https://doi.org/10.31891/2219-9365-2026-86-5
УДК 004.75

БАРАБАШ Анатолій
Хмельницький національний університет
https://orcid.org/0009-0000-9607-1775
e-mail: wkondrael@gmail.com
ЛИГУН Олексій
Хмельницький національний університет
https://orcid.org/0009-0004-5727-5096
e-mail: oleksii.lyhun@gmail.com
ДРОЗД Андрій
Хмельницький національний університет
https://orcid.org/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

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.

Стаття надійшла до редакції / Received 10.03.2026
Прийнята до друку / Accepted 22.04.2026
Опубліковано / Published 31.05.2026

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

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 ОСТАННІХ ДОСЛІДЖЕНЬ І 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:

$$\begin{gathered} R_i \cap W_j = \emptyset, \\ R_j \cap W_i = \emptyset, \\ W_i \cap W_j = \emptyset, \end{gathered} \tag{4}$$

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:

$$\begin{gathered} R(p_i) \cap W(p_j) = \emptyset \\ R(p_j) \cap W(p_i) = \emptyset \\ W(p_i) \cap W(p_j) = \emptyset \end{gathered} \tag{7}$$

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

$$P = P_1 \cup P_2 \cup \dots \cup P_k, \tag{8}$$

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:

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

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

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

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:

$$S_{max} = \frac{1}{\alpha}. \quad (21)$$

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:

$$S_{max} = \frac{1}{0,1} = 10. \quad (22)$$

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.

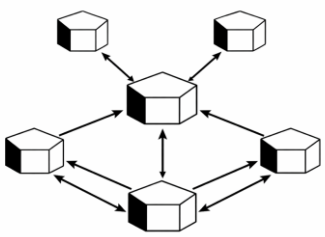

Рис.1 Приклад мережі багатогранних процесів

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 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. 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.

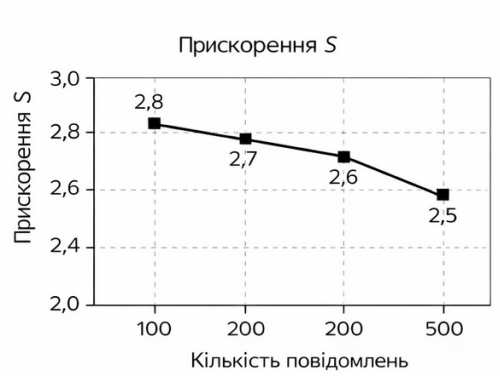

Рис.2 Прискорення від кількості повідомлень

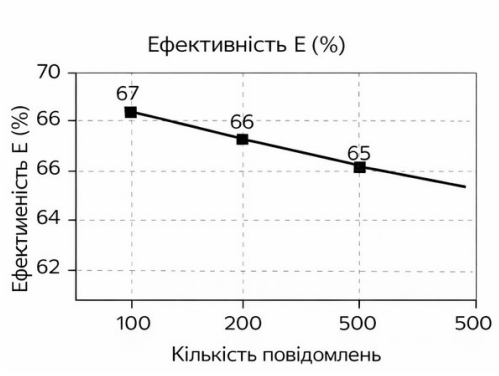

Рис. 3 Ефективність від кількості повідомлень

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 З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

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.

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.