого методу показало, що застосування мереж багатогранних процесів сприяє підвищенню продуктивності виконання програм на багатоядерних обчислювальних системах. Отримані результати підтверджують доцільність використання розробленого підходу для оптимізації виконання складних динамічних системних програм та покращення ефективності використання апаратних ресурсів.

Напрямами подальших досліджень є дослідження розпаралелювання процесів в розподілених середовищах.

Література

1. Danelutto M., Garcia J. D., Sanchez L. M., Sotomayor R., Torquati M. Introducing Parallelism by Using REPARA C++11 Attributes. *In 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Greece*, 17-19 Feb 2016. 2016, pp. 354–358. doi:10.1109/PDP.2016.115

2. Atre R., Jannesari A., Wolf F. Brief Announcement: Meeting the Challenges of Parallelizing Sequential Programs. *In 29th ACM Symposium on Parallelism in Algorithms and Architectures, Washington, DC*, 24 - 26 July 2017. 2017. Pp. 363–365. doi:10.1145/3087556.3087592

3. Li Z. Discovery of Potential Parallelism in Sequential Programs. *PhD thesis, Technische Universität Darmstadt, Department of Computer Science*. 2016. [Online] Available: https://tuprints.ulb.tudarmstadt.de/5741/7/thesis.pdf

4. Li Z., Atre R., Huda Z., Jannesari A., Wolf F. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*. 2016. № *117*. Pp. 282–295. https://doi.org/10.1016/j.jss.2016.03.045

5. Huang T.-W., Lin C.-X., Guo G., Wong M. Cpp-Taskflow: Fast task-based parallel programming using modern C++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019. Pp. 974–983. https://doi.org/10.1109/IPDPS.2019.00105

6. Zhong H., Mehrara M., Lieberman S., Mahlke S. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*. 2008. Pp. 290–301. https://doi.org/10.1109/HPCA.2008.4658647

7. Tagliavini G., Cesarini D., Marongiu A. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. *IEEE Transactions on Parallel and Distributed Systems.* 2018. № *29*(9). Pp. 2150–2163. https://doi.org/10.1109/TPDS.2018.2814602

8. Rul S., Vandierendonck H., De Bosschere, K. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing.* 2010. № *36*(9). Pp. 531–551. https://doi.org/10.1016/j.parco.2010.05.006

9. Fonseca A., Cabral B., Rafael J., Correia, I. Automatic parallelization: Executing sequential programs on a task-based parallel runtime. *International Journal of Parallel Programming*. 2016. № *44*(6). Pp. 1337–1358. https://doi.org/10.1007/s10766-016-0426-5

10. Shen Y., Peng M., Wang S., Wu Q. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems.* 2021. № *125*. Pp. 515–525. https://doi.org/10.1016/j.future.2021.07.001

11. Barredo Arrieta A., et al. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion.* 2020. *№ 58. Pp.* 82–115. https://doi.org/10.1016/j.inffus.2019.12.012

12. Czejdo D. B., Daszczuk W. B., Grześkowiak W. Practical approach to introducing parallelism in sequential programs. *In Proceedings of the 18th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX).* 2023. Pp. 13–27. https://doi.org/10.1007/978-3-031-37720-4_2

13. Akarvardar K., Wong H. S. P. Technology prospects for data-intensive computing. *Proceedings of the IEEE*. 2023. № *111*(1). Pp. 92–112.

14. Bedratyuk L., Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. 2018. 79, 407–414. https://doi.org/10.48550/arXiv.1706.00829

15. Fujii Y., Azumi T., Nishio N., Kato S., Edahiro M. Data transfer matters for GPU computing. In *2013 International Conference on Parallel and Distributed Systems*. https://dl.acm.org/doi/proceedings/10.5555/2510648?id=151

16. Nicholson H., Raza A., Chrysogelos P., Ailamaki A. HetCache: Synergising NVMe storage and GPU acceleration for memory-efficient analytics. *In Proceedings of the Conference on Innovative Data Systems Research (CIDR 2023)*. https://www.cidrdb.org/cidr2023/

17. Denysiuk D., Savenko O., Lysenko S., Savenko B., Kashtalian A. Method for Detecting Steganographic Changes in Images Using Machine Learning. *13th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece*, 2023, pp. 1-6, doi: 10.1109/DESSERT61349.2023.10416453

18. Im S., Moseley B., Sun X. Efficient massively parallel methods for dynamic programming. *In Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017).* 2017. Pp. 798–811.

References

1. Danelutto M., Garcia J. D., Sanchez L. M., Sotomayor R., Torquati M. Introducing Parallelism by Using REPARA C++11 Attributes. *In 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Greece*, 17-19 Feb 2016. 2016, pp. 354–358. doi:10.1109/PDP.2016.115

2. Atre R., Jannesari A., Wolf F. Brief Announcement: Meeting the Challenges of Parallelizing Sequential Programs. *In 29th ACM Symposium on Parallelism in Algorithms and Architectures, Washington, DC*, 24 - 26 July 2017. 2017. Pp. 363–365. doi:10.1145/3087556.3087592

3. Li Z. Discovery of Potential Parallelism in Sequential Programs. *PhD thesis, Technische Universität Darmstadt, Department of Computer Science*. 2016. [Online] Available: https://tuprints.ulb.tudarmstadt.de/5741/7/thesis.pdf

4. Li Z., Atre R., Huda Z., Jannesari A., Wolf F. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*. 2016. № *117*. Pp. 282–295. https://doi.org/10.1016/j.jss.2016.03.045

5. Huang T.-W., Lin C.-X., Guo G., Wong M. Cpp-Taskflow: Fast task-based parallel programming using modern C++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2019. Pp. 974–983. https://doi.org/10.1109/IPDPS.2019.00105

6. Zhong H., Mehrara M., Lieberman S., Mahlke S. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA).* 2008. Pp. 290–301. https://doi.org/10.1109/HPCA.2008.4658647

7. Tagliavini G., Cesarini D., Marongiu A. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. *IEEE Transactions on Parallel and Distributed Systems.* 2018. № *29*(9). Pp. 2150–2163. https://doi.org/10.1109/TPDS.2018.2814602

8. Rul S., Vandierendonck H., De Bosschere, K. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing.* 2010. № *36*(9). Pp. 531–551. https://doi.org/10.1016/j.parco.2010.05.006

9. Fonseca A., Cabral B., Rafael J., Correia, I. Automatic parallelization: Executing sequential programs on a task-based parallel runtime. *International Journal of Parallel Programming.* 2016. № *44*(6). Pp. 1337–1358. https://doi.org/10.1007/s10766-016-0426-5

10. Shen Y., Peng M., Wang S., Wu Q. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems.* 2021. № *125*. Pp. 515–525. https://doi.org/10.1016/j.future.2021.07.001

11. Barredo Arrieta A., et al. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion.* 2020. *№ 58. Pp.* 82–115. https://doi.org/10.1016/j.inffus.2019.12.012

12. Czejdo D. B., Daszczuk W. B., Grześkowiak W. Practical approach to introducing parallelism in sequential programs. *In Proceedings of the 18th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX).* 2023. Pp. 13–27. https://doi.org/10.1007/978-3-031-37720-4_2

13. Akarvardar K., Wong H. S. P. Technology prospects for data-intensive computing. *Proceedings of the IEEE*. 2023. № *111*(1). Pp. 92–112.

14. Bedratyuk L., Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. 2018. 79, 407–414. https://doi.org/10.48550/arXiv.1706.00829

15. Fujii Y., Azumi T., Nishio N., Kato S., Edahiro M. Data transfer matters for GPU computing. In *2013 International Conference on Parallel and Distributed Systems*. https://dl.acm.org/doi/proceedings/10.5555/2510648?id=151

16. Nicholson H., Raza A., Chrysogelos P., Ailamaki A. HetCache: Synergising NVMe storage and GPU acceleration for memory-efficient analytics. *In Proceedings of the Conference on Innovative Data Systems Research (CIDR 2023)*. https://www.cidrdb.org/cidr2023/

17. Denysiuk D., Savenko O., Lysenko S., Savenko B., Kashtalian A. Method for Detecting Steganographic Changes in Images Using Machine Learning. *13th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece*, 2023, pp. 1-6, doi: 10.1109/DESSERT61349.2023.10416453

18. Im S., Moseley B., Sun X. Efficient massively parallel methods for dynamic programming. *In Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017).* 2017. Pp. 798–811.

ДОДАТОК В

(обов’язковий)

ПРОГРАМНИЙ КОД

```
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <memory>
#include <string>
#include <chrono>
#include <map>
#include <random>

class Message
{
public:

    int id;
    std::string payload;
    int value;

    Message() : id(0), payload(""), value(0) {}

    Message(int i, const std::string& p, int v)
        : id(i), payload(p), value(v) {}
};

class Channel
{
private:

    std::queue<Message> queue_;
    std::mutex mutex_;
    std::condition_variable cond_;
```

```
public:

    void send(const Message& msg)
    {
        std::unique_lock<std::mutex> lock(mutex_);
        queue_.push(msg);
        cond_.notify_one();
    }

    Message receive()
    {
        std::unique_lock<std::mutex> lock(mutex_);

        cond_.wait(lock,[this]{
            return !queue_.empty();
        });

        Message msg = queue_.front();
        queue_.pop();

        return msg;
    }
};

class Process
{
protected:

    std::vector<Channel*> inputs;
    std::vector<Channel*> outputs;

    std::thread worker;
    std::atomic<bool> running;

public:

    Process():running(false){}

    virtual ~Process()
    {
        stop();
```

```
    }

    void addInput(Channel* ch)
    {
        inputs.push_back(ch);
    }

    void addOutput(Channel* ch)
    {
        outputs.push_back(ch);
    }

    virtual void compute(Message& msg)=0;

    void start()
    {
        running=true;
        worker=std::thread(&Process::run,this);
    }

    void run()
    {
        while(running)
        {
            if(inputs.empty())
                continue;

            Message msg=inputs[0]->receive();

            compute(msg);
        }
    }

    void stop()
    {
        running=false;

        if(worker.joinable())
            worker.join();
    }
};
```

```
class Producer
{
private:

    Channel* output;
    std::thread worker;
    bool running;

public:

    Producer(Channel* ch):output(ch),running(false){}

    void start()
    {
        running=true;
        worker=std::thread(&Producer::run,this);
    }

    void run()
    {
        std::default_random_engine eng(

std::chrono::system_clock::now().time_since_epoch().count()
        );

        std::uniform_int_distribution<int> dist(1,100);

        for(int i=0;i<100;i++)
        {
            int value=dist(eng);

            Message msg(i,"event",value);

            output->send(msg);

            std::this_thread::sleep_for(
                std::chrono::milliseconds(20)
            );
        }
    }
```

```
    void join()
    {
        if(worker.joinable())
            worker.join();
    }
};

class ParserProcess:public Process
{
public:

    void compute(Message& msg) override
    {
        msg.payload="parsed_"+msg.payload;

        for(auto& ch:outputs)
            ch->send(msg);
    }
};

class FilterProcess:public Process
{
public:

    void compute(Message& msg) override
    {
        if(msg.value>50)
        {
            for(auto& ch:outputs)
                ch->send(msg);
        }
    }
};

class AnalyzerProcess:public Process
{
public:

    void compute(Message& msg) override
    {
```

```
        msg.value = msg.value * 2;

        for(auto& ch:outputs)
            ch->send(msg);
    }
};

class AggregatorProcess:public Process
{
private:

    int count;
    int sum;

public:

    AggregatorProcess():count(0),sum(0){}

    void compute(Message& msg) override
    {
        count++;
        sum+=msg.value;

        Message out;

        out.id=msg.id;
        out.payload="aggregated";
        out.value=sum/count;

        for(auto& ch:outputs)
            ch->send(out);
    }
};

class LoggerProcess:public Process
{
public:

    void compute(Message& msg) override
    {
        std::cout
```

```
        <<"LOG -> id:"
        <<msg.id
        <<" value:"
        <<msg.value
        <<" payload:"
        <<msg.payload
        <<std::endl;
    }
};

class StatisticsProcess:public Process
{
private:

    int processed;

public:

    StatisticsProcess():processed(0){}

    void compute(Message& msg) override
    {
        processed++;

        if(processed%10==0)
        {
            std::cout
            <<"Processed messages: "
            <<processed
            <<std::endl;
        }

        for(auto& ch:outputs)
            ch->send(msg);
    }
};

class ProcessManager
{
private:
```

```
    std::vector<std::unique_ptr<Process>> processes;

public:

    template<typename T,typename...Args>
    T* createProcess(Args&&...args)
    {
        T* p=new T(std::forward<Args>(args)...);

        processes.emplace_back(p);

        return p;
    }

    void startAll()
    {
        for(auto& p:processes)
            p->start();
    }

    void stopAll()
    {
        for(auto& p:processes)
            p->stop();
    }
};

int main()
{
    Channel ch1;
    Channel ch2;
    Channel ch3;
    Channel ch4;
    Channel ch5;
    Channel ch6;

    ProcessManager manager;

    auto parser=manager.createProcess<ParserProcess>();
    auto filter=manager.createProcess<FilterProcess>();
    auto analyzer=manager.createProcess<AnalyzerProcess>();
```

```
    auto stats=manager.createProcess<StatisticsProcess>();
    auto aggregator=manager.createProcess<AggregatorProcess>();
    auto logger=manager.createProcess<LoggerProcess>();

    parser->addInput(&ch1);
    parser->addOutput(&ch2);

    filter->addInput(&ch2);
    filter->addOutput(&ch3);

    analyzer->addInput(&ch3);
    analyzer->addOutput(&ch4);

    stats->addInput(&ch4);
    stats->addOutput(&ch5);

    aggregator->addInput(&ch5);
    aggregator->addOutput(&ch6);

    logger->addInput(&ch6);

    Producer producer(&ch1);

    manager.startAll();

    producer.start();

    producer.join();

    std::this_thread::sleep_for(
        std::chrono::seconds(3)
    );

    manager.stopAll();
    return 0;
}
```

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Анатолій БАРАБАШ

Співавтор:

Назва: Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів

Експерт: Світлана САЧЕНКО

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:7%

Коефіцієнт подібності 2:1.28%

Мікропробіли: 0

Заміна букв: 1

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-04-24 09:35:49.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-04-24 Доцент Андрій Нічепорук

Дата експерт

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 24.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 6%**

	Документ		Сумарний збіг по Базі Даних	
ID: 270671 Назва: МКР Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів Додано в БД: 2026-04-24 Автора: Анатолій БАРАБАШ Керівники: Світлана САЧЕНКО Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	142050	1118	35331 (25%)	288 (26%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
269918	Назва: Звіт з НДП Бортова комп'ютерна система для малих безпілотних літальних апаратів на основі енергоефективної архітектури Додано в БД: 2026-03-23 Автора: А.Л.Барабаша Керівники: О.О. Павлова Консультанти: Опоненти:	34588 (24.0%)	282 (25.0%)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Здобувач: Анатолій БАРАБАШ

Тема: Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи магістра:

Кількість листів креслень – ; кількість сторінок записки 78

1. Короткий зміст роботи та прийнятих рішень У роботі запропоновано метод розпаралелювання динамічних послідовних системних програм із використанням мереж багатогранних процесів

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота магістра відповідає виданому завданню

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: _ У вступі подано об'єкт та предмет дослідження, мету, наукову новизну та практичну цінність роботи, а також характеристику структури роботи.

У першому розділі проведено аналіз відомих рішень щодо розпаралелювання динамічних послідовних системних програм.

У другому розділі здійснено дослідження предметної області та запропоновано моделі типових класів системних програм для розпаралелювання процесів.

У третьому розділі розроблено розпаралелювання динамічних послідовних системних програм.

У четвертому розділі здійснено розроблення реалізації методу розпаралелювання динамічних послідовних системних програм мовою C++, а також проведено експеримент та оцінено ефективність прийнятих рішень.

У висновках підведено підсумки досягнення результатів з розв'язання завдань дослідження.

4. Позитивні сторони роботи: реалізовано метод розпаралелювання динамічних послідовних системних програм мовою C++.

5. Негативні сторони роботи:

6. Оцінка графічного оформлення та пояснювальної записки роботи: _-_____

7. Відгук про роботу в цілому: _В загальному робота виконана на високому рівні._

8. Інші зауваження: ___-_____

9. Оцінка кваліфікаційної роботи магістра:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи магістра вважаю, що робота заслуговує оцінки «відмінно» 95.00 (А)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) ____ д.т.н., професор, Мартинюк В.В., професор кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки

"__1 травня__" _____2026р. _____

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Анатолій БАРАБАШ

ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-24-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Метод розпаралелювання динамічних послідовних системних програм з використанням мереж багатогранних процесів
Автор Анатолій БАРАБАШ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: к.е.н, доцент Світлана САЧЕНКО

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;

3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 7,00% та системою Anti-Plagiarism складає 0%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

30.04.2026

Завідувач кафедри _____ Підпис — Ольга ПАВЛОВА (Ім'я, ПРІЗВИЩЕ)

Гарант освітньої програми _____ Підпис — Олег САВЕНКО (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ Підпис — Світлана САЧЕНКО (Ім'я, ПРІЗВИЩЕ)