Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

ДИФУЧИНА ОЛЕКСАНДРА ЮРІЇВНА

УДК 004.41::004.94

ДИСЕРТАЦІЯ
МЕТОД ОПТИМІЗАЦІЇ ПАРАМЕТРІВ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ НА ОСНОВІ ПЕТРІ-ОБ'ЄКТНОГО МОДЕЛЮВАННЯ

126 Інформаційні системи та технології
12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело_____О.Ю. Дифучина

Науковий керівник Павлов Олександр Анатолійович, д.т.н., професор

Київ – 2024

АНОТАЦІЯ

Дифучина О.Ю. Метод оптимізації параметрів паралельних обчислень на основі Петрі-об’єктного моделювання. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 126 – Інформаційні системи та технології з галузі знань 12 – Інформаційні технології. – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, 2024.

Дисертаційна робота присвячена розробці методів та засобів дослідження впливу параметрів паралельних обчислень на швидкодію обчислень. Сучасні інформаційні технології потребують швидкої роботи алгоритмів, яку можна досягти за рахунок використання паралельних обчислень. Проте, в залежності від параметрів, що визначають характеристики підзадач та механізми їх взаємодії, використання паралельних обчислень може призвести як до прискорення, так і до сповільнення обчислень. На практиці проблему налаштування параметрів паралельних обчислень вирішують шляхом багатократного тестування програми на різних комп’ютерних платформах при різних наборах параметрів, щоб гарантувати коректність та ефективність обчислень. Через велику кількість варіантів захоплення обчислювального ресурсу та, відповідно, варіантів виконання інструкцій окремими процесами паралельних обчислень результат запуску однієї і тієї ж програми в однакових умовах може суттєво відрізнятись. Тому тестування паралельних алгоритмів в реальних умовах є ресурсовитратним.

Математичні методи оцінювання ефективності паралельних обчислень спроможні вказати на існування обмеження на максимально досяжне прискорення за ідеальних умов вільного доступу до обчислювального ресурсу та відсутності синхронізації обчислень (тобто фактично відсутності взаємодії між частинами програми).

Існуючі засоби проєктування програм, такі як UML, дають змогу (досить узагальнено) представити графічно взаємодію окремих частин програми, проте не надають можливості будь-якого чисельного аналізу обчислень. Засоби моделювання, такі як високорівневі мережі Петрі, рекомендовані міжнародним стандартом ISO/IEC 15909-1:2019 як техніка для специфікації паралельних і розподілених систем. На сьогоднішній день є досвід розробки симуляторів обчислень на основі мереж Петрі, проте жоден з них не став на сьогоднішній день широко використовуваним у розробці паралельних обчислень.

Таким чином, на сьогодні не існує уніфікованого методу створення моделі паралельних обчислень і, відповідно, не існує іншого, окрім реальної програми, засобу, який можна використовувати для оптимізації параметрів паралельної програми. Відсутність засобів моделювання стримує розробку високоефективних паралельних обчислень. З огляду на це, створення методів та засобів, спрямованих на вдосконалення процесу налагодження багатопотокових програм та підвищення ефективності використання паралельних обчислень в інформаційній технології, є актуальним науковим завданням.

Метою наукового дослідження є підвищення ефективності використання паралельних обчислень в інформаційних технологіях за рахунок їх проєктування на основі моделей, що можуть бути використані для оцінювання часу виконання паралельного алгоритму, та оптимізації параметрів паралельних обчислень.

У **першому розділі** обґрунтовано необхідність ретельного проєктування паралельних обчислень для досягнення їх ефективності, розглянуті методи та засоби для проєктування паралельних обчислень та аналізу їх ефективності. Виявлено, що математичні методи аналізу ефективності паралельних обчислень здатні оцінити максимально досяжне прискорення за досить ідеальних умов вільного доступу до обчислювального ресурсу та відсутності синхронізації обчислень. На противагу, засоби імітаційного моделювання спроможні достатньо детально відтворювати паралельні процеси і, на відміну від експериментування з реальною програмою паралельних обчислень, не

потребують значних витрат на проведення експериментального дослідження. Серед симуляторів паралельних обчислень багато таких, що використовують формалізм мережі Петрі як засіб опису процесу обчислень. Проте, моделюванню, як правило, підлягають найпростіші елементи паралельної програми: розділення на підзадачі, очікування завершення виконання підзадачі, блокування. Більш складні інструменти взаємодії між потоками, як wait/notify, розглядаються лише в окремих публікаціях. У моделях не враховують обмеженість використовуваного паралельною програмою ресурсу, що, звісно, негативно впливає на їх точність. Розглянуті також засоби тестування паралельних програм, які призначені для виявлення взаємоблокування та/або помилки узгодженості пам'яті у вже розроблених програмах. Такі засоби спрямовані, насамперед, на аналіз коректності виконання обчислень, але не на аналіз ефективності паралельних досліджень. У підсумку, зроблено висновок щодо необхідності розвитку методів та засобів для оцінювання ефективності паралельних обчислень на основі моделей, які враховують механізм захоплення обчислювального ресурсу та механізми взаємодії між підзадачами, що виконуються паралельно.

У **другому розділі** сформульовані мета та задача моделювання паралельних обчислень, визначено формальний опис моделі та виконано розробку методу моделювання паралельних обчислень на основі цього опису. Для формального опису обрано формалізм Петрі-об'єктної моделі у зв'язку з низкою переваг: найбільш точне (у порівнянні зі звичайною мережею Петрі) відтворення структури об'єктно-орієнтованої програми та поведінкових властивостей окремих об'єктів; більш точне (у порівнянні зі звичайною мережею Петрі) відтворення взаємодії потоків/підзадач за рахунок враховування часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу; тиражування об'єктів зі схожою поведінкою із заданими параметрами спрощує створення моделей з великою кількістю підзадач; тиражування груп зв'язків між об'єктами спрощує конструювання

моделі з великою кількістю зв'язків; візуалізація поведінкових властивостей моделі дає змогу розробити програму з найменшою кількістю помилок. Визначені та розроблені шаблони моделювання обчислень багатопотокової програми, які зменшують кількість помилок при розробці моделей, спрощують та прискорюють процес розробки моделі. Набір шаблонів моделювання містить фрагменти мереж Петрі, що відтворюють рутинні інструкції програми, створення, початок та завершення роботи потоку, призупинку потоку на заданий інтервал часу, блокування дій потоку, очікування за умовою, обчислення підзадач пулом потоків. Описано процес розробки Петрі-об'єктної моделі паралельної програми. Усі запропоновані та розроблені методи, підходи та засоби моделювання поєднані у технологію моделювання паралельного алгоритму. Наведено приклади розробки моделей.

У **третьому розділі** сформульована задача оптимізації параметрів паралельних обчислень, обґрунтовано існування оптимальних значень та визначені шляхи пошуку оптимальних значень на моделі, визначені методи та засоби збору даних для моделі, описані методи та засоби розробки Петрі-об'єктної моделі паралельних обчислень. Програмне забезпечення Parallel Program Simulation (PPS) реалізує такі функції для підтримки розробки моделі: розробка мережі Петрі у графічному редакторі мереж Петрі; збереження мережі Петрі у двох форматах - графічне зображення та у вигляді методу; дослідження параметрів часової затримки; експериментальне дослідження моделі; пошук оптимальних значень параметрів паралельних обчислень еволюційним методом.

У **четвертому розділі** метод оптимізації параметрів паралельних обчислень, описаний у розділі 3, апробовано на прикладах паралельного алгоритму імітації дискретно-подійної системи та пулу потоків. Паралельний алгоритм імітації є складним як за кількістю обчислювальних дій, так і за механізмами взаємодії підзадач, які забезпечують злагоджені дії усіх частин моделі. Теоретично та експериментально доведено наявність впливу параметрів паралельного алгоритму на швидкодію його виконання. Встановлено, що

параметром, від якого у найбільшій мірі залежить час виконання алгоритму – це складність одного фрагменту моделі, який запускається на одночасне виконання. Виконано дослідження оптимального значення параметру при зростанні складності моделі. Побудована Петрі-об'єктна модель паралельного алгоритму та виконано пошук оптимальних параметрів на моделі експериментально. Еволюційний алгоритм апробовано на Петрі-об'єктній моделі пулу потоків. Знайдені оптимальні значення в обох випадках достатньо точно відповідають таким, що були виявлені при експериментуванні з паралельними алгоритмами в реальних умовах. Отримані результати свідчать про коректність пошуку оптимальних параметрів на моделі.

Результати, отримані у дисертаційному дослідженні, містять **наукову новизну**:

- **вперше** розроблено технологію моделювання паралельних обчислень на основі Петрі-об'єктного підходу, що надає можливість скоротити ресурсні витрати при розробці паралельних алгоритмів, і, на відміну від існуючих, дає змогу відтворити деталізовано структуру паралельної програми та механізми взаємодії одночасно виконуваних частин програми з урахуванням часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу і спрощує процес побудови моделі за рахунок тиражування фрагментів програми зі схожою функціональністю;
- **удосконалено** моделі базових механізмів синхронізації паралельних обчислень за рахунок підвищення точності відтворення, що забезпечує високу точність результатів моделювання;
- **вперше** розроблено типові фрагменти мереж Петрі, що реалізують механізми багатопотокової технології Java, використання яких прискорює розробку моделі паралельного алгоритму за рахунок зменшення кількості помилок та зменшення загальної кількості елементів, необхідних для розробки моделі;

- **вперше** запропоновано метод оптимізації параметрів паралельних обчислень на основі експериментального дослідження Петрі-об’єктної моделі обчислень, що забезпечує ефективне використання обчислювальних ресурсів і, на відміну від існуючих підходів, дає змогу проводити експериментальне дослідження ефективності паралельних обчислень на моделі замість експериментування на реальній програмі.

Практичне значення результатів дисертаційного дослідження полягає у розробленому програмному забезпеченні для моделювання паралельних обчислень та оптимізації їх параметрів на основі Петрі-об’єктного моделювання, що є одним з результатів виконання науково-дослідної роботи «Методи візуального програмування Петрі-об'єктних моделей» (державний реєстраційний номер 0117U000918). Технологія моделювання паралельних обчислень мережею Петрі впроваджена у рамках навчального дистанційного курсу «Технології паралельних обчислень» (сертифікат ДК № 0098, затверджений протоколом №8 від 02.06.2023 Методичної ради КПІ ім. Ігоря Сікорського).

Результати дисертаційної роботи опубліковано у 9 наукових публікаціях, серед яких 3 статті у періодичних наукових виданнях, проіндексованих у Web of Science Core Collection та Scopus базах даних (дві з них у виднях, віднесених до третього квартиля (Q3)), 1 стаття у фаховому науковому журналі категорії «Б» (зі спеціальності 126), 1 стаття у фаховому науковому журналі з переліку до 12.03.2020 р. (технічні науки), 3 публікації у матеріалах міжнародних наукових конференцій, 1 публікація у матеріалах всеукраїнської наукової конференції.

Ключові слова: багатопотокове програмування, паралельні обчислення, оптимізація, імітаційне моделювання, стохастична мережа Петрі, Петрі-об’єктне моделювання.

Список публікацій здобувача

Наукові праці, в яких опубліковано основні наукові результати дисертації:

Стаття у періодичному науковому виданні, проіндексованому у базах даних Web of Science Core Collection та Scopus:

1. Stetsenko I.V., Pavlov O.A., Dyfuchyna O. (2021) Parallel algorithm development and testing using Petri-object simulation. *International Journal of Parallel, Emergent and Distributed Systems*, 36(6), 549-564. Taylor and Francis Ltd. ISSN 1744-5779. https://doi.org/10.1080/17445760.2021.1955113

2. Stetsenko I.V., Dyfuchyna O. (2020) Thread Pool parameters tuning using simulation. *Advances in Intelligent Systems and Computing*, 938, 78-89. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-030-16621-2_8

3. Stetsenko I.V., Dyfuchyna O. (2019) Simulation of multithreaded algorithms using Petri-object models. *Advances in Intelligent Systems and Computing*, 754, 391-401. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-319-91008-6_39

Статті у наукових виданнях, включених на дату опублікування до переліку наукових фахових видань України:

4. Дифучина О.Ю. (2023) Метод оптимізації параметрів паралельних обчислень. *Технічні науки та технології,* 3(33), 130-140. (Фахове видання, «Б»). ISSN 2411-5363. https://doi.org/10.25140/2411-5363-2023-3(33)_-130-14

5. Стеценко І.В., Дифучина О.Ю. (2017) Моделювання паралельних обчислень стохастичними мережами Петрі. *Вісник Національного технічного університету України «Київський політехнічний інститут». Інформатика, управління та обчислювальна техніка*, 66, 27-31. (Фахове видання)

Публікації у матеріалах міжнародних наукових конференцій:

6. Дифучина О.Ю. (2023). Метод оптимізації параметрів паралельних обчислень на основі Петрі-об’єктного моделювання. МОДС 2023: тези доповідей Вісімнадцятої міжнародної конференції (13 – 15 листопада 2023 р., м. Чернігів). М-во освіти і науки України; Нац. Акад. наук України; Академія технологічних

наук України; Інженерна академія України та ін. Чернігів: НУ «Чернігівська політехніка», 2023. С.25-28. http://ir.stu.cn.ua/123456789/29144

7. Дифучина О.Ю. (2018). Тестування паралельних програм на моделях. Математичне та імітаційне моделювання систем. МОДС 2018: тези доповідей Тринадцятої міжнародної науково-практичної конференції (м. Київ – с. Жукін, 25 червня – 29 червня 2018 р.). М-во осв. і наук. України, Нац. Акад. наук України, Академія технологічних наук України, Інженерна академія України та ін. Чернігів: ЧНТУ, 2018. С.231-234. https://stu.cn.ua/wp-content/uploads/2021/04/mods18-p.pdf

8. Стеценко І.В., Дифучина О.Ю. (2018). Програмне забезпечення моделювання дискретно-подійних систем. Тези доповідей п’ятої міжнародної науково-практичної конференції «Управління розвитком технологій». К.: КНУБА, 2018. С.97-98. https://www.knuba.edu.ua/wp-content/uploads/2022/10/%D0%A2%D0%B5%D0%B7%D0%B8-2018.pdf

Публікації у матеріалах всеукраїнських наукових конференцій:

9. Дифучина О.Ю., Стеценко І.В. (2019). Критерій ефективності використання паралельних обчислень в інформаційній технології. Матеріали ІІІ всеукраїнської науково-практичної конференції молодих вчених та студентів «Інформаційні системи та технології управління» (ІСТУ-2019). Київ.: НТУУ «КПІ ім. Ігоря Сікорського», 20-22 листопада 2019 р. – С.6-8.

ABSTRACT

Dyfuchyna O. Optimization method of parallel computing parameters based on Petri-object modeling.

Thesis for the degree of Doctor of Philosophy in specialty 126 – Information systems and technologies in the field of knowledge 12 - Information technologies. - National Technical University of Ukraine "Ihor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2024.

Ph.D. thesis is devoted to the development of methods and tools for researching the influence of parallel computing parameters on the speed of computing. Modern information technologies require fast operation of algorithms, which can be achieved through the use of parallel computing. However, depending on the parameters that determine the characteristics of the subtasks and the mechanisms of their interaction, the use of parallel computing can lead to both speeding up and slowing down the computing. In practice, the problem of setting the parameters of parallel computing is solved by repeatedly testing the program on different computer platforms with different sets of parameters to guarantee the correctness and efficiency of the calculations. Due to the large number of options for capturing the computing resource and, accordingly, options for executing instructions by individual parallel computing processes, the result of running the same program under the same conditions may differ significantly. Therefore, testing parallel algorithms in real conditions is resource-consuming process.

Mathematical methods for evaluating the efficiency of parallel computing are able to indicate the existence of a limit on the maximum achievable speed up under ideal conditions of free access to the computing resource and the absence of synchronization of calculations (that is, in fact, the absence of interaction between parts of the program).

Existing program design tools, such as UML, allow (quite generalized) to graphically represent the interaction of individual parts of the program, but do not provide any numerical analysis of calculations. Simulating tools such as high-level Petri nets are recommended by the international standard ISO/IEC 15909-1:2019 as a

technique for specifying parallel and distributed systems. To date, there is experience in the development of Petri net-based computing simulators, but none of them have become widely used in the development of parallel computing.

Thus, today, there is no unified method for creating a model of parallel computing and, accordingly, there is no tool other than a real program that can be used to optimize the parameters of a parallel program. The lack of simulation tools hinders the development of highly efficient parallel computing. In view of this, the creation of methods and tools aimed at improving the process of setting up multi-threaded programs and increasing the efficiency of using parallel computing in information technology is an urgent scientific task.

The purpose of scientific research is to increase the effectiveness of parallel computing usage in information technologies by designing it based on models which can be used for performance time estimation of parallel algorithm and parallel computing parameters optimization.

The first section substantiates the need for careful design of parallel computing to achieve their efficiency and provides an overview for methods and tools for parallel computing design and their efficiency analysis. It was found that the mathematical methods of analyzing the efficiency of parallel computing are able to estimate the maximum achievable acceleration under fairly ideal conditions of free access to the computing resource and the absence of synchronization of computing. In contrast, simulation tools are able to reproduce parallel processes in sufficient detail and, unlike experimentation with a real parallel computing program, do not require significant costs for conducting experimental research. Among the parallel computing simulators, there are many that use the Petri net formalism as a tool to describe the computing process. However, as a rule, only the simplest elements of a parallel program are simulated: division into subtasks, waiting for the completion of subtask execution, blocking. More complex tools for interaction between threads, such as wait/notify, are covered only in separate publications. The models do not take into account the limitation of the resource used by the parallel program, which, of course, negatively

affects their accuracy. Also considered are tools for testing parallel programs, which are designed to detect interlocking and/or memory consistency errors in already developed programs. Such tools are aimed, first of all, at the analysis of the correctness of the computing, but not at the analysis of the efficiency of parallel studies. As a result, a conclusion was made regarding the need to develop methods and tools for evaluating the effectiveness of parallel computing based on models that take into account the mechanism of capturing the computing resource and the mechanisms of interaction between subtasks performed in parallel.

In the second section, the purpose and task of simulating parallel computing are formulated, a formal description of the model is defined, and the method of simulating parallel computing is developed based on this description. For the formal description, the formalism of the Petri-object model was chosen due to a number of advantages: the most accurate (compared to the usual Petri net) reproduction of the structure of the object-oriented program and the behavioral properties of individual objects; more accurate (compared to the usual Petri net) reproduction of the interaction of threads/subtasks due to taking into account time delays for performing computational actions and the stochasticity of computing resource capture; replication of objects with similar behavior with given parameters simplifies the creation of models with a large number of subtasks; replication of groups of connections between objects simplifies the construction of a model with a large number of connections; visualizing the behavioral properties of the model allows us to develop a program with the least number of errors. Simulating templates of multithreaded program computations are defined and developed. They reduce the number of errors in model development, simplify and speed up the model development process. A set of modeling templates contains fragments of Petri nets that reproduce routine program instructions, creating, starting and terminating a thread, stopping a thread for a given time interval, blocking thread actions, waiting by condition, computing subtasks by a thread pool. The process of developing a Petri-object model of a parallel program is described. All the proposed

and developed methods, approaches and modeling tools are combined in the parallel algorithm simulation technology. Examples of model development are provided.

In **the third section**, the task of optimizing the parameters of parallel computing is formulated, the existence of optimal values is substantiated and the ways of finding optimal values on the model are determined, the methods and tools of data collection for the model are defined, the methods and tools of developing the Petri-object model of parallel computing are described. The main stages of the method of optimizing the parameters of parallel calculations are formulated and described, in particular, the adaptive step-by-step optimization algorithm and the evolutionary algorithm, which are applied depending on the number of parameters under study. Parallel Program Simulation (PPS) software implements the following functions to support model development: Petri net development in the graphical Petri net editor; saving the Petri net in two formats - a graphical image and in the form of a method; study of time delay parameters; experimental study of the model; search for optimal values of parallel computing parameters by evolutionary method.

In **the fourth section**, the method of optimizing parameters of parallel computing, described in section 3, is tested on the examples of a parallel algorithm for simulating a discrete-event system and a thread pool. This parallel simulation algorithm is complex both in terms of the number of computational operations and in the mechanisms of interaction of subtasks that ensure coordinated actions of all parts of the model. Theoretically and experimentally, it has been proved that the parameters of the parallel algorithm have an influence on the speed of its execution. It was established that the parameter on which the performance time of the algorithm depends to the greatest extent is the complexity of one fragment of the model, which is launched for simultaneous execution. The study of the optimal value of the parameter with increasing complexity of the model was carried out. A Petri-object model of the parallel algorithm was built and the search for optimal parameters was performed on the model experimentally. The evolutionary algorithm is tested on te Petri-object model of thread pool. The optimal values found in both cases quite accurately correspond to those found

during experimentation with parallel algorithms in real conditions. The obtained results indicate the correctness of the search for optimal parameters on the model.

The results obtained in the research contain **scientific novelty**. **For the first time**, a parallel computing simulation technology based on the Petri-object approach has been developed, which provides an opportunity to reduce resource costs in the development of parallel algorithms. Unlike existing ones, it allows to reproduce in detail the structure of a parallel program and the mechanisms of interaction of simultaneously executing parts of the program, taking into account time delays on the execution of computing actions and the stochasticity of capturing the computing resource. Developed technology simplifies the process of building a model due to the replication of program fragments with similar functionality.

The models of the basic mechanisms of synchronization of parallel computing have been **improved** by increasing the accuracy of reproduction, which ensures high accuracy of simulation results.

For the first time, typical fragments of Petri nets implementing the mechanisms of Java multithreading technology have been developed, the use of which accelerates the development of a parallel algorithm model by reducing the number of errors and reducing the total number of elements required for model development.

For the first time, a technology for optimizing the parameters of parallel computing based on an experimental study of the Petri-object model of computing is proposed. It ensures the efficient use of computing resources and, unlike existing approaches, makes it possible to conduct an experimental study of the efficiency of parallel computing on a model instead of experimenting on a real program.

The practical significance of the results of the dissertation research lies in the developed software for modeling and optimizing the parameters of parallel computing based on Petri-object simulation, which is one of the results of the research work "Methods of visual programming of Petri-object models" (state registration number 0117U000918). Technology of parallel computing simulation using Petri net was introduced as part of the distance learning course "Parallel Computing Technologies"

(certificate ДК No. 0098, approved by Protocol No. 8 dated June 2, 2023 of the Igor Sikorsky KPI Methodological Council).

The results of the dissertation were published in 9 scientific publications, including 3 papers in a periodical scientific publication indexed in the Web of Science Core Collection and Scopus databases, 2 papers in a professional scientific journal, 3 publications in the materials of international scientific conferences indexed in Scopus, 1 publication in the materials of the All-Ukrainian scientific conference.

Keywords: multithreaded programming, parallel computing, optimization, simulation, stochastic Petri net, Petri-object simulation.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасні інформаційні технології потребують швидкої роботи алгоритмів, яку можна досягти за допомогою паралельних обчислень. За влучним виразом А. С. Мена, “Паралелізм – хліб і масло сучасної архітектури програмного забезпечення” [1]. Без використання паралельних обчислень важко на сьогодні уявити розробку навіть простих інформаційних систем. Водночас, виникає все більше викликів у зв’язку з коректним та надійним використанням інструментів паралельних обчислень, зростанням складності розробки та тестування програмного забезпеченням, вибором варіантів використання обчислювальних ресурсів.

Багатопотокове програмування є популярною технікою сучасної розробки програмного забезпечення, яка широко використовується у веб-застосунках, графіці, анімації та обчисленнях великих даних, оскільки значно прискорює швидкодію програм. Однак розробка багатопотокової програми часто вимагає значних зусиль для досягнення ефективної реалізації алгоритму, оскільки вона ускладнюється недетермінованим порядком інструкцій, що виконуються потоками.

Функціонування паралельних програм сильно залежить від обчислювальних ресурсів, які використовуються для виконання програми. Це означає, що тестування паралельної програми повинно проводитися на різних комп’ютерних платформах, щоб гарантувати коректність та ефективність обчислень. На швидкодію алгоритму також сильно впливають такі параметри, як кількість потоків та розмір підзадач, що виконуються паралельно. Очевидно, що налаштування всіх параметрів паралельного алгоритму є ресурсовитратною задачею. На жаль, існуючі засоби проєктування та розробки програм ефективні для налагодження та оптимізації продуктивності лише однопотокових програм. З огляду на це, створення методу, спрямованого на вдосконалення процесу налагодження багатопотокових програм та підвищення ефективності

використання паралельних обчислень в інформаційній технології, є актуальним науковим завданням.

У даній роботі для досягнення найбільш точного відтворення складної взаємодії між потоками в одній програмі пропонується використовувати імітаційне моделювання. А саме, моделювання засобами стохастичних мереж Петрі, які дозволяють досліджувати виконання програм з урахуванням часу виконання дій потоків і підвищувати точність побудованої моделі. Формалізм та програмне забезпечення Петрі-об’єктного моделювання сприяють спрощенню побудови та дослідження моделей.

Зв’язок роботи з науковими програмами, планами, темами. Наукове дослідження проводилось у Національному технічному університеті України «Київський політехнічний інститут імені Ігоря Сікорського» у відповідності до напряму “Технологічні засоби та сервіси програмного інжинірингу” переліку пріоритетних тематичних напрямів наукових досліджень і науково-технічних розробок на період до 2023 року, затвердженого постановою Кабінету Міністрів України №942 від 7.09.2011 (в редакції постанови №463 від 09.05.2023), та у відповідності до тематики наукових розробок кафедри. Результати наукового дослідження є результатом участі здобувача у виконанні науково-дослідної роботи «Методи візуального програмування Петрі-об'єктних моделей» (державний реєстраційний номер 0117U000918).

Мета і завдання дослідження. Метою наукового дослідження є підвищення ефективності використання паралельних обчислень в інформаційних технологіях за рахунок моделювання паралельних обчислень та оптимізації їх параметрів. Для досягнення мети поставлені та вирішені такі завдання:

- аналіз існуючих засобів і методів тестування/моделювання багатопотокових програм;
- визначення правил співставлення фрагмента програмного коду частині мережі Петрі;

– розробка технології моделювання паралельних обчислень Петрі-об’єктною технологією;
– оцінювання точності моделей;
– розробка методу збору даних для моделі;
– розробка методу оптимізації параметрів;
– побудова та дослідження моделі паралельного алгоритму імітації дискретно-подійної системи.

Об’єкт дослідження – процес розробки паралельних обчислень в інформаційних технологіях.

Предмет дослідження – методи та засоби моделювання та оптимізації паралельних обчислень в інформаційних технологіях.

Методи дослідження – методи імітаційного моделювання і математичної статистики, теоретичні методи оцінювання складності алгоритмів, експериментальні методи, мережі Петрі.

Наукова новизна отриманих результатів:

- **вперше** розроблено технологію моделювання паралельних обчислень на основі Петрі-об’єктного підходу, що надає можливість скоротити ресурсні витрати при розробці паралельних алгоритмів, і, на відміну від існуючих, дає змогу відтворити деталізовано структуру паралельної програми та механізми взаємодії одночасно виконуваних частин програми з урахуванням часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу і спрощує процес побудови моделі за рахунок тиражування фрагментів програми зі схожою функціональністю;
- **удосконалено** моделі базових механізмів синхронізації паралельних обчислень за рахунок підвищення точності відтворення, що забезпечує високу точність результатів моделювання;
- **вперше** розроблено типові фрагменти мереж Петрі, що реалізують механізми багатопотокової технології Java, використання яких прискорює

розробку моделі паралельного алгоритму за рахунок зменшення кількості помилок та зменшення загальної кількості елементів, необхідних для розробки моделі;

- **вперше** запропоновано метод оптимізації параметрів паралельних обчислень на основі експериментального дослідження Петрі-об’єктної моделі обчислень, що забезпечує ефективне використання обчислювальних ресурсів і, на відміну від існуючих підходів, дає змогу проводити експериментальне дослідження ефективності паралельних обчислень на моделі замість експериментування на реальній програмі.

Практичне значення отриманих результатів. Розроблено програмне забезпечення для моделювання та оптимізації параметрів паралельних обчислень на основі Петрі-об’єктного моделювання, що є одним з результатів виконання науково-дослідної роботи «Методи візуального програмування Петрі-об'єктних моделей» (державний реєстраційний номер 0117U000918). Запропонована технологія моделювання та оптимізації паралельних обчислень використовувалась при розробці моделей пулу потоків та паралельного алгоритму імітації дискретно-подійної системи, які були представлені на міжнародних конференціях та отримали позитивні відгуки. Технологія моделювання паралельних обчислень мережею Петрі впроваджена у рамках навчального дистанційного курсу «Технології паралельних обчислень» (сертифікат ДК № 0098, затверджений протоколом №8 від 02.06.2023 Методичної ради КПІ ім. Ігоря Сікорського).

Особистий внесок здобувача. Усі результати наукового дослідження, представлені до захисту, отримані авторкою особисто. У написаних у співавторстві публікаціях авторці належать такі результати: [3] – розробка та дослідження Петрі-об’єктної моделі проблеми дедлоку (deadlock) та її вирішення, а також моделі проблеми доступу до спільних даних та її вирішення; [2] – розробка моделі та дослідження ефективності пулу потоків; [1] – розробка Петрі-об’єктної моделі та дослідження ефективності паралельного алгоритму

імітації дискретно-подійної системи; [5] – розробка бібліотеки шаблонів базових фрагментів багатопотокової програми, яка призначена для моделювання паралельних обчислень стохастичною мережею Петрі; [8] – розробка модуля анімації імітації Петрі-об'єктної моделі; [9] – створення критерію для оцінювання ефективності використання паралельних обчислень в інформаційній технології та спосіб його визначення на основі Петрі-об'єктних моделей.

Апробація результатів дисертації. Основні результати наукового дослідження доповідались та отримали позитивні відгуки на таких наукових конференціях: Mathematical Modeling and Simulation of Systems (MODS 2023, 2018, 2017), International Conference on Computer Science, Engineering and Education Applications (ICCSEEA 2018, 2019).

Публікації. Результати наукового дослідження опубліковано у 8 наукових публікаціях, серед яких 3 статті у періодичному науковому виданні, проіндексованому у Scopus базі даних, 1 стаття у фаховому науковому журналі категорії "В", 3 публікації у матеріалах міжнародних науково-практичних конференцій, 1 публікація у матеріалах всеукраїнської науково-практичної конференції.

Структура та обсяг роботи. Дисертація складається з вступу, 4 розділів, висновків, списку використаних джерел з 68 найменувань та 5 додатків. Загальний обсяг роботи складає 172 сторінки, з них 118 сторінок основного тексту, 35 рисунків, 9 таблиць.

1 ОГЛЯД ІСНУЮЧИХ ЗАСОБІВ ДЛЯ ПРОЄКТУВАННЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА АНАЛІЗУ ЇХ ЕФЕКТИВНОСТІ

1.1 Математичні методи аналізу ефективності паралельних обчислень

Загальноприйнятими у практиці паралельних обчислень є такі показники ефективності [2]: прискорення (speedup), ефективність (effeciency) та вартість обчислень (cost). *Прискорення* визначає, у скільки разів швидше виконується паралельний алгоритм у порівнянні з послідовним:

$$S = \frac{T_s}{T_p}, \tag{1.1}$$

де S – прискорення,

T_s – час виконання послідовного алгоритму,

T_p – час виконання паралельного алгоритму.

Прискорення, розраховане на одиницю використовуваного обчислювального ресурсу, визначає *ефективність* паралельних обчислень:

$$E = \frac{\mathrm{S}}{p} = \frac{T_s}{pT_p}, \tag{1.2}$$

де p – кількість обчислювального ресурсу, що забезпечує паралельність виконання.

Зауважимо, що Амдал формулював свій закон для процесорів. Проте сучасні процесори є багатоядерними, а технологія багатопотокових обчислень надає можливість використовувати багатоядерний процесор для паралельних обчислень. Тому в сучасній інтерпретації закону замість кількості процесорів використовують поняття обчислювального ресурсу, що спроможний виконувати частину обчислень незалежно від інших.

Використовують також показник *вартості* обчислень, який розраховують як загальний час, витрачений на усіх задіяних в обчисленнях процесорах:

$$C = pT_p. \tag{1.3}$$

Найбільш точно показники ефективності визначають за результатами екпериментального дослідження. Для такого дослідження алгоритм спочатку має бути розроблений, а це означає, що для отримання ефективного алгоритму може бути проведено кілька ітерацій розробки та дослідження алгоритму. Щоб отримати уявлення про вплив обчислювального ресурсу необхідно проводити експериментальне дослідження на різних обчислювальних ресурсах. Отже, процес розробки ефективного паралельного алгоритму у такий спосіб є ресурсовитратними. Тому важливо ще на етапі проєктування мати можливість оцінки отримуваного прискорення та ефективності для розроблюваного алгоритму.

Теоретично доведене обмеження згори для отримуваного за рахунок паралельних обчислень прискорення встановлює закон Амдала [3]. Амдал припустив, що для паралельного алгоритму завжди може бути визначена частка обчислень $f \in [0; 1]$, що не можуть бути розпаралелені і виконуються послідовно, та частка обчислень $1 - f$, що виконуються паралельно на p незалежних обчислювальних ресурсах. Тоді час виконання послідовного алгоритму може бути представлений як $T_s = fT_s + (1 - f)T_s$, час виконання паралельного алгоритму – як $T_p = fT_s + (1 - f)\frac{T_s}{p}$, а прискорення визначається співвідношенням:

$$S = \frac{fT_s + (1-f)T_s}{fT_s + (1-f)\frac{T_s}{p}} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}, \tag{1.4}$$

оскільки $\frac{1-f}{p} > 0$.

Звідси, прискорення завжди має обмеження згори $\frac{1}{f}$, якою б великою не була кількість обчислювального ресурсу.

Проте визначення частини послідовних обчислень у практичних умовах є проблематичним і не існує чіткої методики визначення такої частки.

Окрім того, дана формула враховує не всі витрати. Так, не задіяними у даній формулі залишаються витрати на виділення ресурсів та на розбиття задачі

на незалежні підзадачі. Досить часто саме ці витрати виявляються більшими, ніж виграш у прискоренні, від паралелізму.

Іншим законом, що оцінює максимально досяжне прискорення паралельних обчислень, є закон Густавсона-Барсіса [4]:

$$S = f + p(1 - f) = p - f(p - 1), \tag{1.1}$$

де τ – час послідовної частини виконуваних обчислень,

π – час паралельної частини виконуваних обчислень,

p – кількість процесорів,

$f = \frac{\tau}{\tau + \frac{\pi}{p}}$ – частка послідовних обчислень,

$S = \frac{\tau + \pi}{\tau + \frac{\pi}{p}}$ – прискорення.

Згадані закони надають оцінку лише верхньої межі прискорення, яка можлива лише за ідеальних умов використання паралельних обчислень. За реальних обставин, ефективність паралельного алгоритму залежить від належного налаштування його параметрів та від обчислювальних ресурсів, на яких він виконується.

Застосування закону Амдала для теоретичного оцінювання часу багатопотокової програми запропоновано у статті [5]:

$$t = \sum_{i=1}^{n} \max_j t_{ij} + t_s,$$

де t_{ij} – час виконання *j*-ої підзадачі *i*-ої ділянки програми, $j = \overline{1, p}$,

$\max_j t_{ij}$ – час виконання *i*-ої ділянки програми,

t_s – час виконання послідовних ділянок програми.

Проте виведення авторами формули виконано у значних припущеннях: 1) кількість обчислювального ресурсу необмежена, тобто є обчислювальний ресурс для кожної підзадачі, 2) задачі не блокують одна одну, тобто відсутнє очікування задач одна одної.

Методи аналітичної оцінки прискорення мають обмежену точність, оскільки вони не беруть до уваги деталі паралельного алгоритму. Урахування синхронних обчислень у паралельних алгоритмах ускладнює оцінку фактичної частки обчислень, які виконуються паралельно. Тому, якщо в обчисленнях використовуються n процесорів, це не означає, що обчислення будуть прискорені в n разів. З цього приводу виникає потреба у технології моделювання паралельних обчислень, яка враховує параметри паралельного алгоритму та обсяг обчислювальних ресурсів.

По сьогоднішній день найпопулярнішим засобом проєктування програмного забезпечення залишається стандарт Unified Modeling Language (UML). Це уніфікована мова моделювання загального призначення в галузі програмної інженерії, призначена для надання стандартного способу візуалізації дизайну системи. Дехто вважає UML невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. Для проєктування логіки програми та візуалізації складових системи та їх взаємозв'язків цей інструмент є дійсно достатнім. Однак для проєктування багатопотокових програм, специфіка яких полягає у стохастичності їх поведінки, даного інструменту буде недостатньо.

1.2 Засоби моделювання паралельних обчислень

Розробка паралельних алгоритмів є важким завданням, що часто вимагає фундаментальної реконструкції послідовного алгоритму. Програміст намагається максимально ефективно використовувати обчислювальні ресурси для досягнення найвищої швидкості. Однак використання синхронізованих процесів і вартість підтримки багатопотоковості можуть звести нанівець його зусилля. Це може відбутися з тієї причини, що синхронізація завжди сповільнює роботу алгоритму, а потоки завжди потребують додаткових ресурсовитратних операцій, щоб контролювати їх роботу. Таким чином, програміст повинен знайти баланс між складністю розробки, споживанням ресурсів потоками і накладними

витратами на взаємодію між потоками та бути впевненим, що витрати на багатопотоковість менші, ніж переваги від неї [6].

Перед розробкою алгоритму необхідно прийняти рішення щодо ефективності паралельної реалізації. Також корисно визначити рівень складності завдання, за якого паралельна реалізація завдання буде ефективною. На жаль, на даний момент не існує інструментів чи методів для прогнозування прискорення, що забезпечується паралелізмом. Як зазначалось вище, закон Амдала визначає лише верхню межу прискорення. Водночас, залишається проблемою оцінка прискорення, яке спостерігатиметься в реальних умовах

Ще однією проблемою, що виникає при розробці паралельного алгоритму є відсутність вказівок щодо налаштування параметрів інструментів багатопотоковості, які б забезпечили ефективну роботу відповідного інструменту. Наприклад, для таких інструментів як Fork Join Pool або Thread Pool в документації Java не надається інформація про те, який розмір підзадач має бути. У статті [7] було виявлено значний вплив розміру підзадач, що виконуються незалежно у потоках, на ефективність роботи паралельного алгоритму. У статті [8] досліджено та виявлено мінімальний розмір підзадач у пулі потоків, при якому спостерігається прискорення паралельного алгоритму. З цих досліджень випливає, що питання визначення розміру підзадач для потоків є ключовим при розпаралелюванні алгоритму.

Тож, при розробці завжди постає велике питання: які параметри паралельного алгоритму забезпечать найбільше прискорення для конкретного завдання і конкретних ресурсів комп'ютера? Експериментальне дослідження параметрів є ресурсовитратним процесом. Тому краще використовувати модель, яка відтворює паралельні обчислення.

Розглянемо існуючі засоби моделювання паралельних обчислень. На сьогоднішній день таких засобів відомо небагато, проте рух досліджень у цьому напрямку є. Так, у роботі [9] для моделювання пулу потоків використовуються

синхронізовані автомати. Модель показує значне скорочення тривалості виконання завдань у випадку паралельних потоків.

Іншим засобом моделювання є Renew [10]. Це симулятор мереж Петрі, розроблений мовою Java, який забезпечує гнучкий підхід до моделювання саме програм на основі reference мереж. Він реалізує концепцію nets-within-nets (мережа в мережі), де маркери позицій можуть бути посиланнями на фрагмент мережі Петрі (звідси і назва, «reference» – посилання). Не дивлячись на те, що в Renew присутні часові затримки, все ж таки розробники відійшли від класичних мереж Петрі і створили надбудову над ними, ускладнивши процес моделювання ієрархічністю та великою кількістю повторюваних елементів.

CPN Tools [11] - ще один програмний засіб для моделювання складних систем, включаючи розподілені. Моделювання відбувається з використанням розфарбованих мереж Петрі. Інструмент реалізує ієрархічну структуру моделей за принципом «мережа в мережі», з'єднання мереж відбуваються тільки через перехід. У CPN Tools присутня велика кількість додаткових параметрів, які призводять до перевантаження моделі та роблять її не читабельною. Не дивлячись на те, що у даному програмному засобі є приклади моделювання розподілених систем, моделювання паралельних програм, які вирізняються великою кількістю повторюваних фрагментів, за допомогою кольорових мереж Петрі та інструменту CPN Tools є недоречним. Наразі, є оновлена версія даного засобу – CPN IDE [12], яка повністю замінила CPN Tools. Це розширений інструмент, який містить додаткові функції, наприклад створення журналу подій. Однак основні принципи роботи залишились такими ж, як і в CPN Tools.

Як бачимо, існуючі засоби моделювання багатопотокових програм задіюють мережі Петрі. Річ у тім, що мережі Петрі були спеціально створені для відтворення саме паралельних процесів. Однак частіше за все їх використовують для моделювання паралельних процесів у сфері бізнесу та виробництва, а не в програмуванні.

Першим в історії, хто застосував мережі Петрі для моделювання паралельних обчислень, був Джеймс Пітерсон. У своїй книзі «Petri Net Theory and the Modeling of Systems» [13] у 1981 році він зобразив моделі таких відомих механізмів та проблем багатопотокових програм, як операції fork та join, проблема доступу до спільних даних, синхронізація процесів, проблема producer-consumer, проблема взаємоблокувань. Дотепер найвідомішою моделлю з цієї книги залишається модель задачі про 5 філософів, що обідають, запропоновану ще Дейкстра. Задача демонструє проблему взаємоблокувань та її рішення. Однак дана модель, як і всі інші наведені у книзі моделі паралельних обчислень, дуже поверхово описує лише логіку роботи фрагменту алгоритму, не деталізуючи опис програми.

Серед досліджень паралельних обчислень є багато інших випадків моделювання їх засобами мереж Петрі. Так, наприклад, використання класичних мереж Петрі для моделювання багатопотокових застосунків викладено в [14]. Розробка інструменту візуалізації багатопотокової java-програми з використанням класичних мереж Петрі представлена в [15]. У роботі [16] описано можливі сценарії виникнення зависання багатопоточної програми та розглянуто моделювання цих сценаріїв класичною мережею Петрі та операційним графом, який запропонований авторами публікації, на прикладі простого застосунку.

Алоїз Ферша розглядав використання мережі Петрі для прогнозування часу виконання паралельної програми на етапі проєктування [17]. Запропоновані моделі та технології були втілено у спеціалізоване програмне забезпечення Computer Aided Parallel Software Engineering (CAPSE) [18]. Припускається, що паралельні обчислення реалізуються технологією обміну повідомленнями. У роботі для опису паралельних процесів обчислень використовувалися фрагменти з мережами Петрі, що моделювали send та receive повідомлення процесом, fork або join завдання. Для враховування впливу обчислювального ресурсу на виконання паралельного алгоритму до звичайної мережі Петрі додані Resource-

net та Mapping-net, що відтворюють використання та розподіл ресурсів між процесами. З використанням запропонованої технології моделювання розроблено та досліджено паралельний алгоритм множення матриць з метою визначення найкращого розбиття матриць [19]. На жаль, результати, отримані на моделі, не порівнюються з такими, що отримані в реальній обчислювальній системі. Алгоритм множення матриць не містить ніякої складної взаємодії між процесами, тому такий приклад не дає можливості усвідомити переваги технології. Наведені в роботі експерименти розглядають розбиття матриць дуже невеликого розміру (6×8 та 8×12).

У статті [20] автори запропонували використання кольорових мереж Петрі при розробці паралельних програм, які виконуються на суперкомп'ютерних системах різної архітектури. Створена ієрархічна мережа є узагальненою, адже вона не враховує особливості архітектури системи, на якій виконується паралельний алгоритм.

У роботі [21], де запропоновано фреймворк PN-PEM, теж використовуються кольорові мережі Петрі. Даний фреймфорк є моделлю для паралельного виконання програм, здебільшого орієнтований на симетричні мультипроцесорні системи. Представлення паралельних алгоритмів кольоровими мережами Петрі в даній роботі не є досить детальним.

Автори роботи [22] розглянули розробку програмного забезпечення для моделювання інформаційних систем. Мова моделювання інформаційних систем була запропонована як розширення PNML (Petri Net Markup Language). Опис моделі поділяється на інформаційну модель і модель процесу, яка маніпулює інформаційною моделлю.

У статті [23] розглянута концепція розробки коду програмного забезпечення за його моделлю, описаною мережею Петрі. За цією концепцією модель будується за вимогами, які висуваються до програмного забезпечення. На наступному етапі програмне забезпечення розробляється у відповідності до побудованої моделі та використовує модель для верифікації. Автоматизована

генерація коду за моделлю програми у такій концепції гарантує її відповідність вимогам, а мережі Петрі мають бути складовою частиною розробки програмного забезпечення. Проте втілення цієї концепції у подальших роботах не знайдено.

Варто зазначити, що використання високорівневої мережі Петрі для проєктування програмного забезпечення зазначено в міжнародному стандарті ISO/IEC 15909-1:2019 [24] як техніка для специфікації паралельних і розподілених систем. Незважаючи на це, на даний момент відомо не так багато практичного досвіду з розробки паралельних алгоритмів з використанням мереж Петрі.

Таким чином, на сьогодні не існує уніфікованого методу створення моделі паралельного алгоритму. Відсутність засобів моделювання стримує розробку високоефективних паралельних програм.

Важливим при моделюванні паралельних обчислень є деталізація опису програми. Інструмент, що моделює програму, має утримувати баланс між точністю моделі та її простотою. Тобто, процес побудови моделі та її обчислення має бути легким, але при цьому програма має відтворюватися достатньо точно. Стохастичні мережі Петрі та Петрі-об’єктне моделювання дозволяють фокусуватися на потрібних фрагментах та відтворювати їх детально, при цьому інші фрагменти, деталізація яких не важлива, можуть бути імітовані стисло за допомогою часових затримок. Це дозволяє дослідити параметри та спрощує цей процес у порівнянні з експериментальним шляхом.

1.3 Засоби тестування паралельних програм

Тестування паралельної програми виконується після її розробки з метою виявлення помилок. Розробки засобів автоматизованого тестування спрямовані, в основному, на виявлення дедлоків як найбільш проблемної ситуації в паралельній програмі. Метод статичного виявлення взаємоблокувань запропоновано в [25]. Крім того, були розроблені спеціальні інструменти для

виявлення взаємоблокувань [26] або помилок узгодженості пам'яті [27] у паралельній програмі.

Тестування паралельної програми з використанням техніки аналізу потоків даних розглядається в роботі [28]. Алгоритм відстеження виконання програми допомагає досліджувати міжпроцесний зв’язок. Запропонована у статті методика може бути застосована лише для програми MPI.

Платформа для тестування паралельного алгоритму Java з використанням апаратного забезпечення з 60 ядрами запропонована в роботі [29] та наведено ряд експериментальних результатів для різних широко відомих алгоритмів. Час виконання та прискорення для всіх алгоритмів залежать від кількості потоків, які використовуються для реалізації алгоритму.

Популярності набувають також засоби самоналаштування паралельних програм – автотюнери. Більше того, у статі [30] та дисертації [31] запропоновані методи автоматизації генерації таких автотюнерів. Суть автотюнінга полягає в тому, що оптимізація виконується автоматизовано шляхом емпіричних випробувань, які здійснюються програмою-автотюнером на тому ж обладнанні, на якому в подальшому буде виконуватися оптимізована застосункова програма. Тобто автотюнери виключають можливість врахування та варіації обсягу задіяного обчислювального ресурсу при виконання паралельної програми.

1.4 Висновки до розділу 1

Виконано огляд існуючих показників ефективності паралельних обчислень та методів їх оцінювання.

Математичні методи аналізу ефективності паралельних обчислень здатні оцінити максимально досяжне прискорення за досить ідеальних умов вільного доступу до обчислювального ресурсу та відсутності синхронізації обчислень. Спостережуване прискорення реального алгоритму буде значно меншим. Для більш точної оцінки необхідно використовувати більш точні моделі паралельних обчислень.

У розділі розглянуті засоби моделювання паралельних обчислень та їх здатність оцінювати ефективність. Достатньо багато з них, використовують мережі Петрі. Проте звичайні мережі Петрі при детальному описі програми призводять до надто складних моделей. Тому моделюванню підлягають або окремі елементи програми, або достатньо спрощені для їх реалізації мережею Петрі. Моделюванню, як правило, підлягають найпростіші елементи паралельної програми: розділення на підзадачі, очікування завершення виконання підзадачі, блокування. Більш складні інструменти взаємодії між потоками, як wait/notify, розглядаються лише в окремих публікаціях. Майже усі засоби моделювання не враховують обмеженість використовуваного паралельною програмою ресурсу, що, звісно, зменшує точність таких моделей.

Розглянуті також засоби тестування паралельних програм, які здатні виявляти у вже розроблених програмах взаємоблокування та/або помилки узгодженості пам’яті. Це помилки розробки паралельної програми, які найважче виявляються і відлагоджуються. Засоби тестування програм спрямовані, насамперед, на аналіз коректності виконання обчислень, але не на аналіз ефективності паралельних досліджень.

У підсумку, моделі паралельних обчислень потенційно спроможні використовуватись для оцінювання ефективності паралельних обчислень. Мережі Петрі мають низку переваг для представлення паралельної програми: деталізація опису паралельних обчислень, візуалізація представлення станів програми.

Існуючі засоби моделювання недостатньо точно відтворюють механізми захоплення обчислювального ресурсу та механізми взаємодії підзадач паралельної програми, тому існує необхідність розвитку методів та засобів для оцінювання ефективності паралельних обчислень на основі моделей.

2 ПЕТРІ-ОБ'ЄКТНЕ МОДЕЛЮВАННЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

2.1 Змістовна постановка задачі моделювання

Найбільш важливим показником ефективності паралельних обчислень, як було вказано у підрозділі 1.1, є прискорення, що розраховується як час виконання послідовного алгоритму поділений на час виконання паралельного алгоритму. Отже, для розрахунку прискорення необхідною є побудова двох моделей обчислень – послідовного та паралельного алгоритму, що збільшує вдвічі зусилля з побудови моделей та їх експериментального дослідження. У цьому розділі розглядається побудова моделі паралельних обчислень. Модель послідовних обчислень у багатьох випадках можна отримати з моделі паралельних обчислень переходом від багатьох задач, що запускаються на обчислення, до однієї та видаленням механізмів управління, що потребують наявності в програмі не менше двох потоків.

Метою моделювання паралельних обчислень визначимо дослідження впливу їх параметрів на час виконання паралельних обчислень. Отже, вихідною величиною моделі паралельних обчислень визначимо час виконання паралельних обчислень.

Паралельні обчислення, що реалізуються сучасними програмними засобами, мають у своїй основі схожі механізми розподілу на підзадачі та взаємодії підзадач. Управління підзадачами, що виконуються на різних обчислювальних ресурсах, як правило, передбачає такі основні дії, як запуск підзадачі на виконання, призупинка до виконання певних умов та завершення виконання підзадачі. Призупинка виконання реалізує, по суті, синхронізацію виконання однієї задачі з іншими підзадачами.

Багатопотокова технологія паралельних обчислень ґрунтується на поділі обчислень, які виконує алгоритм, на підзадачі, що можуть виконуватись одночасно і, бажано, незалежно одна від одної. Якщо незалежність виконання

підзадач неможливо організувати, то виникає необхідність використовувати механізми управління потоками такі, як:

- призупинка на заданий інтервал часу,
- призупинка до завершення обчислень іншим потоком,
- очікування звільнення заблокованого об’єкта, який контролює виконання дій зі спільними об’єктами,
- очікування на виконання умови, яка є необхідною для продовження обчислювальних дій.

Будь-яка програма складається з інструкцій. Виконання інструкції призводить до перетворення стану програми і кожний новий стан є передумовою для виконання наступної інструкції. Одна інструкція програми – це одна або кілька обчислювальних дій. Дія може бути елементарною арифметичною чи логічною дією, і більш складною як, наприклад, виклик методу, що виконує певні послідовні обчислення. Обов’язковою умовою для поділу алгоритму на інструкції є умова, що в межах однієї інструкції обчислення можуть виконуватись виключно послідовно.

Сформулюємо задачу моделювання паралельних обчислень:

для заданого алгоритму паралельних (багатопотокових) обчислень, заданої кількості підзадач, що виконуються паралельно, та заданих параметрів обчислювального ресурсу (кількість ядер, тривалість обробки основних обчислювальних дій), що використовується для виконання паралельних обчислень, визначити час виконання обчислень.

Формально модель можна представити як перетворення множини вхідних змінних у вихідні змінні:

$$\{I_k, c, R \mid k = \overline{1,n}\} \longrightarrow q, \tag{2.1}$$

де $I = \{I_k | k = \overline{1,n}\} -$ множина, кожний елемент якої I_k представляє послідовність інструкцій k-ої задачі,

c – кількість ядер, задана натуральним числом $c \geq 2$,

$R = \{r_j | j = \overline{1,m}\}$ – тривалості обробки обчислювальних дій алгоритму за умови виконання на заданому обчислювальному ресурсі, які задані невід'ємними дійсними числами,

q – час виконання алгоритму.

Кожній інструкції I_{ki} може бути співставлена тривалість її виконання t_{ki} як лінійна комбінація тривалостей обчислювальних дій:

$$I_{ki} \longrightarrow t_{ki} = \sum_j d_{kij} r_j,\ r_j \in R,\ d_{kij} \in \mathrm{N} \cup \{0\},$$

де d_{kij} – кількість дій з однаковою тривалістю r_j.

Навіть у випадку незалежного виконання підзадач оцінювання часу виконання обчислень є нетривіальною задачею. Наведемо простий приклад. Нехай n підзадач з загальними тривалостями виконання $t_k = \sum_i \sum_j d_{kij} r_j, k = 1..n$ запустили на одночасне виконання. Якщо кількість ядер $c \geq n$, то час виконання програми $q = \max\limits_k t_k$. В іншому випадку час виконання визначається наборами підзадач, які виконались на кожному ядрі. Програміст не може вказати операційній системі, на якому ядрі виконувати підзадачу, він може тільки запустити на одночасне виконання множину підзадач та контролювати кількість створених потоків на виконання. Отже, набори підзадач, які потрапили на виконання в кожне ядро є випадковими. Час виконання визначається найдовшою тривалістю виконання набору підзадач, що виконується на одному ядрі. У випадку $c = 2$, можна скласти такі умови для визначення мінімально можливого часу:

$$q = \max\left\{\sum_{j \in A} t_j, \sum_{i \in B} t_i\right\},$$

$$A \cap B = \emptyset, A \neq \emptyset, B \neq \emptyset, A \cup B = \{1, 2 \ldots n\},$$

$$A, B: \left|\sum_{j \in A} t_j - \sum_{i \in B} t_i\right| \longrightarrow min,$$

де A, B – множини індексів, що вказують на підзадачі, які потрапили на виконання у перше та друге ядро.

У випадку довільного c мінімально можливий час одночасного виконання обчислень n підзадач може бути визначений з таких умов:

$$q = \max_k \sum_{j \in A_k} t_j \, , k = \overline{1,n},$$

$$\forall i,j \; A_i \cap A_j = \emptyset, A_i \neq \emptyset, \bigcup_k A_i = \{1, 2 \dots n\},$$

$$A_1, A_2 \dots A_c : \; \sum_{i,j} \left| \sum_{k \in A_i} t_k - \sum_{k \in A_j} t_k \right| \longrightarrow min,$$

де $A_1, A_2 \dots A_c$ – множини індексів, що вказують на підзадачі, які потрапили на виконання у $1, 2 \dots c$ ядро.

Складність перебору усіх можливих варіантів є неполіноміальною, тому оцінити час у такий спосіб можливо тільки при достатньо невеликій кількості підзадач. Операційна система не може гарантувати мінімально можливий час виконання підзадач, оскільки час їх виконання їй невідомий. Проте, якщо задачі відсортовані у порядку зменшення їх тривалості, то час виконання буде близьким до найменш можливого.

Враховування взаємодії підзадач означає, що виконання підзадачі може призупинятись до виконання певних умов, що залежать від перебігу обчислень в іншій підзадачі, а отже, час виконання може бути більшим за час виконання без взаємодії:

$$I_{ki} \longrightarrow \; t_{ki} + \; \delta,$$

де δ – тривалість очікування, що залежить від використовуваного механізму взаємодії та перебігу обчислень.

Якщо, наприклад виконання інструкцій I_{ki} та I_{nj} синхронізовано, то час виконання кожної з них збільшується на час виконання іншої (оскільки виконання їх перетворюється у послідовне):

$$I_{ki} \longrightarrow t_{ki} + \; t_{nj}, \; I_{nj} \longrightarrow t_{nj} + t_{ki}.$$

У випадку очікування завершення однією підзадачею іншою (інструкція join), то час завершення відповідної інструкції може збільшитись за рахунок очікування. Якщо інструкція підзадачі I_{kw} є join-інструкцією для підзадачі I_n, то

час завершення виконання інструкції I_{kw} збігається з завершенням виконання підзадачі I_n. Якщо підзадачі почали своє виконання одночасно і тривалість виконання підзадачі I_n більша за виконання усіх інструкцій підзадачі I_k, що передували join-інструкції, то час виконання інструкції I_{kw} з урахуванням очікування можемо визначити з такого рівняння:

$$\begin{cases} \sum_{i=1}^{w-1} t_{ki} + t_{kw} = t_n \\ t_n > \sum_{i=1}^{w-1} t_{ki} \end{cases}.$$

Звідси,

$$t_{kw} = t_n - \sum_{i=1}^{w-1} t_{ki} > 0.$$

Отже, оцінка часу очікування є складним завданням і піддається оцінюванню математичним виразом тільки у дуже простих випадках. Перешкодою до виведення такого математичного виразу є також наявна стохастичність у захопленні ресурсу. Тому тільки відтворення послідовності обчислювальних дій в часі з урахуванням їх взаємодії можуть допомогти вирішити задачу оцінювання часу виконання алгоритму без його реалізації. Тому методом моделювання обраний метод імітації.

2.2 Метод Петрі-об’єктного моделювання

2.2.1 Загальна концепція Петрі-об’єктного моделювання

Метод Петрі-об’єктного моделювання вперше був запропонований у роботі [32], а в роботі [33] сформульовані теоретичні положення Петрі-об’єктного моделювання. Метод ґрунтується на об’єктно-орієнтованому представленні моделі та описі динаміки функціонування елементів моделі мережею Петрі. Один раз розроблена мережа Петрі може використовуватись багатократно з заданими параметрами для конструювання багатьох елементів за рахунок використання механізму тиражування об’єктів, що використовується об’єктно-орієнтованими мовами програмування. За означенням, Петрі-об’єктом є об’єкт суперкласу, що містить опис динаміки стохастичною мережею Петрі.

Такий суперклас містить поле net та методи, які змінюють стан мережі Петрі (тобто значення поля net) при здійсненні подій. Кожна подія відтворюється в мережі Петрі виходом маркерів з переходу та входом маркерів в переходи. Відтворена в часі послідовність подій утворює імітацію процесу, що моделюється.

Для передачі інформації про зміну стану одного Петрі-об’єкта іншому необхідно вказати ототожнення потрібних позицій цих об’єктів. Таке зв’язування Петрі-об’єктів забезпечує, що утворена модель визначається мережею Петрі, яка є об’єднанням усіх мереж Петрі-об’єктів.

Конструювання моделі відбувається для заданого списку Петрі-об’єктів, з’єднаних один з одним. В роботі [34] запропоновано використовувати поняття конектора для множини ототожнень позицій між парою Петрі-об’єктів та використовувати тиражування конекторів для визначення зв’язків між одним Петрі-об’єктом та кількома іншими. Концепцію Петрі-об’єктного моделювання було розвинуто також у згаданій роботі за рахунок створення групи та колекції Петрі-об’єктів, що спрощують розробку моделі з великою кількістю однотипних елементів.

2.2.2 Стохастична мережа Петрі з багатоканальними та конфліктними переходами, з інформаційними дугами

Побудова Петрі-об’єктної моделі базується на стохастичній мережі Петрі та об'єктно-орієнтованій технології. Стохастичні мережі Петрі виникли як розширення класичних мереж Петрі, винайдених німецьким математиком та інформатиком Карлом Адамом Петрі у 1939 році та опублікованих у 1962 році в його дисертації на тему взаємодії з автоматами [35]. Він був першим вченим, який визначив паралелізм як фундаментальний аспект обчислень. Його теорія мереж Петрі не тільки була цитована сотнями тисяч наукових публікацій, але й значно просунула галузі паралельних і розподілених обчислень.

Математичний опис стохастичної мережі Петрі з багатоканальними переходами може бути представлено мультиможиною [36]:

$$PetriNet = (\mathbf{P}, \mathbf{T}, \mathbf{A}, \mathbf{W}, \mathbf{K}, \mathbf{R}), \quad (2.1)$$

де $\mathbf{P} = \{P\}$ – множина позицій,

$\mathbf{T} = \{T\}$ – множина переходів, $\mathbf{P} \cap \mathbf{T} = \emptyset$,

$\mathbf{A} \subseteq \mathbf{P} \times \mathbf{T} \cup \mathbf{T} \times \mathbf{P}$ – множина вхідних і вихідних дуг,

$\mathbf{W}: \mathbf{A} \to$ N, множина натуральних чисел, що задають кратності дуг,

$\mathbf{K} = \{(c_T, b_T) | T \in \mathbf{T}, c_T \in \mathrm{N}, b_T \in [0; 1]\}$ множина пар значень, що визначають пріоритет та ймовірність запуску переходу,

$\mathbf{R}: \mathbf{T} \to \Re_+$ – множина невід'ємних дійсних значень, що характеризують часові затримки в переходах і можуть бути або детермінованим значенням, або випадковою величиною з заданим законом розподілу,

N – множина натуральних чисел,

$\Re_+$ – множина дійсних невід'ємних чисел.

Мережа Петрі є коректно побудованою, якщо 1) є хоча б один перехід з ненульовою часовою затримкою, 2) кожний перехід мережі Петрі має хоча б одну вхідну та хоча б одну вихідну позицію (вхідні позиції, з'єднані з переходом однією інформаційною дугою, не враховуються).

Стан мережі Петрі в будь-який момент часу t повністю визначається станом її позицій та переходів:

$$\mathbf{S}(t) = \big(\mathbf{M}(t), \mathbf{E}(t)\big),$$

де $\mathbf{S}(t)$ – стан мережі Петрі,

$\mathbf{M}(t) = \{M_P(t) \mid M_P(t) \in \mathrm{N} \cup \{0\}, P \in \mathbf{P}\}$ – стан позицій,

$\mathbf{E}(t) = \{E_T(t) | T \in \mathbf{T}\}$ – стан переходів.

Петрі-об'єктна модель використовує мережу Петрі з *багатоканальними переходами*, що означає можливість багатократного входу маркерів у перехід. При кожному вході маркерів у перехід запам'ятовується нове значення виходу з переходу. Тому стан переходу $E_T(t)$ визначається множиною моментів виходу маркерів з каналів переходу:

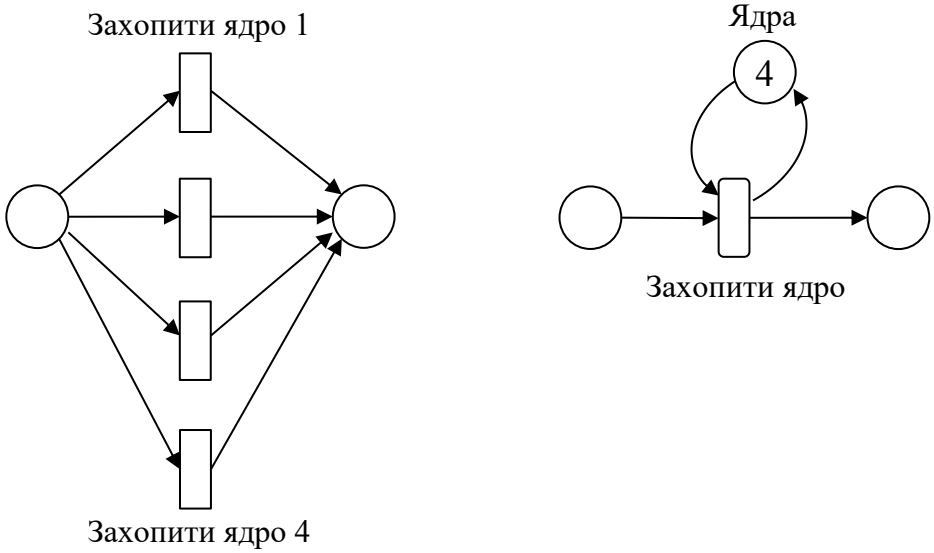
$$E_T(t) = \{ [E_T(t)]_q \},$$

де q - номер каналу переходу, $q = 1, 2 \ldots |E_T(t)|$,

$|E_T(t)|$ - кількість каналів, з яких очікується вихід маркерів.

Коли $|E_T(t)| = 0$, тобто не очікується вихід маркерів з переходу в найближчий час, то стан переходу $E_T(t) = \{\infty\}$.

Зауважимо, що використання багатоканальних переходів значно зменшує кількість елементів, необхідних для моделювання подій, що відбуваються одночасно. Наприклад, для відтворення використання одного з k ядер комп'ютера звичайною мережею Петрі потрібно n переходів, 2 позиції та $2k$ дуг, а для відтворення з багатоканальним переходом – 3 позиції та 4 дуги при будь-якому великому k. На рисунку 2.1 представлено захоплення одного з 4 ядер. Позначенням прямокутника зі заокругленими кутами підкреслюємо при зображенні стохастичної мережі Петрі, що перехід є багатоканальним (на відміну від традиційного позначення звичайним прямокутником з гострими кутами).

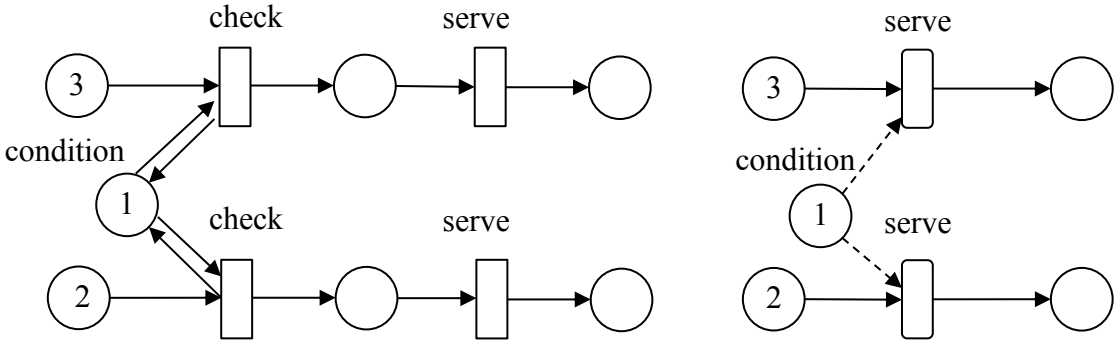


(а) (б)

Рисунок 2.1 – Еквівалентне представлення багатоканального переходу (б) одноканальними переходами (а).

Використання *інформаційної дуги* відкриває можливість використання стану позиції як додаткової умови для входу маркерів в перехід. За визначенням,

якщо для дуги вказано, що вона є інформаційною, то при перевірці умови входу маркерів у перехід інформаційна дуга враховується як звичайна, проте при вході маркерів в перехід вздовж цієї дуги маркери з позиції не віднімаються. Отже, наявність маркерів у позиції, що з'єднана з переходом, у кількості, рівній кратності дуги, є необхідною умовою входу маркерів в перехід (як і для звичайної), проте кількість маркерів в такій позиції не змінюється при вході маркерів в перехід. Це означає, що така умова є необхідною для здійснення події, проте виконання події не впливає на зміну умови (вона не зникає). Наприклад, на рисунку 2.2, представлений фрагмент мережі Петрі, в якому одна й та сама умова condition використовується для дозволу на виконання подій serve, які виконують два процеси одночасно, двома способами – звичайною мережею Петрі та мережею Петрі з інформаційними дугами. У випадку звичайної мережі Петрі (рис. 2.2, а), доводиться спочатку робити перевірку дозволу (подія check з часовою затримкою d=0) і тільки потім виконувати подію serve. Тобто дозвіл на виконання буде використовуватись тільки послідовно, оскільки маркер на час спрацьовування події check з позиції condition зникає. У випадку мережі Петрі з інформаційними дугами (рис. 2.2, б) обидва переходи serve матимуть одночасно виконану умову запуску і немає перешкоди для одночасного їх запуску. Якщо прийняти до уваги багатоканальність переходу, то різниця у застосуванні звичайної та інформаційною дугою стає ще більш очевидною: дозвіл з інформаційною дугою може безперешкодно використовуватись будь-якою кількістю каналів переходу, проте дозвіл зі звичайною дугою може бути використаний тільки послідовними запусками і фактично стає обмеженням на виконання події по одній. Отже, інформаційні дуги є важливим елементом мережі Петрі, який дає змогу відтворювати більш точно умови виконання подій і при цьому потребує менше елементів для представлення та менше кроків імітації для відтворення.



(а) (б)

Рисунок 2.2 – Порівняння представлення події, що виконується за додаткової умови, мережею Петрі (а) зі звичайними дугами та (б) з інформаційною дугою.

Момент найближчої події визначається мінімальним значенням зі всіх моментів виходу:

$$t_n = \min_T \left(\min_q [E_T(t_{n-1})]_q \right), t_n \geq t_{n-1}.$$

Виконання події в момент t_n відбувається у два кроки: 1) вихід маркерів з переходів, 2) багатократний вхід маркерів в переходи. Першому кроку відповідає перетворення стану мережі Петрі, яке позначається D^+, а другому кроку – перетворення $(D^-)^m$. Позначення + та – підказують, що при першому перетворенні відбувається додавання маркерів у вихідні позиції, а при другому – багатократне віднімання маркерів з вхідних позицій. Між першим та другим кроком є перехідний стан мережі Петрі $\mathbf{S}^+$, який є результатом виходу маркерів з переходів:

$$\mathbf{S}(t_{n-1}) \xrightarrow{D^+} \mathbf{S}^+(t_n) \xrightarrow{(D^-)^m} \mathbf{S}(t_n). \quad (2.2)$$

Перетворення D^+, D^- можуть бути представлені логіко-алгебраїчними рівняннями [36], які математично описують результат таких дій:

– (D^+) Для всіх переходів мережі Петрі, якщо момент часу співпадає з моментом виходу з переходу, то в усі вихідні позиції переходу додати кількість маркерів, що дорівнює кратності відповідної дуги, та видалити з

множини моментів виходу переходу відповідне значення, якщо в множині більше ніж 1 елемент, або замінити значення на ∞, якщо в множині тільки 1 елемент.

– (D^-) Для всіх переходів мережі Петрі, якщо виконана умова входу маркерів в перехід, то з усіх вхідних позицій переходу відняти кількість маркерів, що дорівнює кратності відповідної дуги, та додати до множини моментів виходу переходу нове значення, якщо момент найближчої події менший за ∞, або замінити значення на нове в противному випадку. Нове значення виходу маркерів розраховується як поточний момент часу плюс значення часової затримки в переході.
– ($(D^-)^m$) Вхід маркерів повторюється багатократно доки є хоч один перехід, для якого виконана умова входу маркерів. Число m є кількістю входів маркерів до досягнення стану мережі Петрі, в якому для жодного з її переходів не виконана умова входу маркерів.

Умова входу маркерів в перехід виконана тоді і тільки тоді, коли в усіх вхідних позиціях переходу наявна кількість маркерів не менша за кратність дуги, що з'єднує позицію з переходом. Математично сформулювати виконання умови для переходу T в момент часу t_n можна через такий предикат $I(T, t_n)$:

$$I(T, t_n) = \begin{cases} 1, & \forall P: (P,T) \in \boldsymbol{A} \quad M_{\boldsymbol{P}}(t_{n-1}) \geq W_{P,T} \\ 0, & \exists P: (P,T) \in \mathbf{A} \quad M_{\boldsymbol{P}}(t_{n-1}) < W_{P,T} \end{cases} \tag{2.3}$$

Отже, умова входу маркерів в перехід гарантує, що при виконанні перетворення D^- не може з'явитись від'ємне маркірування в позиції мережі Петрі.

У зв'язку з входом маркерів в переходи обов'язково потрібно вказувати як вирішуються конфлікти переходів. Конфлікт виникає, коли одночасно виконана умова для більш ніж одного переходу. Оскільки від послідовності запуску переходів може залежати у значній мірі результат моделювання, то спосіб, який використовує алгоритм імітації для вирішення конфлікту, є важливим. Стохастична мережа Петрі з багатоканальними та конфліктними переходами

визначає таке правило для розв'язання конфлікту переходів [36]: з усіх переходів, для яких виконана умова входу маркерів, обираються переходи з найвищим пріоритетом і, якщо їх більше одного, то подальший вибір здійснюється за параметром ймовірності запуску. Математично це можна представити таким виразом:

$$T_k \in \mathbf{T}: \ (I(T_k, t) = 1) \wedge \left(c_k = \max_j c_j\right) \wedge \left(\xi < \frac{\sum_{i=1}^{k} b_i}{\sum_{i=1}^{n} b_i}\right),$$

де k – індекс переходу, який виграв конфлікт,

ξ – випадкове число, рівномірно розподілене в інтервалі (0,1),

$n = |\mathbf{U}|, \mathbf{U} = \left\{T \in \mathbf{T}: \ (I(T, t) = 1) \wedge \left(c_T = \max_j c_j\right)\right\}$ – кількість переходів з виконаною умовою запуску та найвищим пріоритетом,

$c_j, j = \overline{1, n}$ – значення пріоритету переходу T_j,

$b_j, \ j = \overline{1, n}$ – значення ймовірності запуску переходу T_j.

Кожного разу в результаті вирішення конфлікту визначається один перехід і здійснюється перетворення D^-. Після зміни стану мережі Петрі знову виконується спроба виконати вхід маркерів в перехід і якщо тільки досягнуто стан, в якому вхід маркерів неможливий, то спроби припиняються, а алгоритм переходить до наступного свого кроку - визначення моменту найближчої події.

Наприклад, на рисунку 2.3 (а) для всіх трьох переходів виконана умова запуску і якщо вони усі вказані з однаковим значенням пріоритету та однаковою ймовірністю, то можливими є послідовності запусків :

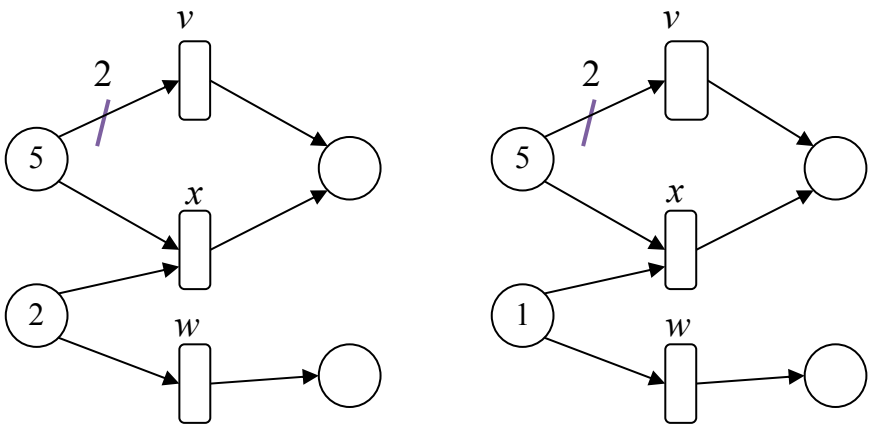
$v \to v \to x \to w$
$v \to v \to w \to x$
$v \to v \to w \to w$
$v \to w \to v \to w$
$v \to x \to x$
$v \to x \to w$
$v \to w \to x$

$x \to v \to v \to w$
$x \to w \to v \to v$
$x \to v \to w \to v$
$x \to v \to v \to w$
$x \to x \to v \to v$
$x \to v \to v \to x$
$x \to v \to x \to v$

$w \to v \to v \to x$
$w \to x \to v \to v$
$w \to w \to v \to v$
$w \to v \to v \to w$
$w \to v \to w \to v$
$w \to x \to v \to x$
$w \to x \to v$
$w \to v \to x$

Зазначені послідовності призводять до різного кінцевого маркірування, проте усі вони визначаються досягненням стану, в якому жоден з переходів мережі Петрі не запускається.

На рисунку 2.3 (б) перехід T2 вказаний з більш високим значенням пріоритету (візуально це помітно по більшій ширині графічного елементу). При перевірці умови входу маркерів тільки він один виявиться у групі з найвищим пріоритетом, тому здійснюється вхід маркерів саме в цей перехід. Маємо тільки такі можливі послідовності входів в переходи: $v \to v \to x$ або $v \to v \to w$.



(а) (б)

Рисунок 2.3 – Конфліктні переходи стохастичної мережі Петрі

Розв’язання конфліктів відтворює стохастичність перебігу подій, які відбуваються в системі. У контексті паралельних обчислень стохастичність виникає, наприклад, при захопленні локера тим чи іншим потоком.

Рівняння стану стохастичної мережі Петрі мають вигляд [37]:

$$\begin{cases} t_n = \min\limits_{T}\left(\min\limits_{q}[E_T(t_{n-1})]_q\right), \\ \mathbf{S}(t_n) = (D^-)^m D^+\left(\mathbf{S}(t_{n-1})\right), \\ m\colon \bigvee\limits_{T} I(T, t_n) = 0, \\ t_n \le T_{mod}, \end{cases} \tag{2.4}$$

де t_n, $n = 1, 2 \ldots$ - моменти часу, в які відбуваються виходи маркерів з мережі Петрі,

$I(T, t_n)$ – предикат, визначений виразом (2.2),

T_{mod} – час моделювання.

Перше рівняння системи (2.4) визначає просування часу, друге рівняння – зміну стану мережі Петрі, третє – кількість багатократних входів до досягнення стану, в якому жоден з переходів мережі Петрі не запускається, четверте – умову завершення імітації. У початковому стані мережі Петрі для всіх переходів, як правило, вказують стан $E_T(t_0) = \{\infty\}$, а в позиціях вказують початкове маркірування. Оскільки поточне значення часу (t = 0) не співпадає із значенням моменту виходу маркерів з переходу ($0 \neq \infty$), то для жодного переходу вихід не виконується і результатом перетворення D^+ у цьому випадку буде той же стан: $D^+\left(\mathbf{S}(t_0)\right) = \mathbf{S}(t_0)$. За наступним кроком відбувається перетворення $(D^-)^m$, що змінює маркірування позицій та стан переходів. Якщо хоч для одного переходу був виконаний вхід маркерів, то нове значення моменту виходу з переходу може стати моментом найближчої події. Після просування часу в цей момент відбудеться вихід маркерів і знову вхід маркерів в переходу. Просування часу відбувається, доки новий момент часу не перевищує задане значення часу імітації.

2.2.3 Поняття Петрі-об’єктної моделі

Припустимо, що суперклас Λ містить поле, яке зберігає інформацію про мережу Петрі, та методи, які необхідні для змінювання стану мережі Петрі у відповідності до перетворень D^- та D^+ відповідно. Тоді будь-який об’єкт такого класу може відтворювати запуски переходів мережі Петрі, заданої в його полі

`net`. При цьому гарантується, що усі об'єкти суперкласу відтворюють запуски переходів мережі Петрі однаково.

Отже, об'єкт o є Петрі-об'єктом, якщо і тільки якщо він є об'єктом суперкласу Λ:

$$o \subset \Lambda. \tag{2.5}$$

Ототожнення позицій мереж Петрі-об'єктів дає змогу передати інформацію про змінювання в одному з них іншому:

$$a.net.P_j = b.net.P_i ,$$

де $a \subset \Lambda$, $b \subset \Lambda$ – Петрі-об'єкти,

$a.net, b.net$ – їх мережі Петрі,

$a.net.P_j, b.net.P_i$ – позиції мереж Петрі, які ототожнюються.

Для пари Петрі-об'єктів може бути визначено кілька пар ототожнень, які утворюють конектор $c(a,b)$ цих об'єктів:

$$c(a,b) = \{(a.net.P_j , b.net.P_i)| P_j \in a.net.\mathbf{P}, P_i \in b.net.\mathbf{P} \},$$

де $a.net.P \in a.net.\mathbf{P}$, $b.net.P \in b.net.\mathbf{P}$ – позиції мереж Петрі-об'єктів a та b, $a.net, b.net$ – мережі Петрі-об'єктів a та b.

Петрі-об'єктна модель – це модель, що конструюється з Петрі-об'єктів та конекторів, які утворюють зв'язки між ними:

$$U = \{O, C \mid O = \{o_i, i = \overline{1,h}, \mid o_i \subset \Lambda \}, C = \{(o_i, o_j)| \ i \neq j\}\}. \tag{2.6}$$

Зв'язки між Петрі-об'єктами, виконані за допомогою ототожнень, гарантують, що динаміку Петрі-об'єктної моделі повністю визначає мережа Петрі, яка є об'єднанням мереж її Петрі-об'єктів:

$$U.net = \bigcup_j o_j.net. \tag{2.7}$$

Тому алгоритм імітації Петрі-об'єктної моделі відтворює функціонування мережі Петрі, яка є об'єднанням мереж Петрі-об'єктів. Проте, як доведено в роботі [33], перетворення D^- та D^+ мережі $U.net$ можуть бути розділені на перетворення D^- та D^+ її Петрі-об'єктів:

$$D^-(\mathbf{S}(t_n)) = \begin{Bmatrix} D^-(\mathbf{S_1}(t_n)) \\ D^-(\mathbf{S_2}(t_n)) \\ \dots \\ D^-(\mathbf{S_h}(t_n)) \end{Bmatrix},\ D^+(\mathbf{S}(t_n)) = \begin{Bmatrix} D^+(\mathbf{S_1}(t_n)) \\ D^+(\mathbf{S_2}(t_n)) \\ \dots \\ D^+(\mathbf{S_h}(t_n)) \end{Bmatrix}, \quad (2.8)$$

де $\mathbf{S_j}(t_n),\ j = \overline{1,h}$ – стан мережі $o_j.net$.

З урахуванням розділення перетворень (2.8) рівняння станів моделі (2.4) набувають вигляду:

$$\begin{cases} t_n = \min\limits_T \left(\min\limits_q [E_T(t_{n-1})]_q \right), \\ \mathbf{S_j}(t_n) = (D^-)^{m_j} D^+(\mathbf{S}(t_{n-1})),\ m_j : \bigvee\limits_T I(T, t_n) = 0, \quad j = \overline{1,h}, \\ n = 1,2 \dots \end{cases} \quad (2.9)$$

Отже, перетворення стану моделі відбувається перетворенням стану її Петрі-об’єктів. За рахунок такого розбиття досягається більша швидкодія алгоритму імітації, оскільки замість перегляду усіх переходів мережі Петрі при кожному просуванні часу здійснюється перегляд стану переходів Петрі-об’єкту, момент виходу маркерів якого співпадає з поточним моментом часу. Кількість переходів одного об’єкта значно менша за кількість переходів в усій мережі, то отриманий виграш в часі виконання імітації буде відповідно великим. Окрім цього, кожен об’єкт запам’ятовує своє значення найближчої події, тому при пошуку моменту найближчої події пошук мінімального значення відбувається з множини значень для всіх об’єктів замість з множини значень для всіх переходів мережі Петрі. Таким чином, окрім спрощення розробки моделі досягається також прискорення імітації процесу, який моделюється.

Важливою перевагою використання Петрі-об’єктів є можливість тиражування об’єктів на основі один раз створеної мережі Петрі, що забезпечується об’єктно-орієнтованим підходом. Тиражування об’єктів можна виконувати з заданими параметрами, які передаються в конструкторі об’єкта і можуть бути використані для налаштування чисельних параметрів мережі Петрі-об’єкта. Хорошою практикою, що допомагає у розробці складних моделей, є також створення класів Петрі-об’єктів, в конструкторі яких вказана

використовувана мережа Петрі та застосування аргументів конструктора до параметрів мережі Петрі:

$$o \subset \Upsilon - \triangleright \Lambda, \tag{2.10}$$

де символом $-\triangleright$ позначено операцію спадкування,

Υ – клас-нащадок класу Λ.

В термінах об'єктно-орієнтованого програмування для конструювання моделі з об'єктів використовується операція агрегування:

$$U \text{—}\diamond \{\Upsilon_1, \Upsilon_2, .. \Upsilon_l\}, \Upsilon_j - \triangleright \Lambda\ , \tag{2.11}$$

де символ —◇ позначає операцію агрегування в термінах UML [38].

2.3 Механізми паралельних обчислень у багатопотоковій технології Java

З погляду побудови програми паралельних обчислень розрізняють паралелізм за даними та за завданнями. Паралелізм за даними означає, що одні й ті самі обчислювальні дії окремі процеси програми виконують одночасно для різних даних. Декомпозиція задачі полягає у декомпозиції даних на множини, що не перетинаються, і запускаються одночасно на обчислення. Паралелізм за завданнями означає, що на одних і тих самих даних запускаються на обчислення різні функції. Декомпозиція задачі в цьому випадку виконується за обчислювальними діями, що можуть бути виконані одночасно. Кожний спосіб має свої переваги та недоліки, які оглянуті у виданні [39]. Можливість використовувати спільні дані зменшує час, витрачений на передачу даних між окремими потоками. Проте використання спільних даних потоками обчислень потребує використання синхронізації, що може сповільнити роботу програми. На противагу паралелізму за завданнями, обробка окремих наборів даних обчислювальними процесами зменшує необхідність використання синхронізації доступу до даних, а також спрощує рівномірне розподілення обчислювальних дій між процесами. У складних програмах, як правило, використовують обидва підходи, щоб досягти найліпшого результату.

З появою багатоядерної архітектури розробники програмного забезпечення отримали можливість одночасного використання обчислювального ресурсу навіть за умови однопроцесорного комп'ютера. Натомість необхідно докласти додаткові зусилля щодо розробки програми для паралельного виконання. Основні методики розробки паралельної програми під багатоядерну архітектуру описані у [40]. Вичерпний опис небезпек, що приховуються в багатопотокових програмах на Java можна знайти в роботі [41].

Паралельна програма — це набір процесів, що виконуються одночасно. Процес асоціюється з обчислювальним підзавданням, яке описується послідовністю запрограмованих інструкцій Процес працює в операційній системі, яка керує використанням обчислювального ресурсу серед набору процесів. Потік - це елементарний процес. Будь-який процес складається з одного або кількох потоків і має виокремлену пам'ять, до якої не мають доступу інші процеси. Потоки, що належать одному процесу, можуть використовувати спільну пам'ять. Тож, виконання багатопотокової програми — це одночасне виконання кількох потоків.

Потік виконує код тільки у тому випадку, якщо відбувається захоплення ресурсів (ядер процесору, спільних даних). Обмеження ресурсів викликає конфлікт запущених на виконання потоків, що призводить до неправильного функціонування програми.

Операції потоку виконуються асинхронним або синхронним способом. Асинхронне виконання означає, що потік може виконуватися незалежно від інших потоків (окрім основного потоку). Синхронне виконання означає, що потік має очікувати сигналу від інших потоків, щоб продовжити своє виконання. Координація роботи потоків потрібна, наприклад, коли відбувається доступ до спільних даних.

Базовим механізмом паралельних обчислень є призупинка потоку. Він реалізується методами sleep та join в залежності від потрібного виду взаємодії

між потоками. Метод `sleep` – блокує виконання потоку на визначений проміжок часу. Метод `join` дозволяє одному потоку чекати завершення іншого.

Іншим засобом взаємодії між потоками є методи `wait()` і `notify()/notifyAll()`. Метод `wait()` змушує поточний потік чекати, поки інший потік не викличе методи `notify()` або `notifyAll()` для цього об’єкта. Метод `notify()` активує один потік, який очікує на моніторі цього об’єкта. Метод `notifyAll()` активує всі потоки, які очікують на моніторі цього об’єкта. Потік очікує на моніторі об’єкта, якщо він викликав один із методів `wait()`.

Атомарна дія - це дія, яка відбувається або повністю, або зовсім не відбувається, без можливості зупинки посередині. Побічні ефекти атомарної дії не стають видимими до того моменту, поки дія не завершиться.

Існують дії, які можна вважати атомарними:

- операції читання та запису є атомічними для змінних посилання та для більшості простих типів (крім long та double);
- операції читання та запису є атомічними для всіх змінних, які були оголошені з ключовим словом "volatile" (включаючи long та double).

Такі атомарні дії не можуть бути перервані іншими діями, що дозволяє їх використовувати без страху втручання потоків. Однак, це не усуває потреби синхронізації атомарних дій, оскільки існує ризик помилок у збереженні пам'яті. Використання ключового слова "volatile" допомагає зменшити цей ризик, оскільки гарантує, що зміни змінних будуть видимими іншим потокам. Варто звернути увагу на те, що при читанні змінної "volatile" потік бачить не тільки її останнє значення, але й всі побічні ефекти, що призвели до цієї зміни. Тож, використання атомарних змінних є ефективнішим, ніж доступ до змінних через синхронізований код, але вимагає від програміста більшої обережності щодо уникнення помилок узгодженості пам’яті.

Синхронізовані колекції - це ще один інструмент для безпечної взаємодії потоків. Найбільш використовувані типи синхронізованих колекцій це

`BlockingQueue` та `ConcurrentMap`. `BlockingQueue` – черга, яка додатково підтримує операції очікування, поки черга стане непорожньою під час отримання елемента, та очікування, поки в черзі звільниться місце під час збереження елемента. `ConcurrentMap` – це паралельний аналог колекції `Map`, що забезпечує безпеку потоків і гарантії атомарності. Операції видалення або заміни пари ключ-значення в такій колекції відбувається, лише якщо ключ присутній, так само, як і операція додавання пари ключ-значення відбувається, лише якщо ключ відсутній. Таким чином, синхронізовані колекції допомагають уникнути помилок узгодженості пам’яті.

Загалом, дефекти синхронізації, що спричиняють помилки, досить часто виникають у багатопотокових програмах [42]. Головними проблемами, що трапляються при розробці паралельної програм, є взаємоблокування (deadlock) та неузгодженість даних. Взаємоблокування виникає, коли два або більше потоків очікують сигналу, який неможливо отримати. Наприклад, потік А чекає сигналу від потоку В, але останній чекає сигналу від потоку А. Тому вони потрапили в стан взаємного очікування один одного. Взаємоблокування має гарний опис за допомогою звичайної мережі Петрі. Відомий приклад п'яти філософів описує Дж. Пітерсон [13]. Для дослідження ситуацій взаємоблокування у багатопотокових програмах у роботі [43] автори пропонують спеціальний клас звичайної мережі Петрі під назвою мережа Гадара.

Інша проблема багатопотоковості – неузгодженість даних – означає помилку обчислення, коли два або більше потоків одночасно модифікують спільні дані. У [44] запропоновано розгорнутий метод виявлення помилок неузгодженості даних у багатопотокових програмах з використанням специфічних мереж Петрі з даними (PD-net).

У роботі [45] запропоновано фреймворк під назвою PVcon для динамічного виявлення помилок паралелізму. Експериментальні результати показують, що PVcon може ефективно виявляти вдвічі більше помилок у багатопотокових програмах, ніж інші методи. Однак, автори вказують, що у

даній роботі вони зосередились на помилках багатопотоковості за виключенням проблем взаємоблокувань.

У роботі [46] запропоновано підхід для перевірки паралельних алгоритмів та програм за допомогою кольорових мереж Петрі. Моделі, що будуються за кодом, можна перевірити на правильність, щоб довести відсутність помилок взаємного блокування.

Огляд існуючих методів зневадження паралельних та розподілених програм міститься в роботі [47]. Дана робота підтверджує необхідність уніфікованого засобу налагодження багатопотокових програм.

На закінчення, поведінка паралельного алгоритму сильно залежить від ресурсів і співвідношення часу виконання інструкцій. Тому точна модель, що описує таку поведінку, може бути побудована за допомогою стохастичної мережі Петрі.

2.4 Метод Петрі-об’єктного моделювання паралельних обчислень

2.4.1 Шаблони моделювання паралельних обчислень мережею Петрі

У роботах [48, 49, 50] було запропоновано дослідження паралельних програм на моделях, представлених стохастичною мережею Петрі. Концепція використання шаблонів та ключові шаблони для проєктування багатопотокових обчислень запропоновані в роботі [51]. Основне призначення таких шаблонів – прискорення процесу розробки моделей за рахунок спрощення та зменшення кількості помилок. Шаблони моделювання паралельних обчислень розроблені в контексті багатопотокової технології Java, проте механізми, які розглядаються, притаманні усім реалізаціям багатопотоковості іншими мовами програмування. У роботі [52, 53] міститься опис розробки початкових моделей базових механізмів багатопотокової програми та впровадження їх у програмному забезпеченні Петрі-об’єктного моделювання для полегшення процесу побудови моделі паралельної програми. Варто зазначити, що у порівнянні із зазначеною

роботою у даному дисертаційному дослідженні частина моделей удосконалена за рахунок більш точного відтворення деталей функціонування механізмів багатопотоковості.

Ключовими шаблонами моделювання рутинних інструкцій програми є цикл for, цикл while, оператор if-then-else.

Цикл for може бути представлений двома подіями: перша починає виконання однієї ітерації циклу, якщо кількість повторень не вичерпано, або завершує виконання циклу в іншому випадку, а друга завершує виконання ітерації (рис. 2.4, а). Позиція “numIteration” міститиме кількість маркерів m, яка дорівнює кількості ітерацій, які залишилися для виконання. Позиція "for" обмежить виконання ітерацій одна за одною. Коли всі ітерації завершені, *m* маркерів будуть у вхідній позиції переходу "forFinish", і перехід спрацює.

Цикл while відрізняється від циклу for лише перевіркою умови для продовження ітерації (рис. 2.4, б). Ця умова представлена позицією "cond", пов’язаною із переходом "while start" інформаційною дугою. Пріоритет переходу "while start" повинен бути встановлений на вище значення, ніж пріоритет переходу "while finish".

Оператор if-then-else представлений на рисунку 2.4 (в). Перевірка стану виконується перевіркою маркера в позиції "condition". Конфлікт між альтернативними подіями "if" та "else" вирішується встановленням вищого пріоритету для переходу "if". Лише коли умова не відповідає дійсності, може спрацювати перехід "else".

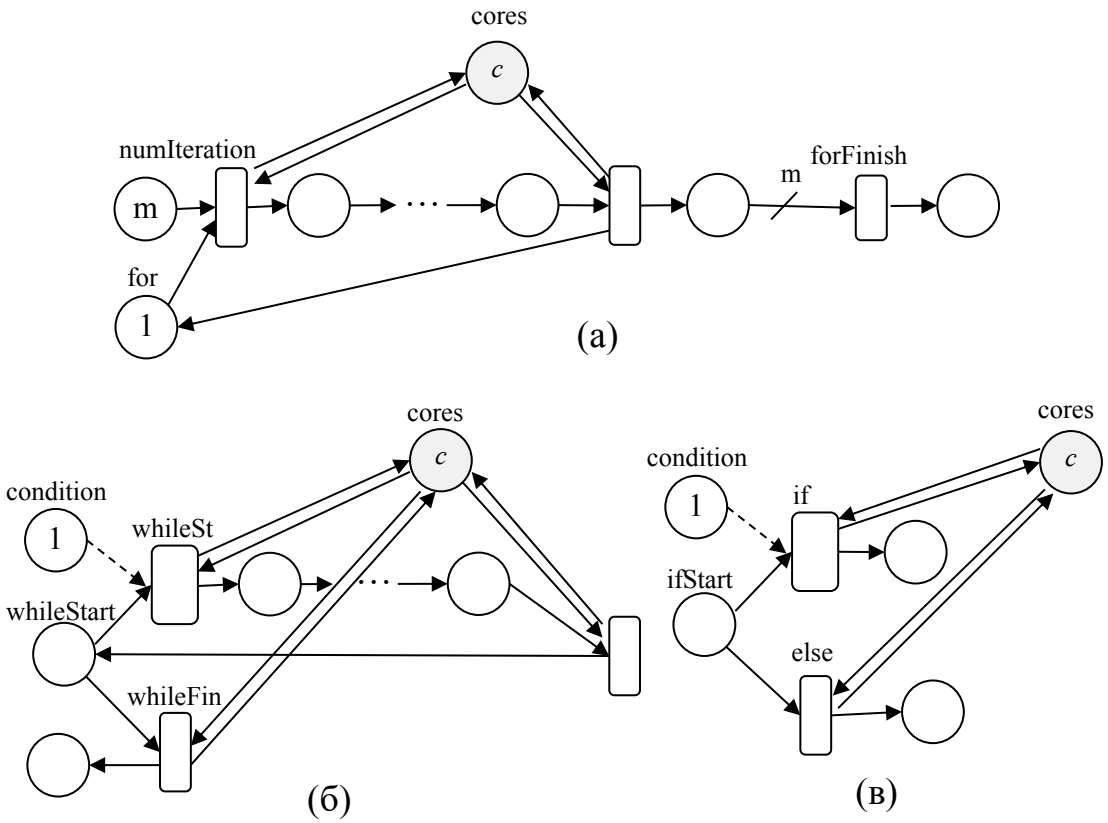


Рисунок 2.4 – Моделювання циклу for (а), циклу while (б) та оператора if-then-else (в) стохастичною мережею Петрі.

Кожна інструкція в програмі потребує захоплення обчислювального ресурсу. Тому всі переходи, які відповідають обчислювальним діям, мають вхідну та вихідну позицію "cores", яка містить інформацію про кількість наявного обчислювального ресурсу. Обмеження ресурсу означає, що не більше *c* обчислювальних дій можуть виконуватись одночасно.

Варто відмітити, що всі переходи, які відтворюють інструкції програми, задіюють обчислювальний ресурс. Однак для більш простого візуального представлення захоплення ресурсу не зображатиметься для кожного переходу, а наявність позиції "cores" та зв’язків з нею кожного переходу передбачається за замовченням.

Позиції мережі Петрі, які будуть використовуватись далі для обміну інформацією про свій стан з іншими Петрі-об’єктами, відмічаються сірим кольором. Тим самим символізуємо, що змінювання їх маркіркування може бути

спричинено також іншими Петрі-об’єктами, оскільки такі позиції можуть бути спільними з іншими об’єктами.

Будь-яка програма в java є потоком, і `main`-метод запускається Main-потоком програми. Створення іншого потоку, початок та завершення його роботи ініціюються такими інструкціями програми:

```
public static void main(String[] args) {
        Thread thread = new Thread(new Runnable(){
            @Override
            public void run() {
              …  // код дій потоку
            }
        });
        thread.start();
}
```

Для створення `Runnable`-об’єкта можна скористатись також лямбда-виразом:

```
Thread thread = new Thread(() -> {
              …  // код дій потоку
});
```

При створенні потоку створюється об’єкт, який підтримує інтерфейс `Runnable`. При цьому потік, який ініціює створення нового потоку продовжує свою роботу. Запуск потоку на виконання відбувається викликом методу `start`, який виконує дії, вказані в методі `run` потоку. Потік не завершує роботу доки усі потоки, продуковані ним, не завершили свою роботу, тому метод `main` завершить своє виконання тільки після завершення роботи створеного ним потоку. На рисунку 2.5 представлена мережа Петрі, що моделює роботу обох потоків – `main` та створеного ним `thread`. Між start та end, runSt та runEnd розміщуються події, які відповідають обчислювальним діям, що виконуються потоком (позначені на рисунку трьома крапками).

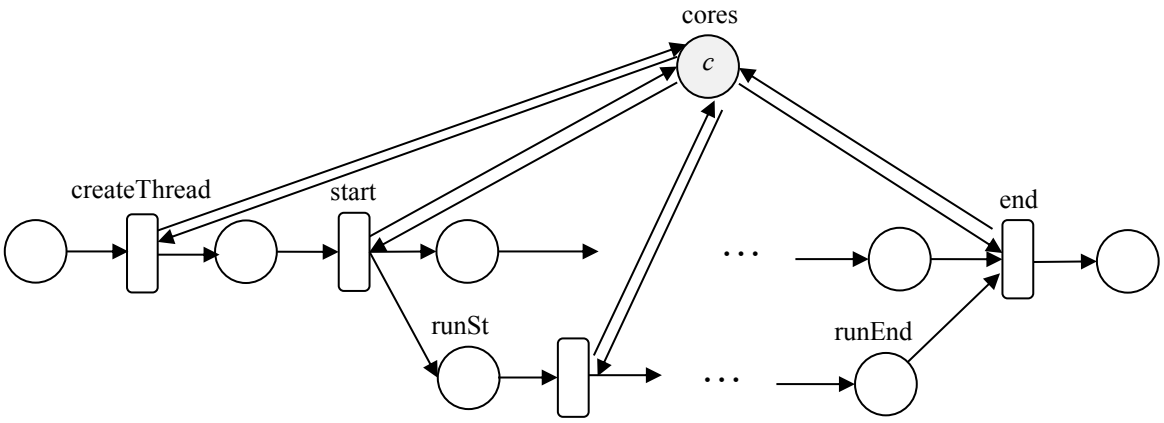


Рисунок 2.5– Фрагмент мережі Петрі, що моделює створення, початок та завершення роботи потоку.

Метод `sleep` є статичним і вказує на необхідність призупинки виконання обчислювальних дій в тому місці програми, де метод викликаний. Призупинено буде виконання дій того потік, який виконує обчислювальні дії у цьому місці коду. Метод `sleep` звільнює обчислювальний ресурс на час призупинки, тому він моделюється трьома подіями: звільнення ресурсу, завершення інтервалу призупинки, захоплення ресурсу (рис. 2.6). Звідси ясно, що продовжити свою роботу потік зможе тільки при наявності вільного обчислювального ресурсу. Отже фактичний час, на який було призупинено роботу може бути більшим, ніж вказаний в аргументі `sleep` методу.

Метод `join` є одним з найпростіших методів узгодження роботи потоків і водночас найбільш використовуваним. Він викликається для вказаного потоку *B* і зобов'язує програму в даному місці коду (потоку *A*) дочекатись, доки потік *B* завершить свою роботу. Достатньо використати одну подію для відтворення такого очікування (див. рис. 2.6). Найпростіший приклад використання `join`-методу – це очікування в `main`-методі завершення обчислень усіх потоків для початку формування та виведення результату обчислень.

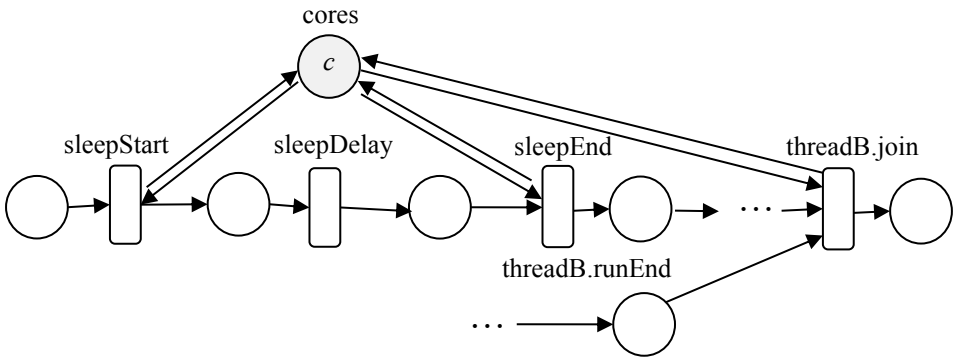


Рисунок 2.6 – Фрагмент мережі Петрі, що моделює призупинку потоку та очікування на завершення виконання дій потоку .

У багатопотоковому програмуванні, коли кілька потоків мають доступ до спільного розділеного ресурсу, необхідно гарантувати, що цей ресурс використовується тільки одним потоком одночасно. Цей процес називається синхронізацією. Для досягнення цієї мети використовуються механізми, такі як синхронізований метод, блокування потоків та блок синхронізації дій.

Якщо для об’єкта викликається синхронізований метод, його монітор буде захоплено, якщо він доступний. Монітором в java називають внутрішній локер, яким оснащений будь-який об’єкт. Лише один потік може бути власником монітора. Якщо потік намагається захопити монітор, коли він заблокований іншим потоком, потік стає заблокованим, доки монітор об’єкта не стане доступним для захоплення. На рисунку 2.7 представлена найпростіша реалізація блокування монітором об’єкта фрагментом мережі Петрі.

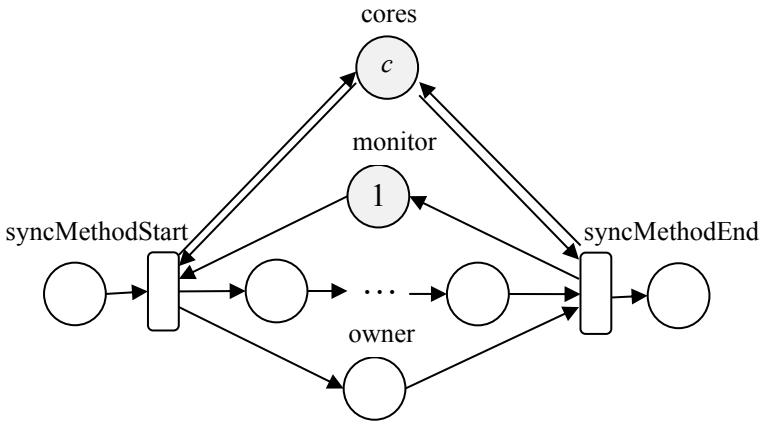


Рисунок 2.7 – Фрагмент мережі Петрі, що моделює захоплення монітора об’єкта, для якого викликано синхронізований метод

Слід зазначити, що потік, який є власником монітора, може будь-скільки разів успішно захопити монітор. Таку властивість об’єкта, що використовується для блокування, називають reentrance. Лічильник блокувань забезпечує кількість розблокувань, що дорівнює кількості блокувань. Звичайний об’єкт, який може виступати монітором, не має методів, які надають можливість контролювати кількість захоплень одним потоком. Представимо цю можливість на прикладі використання локерів (об’єкти типу Lock). Використання локерів надає програмісту більш широкі можливості контролювати захоплення локерів, використовувати різні варіанти захоплення (з очікуванням на звільнення локера, або без нього), а також розвинуті можливості для дебагінгу стану локера в ході виконання програми. Фрагмент java-коду для захоплення локера має вигляд:

```
private final Lock lock = new ReentrantLock();
try{
     lock.lock()){
     …// synchronized actions
}
} finally {
    lock.unlock();
}
```

На рисунку 2.8 представлено фрагмент мережі Петрі для моделювання захоплення локера з урахуванням можливості його повторного захоплення потоком, який вже є власником цього локеру. Заради зручності сприйняття фрагментів мереж Петрі та спрощення їх представлення, *на наступних рисунках будемо вважати за замовчуванням, що наявні зв’язки усіх переходів з позицією cores*. Подія "lock" буде спрацьовувати при першому запиті на захоплення локера і змінює стан потока на власника. При подальших запитах, якщо потік вже є власником локера, буде спрацьовувати перехід "asOwner" (він має менший пріоритет у порівнянні з "lock"). Позиція "holdCount" містить інформацію про кількість захоплень локера як власника (загальна кількість захоплень на 1 більша). При кожному завершенні опрацювання дій, що призначені для блокування, спрацьовує подія "release" (має більший пріоритет у порівнянні з "unlock"), яка зменшує кількість захоплень локера як власника, якщо він має

додатнє значення. Як тільки лічильник захоплень "holdCount" досяг нульового значення, спрацьовує подія "unlock", яка розблоковує локер. Таким чином контролюється, що кількість захоплень локера дорівнює кількості звільнень локера.

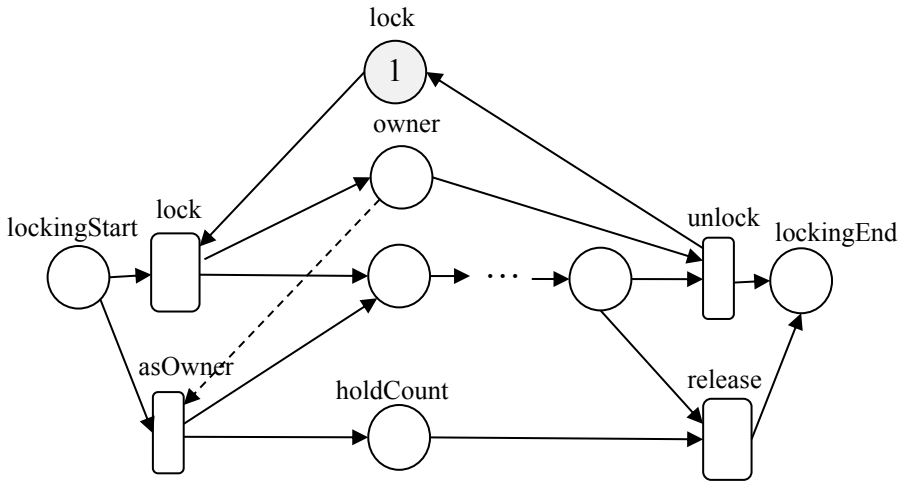


Рисунок 2.8 – Фрагмент мережі Петрі, що моделює захоплення локера об’єкта з урахуванням можливості багаторазового захоплення.

За умови, що в програмі гарантовано не відбувається повторний запит локера, в моделі можна використовувати більш спрощений варіант з урахуванням виключного однократного захоплення локера (рис. 2.7). Це, звісно, спрощує представлення моделі в цілому, оскільки фрагмент на рисунку 2.7 містить вдвічі менше переходів, ніж той, що на рисунку 2.8.

Метод `tryLock()` інтерфейсу `Lock` надає можливість перевірити результат захоплення локера і прийняти рішення, що робити далі в програмі. На відміну від `lock()` методу цей метод не зобов’язує програму очікувати звільнення локера [54]. Програміст має вирішити і вказати програмі, що робити у разі хибної спроби захопити локер. У наведеному далі фрагменті синхронізовані дії будуть виконуватись тільки у разі успішного захоплення локера, в іншому випадку програма продовжить виконання наступної інструкції коду:

```
private final Lock lock = new ReentrantLock();
try{
```

```
     if (lock.tryLock()){
          …// synchronized actions
}
…// asynchronnous actions
} finally {
    lock.unlock();
}
```

Представимо спочатку спрощений варіант однократного захоплення локера (рис. 2.9). Перехід з вищим пріоритетом "tryLockTrue" спрацює як тільки є запит на блокування і доступ до локера вільний, в противному випадку спрацює перехід "tryLockFalse".

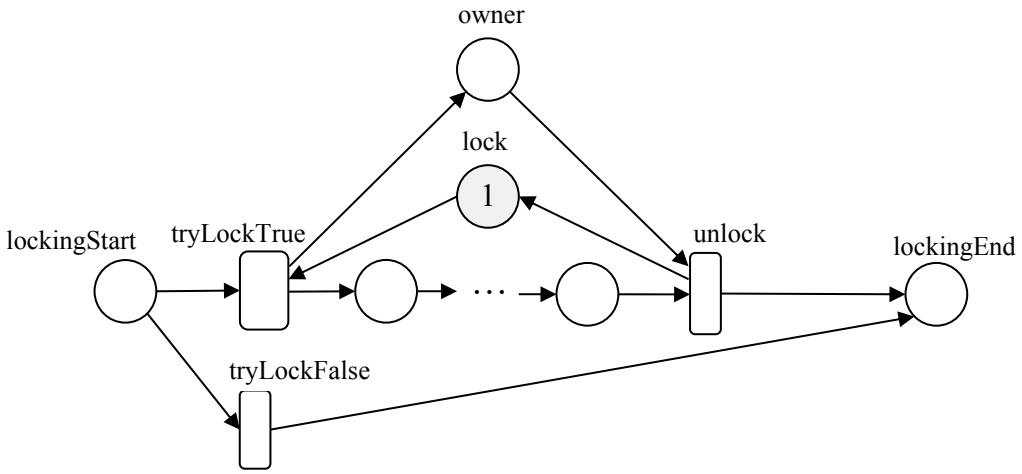


Рисунок 2.9 – Фрагмент мережі Петрі, що моделює захоплення локера без очікування його звільнення

У випадку негайного (тобто без очікування звільнення) захоплення локера з можливістю багатократного захоплення відповідний фрагмент мережі Петрі представлений на рисунку 2.10. Переходи "tryLockTrue", "tryAsOwner", "tryLockFalse" мають пріоритети найвищий, середній та найнижчий відповідно. Переходи "release", "unlock" – найвищий та найнижчий відповідно.

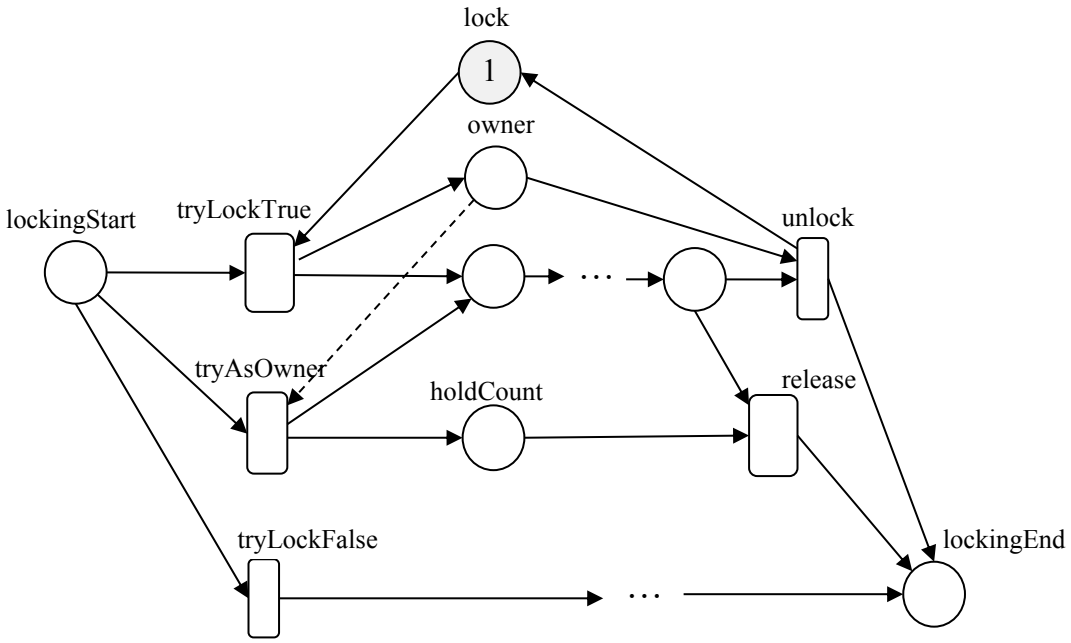


Рисунок 2.10 – Фрагмент мережі Петрі, що моделює негайне (без очікування звільнення) захоплення локера з урахуванням можливості багаторазового захоплення

Наведемо простий приклад застосування блокування у програмі. У випадку, коли потоки виконують неатомарні операції з даними, їх дії можуть виконуватись у порядку, що призводить до помилки, яку називають гонитвою (race condition) [55]. Навіть проста дія - збільшення значення для спільної змінної - при одночасному виконанні потоками може виконуватись помилково, оскільки в машинному коді вона буде представлена трьома діями: читання, збільшення та записування значення. Для правильного функціонування необхідно блокувати обчислювальні дії зі спільними даними, наприклад, синхронізованим методом:

```
public synchronized void  incMethod(){
            local++;
}
```

Фрагмент мережі Петрі, зображений на рисунку 2.11, був використаний у роботі [51] для дослідження конфлікту потоків при доступі до спільних даних в залежності від кількості потоків та кількості обчислювального ресурсу. У фрагменті присутні події "read", "modify", "write", які відтворюють зчитування,

змінювання та запис значення відповідно. При виконанні події відбуваються відповідні обчислювальні дії: local = shared, local ++, shared = local.

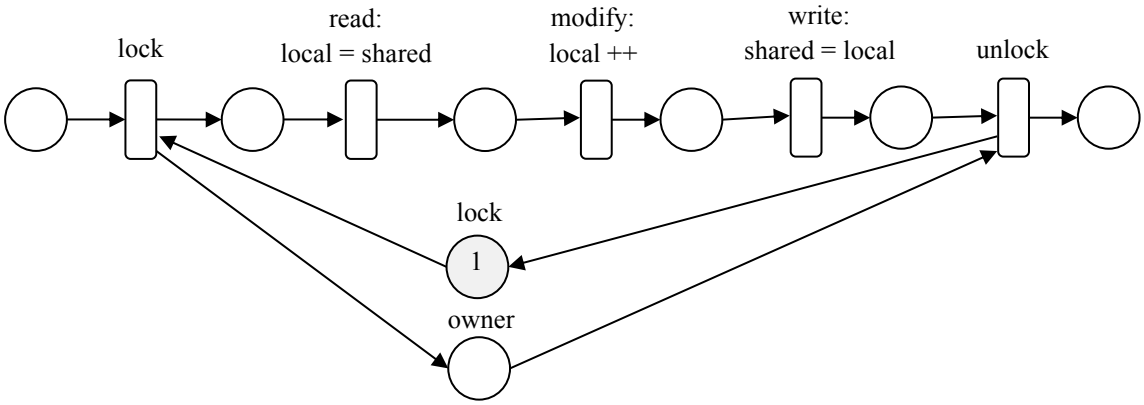


Рисунок 2.11– Фрагмент мережі Петрі, що моделює синхронізований доступ потоку до спільного значення.

Важливим механізмом синхронізації, який дозволяє перевести один з потоків у стан очікування доки певна умова не буде виконана, є механізм очікування за умовою wait/notify. Інший потік повинен виконати метод-сповіщення `notifyAll()` або `notify()`, щоб надіслати сигнал першому потоку. Отриманий сигнал примусить потік, який очікує, припинити очікування та перевірити умову очікування знову. Механізм є важливим, оскільки дає змогу організувати взаємодію потоків без зайвої перевірки умови в нескінченному циклі. Замість цього перевірка умови буде виконуватись тільки тоді, коли змінились умови, про що повідомляє сигнал.

Патерн застосування wait/notify передбачає спільне використання об’єкта, для якого перевіряється умова. Тому очікування за умовою завжди виконується в синхронізованому фрагменті коду. Програмний код методу, який викликається потоками, виглядає так [56]:

```
public synchronized void method() throws InterruptedException{
   while (!condition()) {
     wait();
   }
     …// some actions
   notifyAll();
}
```

Відповідний фрагмент мережі Петрі, очевидно, має містити події "syncStart", "check", "wait", "getSignal", "syncEnd", які відтворюють блокування, перевірку умови, отримання сигналу та розблокування відповідно. Коли спрацьовує `wait()` метод, то відбувається розблокування монітора об’єкта (це зазначено в документації методу [57]) і потік втрачає статус власника монітора об’єкта, а спрацьовування методу `notifyAll()` відбудеться тільки за умови повернення потоку статусу власника монітора об’єкта. Механізм отримання сигналу передбачає, що він буде отриманий тільки за умови, якщо потік у стані waiting, інакше він буде просто попущений. Для відтворення цієї події додамо перехід "isNotWaiting". Якщо сигнал отримано, коли потік у стані waiting, то наступною дією буде захоплення монітора об’єкта. Для відтворення цього захоплення додамо подію "catchMonitor". На рисунку 2.12 представлено відповідний фрагмент мережі Петрі. Кожного разу, коли отримано сигнал від іншого потоку, виконується спроба захоплення монітора об’єкта та перевірка умови (позиція "condition"). Ці дії будуть продовжуватись доки потік не зможе натрапити на умову, яку очікує. У цьому і проявляється цикл while з наведеного фрагменту коду. Зазначимо, що у випадку, коли кількість потоків у програмі перевищує два, до позиції "signal From" маркер надходить від будь-якого потоку, а з позиції "signalTo" маркером може скористатись будь-який потік.

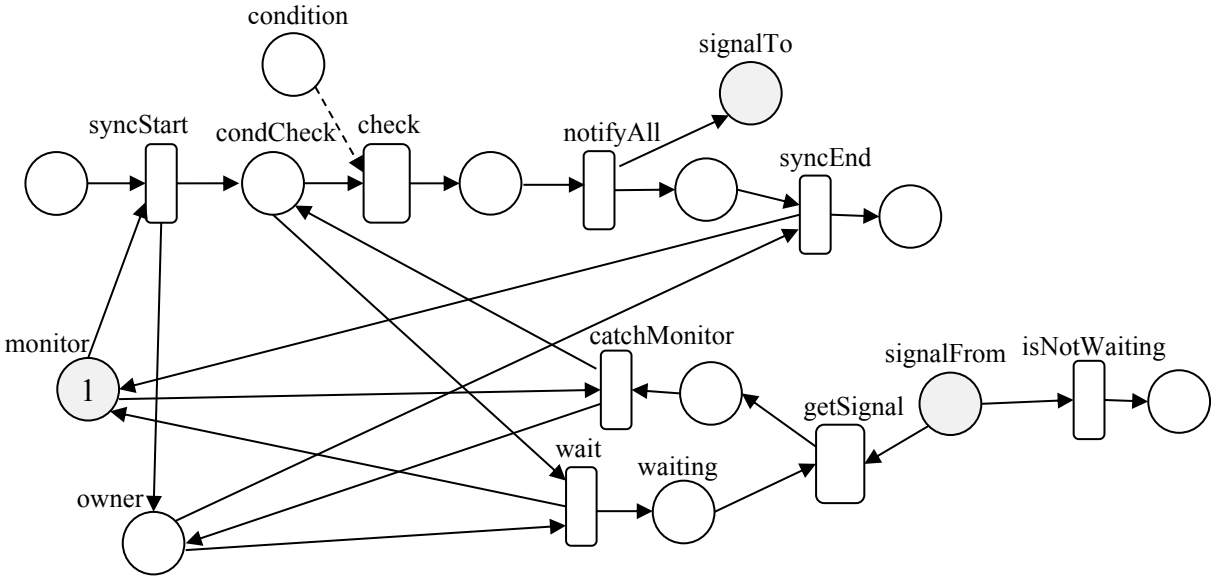


Рисунок 2.12 – Фрагмент мережі Петрі, що моделює очікування за умовою.

Використання потоків є ресурсовитратним, тому їх збільшення в програмі негативно впливає на швидкодію. При великій кількості підзадач у програмі ефективним буде використання пулу потоків (Thread Pool). Замість створювати для кожної окремої підзадачі окремий потік, пул потоків пропонує використання одного й того ж набору обмеженої кількості потоків для обробки черги завдань. Після свого створення пул потоків знаходиться у постійному очікуванні задач на обробку, доки не буде викликаний метод, який закриває можливість надходження нових завдань в чергу пулу потоків. Об’єкти, які можуть завантажуватись на обробку, мають бути Runnable (або Callable) об’єктом, тобто містити метод run() (або call()), в якому прописані обчислювальні дії підзадачі. Типовий фрагмент коду для створення пулу *k* потоків, завантаження в нього *w* задач та завершення його роботи має вигляд:

```
ExecutorService pool = Executors.newFixedThreadPool(k);
for (int j = 0; j < w; j++) {
   pool.execute(new Runnable(){
                @Override
                public void run() {
                  …  // код дій підзадачі
                }
   });
}
pool.shutdown();
```

Для створення пулу потоків у наведеному вище фрагменті використана фабрика пулу потоків з бібліотечного класу Executors. Замість методу `execute()` для завантаження задачі в пул може бути також використаний метод `submit()`, який окрім виконання заданих у методі `run()` (або `call()`) дій, повертає результат виконання. На випадок, коли в наступній після `shutdown()` інструкції потрібно гарантувати завершення роботи усіх підзадач пулу, документація java рекомендує використовувати метод awaitTerminaton().

Отже, події, які відтворюють роботу пулу потоків, це "poolCreate", "execute", "runSt", "runEnd", "shutdown", "awaitTerminaton". Після створення пул завантажує задачі по одній, а після завантаження усіх переходить до інструкції `shutdown`. Завантажені задачі очікують вільного потоку, щоб виконати свій метод run(). Виконання методу у найпростішому випадку може бути відтворено однією подією "runSt", більш загальним буде відтворення двома подіями "runSt" та "runEnd", між яким можуть бути додані інші події, що відповідають обчислювальним діям методу `run()`. Відповідний фрагмент мережі Петрі представлений на рисунку 2.13. Перехід "execute" вказаний з більшим пріоритетом, ніж перехід "shutdown". Тому перехід "shutdown" спрацює тільки, коли для "execute" не виконана умова запуску, тобто коли усі *w* маркерів з позиції "tasks" будуть вичерпані. Для виконання переходу "awaitTerminaton" необхідно завершення виконання усіх завантажених задач.

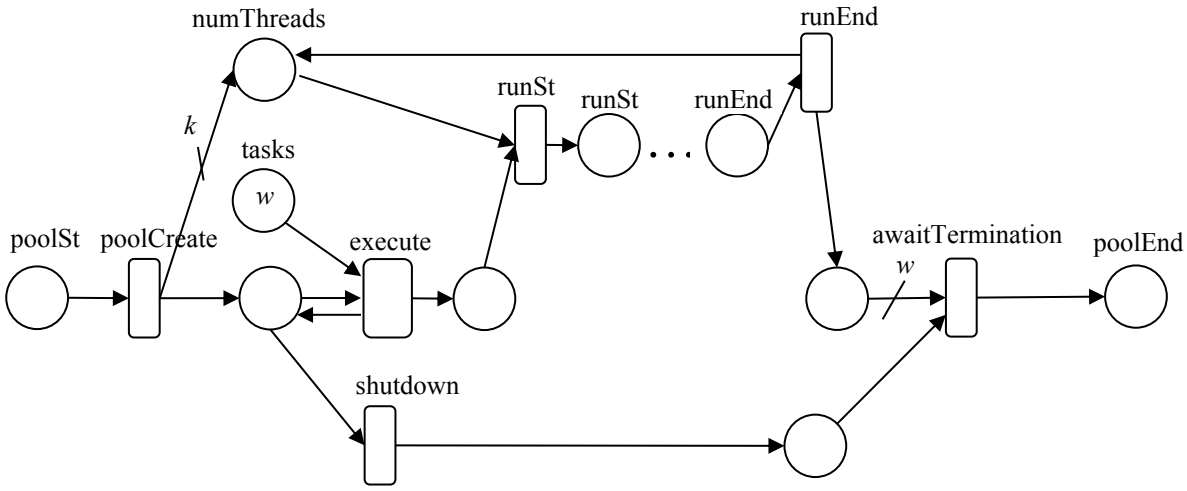


Рисунок 2.13– Модель обчислень пулом потоків.

Тестування шаблонів, запропонованих у підрозділі, виконувалось у програмному забезпеченні при різних потенційно можливих варіантах початкового маркірування. Перебіг подій перевірявся за допомогою протоколу подій. Використовувалась також анімація для візуалізації спрацьовування переходів. Коректність побудованих у підрозділі шаблонів моделювання підтверджується їх успішним використанням для побудови достатньо точних моделей паралельних програм, які описані у наступних розділах дисертації. Програмний код методів-кріейторів мереж Петрі шаблонів наведений у додатку А.

2.4.2 Розробка мережі Петрі-об’єкта, що імітує Runnable / Callable об’єкт багатопоточної програми, на основі шаблонів моделювання обчислювальних дій.

Мережа Петрі об’єкту, що імітує Runnable / Callable об’єкт багатопоточної програми, обов’язково містить фрагменти створення, запуску та завершення (рис. 2.4). Між подіями runSt та runEnd відтворюється послідовність інструкцій, зазначена у run / call методі потоку чи підзадачі. Кожній інструкції, яка не передбачає взаємодії з іншими потоками, ставиться у відповідність подія з часовою затримкою або з нульовою затримкою, якщо затримкою можна

знехтувати (вона мала у порівнянні з іншими). Якщо виконано дослідження часової затримки для кількох послідовних інструкцій, то замість послідовності подій, цій групі інструкцій ставиться у відповідність одна подія з визначеною часовою затримкою. Коли трапляється інструкція, що здійснює взаємодію з іншим потоком чи підзадачею, то використовується відповідний фрагмент мережі Петрі, а позиції, які будуть спільними з іншим потоком чи підзадачею, відмічаються як контактні (на зображенні зафарбовуються сірим кольором).

Розроблену мережу Петрі потрібно зберегти у вигляді методу, що для заданих параметрів створює елементи мережі Петрі, конструює з них мережу Петрі та повертає як результат у вигляді об'єкту `PetriNet`. Домовимось усі такі методи називати кріейторами мереж Петрі та називати, починаючи з `createNet`:

```
public static PetriNet createNetName(args),
```

де *Name* – назва мережі,

args - перелік аргументів.

У переліку аргументів метода-кріейтора мережі Петрі зручно передавати параметри, які можуть варіюватись при дослідженні моделі. Наприклад, тривалість обробки даних залежить від їх кількості, тоді часова затримка у відповідному переході залежить від кількості даних. У випадку, коли дані обробляються порціями за певних умов, то процес обробки деталізується і тоді кількість даних вказується у позиції як початкове маркірування.

Розробка методу-кріейтора мережі Петрі забезпечує можливість створювати будь-скільки фрагментів зі схожою поведінкою, що для конструювання моделей паралельних обчислень надзвичайно важливо. У пул потоків, наприклад, можуть бути завантажені десятки і сотні підзадач з однаковими діями, але на різних даних.

Код методу-кріейтора містить (у відповідності до формального представлення мережі Петрі, див. (2.1)):

- створення масивів позицій, переходів, вхідних дуг та вихідних дуг мережі Петрі,
- створення позицій мережі Петрі та додавання їх у масив позицій,
- створення переходів мережі Петрі та додавання їх у масив переходів,
- створення вхідних дуг мережі Петрі та додавання їх у вхідних дуг,
- створення вихідних дуг мережі Петрі та додавання їх у масив вихідних дуг,
- конструювання мережі Петрі зі створених елементів мережі Петрі.

При конструюванні мережі формуються множини вхідних та вихідних позицій для кожного переходу та перевіряється коректність побудови мережі Петрі.

Наприклад, найпростіший метод зі створення Петрі-об'єкта, що виконує одну обчислювальну дію з випадковою експоненціально розподіленою часовою затримкою з середнім значенням `timeMean`, має такий код:

```
public static PetriNet CreateNetRunnableSimple(double timeMean)
 throws ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> places = new ArrayList<>();
     ArrayList<PetriT> transitions = new ArrayList<>();
     ArrayList<ArcIn> arcsIn = new ArrayList<>();
     ArrayList<ArcOut> arcsOut = new ArrayList<>();
     places.add(new PetriP("runSt",0));
     places.add(new PetriP("runEnd",0));
     places.add(new PetriP("cores",0));
     transitions.add(new PetriT("run", timeMean,Double.MAX_VALUE));
     transitions.get(0).setDistribution("exp",
                              transitions.get(0).getTimeServ());
     arcsIn.add(new ArcIn(places.get(0),transitions.get(0),1));
     arcsIn.add(new ArcIn(places.get(2),transitions.get(0),1));
     arcsOut.add(new ArcOut(transitions.get(0),places.get(1),1));
     arcsOut.add(new ArcOut(transitions.get(0),places.get(2),1));
     PetriNet d_Net = new PetriNet("Runnable",
                                   places,
                                   transitions,
                                   arcsIn,
                                   arcsOut);
     return d_Net;
}
```

Петрі-об’єкт створюється конструктором суперкласу, який реалізує перетворення станів мережі Петрі (див. 2.2.3). У програмному забезпеченні цей клас названий PetriSim. Отже, Петрі-об’єкт створюється таким фрагментом коду:

```
PetriSim obj = new PetriSim(createNetName(args));
```

До прикладу, можна створити Петрі-об’єкт з описаним вище методом (в аргументі методу вказуємо чисельне значення часової затримки, необхідне на виконання інструкції run):

```
PetriSim obj = new PetriSim(CreateNetRunnableSimple(10.0));
```

Після створення об’єкт може використовуватись для зв’язування з іншими об’єктами та передаватись у список Петрі-об’єктів для конструювання моделі.

2.4.3 Розробка Петрі-об’єктної моделі паралельної програми

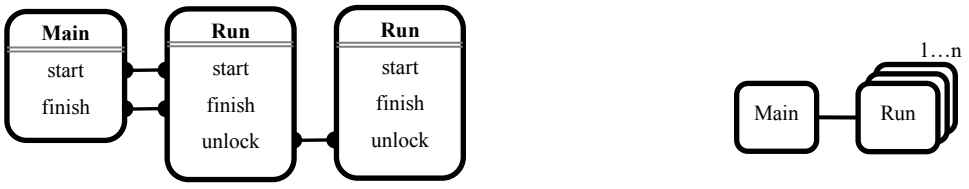
Розробка Петрі-об’єктної моделі включає такі кроки:

- створення списку Петрі-об’єктів,
- розробка конекторів Петрі-об’єктів,
- створення моделі викликом конструктора Петрі-об’єктної моделі, в аргументі якого вказаний список Петрі-об’єктів,
- розробка методу-кріейтора моделі.

Зв’язування об’єктів один з одним конекторами може бути виконано або до конструювання моделі, або після. Для Runnable об’єкту щонайменше його позиції "runSt" та "runEnd" ототожнюються з відповідними позиціями об’єкту, який виступає ініціатором виконання підзадачі.

Для представлення зв’язків між Петрі-об’єктами використовуємо діаграму ототожнень (рис.2.14, а) та діаграму побудови моделі (рис. 2.14, б). На діаграмі ототожнень кожен блок представляє Петрі-об’єкт з переліком позицій-конекторів (тобто позицій, які позначені як такі, що використовуються для зв’язування з іншими об’єктами). Зв’язки представляють ототожнення позицій Петрі-об’єкта з позиціями інших Петрі-об’єктів. На рисунку 2.14 позиції start та finish Петрі-об’єкта Run ототожнюються з позиціями start та finish Петрі-об’єкта

Main, а його позиція unlock ототожнюється з позицією unlock іншого Петрі-об’єкта Run. Таким чином, на діаграмі представлені попарні ототожнення позицій Петрі-об’єкта з позиціями інших Петрі-об’єктів. На діаграмі побудови моделі представлені усі Петрі-об’єкти, з яких складається модель. Якщо об’єкти утворюються тиражуванням, то такі об’єкти утворюють групу. Зв’язки між об’єктами на цій діаграмі представляються вже без деталізації ототожнень позицій Петрі-об’єктів. На рисунку 2.14 модель утворюється групою Петрі-об’єктів Run, що утворюються тиражуванням у заданій кількості n та одним Петрі-об’єктом Main.



(а) (б)

Рисунок 2.14– Діаграми Петрі-об’єктної моделі:

(а) діаграма ототожнень Петрі-об’єкта, (б) діаграма побудови моделі

Для створення моделі необхідно, щоб у списку об’єктів був хоча б один Петрі-об’єкт. Метод-кріейтор такої моделі матиме вигляд:

```
public static PetriObjModel createModel()
            throws ExceptionInvalidNetStructure,
                        ExceptionInvalidTimeDelay{
     ArrayList<PetriSim> list = new ArrayList<>();
     list.add(obj);
     return new PetriObjModel(list);
}
```

З’єднання Петрі-об’єктів один з одним виконується ототожненням позицій. Майже завжди необхідне з’єднання для позиції "cores". Якщо два Петрі-об’єкти obj та other мають позицію cores у своєму списку останньою, то ототожнення має вигляд:

```
obj.getNet().getListP()[obj.getNet().getListP().length] =
other.getNet().getListP()[other.getNet().getListP().length];
```

Проілюструємо розробку моделі на прикладі моделі простої програми, в якій main-метод створює два потоки, що виконують свої run-методи із заданими часовими затримками. Потрібно створити два Петрі-об'єкти та з'єднати їх з Петрі-об'єктом, що імітуватиме виконання дій main-методу. Скористаємось шаблоном мережі Петрі для створення, запуску та завершення потоку (див. рис. 2.5) і на його основі створимо мережу Петрі-об'єкта, що імітуватиме виконання Main-методу (рис. 2.15). Позицію cores та зв'язки з нею не представляємо, оскільки за замовчуванням вона є в усіх моделях та з'єднана з усіма переходами.

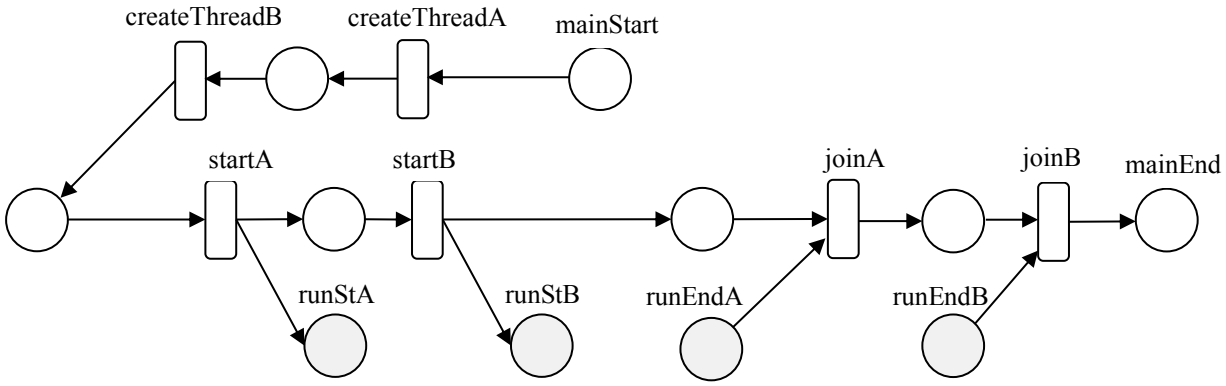


Рисунок 2.15– Мережа Петрі-об'єкта Main для простої моделі створення двох потоків.

Код відповідного метода-кріейтора мережі має вигляд:

```
public static PetriNet CreateNetMain(double delayCreate,
                                      double delayStart,
                                      double delayJoin)
 throws ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
    ArrayList<PetriP> d_P = new ArrayList<>();
    ArrayList<PetriT> d_T = new ArrayList<>();
    ArrayList<ArcIn> d_In = new ArrayList<>();
    ArrayList<ArcOut> d_Out = new ArrayList<>();
    d_P.add(new PetriP("mainStart",1));
    d_P.add(new PetriP("",0));
    d_P.add(new PetriP("",0));
    d_P.add(new PetriP("",0));
    d_P.add(new PetriP("",0));
    d_P.add(new PetriP("runStA",0));
    d_P.add(new PetriP("runStB",0));
    d_P.add(new PetriP("",0));
    d_P.add(new PetriP("mainEnd",0));
    d_P.add(new PetriP("runEndA",0));
    d_P.add(new PetriP("runEndB",0));
    d_T.add(new PetriT("createThreadA", delayCreate));
```

```
        d_T.add(new PetriT("createThreadB", delayCreate));
        d_T.add(new PetriT("startA", delayStart));
        d_T.add(new PetriT("startB", delayStart));
        d_T.add(new PetriT("joinA", delayJoin));
        d_T.add(new PetriT("joinB", delayJoin));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(7),d_T.get(5),1));
        d_In.add(new ArcIn(d_P.get(9),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(10),d_T.get(5),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(4),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(5),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(6),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(7),1));
        d_Out.add(new ArcOut(d_T.get(5),d_P.get(8),1));
        d_P.add(new PetriP("cores",2));
        for (PetriT transition : d_T) {
           transition.setDistribution("unif",
                                transition.getTimeServ());
           transition.setParamDeviation(transition.getTimeServ()/2);
        }
        for (PetriT transition : d_T) {
               d_In.add(new ArcIn(d_P.get(11), transition, 1));
               d_Out.add(new ArcOut(transition, d_P.get(11), 1));
        }
        PetriNet d_Net = new PetriNet("Main",d_P,d_T,d_In,d_Out);
        return d_Net;
}
```

Для створення Петрі-об’єктів, які виконують дії своїх run() методів з відомою часовою затримкою, підійде мережа Петрі, для якої у підрозділі 2.4.2 розроблено метод-кріейтор `CreateNetRunnableSimple()`.

Для того, щоб усі три створені Петрі-об’єкти відтворювали події у зв’язку з подіями в інших Петрі-об’єктах, потрібно встановити ототожнення позицій між парами об’єктів. Окрім позиції cores ототожнемими мають бути позиції runSt та runEnd об’єктів, що відтворюють потоки, та об’єкта Main. Діаграми для конструювання моделі представлена на рисунку 2.16.

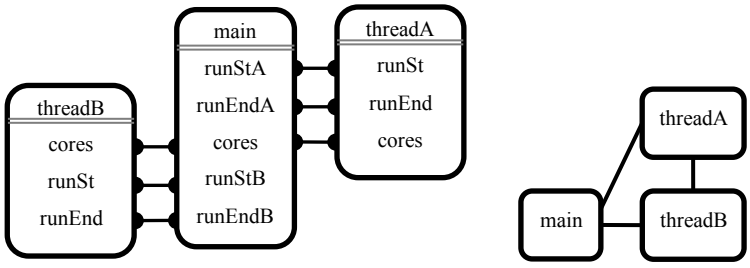


Рисунок 2.16 – Побудова простої Петрі-об’єктної моделі з трьох Петрі-об’єктів

Отже, виконуємо створення Петрі-об’єктів, з’єднуємо Петрі-об’єкти один з одним та додаємо їх у список Петрі-об’єктів. Маємо такий java-код методу-кріейтора моделі:

```
public static PetriObjModel createModel(double runA, double
runB)
            throws ExceptionInvalidNetStructure,
                          ExceptionInvalidTimeDelay{
     ArrayList<PetriSim> list = new ArrayList<>();
     PetriSim main = new PetriSim(CreateNetMain();
     PetriSim threadA =
                 new PetriSim(CreateNetRunnableSimple(runA));
     PetriSim threadB =
                 new PetriSim(CreateNetRunnableSimple(runB));
     threadA.getNet().getListP()[0] =
                              main.getNet().getListP()[5];
     threadB.getNet().getListP()[0] =
                              main.getNet().getListP()[6];
     threadA.getNet().getListP()[1] =
                              main.getNet().getListP()[9];
     threadB.getNet().getListP()[1] =
                              main.getNet().getListP()[10];
     threadA.getNet().getListP()[2] =
                              main.getNet().getListP()[11];
     threadB.getNet().getListP()[2] =
                              main.getNet().getListP()[11];
     list.add(main));
     list.add(threadA);
     list.add(threadB);
     return new PetriObjModel(list);
}
```

Таким чином, після розробки мереж Петрі-об’єктів розробник фокусує свою увагу на обміні інформацією про стан об’єктів та розробляє ототожнення позицій для кожної пари об’єктів. Якщо спосіб з’єднання об’єктів однаковий

(однакові ототожнення позицій), то він може бути винесений в окремий метод-конектор і застосовуватись для зазначених пар об’єктів. Структура багатопоточної програми точно відтворюється у моделі: кожному об’єкту в програмі відповідає Петрі-об’єкт моделі.

Зауважимо, що кодування методів-кріейторів без спеціалізованого програмного забезпечення є хоч і можливою, але непростою задачею через можливість зробити помилку при зв’язуванні елементів мережі Петрі. У розділі 4 буде розглянуто таке програмне забезпечення з можливістю автоматизованого створення методу-кріейтора за його графічним представленням.

2.5 Технологія моделювання паралельного алгоритму

2.5.1 Співставлення фрагмента Java-коду шаблонам моделювання паралельних обчислень

У підрозділі 2.4.1 розроблені такі шаблони моделювання паралельних обчислень мережею Петрі:

- циклу for (рис. 2.4, а),
- цикл while (рис. 2.4, б),
- оператор if-then-else (рис. 2.4, б),
- створення, початку та завершення роботи потоку (рис. 2.5),
- призупинки потоку (рис. 2.6),
- очікування на завершення виконання дій потоку (рис. 2.6),
- захоплення монітора об’єкта (рис. 2.7),
- негайне захоплення локера без та з урахуванням можливості багаторазового захоплення (рис. 2.8, рис. 2.9),
- очікування на умову (рис. 2.12),
- пул потоків (рис. 2.13).

Зазначені шаблони охоплюють усі основні дії та механізми взаємодії підзадач у багатопоточній програмі, включно з високорівневими інструментами.

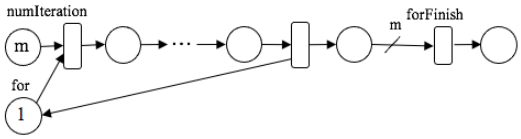
Розробка Петрі-об’єктної моделі починається з розбиття на Петрі-об’єкти. У випадку моделювання паралельної програми Петрі-об’єкти визначаємо у відповідності до структурних елементів паралельної програми, тобто її процесам, потокам та підзадачам. Отже, у випадку Java-коду усі об’єкти типу Process, Runnable, Callable, ForkJoinTask обов’язково підлягають моделюванню окремими Петрі-об’єктами. Інші об’єкти програми представляються Петрі-об’єктами за потреби.

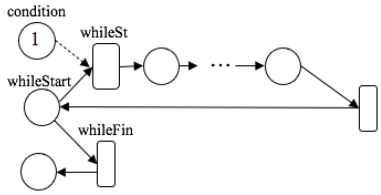
Наступним кроком побудови Петрі-об’єктної моделі є розробка мереж Петрі об’єктів.

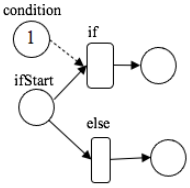
Наведемо співставлення фрагментів Java-коду та розроблених фрагментів мережі Петрі у вигляді таблиці 2.1.

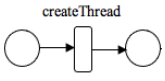
Таблиця 2.1 - Співставлення фрагментів Java-коду шаблонам моделювання стохастичною мережею Петрі з багатоканальними переходами (усі переходи фрагментів мереж Петрі за замовчуванням мають зв’язки з позицією cores, що відтворюють захоплення обчислювального ресурсу).

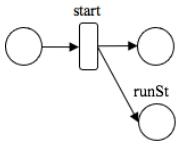
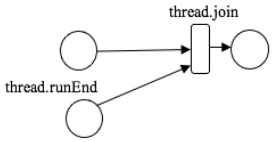
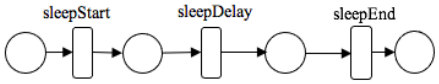
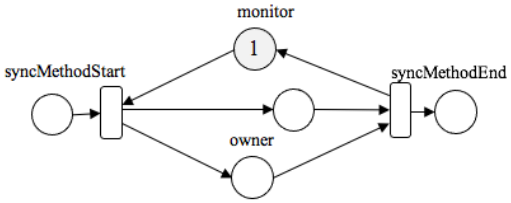
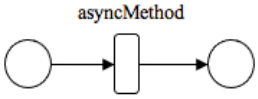
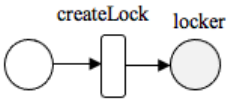
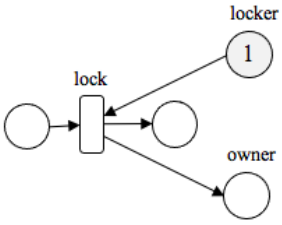
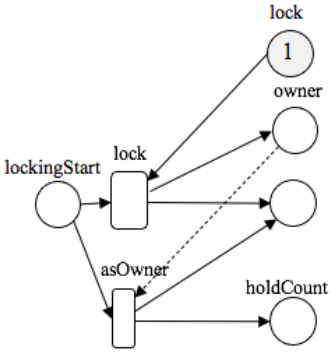
Фрагмент Java-коду	Фрагмент мережі Петрі

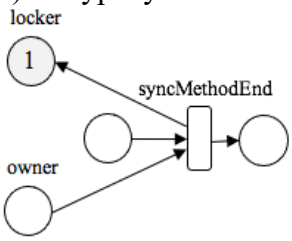
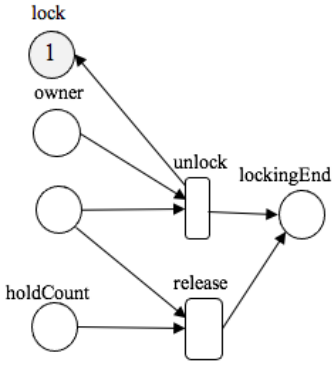
`for (int j=0; j<m; j++){`
`  …// елементарні дії`
`}`
	

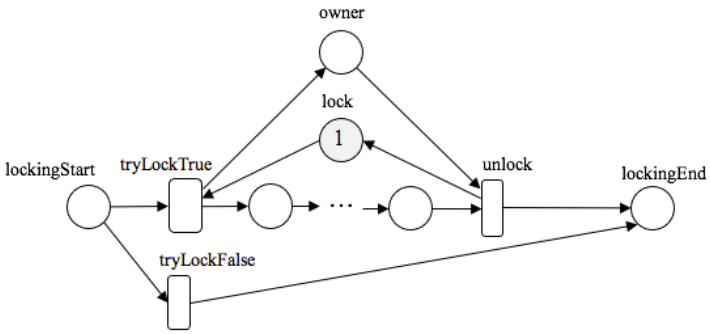
`while (condition){`
`  …// елементарні дії`
`}`
	

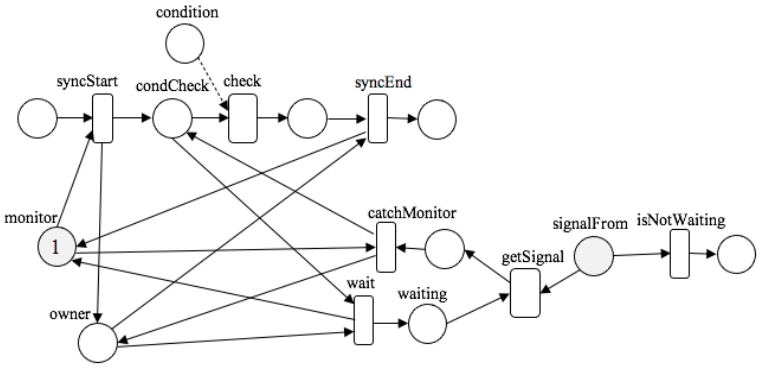
`if (condition){`
`  …// елементарні дії`
`}else{`
`  …// елементарні дії`
`}`
	

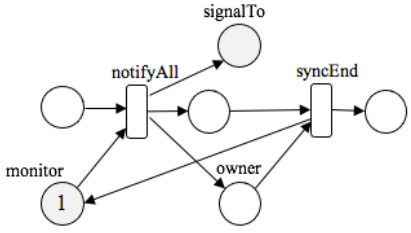
`new Thread(()->{`
`…// елементарні дії`
`});`
	

Фрагмент Java-коду	Фрагмент мережі Петрі
`thread.start();`	
`thread.join();`	
`Thread.sleep(delay);`	 для переходу sleepDelay не встановлюються зв’язки з позицією cores
`obj.syncMethod();`	
`obj.asyncMethod();`	
`Lock lock = new ReentrantLock();`	
`lock.lock();`	а) без урахування багаторазового захоплення  б) з урахуванням багаторазового захоплення 

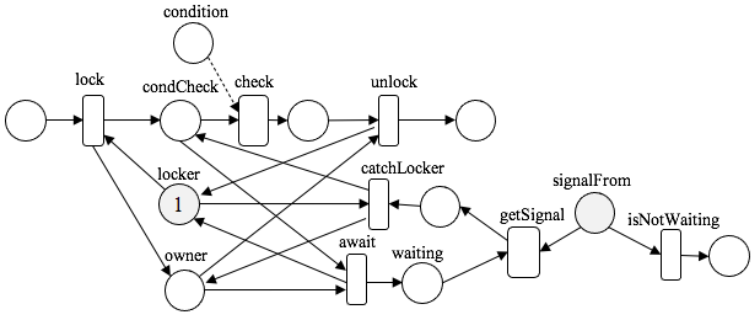
Фрагмент Java-коду	Фрагмент мережі Петрі
`lock.unlock();`	а) без урахування багаторазового захоплення  б) з урахуванням багаторазового захоплення 

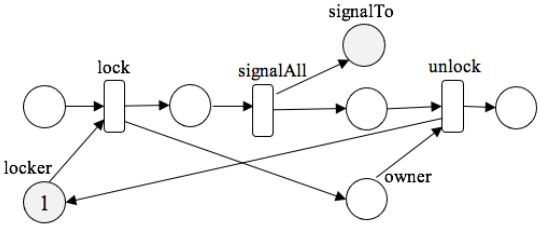
`try{`
`  if (lock.tryLock()){`
`  …// елементарні дії`
`  }`
`} finally {`
`  lock.unlock();`
`}`
	

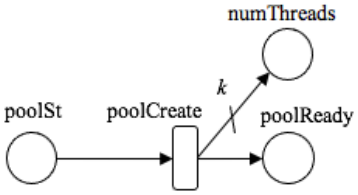
`public synchronized void method() throws InterruptedException{`
`while (!condition()) {`
`  wait();`
`  }`
`}`
	

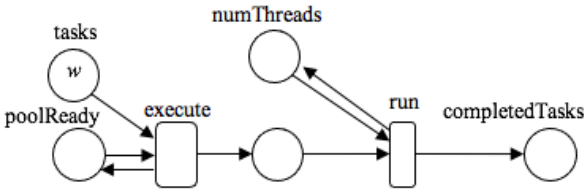
`public synchronized void method() throws InterruptedException{`
`  notifyAll();`
`}`
	

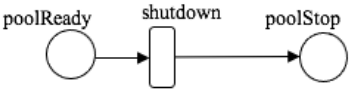
Фрагмент Java-коду	Фрагмент мережі Петрі

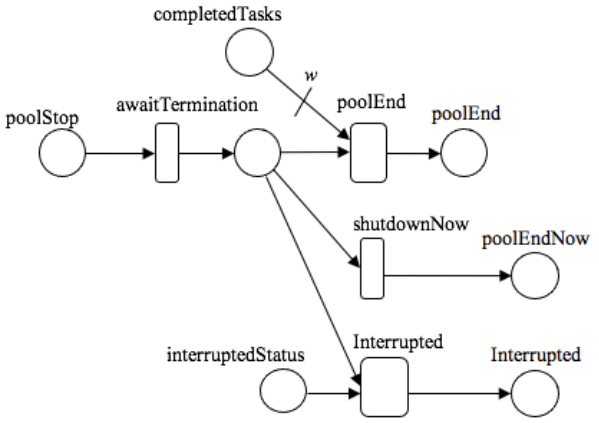
`lock.lock();`
`try {`
`  while (!condition) {`
`  objCond.await();`
`  }`
`}finally {`
`  lock.unlock();`
`}`
	

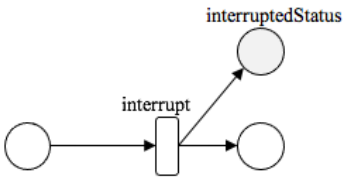
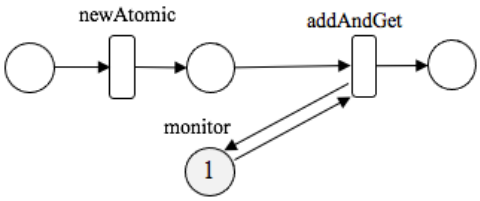
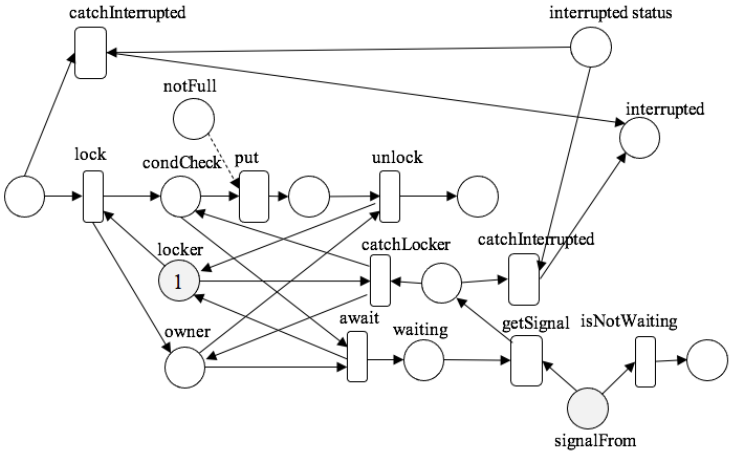
`lock.lock();`
`try {`
`condition.signalAll();`
`}finally {`
`  lock.unlock();`
`}`
	

`ExecutorService pool =`
`Executors.`
`newFixedThreadPool(k);`
	

`pool.execute(`
`  new Runnable(){`
`  @Override`
`  public void run() {`
`  …// елементарні дії`
`  }`
`});`
	

`pool.shutdown();`	

`try {`
`if(!pool.`
`awaitTermination(10,`
`TimeUnit.MINUTES)){`
`pool.shutdownNow();`
`}`
`} catch`
`(InterruptedException e)`
`{`
`…// повідомлення про помилку`
`}`
	

Фрагмент Java-коду	Фрагмент мережі Петрі
`thread.interrupt();`	
`AtomicInteger counter = new AtomicInteger(); counter.addAndGet(v);`	
`blockingQueue. put(produceObj());`	

Зауважимо, що часові затримки на виконання обчислювальних дій, які вказуються в переходах мережі Петрі, мають бути встановлені за результатами дослідження фактичних затримок, що мають місце для заданої операційної системи. У деяких випадках розробки моделі достатньо оцінювання часових затримок в умовних одиницях складності обчислювальних дій.

Отже, за кодом паралельної програми може бути складена модель. Як завжди при розробці моделі, доводиться знаходити компроміс між складністю моделі та її точністю. В залежності від контексту задачі потрібно приймати рішення про деталізоване представлення дій, які виконуються в програмі. Наприклад, кілька рутинних дій (без взаємодії з іншими потоками чи підзадачами) можуть бути замінені на одну дію і відтворюватись одним

переходом мережі Петрі. Саме через це автоматизована розробка моделі за програмним кодом є нетривіальною і досі не вирішеною задачею.

2.5.2 Приклад розробки моделі, що використовує tryLock() для вирішення проблеми взаємного блокування потоків

Офіційний туторіал Java Concurrency від компанії Oracle [58] розглядає проблему взаємоблокування потоків (deadlock) на прикладі взаємодії потоків, що відтворюють вітання друзів один з одним. Вітання потік може розпочати, як тільки він не зайнятий вітанням. Вітання, що успішно почалось, може завершись тільки за умови отримання відповіді від друга, з яким вітаються. Виклик синхронізованого методу з синхронізованого призводить до взаємного блокування моніторів двох друзів, що намагаються привітати один одного, і програма не може продовжити виконання.

Для вирішення конфлікту потоків, що може виникнути, застосовується механізм блокування потоків локерами. Модель, яка демонструє цей процес, створена на основі програмного коду, наведеного у [59]. Суть алгоритму наступна: коли один з друзів (об’єкт класу Friend) намагається привітатись з іншим, який вже зайнятий вітанням з іншим своїм другом(виконує метод bow в своєму потоці, який блокує інший об’єкт класу Friend), він веде рахунок невдалих спроб вітання (невдачі у спробах захопити об’єкт "lock", коли потік знаходиться у стані очікування). Таким чином відбувається фіксація конфліктів потоків та вдалих спроб взаємодії потоків.

Спершу на основі коду була побудована мережа Петрі, що відтворює роботу об’єкта класу Friend, вона подана на рисунку 2.17. Позиція lock відтворює стан локера об’єкта класу Friend. Позиція lockOther відтворює стан локера, який намагається захопити об’єкт класу Friend. Тестування Петрі-об’єкта виконується при різних маркіруваннях позицій lock та lockOther. Якщо обидва локери зайняті (стан маркірування: 0), то кількість вдалих запусків методу bow рівна 0. Якщо обидва локери вільні (стан маркірування: 1), то кількість вдалих

запусків методу bow співпадає з початковим маркірування позиції bowLoop. Якщо локери перебувають у стані lock=1, lockOther=0, то кількість вдалих запусків методу bow рівна 0. Якщо локери перебувають у стані lock=0, lockOther=1, то кількість вдалих запусків методу bow також рівна 0. Отже, мережа Петрі правильно відтворює механізм блокування потоків. Позиції lock та lockOther відмічені сірим кольором, оскільки вони використовуються для зв'язування мережі з мережею іншого Петрі-об'єкта. Сірий колір позначає, що стан позиції залежить не тільки від стану елементів мережі Петрі цього об'єкта, але і від стану мереж Петрі, з яким він буде зв'язаний.

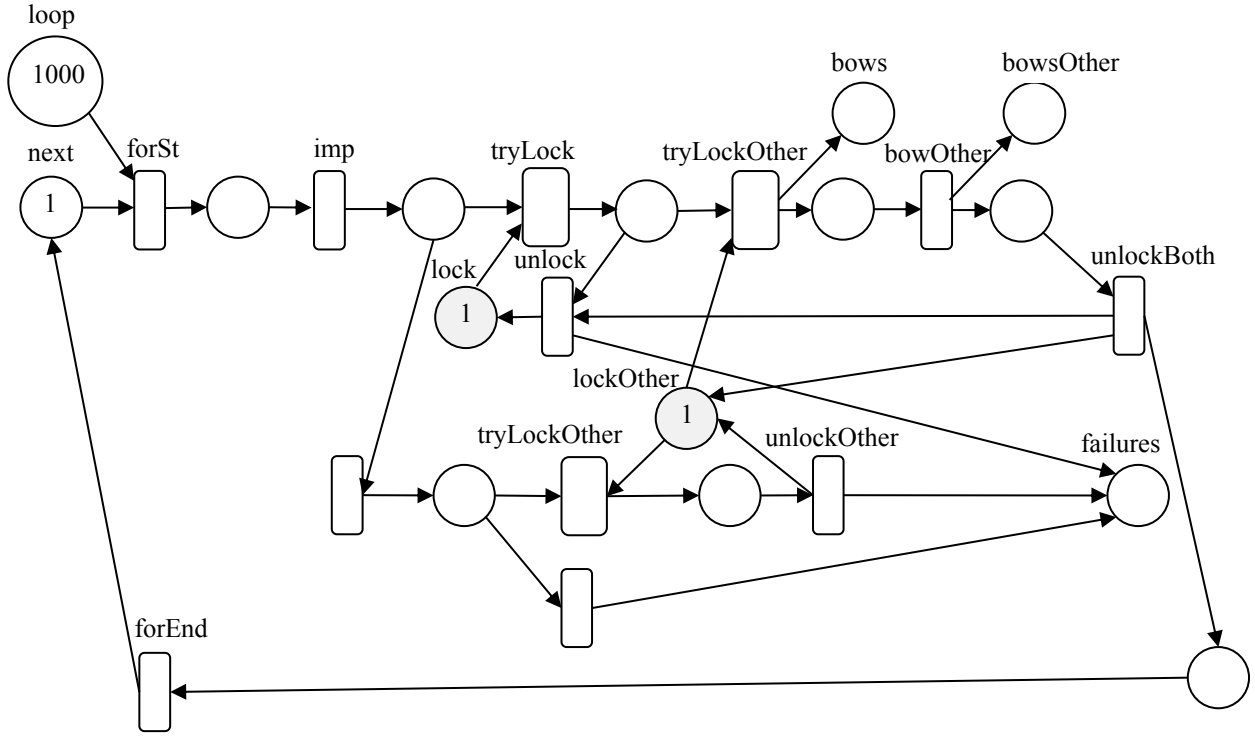


Рисунок 2.17 – Мережа Петрі-об`єкта "Friend".

Після побудови ця мережа має бути збережена як метод `CreateNetFriend(double sleepDelay)` з параметром, що задає часову затримку між повторюваними в циклі вітаннями. На її основі створено клас `Friend` для зручності тиражування об'єктів з заданими властивостями. Конструктор класу `Friend (String name, double delay)` створює мережу Петрі для нового об'єкту викликом методу `CreateNetFriend(double delay):`

```
public Friend(String name,double delay)
 throws ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        super(NetLibrary.CreateNetFriend(delay));
}
```

Таким чином об’єкт створюється з заданою поведінкою та заданим значенням параметра, який може варіюватись. Для зв’язування з іншими об’єктами у класі створено метод `addFriend(Friend other)`, в якому ототожнюються пари локерів:

```
public void addFriend(Friend other) {
     this.getLockOther() = other.getLock();
     this.getCores() = other.getCores();
}
```

Кожний з get-методів у наведеному фрагменті коду повертає відповідну позицію мережі Петрі для ототожнення:

```
public PetriP getLock(){
return this.getNet().getListP()[15];
}
public PetriP getLockOther(){
return this.getNet().getListP()[7];
}
```

Петрі-об’єктна модель складається з кількох Петрі-об’єктів класу Friend. Наприклад, якщо модель містить 4 об’єкти класу Friend `a`, `b`, `c`, `d`, то зв’язки між об’єктами утворюються попарно:

```
a.addFriend(b); b.addFriend(a);
a.addFriend(c); c.addFriend(a);
a.addFriend(d); d.addFriend(a);
b.addFriend(c); c.addFriend(b);
c.addFriend(d); d.addFriend(c);
```

У класі Friend для проведення тестування та дослідження моделі зручно створити метод getResult(), що повертає частоту невдалих спроб:

```
public double getResult() {
     double failures =
          (double) this.getNet().getListP()[6].getMark();
     double bows =
          (double) this.getNet().getListP()[8].getMark();
    return failures / (failures + bows);
}
```

Для загального випадку конструювання моделі з *n* об’єктів потрібно тиражувати Петрі-об’єкти класу Friend у кількості *n* і зв’язати їх. Діаграма зв’язків представлена на рисунку 2.18.

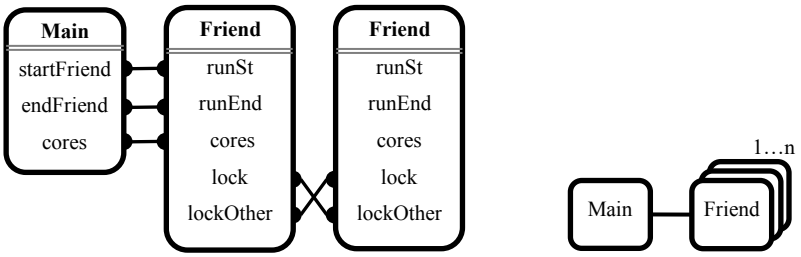


Рисунок 2.18– Побудова Петрі-об’єктної моделі потоків із взаємним захопленням локерів.

Оскільки модель будується для дослідження конфліктів потоків при доступі до локерів (не для дослідження часу виконання програми), то об’єкт Main можна не враховувати, оскільки він впливає на роботу програми тільки на її початку та наприкінці (рис. 2.19). Ототожнення позиції "cores" з позицією "cores" об’єкта Main зберігаємо, додаючи ототожнення з позицією "cores" іншого об’єкта Friend.

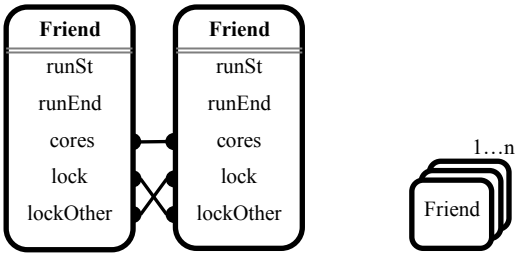


Рисунок 2.19 – Побудова спрощеної Петрі-об’єктної моделі потоків із взаємним захопленням локерів

Метод-кріейтор моделі має такий код:

```
public static PetriObjModel createModel(int n, double d)
                    throws ExceptionInvalidNetStructure,
                                  ExceptionInvalidTimeDelay{
     ArrayList<PetriSim> list = new ArrayList<>();
     Friend[] friends = new Friend[n];
     for(int j=0;j<n;j++){
          friends[j] = new Friend ("Friend_"+j, d);
     }
     for(int j=0;j<n;j++){
          for(int i=0;i<n;i++)
```

```
                if(i!=j) friends[j].addFriend(friends[i]);
        }
        for(int j=0;j<n;j++){
                    list.add(friends[j]);
        }
        return new PetriObjModel(list);
}
```

Програмний код Петрі-об’єктної моделі наведений у додатку Б. Експериментальне дослідження моделі проводилось при кількості ядер 2 (початкове маркірування позиції cores). Для запуску моделі на імітацію в заданому інтервалі часу timeMod використовуємо метод `void go(double time)`:

```
PetriObjModel model = createModel(int 4, double 100);
model.go(1000000);
```

Кількість вдалих та невдалих спроб вітання реєструється. Результати дослідження показують, що ймовірність виникнення конфлікту потоків сильно залежить від часових затримок. На моделях, що містили відповідно два та чотири Петрі-об’єкти Friend були проведені експерименти з метою дослідження впливу часових затримок переходів. Кількість обчислювального ресурсу в усіх експериментах була незмінною і дорівнювала 2 (початкове маркірування позиції ‘cores’). Часовій затримці переходу ‘forSt’ було надано значення d, а іншим переходам – значення $d \cdot r$, де r – співвідношення часових затримок переходів. Невдала спроба стається у випадку якщо один об’єкт класу Friend не може захопити об’єкт ‘lockOther’ іншого об’єкта класу Friend. Тобто значення відносної частоти появи невдалої спроби f характеризує частоту виникнення конфліктів потоків. Результати експерименту, які свідчать про сильну залежність значень f та r, зображені на рисунку 2.20. Справді, випадки, коли часова затримка в переході ‘forSt’ є значно більшою за затримки в інших переходах мережі, відображають ситуацію, коли час, витрачений на дії, які виконуються потоками, значно менше, ніж інтервал між ними. Тож, ймовірність одночасної дії потоків зменшується.

На графіку видно, що для всіх трьох випадків (*d*=100, *d* =10, *d*=1), ймовірність виникнення конфлікту потоків була мінімальною, коли значення співвідношення часових затримок переходів було мінімальним (часова затримка в переході ‘forSt’ значно перевищувала часові затримки в інших переходах).

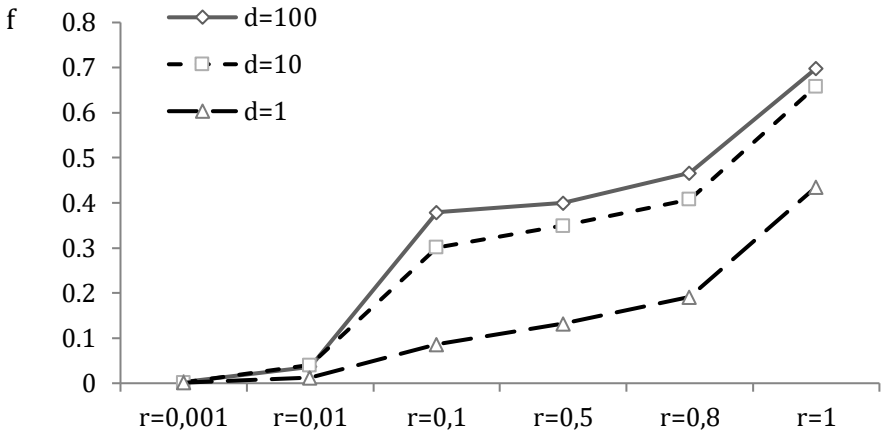


Рисунок 2.20 – Залежність виникнення конфлікту переходів від значень часових затримок

Порівняння відносної частоти вдалої спроби (1-*f*), визначеної за результатами моделювання, з результатами, отриманими при виконанні програми, наведено у таблиці 2.2. З цієї таблиці можна бачити, що значення похибки менше 0,5%, що свідчить про високу точність побудованої моделі. Точність визначення невдалої спроби (*f*) оцінюється значеннями похибки значно більшими – 13% та 12% відповідно для 2 та 4 потоків, що також є прийнятним результатом. Дослідження моделі захоплення локерів опубліковано у роботах [51, 52].

Таблиця 2.2 – Оцінювання точності визначення відносної частоти вдалої спроби за результатами моделювання

Кількість потоків	Частота вдалої спроби за результатами виконання паралельної програми	Частота вдалої спроби за результатами моделювання (d=100, r=0,01)	Похибка результату моделювання
2	0,981450	0,978650	0,29%
4	0,959042	0,963417	0,46%

2.5.3 Приклад розробки моделі пулу потоків

Проілюструємо тиражування об’єктів та використання шаблону моделювання пулу потоків на прикладі розробки моделі багатопотокової програми, в якій запускаються Runnable-задачі на виконання в пул потоків. Задачі виконують однаковий код асинхронно для різних даних. Для відтворення їх методу `run()` можемо скористатись одним переходом з часовою затримкою, що залежить від кількості виконуваних обчислювальних дій. Отже, для створення відповідних об’єктів скористаємось методом-кріейтором `PetriNet CreateNetRunnableSimple (double timeMean)` (див. підрозділ 2.4.2).

Створимо метод-кріейтор мережі Петрі-об’єкта, що відтворює функціонування пулу потоків (див. рис. 2.13) з аргументами, які вказують на кількість задач (`w`), кількість потоків (`k`) та часові затримки у переходах:

```
public static PetriNet CreateNetFixedPool(int w, int k,
                                           double delayCreate,
                                           double delayExecute,
                                           double   delayAwait)
throws ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("poolSt", 0));
        d_P.add(new PetriP("", 0));
        d_P.add(new PetriP("tasks", w));
        d_P.add(new PetriP("numTasks", 0));
```

```
        d_P.add(new PetriP("", 0));
        d_P.add(new PetriP("", 0));
        d_P.add(new PetriP("", 0));
        d_P.add(new PetriP("runSt", 0));
        d_P.add(new PetriP("runEnd", 0));
        d_P.add(new PetriP("poolEnd", 0));
        d_P.add(new PetriP("cores",0));
        d_T.add(new PetriT("poolCreate", delayCreate));
        d_T.get(0).setDistribution("unif",
                                        d_T.get(0).getTimeServ());
        d_T.get(0).setParamDeviation(delayCreate/2);
        d_T.add(new PetriT("execute", delayExecute));
        d_T.get(1).setDistribution("unif",
                                    d_T.get(1).getTimeServ());
        d_T.get(1).setParamDeviation(delayExecute/2);
        d_T.get(1).setPriority(6);
        d_T.add(new PetriT("runSt", 0.0));
        d_T.add(new PetriT("shutdown", 0.0));
        d_T.add(new PetriT("runEnd", 0.0));
        d_T.add(new PetriT("awaitTermination", delayAwait));
        d_T.get(5).setDistribution("unif",
                                    d_T.get(5).getTimeServ());
        d_T.get(5).setParamDeviation(delayAwait/2);
        d_In.add(new ArcIn(d_P.get(0), d_T.get(0), 1));
        d_In.add(new ArcIn(d_P.get(1), d_T.get(1), 1));
        d_In.add(new ArcIn(d_P.get(2), d_T.get(1), 1));
        d_In.add(new ArcIn(d_P.get(4), d_T.get(2), 1));
        d_In.add(new ArcIn(d_P.get(1), d_T.get(3), 1));
        d_In.add(new ArcIn(d_P.get(3), d_T.get(2), 1));
        d_In.add(new ArcIn(d_P.get(8), d_T.get(4), 1));
        d_In.add(new ArcIn(d_P.get(6), d_T.get(5), w));
        d_In.add(new ArcIn(d_P.get(5), d_T.get(5), 1));
        d_Out.add(new ArcOut(d_T.get(0), d_P.get(1), 1));
        d_Out.add(new ArcOut(d_T.get(1), d_P.get(1), 1));
        d_Out.add(new ArcOut(d_T.get(0), d_P.get(3), k));
        d_Out.add(new ArcOut(d_T.get(1), d_P.get(4), 1));
        d_Out.add(new ArcOut(d_T.get(2), d_P.get(7), 1));
        d_Out.add(new ArcOut(d_T.get(4), d_P.get(6), 1));
        d_Out.add(new ArcOut(d_T.get(5), d_P.get(9), 1));
        d_Out.add(new ArcOut(d_T.get(3), d_P.get(5), 1));
        d_Out.add(new ArcOut(d_T.get(4), d_P.get(3), 1));
        for (PetriT tr: d_T){
            d_In.add(new ArcIn(d_P.get(d_P.size()-1),tr,1));
            d_Out.add(new ArcOut(tr, d_P.get(d_P.size()-1),1));
        }
        PetriNet d_Net =
               new PetriNet("FixedPool", d_P, d_T, d_In, d_Out);
        return d_Net;
    }
```

Далі створимо метод-кріейтор для мережі Петрі-об’єкта, що моделює роботу класу `Main`. Як аргумент вказується кількість обчислювальних ресурсів (ядер процесора):

```
public static PetriNet CreateNetMain(int m)
throws ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("mainStart",1));
     d_P.add(new PetriP("poolStart",0));
     d_P.add(new PetriP("poolEnd",0));
     d_P.add(new PetriP("mainEnd",0));
     d_P.add(new PetriP("cores",m));
     d_T.add(new PetriT("start",0.1));
     d_T.add(new PetriT("finish",0.1));
     d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(2),d_T.get(1),1));
     d_In.add(new ArcIn(d_P.get(4),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(4),d_T.get(1),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(4),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(4),1));
     PetriNet d_Net = new PetriNet("Main",d_P,d_T,d_In,d_Out);
     return d_Net;
}
```

У методі-кріейторі моделі створюємо список Петрі-об’єктів з одного об’єкту main, одного об’єкту pool та w об’єктів task. В аргументи методу передамо кількість обчислювальних ресурсів, кількість потоків в пулі, кількість задач, що будуть оброблені, та часові затримки, які будуть змінюватись при тестуванні та експериментуванні з моделлю. Об’єкти task створюємо за допомогою метода-кріейтора `PetriNet CreateNetRunnableSimple (double timeMean)` та з’єднуємо їх з пулом через спільні позиції (рис. 2.21).

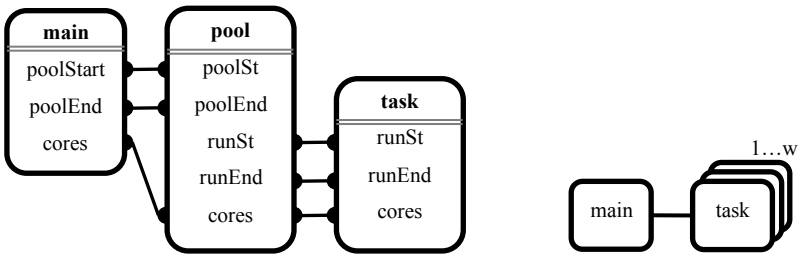


Рисунок 2.21 – Побудова Петрі-об’єктної моделі пулу потоків.

Отримуємо такий код методу-кріейтора моделі:

```
public static PetriObjModel createModel(int m, int w, int k,
                                        double delayCreate,
                                        double delayExecute,
                                        double delayAwait,
                                        double delayRun)
    throws ExceptionInvalidNetStructure,ExceptionInvalidTimeDelay{
   ArrayList<PetriSim> list = new ArrayList<>();
   PetriSim main = new PetriSim(NetLibrary.CreateNetMain(m));
   PetriSim pool = new PetriSim(NetLibrary.CreateNetFixedPool(w, k,
                           delayCreate, delayExecute, delayAwait);
   PetriSim[] tasks = new PetriSim[w];
   for (int j=0; j<w; j++){
     tasks[j]              =              new              PetriSim(
     NetLibrary.CreateNetRunnableSimple(delayRun));
     tasks[j].getNet().getListP()[0] = pool.getNet().getListP()[7];
     tasks[j].getNet().getListP()[1] = pool.getNet().getListP()[8];
     tasks[j].getNet().getListP()[2] = main.getNet().getListP()[4];
   }
   pool.getNet().getListP()[0] = main.getNet().getListP()[1];
   pool.getNet().getListP()[9] = main.getNet().getListP()[2];
   pool.getNet().getListP()[10] = main.getNet().getListP()[4];
   list.add(main);
   list.add(pool);
   for(int j=0;j<w; j++){
     list.add(tasks[j]);
   }
   return new PetriObjModel(list);
}
```

Для запуску моделі на імітацію в заданому інтервалі часу використовуємо метод `void go(double time)`:

```
PetriObjModel model= createModel(2, 2000, 2, 140000, 140000,
0.1);
model.go(1000000);
```

Часові затримки для подій в моделі визначалися шляхом статистичного аналізу відповідних значень під час реальних обчислень на двоядерному

процесорі. Є програмні інструкції, виконання яких вимагає значно більше часу, ніж інші інструкції. Наприклад, час створення пулу потоків є близько в 20 разів більшим, ніж час створення нового об’єкта, тоді як останній у свою чергу більший за час найпростішої арифметичної операції в 30000 разів.

З допомогою розробленої моделі було проведено експериментальне дослідження ефективності пулу потоків. Варіюючи обчислювальну складність паралельного алгоритму від 10^3 до 10^9, різке збільшення прискорення (приблизно до 2) спостерігається лише при складності 10^8 (рис. 2.22). При складності алгоритму менше за 10^6, робота пулу потоків є не ефективною, оскільки прискорення відсутнє (менше 1). Порівнюючи результати моделювання та реального запуску програми, можна зробити висновок про правильність роботи моделі (таблиця 2.3). Слід відмітити, що модель виявила обмеженість ресурсів: максимальне прискорення сягає позначки 2, що відповідає кількості ядер процесора, на якому проводились експерименти. Таблиця 2.4 містить результати пошуку параметрів пулу потоків для досягнення максимального прискорення.

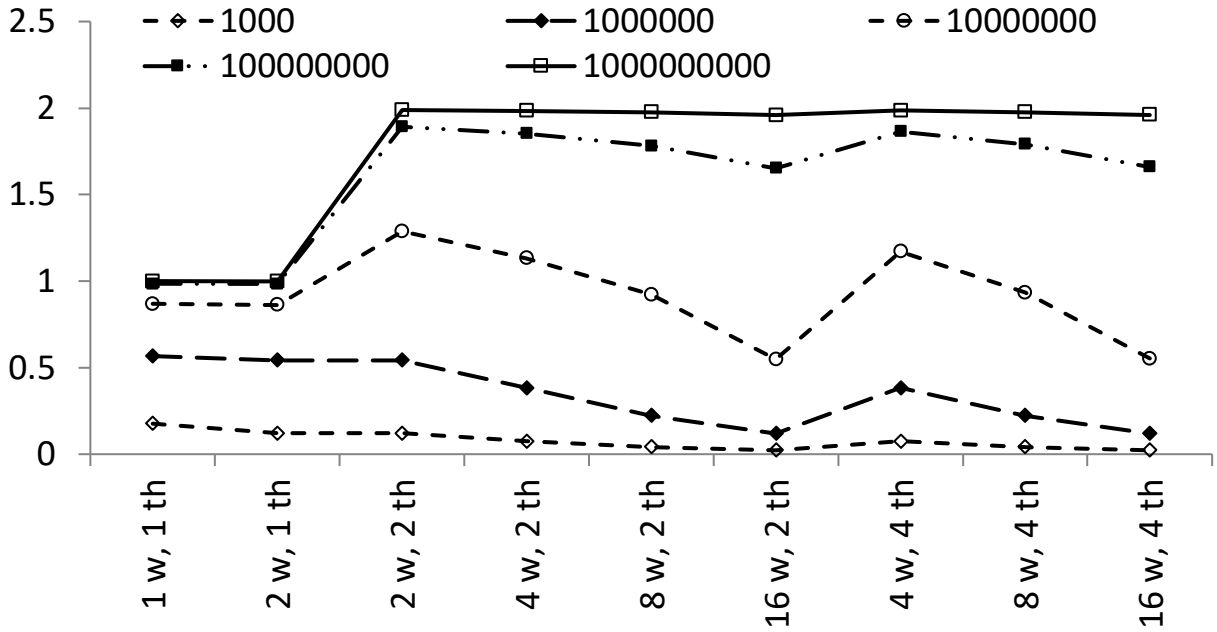


Рисунок 2.22 – Залежність прискорення від параметрів пулу потоків (th – потоки, w - завдання)

Таблиця 2.3 – Точність результатів імітації (складність обчислень – 10^9)

Кількість завдань (w) та потоків (th)	1 w, 1 th	2 w, 1 th	2 w, 2 th	4 w, 2 th	8 w, 2 th	16 w, 2 th	4 w, 4 th	8 w, 4 th	16 w, 4 th
Програма	0,985	0,990	1,859	1,869	1,901	1,856	1,907	1,828	1,855
Модель	0,998	0,998	1,989	1,984	1,976	1,977	1,985	1,977	1,960
Похибка	1%	1%	7%	6%	4%	7%	4%	8%	6%

У таблиці 2.4 наведена точність моделювання прискорення. Значення точності розраховується як різниця між прискоренням, що отримане в моделі, і в реальній багатопотоковій програмі. Отримана різниця виражена у відсотках від значення прискорення у реальній програмі.

Таблиця 2.4 – Параметри пулу потоків (th – потоки, w – завдання) для досягнення максимального прискорення

Параметри / Складність	1 w, 1 th	2 w, 1 th	2 w, 2 th	4 w, 2 th	8 w, 2 th	16 w, 2 th	4 w, 4 th	8 w, 4 th	16 w, 4 th
1000	**0,18**	0,12	0,12	0,07	0,04	0,02	0,07	0,04	0,02
1000000	**0,57**	0,54	0,54	0,38	0,22	0,12	0,38	0,22	0,12
10000000	0,87	0,86	**1,29**	1,13	0,92	0,55	1,17	0,93	0,55
100000000	0,98	0,98	**1,89**	1,85	1,78	1,65	1,86	1,79	1,66
1000000000	1,00	1,00	**1,99**	1,98	1,98	1,96	1,99	1,98	1,96

Результати дослідження ефективності пулу потоків на розробленій моделі детально викладені у статті [8].

2.5.4 Етапи технології моделювання паралельних обчислень

Таким чином, на основі наведених прикладів можна сформулювати основні етапи технології моделювання паралельних обчислень.

1. Визначення структури моделі, а саме, множини Петрі-об’єктів та зв’язків між ними.

На цьому етапі потрібно виокремити в алгоритмі об’єкти, вирішити, які з них потребують детального опису функціонування, та поставити у відповідність їм Петрі-об’єкти. Виділити серед Петрі-об’єктів групи зі схожою зміною станів. Якщо об’єкти в алгоритмі обмінюються інформацією про свій стан, то вказати наявність зв’язку між відповідними ними.

2. Розробка мереж Петрі.

На цьому етапі потрібно побудувати мережі груп Петрі-об’єктів у відповідності до функціонування об’єктів програми, що моделюється, та зберегти їх як методи-кріейтори мереж. Розроблені мережі Петрі перевіряються на коректність запуском імітації при різних можливих початкових маркіруваннях та різних часових затримках. Результатом запуску є протокол подій, в якому фіксується зміна стану мережі Петрі та момент часу, в який відбулась зміна. Якщо мережа правильно реагує на змінювання параметрів, то вважається, що вона є коректною. Мережа Петрі, що успішно пройшла перевірку на коректність, використовується для створення метода-кріейтора мережі Петрі-об’єкта. Отже, результатом цього етапу є методи-кріейтори Петрі-об’єктів.

3. Створення списку Петрі-об’єктів.

На цьому етапі потрібно створити список Петрі-об’єктів. Для груп об’єктів, що мають схожу функціональність, використовується тиражування, тобто створення множини Петрі-об’єктів за однаковим зразком мережі Петрі, що постачається (продукується) методом-кріейтором.

4. Встановлення зв’язків між парами Петрі-об’єктами.

На цьому етапі потрібно встановити зв’язки між Петрі-об’єктами. Для цього з’єднати Петрі-об’єкти між собою за допомогою ототожнення позицій та/або переходів. Якщо для груп об’єктів використовуються схожі ототожнення, то розробити методи, які створюють зв’язки між об’єктами за зразком.

5. Конструювання Петрі-об’єктної моделі.

На цьому етапі потрібно створити Петрі-об’єктну модель на основі списку Петрі-об’єктів. Побудовану модель потрібно перевірити на коректність запуском імітації при різних значеннях параметрів. Якщо модель правильно реагує на змінювання параметрів, то вважається, що вона є коректною. Модель, що успішно пройшла перевірку на коректність, використовується для створення метода-кріейтора моделі.

6. Експериментальне дослідження часових затримок.

Для встановлення параметрів часових затримок переходів виконати експериментальне дослідження затримок на виконання обчислювальних дій у реальних умовах. Важливо встановити співвідношення між тривалостями затримок, що витрачаються на виконання основних дій в моделі, та відтворити їх в моделі.

7. Верифікація моделі.

Виконати верифікацію моделі запуском імітації при різних значеннях параметрів. Зіставити протокол подій, отриманий за результатом запуску імітації з перебігом подій в реальній програмі.

8. Валідація моделі.

Виконати валідацію моделі зіставленням результату, отриманого на моделі та в реальній паралельній програмі.

У підсумку, технологія Петрі-об’єктного моделювання паралельної програми має такі переваги:

- використання формалізованого опису у вигляді Петрі-об’єктної моделі дає змогу найбільш точно відтворити структуру об’єктно-орієнтованої програми та поведінкові властивості окремих об’єктів;
- використання стохастичної мережі Петрі дає змогу відтворювати більш точно взаємодію потоків/підзадач за рахунок враховування часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу;

- використання типових фрагментів мереж Петрі (шаблонів) для моделювання основних інструментів паралельного програмування спрощує та прискорює розробку моделі;
- тиражування об’єктів зі схожою поведінкою із заданими параметрами спрощує створення моделей з великою кількістю підзадач;
- тиражування груп зв’язків між об’єктами спрощує конструювання моделі з великою кількістю зв’язків;
- візуалізація поведінкових властивостей моделі дає змогу розробити програму з найменшою кількістю помилок.

2.6 Висновки до розділу 2

У розділі сформульовані мета та задача моделювання паралельних обчислень, визначено формальний опис моделі та виконано розробку методу моделювання паралельних обчислень на основі цього опису. Для формального опису обрано формалізм Петрі-об’єктної моделі у зв’язку з низкою переваг:

- найбільш точне (у порівнянні зі звичайною мережею Петрі) відтворення структури об’єктно-орієнтованої програми та поведінкових властивостей окремих об’єктів;
- більш точне (у порівнянні зі звичайною мережею Петрі) відтворення взаємодії потоків/підзадач за рахунок враховування часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу;
- тиражування об’єктів зі схожою поведінкою із заданими параметрами спрощує створення моделей з великою кількістю підзадач;
- тиражування груп зв’язків між об’єктами спрощує конструювання моделі з великою кількістю зв’язків;
- візуалізація поведінкових властивостей моделі дає змогу розробити програму з найменшою кількістю помилок.

Розроблені шаблони моделювання обчислень багатопотокової програми для зменшення кількості помилок при розробці моделей та прискорення процесу розробки. Набір шаблонів моделювання містить фрагменти мереж Петрі, що відтворюють рутинні інструкції програми, створення, початок та завершення роботи потоку, призупинку потоку на заданий інтервал часу, блокування дій потоку, очікування за умовою, обчислення підзадач пулом потоків.

Описано процес розробки Петрі-об’єктної моделі паралельної програми. Усі запропоновані та розроблені методи, підходи та засоби моделювання поєднані у технологію моделювання паралельного алгоритму. Розглянута її реалізація мовою Java. Наведено приклад розробки моделі вирішення проблеми взаємного блокування потоків, а також моделі пулу потоків.

3 МЕТОД ОПТИМІЗАЦІЇ ПАРАМЕТРІВ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

3.1 Постановка задачі оптимізації та обґрунтування існування оптимальних значень параметрів

Параметрами паралельного алгоритму є такі параметри, як кількість підзадач, їх складність, кількість потоків. До цього набору можуть бути додані специфічні умови, які впливають на механізми взаємодії між підзадачами. Якщо кількість обчислювальних дій в алгоритмі Q, то кількість підзадач k не може бути більшою за Q (інакше підзадача міститиме менше 1 дії) і не може бути меншою за 2 (інакше не буде кількох одночасно виконуваних задач). Отже, $2 \leq k \leq Q$. Аналогічне обмеження має кількість потоків p, що виконують одну задачу обчислювальної складності Q: $2 \leq p \leq Q$. За умови однакової складності підзадач при розділенні задачі на k підзадач, їх обчислювальна складність буде дорівнювати $g = Q/k$. З умови $2 \leq k \leq Q$ випливає умова $Q/2 \geq Q/k \geq 1$. Отже, $1 \leq g \leq Q/2$.

На обмеженому наборі параметрів паралельного алгоритму обов’язково існують оптимальні значення параметрів, які можуть бути знайдені повним перебором можливих варіантів параметрів або спрямованим до оптимуму перебором, якщо повний перебір містить надто багато варіантів. Здійснити перебір шляхом прогонів програми змінюючи значення параметрів означає провести численну кількість запусків програми та, відповідно, витратити час на очікування її виконання. Адже навіть при невеликій кількості параметрів кожний експеримент має бути повторений не менше чотирьох разів, щоб забезпечити точність результатів. Проведення експериментів із здійсненням перебору значень параметрів на моделі спростить та пришвидшить пошук оптимальних значень. До того ж, модель дозволяє відокремити випадкові фактори запуску такі, як, наприклад, зайнятість обчислювального ресурсу (ядра процесора) іншими задачами.

3.2 Методи та засоби збору даних для моделі

Для експериментального дослідження впливу часових затримок на виконання обчислювальних дій алгоритму, що моделюється, необхідно використовувати статистичні методи обробки даних.

Перед тим як обробляти дані, спершу проводиться їх збір. Для того, щоб точно заміряти час виконання тієї чи іншої інструкції/набору інструкцій, важливо проводити заміри за умови найменшого завантаження обчислювального ресурсу сторонніми задачами. Тобто, всі сторонні програми та застосунки мають бути вимкненими. Не слід забувати також про фонову роботу застосунків, наприклад оновлення, які можуть вплинути на результати замірів. Щоб виключити цей фактор, варто вимкнути доступ до інтернету. Також, якщо заміри здійснюються на ноутбуці, важливо проводити їх з увімкненим живленням, оскільки при використані вбудованої батареї ноутбук працює в енергозберігаючому режимі і не задіює повну потужність обчислювальних ресурсів. Коли мова йде про проведення замірів часу виконання паралельних обчислень в Java, не можна не згадати про загально відому особливість Java Virtual Machine – так званий розгін. Річ у тім, що при збільшенні завантаженості обчислювального ресурсу відбувається прискорення обчислень. З огляду на це, необхідним є додавання розігріву (warm up) – попереднього завантаження ресурсу певним обсягом обчислювальних операцій [60].

Точність замірів досягається за рахунок усереднення значень, отриманих в результаті серії прогонів програми. Оцінка кількості прогонів програми, необхідних для точного заміру часу виконання інструкції, за умови заданої точності обраховується за нерівністю Чебишева [61]:

$$p = \frac{\sigma^2}{\varepsilon^2(1-\beta)},$$

де p – кількість прогонів,

σ^2 – дисперсія часу виконання,

ε – точність вимірювання,

β – довірча ймовірність.

Наприклад, за довірчої ймовірності $\beta = 0{,}95$ та заданої точності $\varepsilon = \sigma$ кількість прогонів складатиме $p = 20$ прогонів. Аналогічно, при $\beta = 0{,}95$ та $\varepsilon = {\sigma}/{2}$ кількість прогонів становитиме $p = 80$.

Говорячи про замір часу, варто зазначити як правильно програмно організувати такий замір. При вимірюванні часу виконання однієї інструкції (або набору інструкцій) безпосередньо перед нею фіксується поточний час у обраних одиницях, це значення є початком відліку часу виконання. Після інструкції обраховується різниця поточного часу та початку відліку. Слід пам'ятати, що інструкції з виведення на консоль не мають входити до фрагменту програми, час виконання якого заміряється. Адже операції виведення на консоль, будучи не суттєвими для алгоритму, витрачають обчислювальний ресурс. Оголошення змінної, що зберігатиме час виконання, має здійснюватися до початку заміру. Нижче наведено приклад заміру часу (в наносекундах) виконання однієї інструкції створення пулу потоку в Java:

```
long createTime;
long currentTime = System.nanoTime();
ExecutorService pool = Executors.newFixedThreadPool(2);
createTime = System.nanoTime() - currentTime;
```

При дослідженні часових затримок важливою є ідентифікація закону розподілу випадкової величини. Досить часто час виконання інструкції паралельного алгоритму може бути недетермінованим, адже він може залежати від параметрів. У такому випадку визначення закону розподілу допоможе встановити часову затримку в моделі та сприятиме точному відтворюванню роботи паралельного алгоритму. Для ідентифікації закону розподілу необхідно спершу сформувати масив значень випадкової величини, тобто зробити певну кількість прогонів програми з різними параметрами та зафіксувати час виконання. Далі за отриманими значеннями слід побудувати гістограму частот. Наступним кроком йде формування гіпотези про вид закону розподілу випадкової величини. На основі припущення про вид розподілу здійснюється

оцінка значень параметрів закону розподілу. Після цього перевіряється відповідність досліджуваних значень обраному закону розподілу за допомогою критерію згоди. Якщо розраховане значення критерію згоди менше за табличне, тоді гіпотеза про закон розподілу випадкової величини підтверджується. У протилежному випадку варто змінити параметри закону розподілу або змінити припущення про вид закону розподілу [62]

Вичерпна колекція методів для статистичного дослідження випадкової величини міститься у програмному забезпеченні Stochastic Simulation in Java (SSJ) [63].

3.3 Методи та засоби розробки моделі

Розробка моделі паралельних обчислень виконується методом Петрі-об’єктного моделювання, викладеним у підрозділі 2.4. Технологія моделювання паралельних обчислень описана детально у підрозділі 2.5. Оскільки дії з моделювання на окремих етапах доводиться виконувати багатократно, то програмна їх підтримка пришвидшує та полегшує процес розробки моделі. Для підтримки процесу розробки моделі розроблено програмне забезпечення Parallel Program Simulation (PPS) [64, 52]. Програмне забезпечення PPS реалізує такі функції для підтримки розробки моделі:

- розробка мережі Петрі у графічному редакторі мереж Петрі;
- збереження мережі Петрі у двох форматах - графічне зображення та у вигляді методу;
- редагування параметрів моделі;
- анімація імітації як тестування моделі;
- виведення протоколу подій з деталізованим відтворення зміни стану моделі;

Розробка мережі розпочинається із запуску застосунку, внаслідок чого відкривається графічний редактор. Найважливіший елемент інтерфейсу на даному етапі – панель інструментів та шаблонів – розміщується у головному

вікні застосунку (рис. 3.1). За допомогою даних інструментів, власне, і відбувається побудова мережі Петрі.

Рисунок 3.1 – Панель інструментів та шаблонів

Панелі інструментів та шаблонів окрім стандартних інструментів розробки мережі Петрі містить також шаблони для моделювання паралельних обчислень, описані у підрозділі 2.4.

Побудова мережі відбувається у робочій панелі, розміщеній в центрі вікна (рис. 3.2). Робоча панель розділена на чотири області: панель збережених методів-кріейторів мереж Петрі, панель інструментів та шаблонів, панель графічного редактора мережі Петрі, панель виводу результатів моделювання (протокол подій та статистичні значення). Варто відмітити наявність редактора параметрів елементів мережі Петрі, який відкривається в окремому вікні. У випадку вказування параметру у вигляді рядкового значення (замість числового) такий параметр інтерпретується як аргумент метода-кріейтора мережі Петрі.

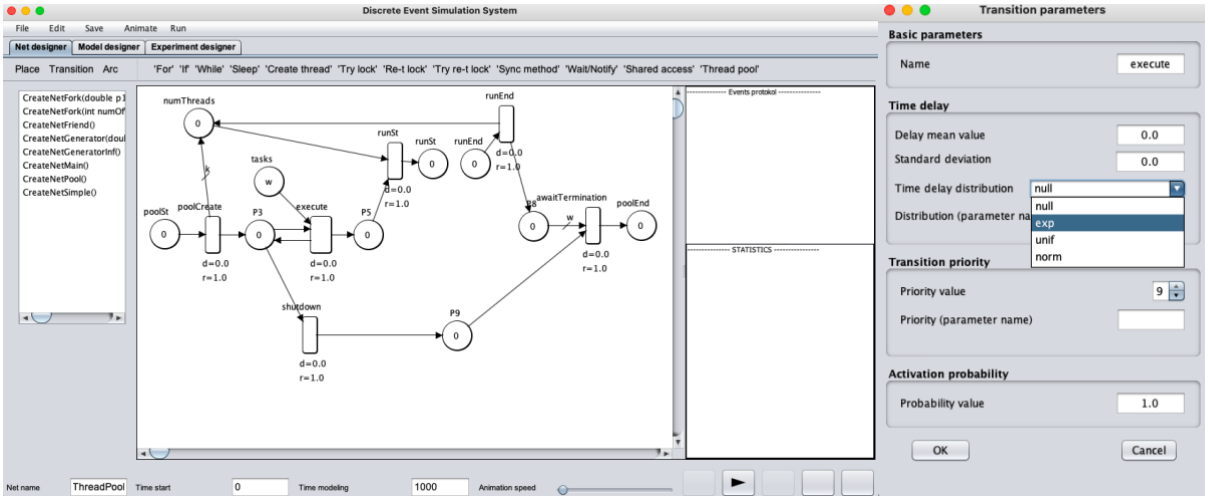


Рисунок 3.2 – Робоча панель з відкритим шаблоном «пул потоків». Вікно зміни параметрів переходу "execute" шаблону «пул потоків»

Тестування розробленої мережі Петрі здійснюється запуском імітації при різних початкових маркіруваннях. Результат імітації, представлений протоколом подій та статистичними значеннями, що характеризують стан позицій та

переходів, перевіряється на відповідність очікуваному сценарію подій та їх чисельному результату.

Відтворення подій в часі відбувається за алгоритмом імітації, що складений у відповідності до математичного опису Петрі-об’єктної моделі [65]. Алгоритм імітації складається з таких основних кроків:

- введення початкових даних для імітації (елементи моделі, час моделювання);
- доки не виконана умова завершення імітації:
  - відшукати момент найближчої події (рівний найменшому з усіх значень найближчої події, що визначені у Петрі-об’єктах за найменшим з усіх значень виходу маркерів з переходів мережі Петрі); якщо Петрі-об’єктів з найближчою подією декілька, то розв’язати конфлікт між Петрі-об’єктами;
  - просунути час у момент найближчої події;
  - для усіх Петрі-об’єктів, час виходу маркерів яких співпадає з поточним моментом, виконати (багатократний) вихід маркерів з переходів мережі Петрі та (багатократний) вхід маркерів в переходи мережі Петрі;
  - для усіх Петрі-об’єктів виконати вхід маркерів в переходи мережі Петрі (якщо змінився стан спільних позицій);
- вивести результати імітації.

Умовою завершення моделювання у класичному алгоритмі імітації є досягнення моменту часу, більшого за заданий час моделювання. У випадку імітації паралельних обчислень з метою оцінювання швидкодії умовою завершення імітації є досягнення стану моделі, який означає завершення процесу обчислень (маркірування відповідної позиції приймає значення 1), а значення часу на момент досягнення цього стану є оцінкою часу виконання обчислень.

Візуалізація покрокового відтворення подій підтримується анімацією змінювання стану мережі Петрі (рис. 3.3). Дана опція сприяє швидкому

виявленню помилок при конструюванні мережі. Адже стає можливим відстеження входів та виходів маркерів з переходів.

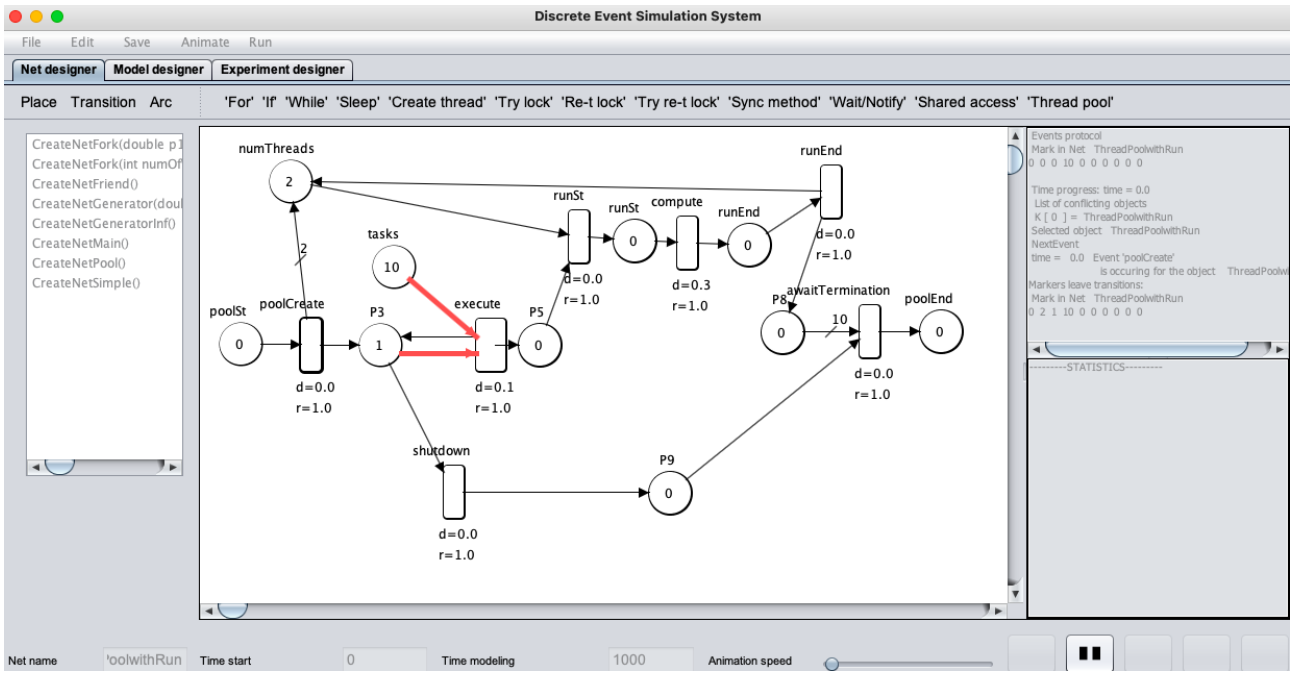


Рисунок 3.3 – Анімація змінювання стану мережі Петрі.

Після побудови мережі є можливість зберегти розроблену мережу як метод-кріейтор класу NetLibrary, який створюється автоматизовано на основі графічно побудованої мережі. Окрім цього, є також можливість збереження мережі у графічному вигляді для подальшого її редагування за потреби. Збережений метод-кріейтор можна використовувати для створення Петрі-об’єктів. При збереженні мережі Петрі відбувається перевірка на коректність побудови мережі: наявність хоча б однієї вхідної та хоча б однієї вихідної позиції у кожного переходу; наявність хоча б однієї звичайної вхідної позиції у випадку наявності вхідної позиції з інформаційною дугою. Невід’ємність генерованих значень часових затримок перевіряється в ході імітації.

Розробка Петрі-об’єктної моделі підтримується класами та методами бібліотеки PetriObjLib[66]. Клас PetriSim цієї бібліотеки є суперкласом для створення Петрі-об’єктів. Конструктор цього класу використовує в своєму аргументі мережу Петрі. Така мережа може бути продукована методом-кріейтором. Саме такий спосіб дозволяє створити потрібну кількість Петрі-

об’єктів з однаковою динамікою, тобто тиражувати Петрі-об’єкти. При цьому параметри Петрі-об’єктів можна змінювати. Після створення Петрі-об’єктів важливо вказати зв’язки між ними через спільні позиції. Для цього позиція одного Петрі-об’єкта ототожнюється з позицією другого.

Клас PetriObjModel згаданої бібліотеки необхідний для створення Петрі-об’єктної моделі. Конструктор даного класу приймає в своєму аргументі список Петрі-об’єктів. Запуск моделі на імітацію здійснюється методом go(), в аргументі якого вказується тривалість імітації в одиницях модельного часу.

Приклади розробки моделей демонструвались у підрозділі 2.5.

3.4 Метод оптимізації параметрів

Загалом, суть методу оптимізації параметрів полягає у визначенні набору параметрів, які впливають на ефективність алгоритму, та варіюванні значень цих параметрів у моделі для визначення оптимальних параметрів, що гарантують найвищу ефективність алгоритму. Критерієм оптимізації є мінімізація часу виконання алгоритму:

$$TimePerformance(n, l) \rightarrow min,$$

n, l – досліджувані параметри паралельного алгоритму.

Під час застосування методу оптимізації на етапі визначення параметрів, що впливають на ефективність паралельного алгоритму, в залежності від кількості параметрів, можливе застосування одного з двох методів пошуку оптимальних параметрів.

У випадку, коли кількість параметрів, що досліджуються, менше або дорівнює 2, слід застосувати адаптивний покроковий алгоритм оптимізації зі змінним кроком. Він є точним, оскільки множина значень кожного параметра є дискретною та обмеженою. Метод полягає в тому, що при фіксованому значенні одного параметра експериментально будується залежність часу виконання паралельного алгоритму по всім можливим значенням другого параметра. Кількість таких експериментальних одновимірних залежностей дорівнює

кількості можливих значень другого параметра. Оптимальне значення задається тією одновимірною експериментальною кривою, на якій досягається мінімальне значення часу виконання паралельного алгоритму. Значення першого параметра відповідає оптимуму відповідної одновимірної експериментальної кривої, значення другого параметра – це значення, для якого ця крива побудована. За побудовою отриманий метод є точним.

У випадку, коли кількість параметрів оптимізації від 2 і більше, пропонується у рамках поставленої задачі реалізація евристичного методу, що базується на еволюційному алгоритмі (узагальнена версія генетичного алгоритму) [61]. Елементом популяції являється набір значень параметрів паралельних обчислень по два або більше параметри (наприклад, складність підзадач та ліміт буфера зовнішніх подій) для кожної особи популяції:

$$A = \{A^{(i)}\},$$

$$A^{(i)} = (n_1, \dots n_k),$$

де $A^{(i)}$ – і-та особа популяції,

n_j – j-тий параметр особи,

k – кількість параметрів особи.

Програмна реалізація особи популяції виглядає як java-клас Individual, у полях якого містяться кількість параметрів і масиви мінімальних та максимальних значень параметрів, а в конструкторі, де через аргумент передається кількість прогонів, створюється масив значень параметрів для однієї особи:

```
public class Individual {
     private static int numParams;
     private final int[] params;
     private static int[] mins;
     private static int[] maxs;

     public Individual(int progon) {
          numParams = mins.length;
          params = new int[numParams];
          for (int i = 0; i < numParams; i++) {
            if (maxs[i] == mins[i]) {
                this.params[i] = mins[i];
```

```
            } else {
                this.params[i] =
                mins[i] + random.nextInt(maxs[i] - mins[i]);
            }
        }
```

Початкова популяція (генерування 0) формується з випадкових значень, рівномірно розкиданих в області допустимих значень параметрів. Кожна особа популяції запускається у «життя» – в імітаційну модель паралельних обчислень. При цьому набір параметрів в осіб генерується так, що в результаті кожний параметр варіюється з кожним. Результатом життєдіяльності елемента популяції є відгук моделі – значення функції допасованості (fitness function), що повертається методом calcFit() і відповідає часу виконання алгоритму в моделі. Нижче наведено конструктор особи популяції для випадку двох параметрів з варіацією двох індексів: j та k; num – інтервал розбиття області допустимих значень на рівномірні проміжки.

```
public Individual(int progon, int j, int k, int num) {
    numParams = mins.length;
    params = new int[numParams];
    if (maxs[0] == mins[0]) {
        this.params[0] = mins[0];
    }
    else {
        this.params[0] =
        mins[0] + (int) Math.rint(j * (maxs[0] - mins[0]) / num);
    }
    if (maxs[1] == mins[1]) {
        this.params[1] = mins[1];
    }
    else {
        this.params[1] =
        mins[1] + (int) Math.rint(k * (maxs[1] - mins[1]) / num);
    }
    calcFit(progon);
}
```

Після генерації початкової популяції масив осіб `population` сортується за зростанням значення відгуку моделі, таким чином першим елементом масиву буде особа з найкращим відгуком моделі, яка запам’ятовується в окремому масиві `fitValues`. Набори параметрів, які отримали в процесі імітації великі значення відгуку моделі, підлягають знищенню і поріг для знищення з кожним

поколінням зменшується. Відтак, відбір елементів популяції відбувається за значенням відгуку моделі. Елементи популяції, пройшовши відбір, піддаються схрещуванню. Схрещування здійснюється для випадково обраних пар елементів популяції шляхом змішування частин наборів параметрів. Нехай для схрещування обрані елементи популяції $A^{(i)}$ та $A^{(j)}$. Внаслідок роботи оператора кроссовера нова особа популяції матиме набір параметрів, кожен з яких є середнім арифметичним відповідних значень параметрів осіб $A^{(i)}$ та $A^{(j)}$:

$$\begin{cases} A^{(i)} = \left(n_1^{(i)} \dots n_k^{(i)}\right) \\ A^{(j)} = \left(n_1^{(j)} \dots n_k^{(j)}\right) \end{cases} \Rightarrow$$

$$A^{(c)} = \left(\frac{n_1^{(i)} + n_1^{(j)}}{2}, \frac{n_2^{(i)} + n_2^{(j)}}{2}, \dots, \frac{n_{k-1}^{(i)} + n_{k-1}^{(j)}}{2}, \frac{n_k^{(i)} + n_k^{(j)}}{2} \right),$$

де $A^{(c)}$- особа популяції створена внаслідок схрещування осіб $A^{(i)}$ та $A^{(j)}$.

Мутація відбувається шляхом додаванням випадкового відхилення до результату, отриманого в результаті схрещування:

$$A^{(m)} = (n_1^{(i)} + \xi_1,\ n_2^{(i)} + \xi_2, \dots\ n_{k-1}^{(j)} + \xi_{k-1},\ n_k^{(j)} + \xi_k),$$

де ξ_i - випадкова величина, що приймає значення -d, 0, d з рівною ймовірністю; d – величина відхилення.

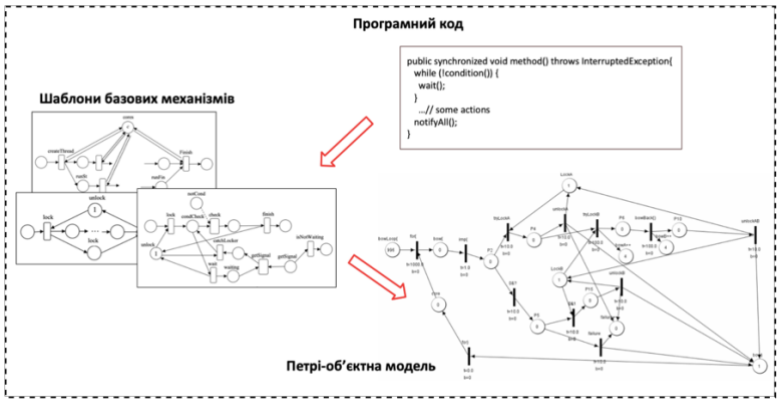
Кожна наступна популяція (генерування *i*) утворюється з k елементів, пройшовших відбір на попередньому генеруванні (генерування *i*-1), та з елементів, створених внаслідок схрещування та мутації. При цьому, якщо k=1, тобто лише одна особа пройшла відбір, то вона схрещується з наступними особами попередньої популяції, що відсортовані за зростанням відгуку моделі. У випадку коли k>1, випадковим чином обираються дві унікальні особи з масиву осіб, що пройшли відбір. Зупинка еволюційного пошуку відбувається після здійснення встановленої користувачем кількості генерувань. Після виконання всіх генерувань повертається набір значень параметрів з найменшим часом виконання алгоритму. Код класів алгоритму наведений у додатку В.

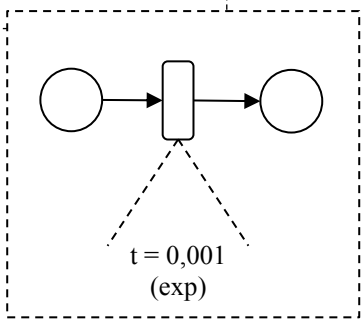
Етапи еволюційного методу пошуку оптимальних значень параметрів:

1. Створення початкової популяції з заданою кількістю осіб.
2. Сортування масиву фітнес-функцій осіб (часу виконання алгоритму) початкової популяції за зростанням.
3. Додати особу з найменшим значенням фітнес-функції до масиву найкращих фітнес-функцій популяцій.
4. Виконати задану кількість генерацій:
   4.1. Генерація наступної популяції:
      4.1.1. Відбір найкращих k осіб з попередньої популяції за значенням фітнес-функції;
      4.1.2. Створення нових осіб шляхом схрещення довільних пар найкращих k осіб та мутації;
      4.1.3. Сортування масиву фітнес-функцій осіб за зростанням.
   4.2. Оцінка зменшення фітнес-функції поточної популяції відносно попередньої популяції:
      4.2.1. Якщо фітнес-функція не зменшується декілька разів підряд – зупинити еволюційний алгоритм, вивести набір параметрів особи з найкращим значенням фітнес-функції з усіх згенерованих популяцій;
      4.2.2. Інакше, продовжити генерацію наступних популяцій.
5. Вивести набір параметрів особи з найкращим значенням фітнес-функції з усіх згенерованих популяцій.

Основні етапи загального методу оптимізації параметрів паралельного алгоритму представлені у таблиці 3.1

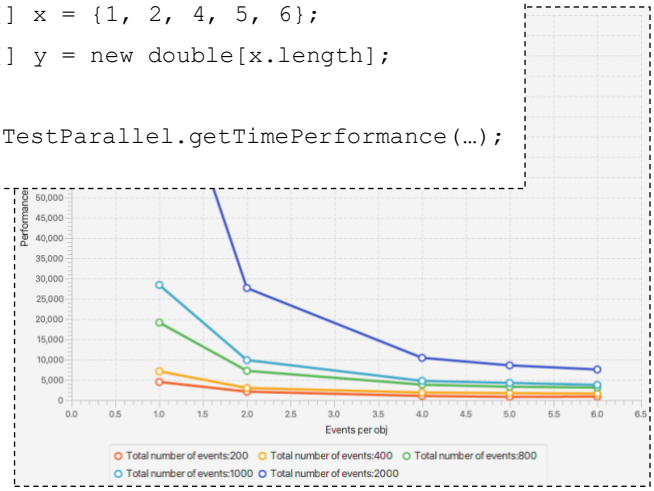
Таблиця 3.1 Метод оптимізації параметрів

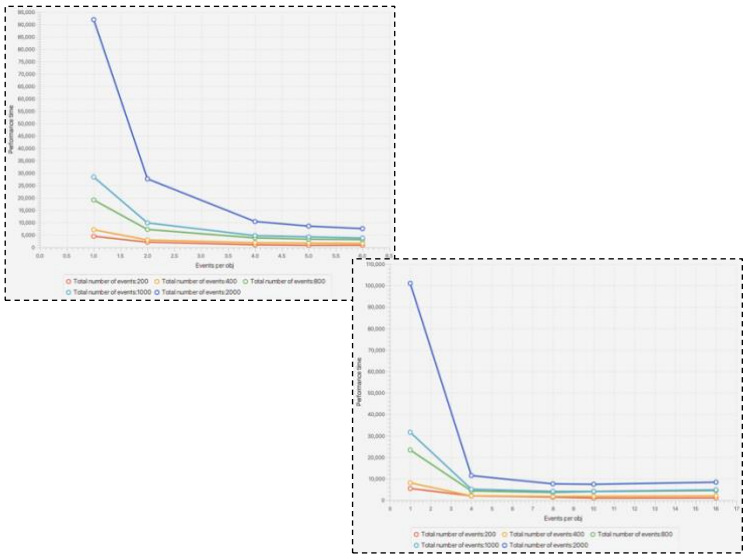
Етап	Зміст	Результат
Побудова моделі	Використовуючи технологію моделювання паралельних обчислень, побудувати Петрі-об’єктну модель паралельного алгоритму.	 Метод createModel()

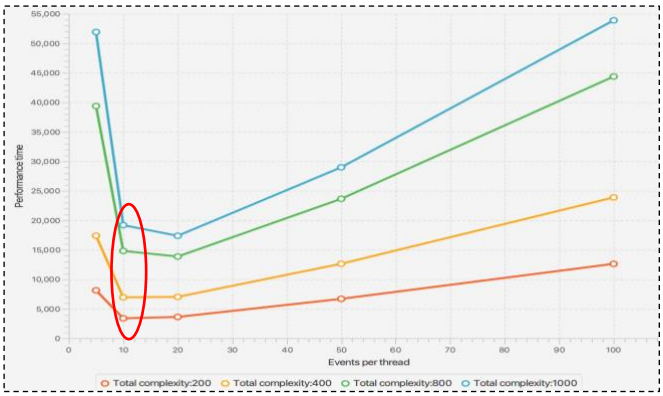
Визначення часових затримок	Експериментальним шляхом визначити часові затримки подій та внести їх до моделі.	`long startTime = System.nanoTime();`
		`...`
		`long stopTime = System.nanoTime();`
		`time = (stopTime - startTime);`
		 

<table>
<tr><th>Етап</th><th>Зміст</th><th>Результат</th></tr>
<tr><td>Визначення параметрів</td><td>Визначити набір параметрів, що впливають на ефективність алгоритму.
Якщо кількість параметрів менше або рівна 2, то далі застосовується метод покрокової оптимізації. У іншому випадку застосовується евристичний метод.</td><td>• кількість потоків;
• кількість підзадач;
• розмір підзадачі;
• обсяг обчислювальних ресурсів;
• інше.
 
createModel(...)</td></tr>
<tr><td>Підготовка до експериментів</td><td>Налаштувати клас Experiment для проведення серій експериментів:
• визначити параметри осей Ox та Oy;
• визначити результати, які наноситимуться на графік (час виконання);
• визначити кількість прогонів.</td><td>
<pre>...
stage.setTitle(title);
//defining the axes
final NumberAxis xAxis = new NumberAxis();
final NumberAxis yAxis = new NumberAxis();
xAxis.setLabel(xLabel);// "Events per obj"
yAxis.setLabel(yLabel);//"Performance time"
//creating the chart
final LineChart<Number, Number> lineChart
       = new LineChart<>(xAxis, yAxis);
for (XYChart.Series s : series) {
     lineChart.getData().add(s);
}
Scene scene = new Scene(lineChart, 800, 600);
stage.setScene(scene);
stage.show();
...</pre>
</td></tr>
</table>

Етап	Зміст	Результат

Проведення експериментів	Провести всі експерименти з варіюванням параметрів (запустити прогони моделі для різних параметрів) і отримати графіки.	`double[] x = {1, 2, 4, 5, 6};`
		`double[] y = new double[x.length];`
		`…`
		`y[i]+= TestParallel.getTimePerformance(…);`
		 

Перевірка умови завершення дослідження	Якщо результат експериментів не проявив зменшення часу при певних параметрах, тоді слід змінити їх і повторити експеримент допоки не виявиться зменшення часу.	

Етап	Зміст	Результат
Аналіз результатів	Проаналізувати графіки та визначити оптимальні значення параметрів.	

На *першому* етапі для побудови моделі використовуються шаблони моделювання механізмів багатопотокової програми. За допомогою готових фрагментів мереж Петрі створюються основи для Петрі-об’єктів та зв’язки між ними. Далі Петрі-об’єкти доповнюються у відповідності до об’єктів програми. Розробка моделі відбувається за технологією моделювання описаною в розділі 2.5. Результатом даного етапу є розроблений метод getModel(), який повертає Петрі-об’єктну модель багатопотокової програми.

На *другому* етапі залучається метод збору даних для моделі. Проводиться експериментальне дослідження часу витраченого на виконання інструкцій та визначаються їх статистичні характеристики. Результатом даного етапу є параметри часових затримок, які вносяться до моделі: середнє значення, середнє квадратичне відхилення та закон розподілу. Якщо відхилення невелике, то може бути прийняте рішення про детерміновану затримку.

На *третьому* етапі проводяться попередні експерименти з моделлю для визначення набору параметрів, що впливають на ефективність. Такими параметрами можуть бути: кількість потоків, кількість підзадач, розмір підзадачі, обсяг обчислювальних ресурсів тощо. Варіювання цих параметрів треба забезпечити при проведенні експериментів з моделлю. Це реалізується шляхом розміщення параметрів у методі getModel() і є результатом даного

етапу. Якщо кількість параметрів менше або дорівнює 2, для пошуку оптимальних параметрів застосовується метод покрокової оптимізації. В іншому випадку, коли кількість параметрів більше 2, застосовується евристичний метод пошуку оптимальних параметрів, який базується на еволюційному алгоритмі.

У випадку, коли кількість параметрів менше або дорівнює 2, на *четвертому* етапі відбувається підготовка до проведення експериментів. Клас Experiment, що підтримує JavaFx Application налаштовується під конкретну серію експериментів, щоб отримати графік результатів після імітації моделі. У даному класі мають бути налаштовані методи getResult() (одна точка на графіку при конкретних значеннях), getSeries() (отримання множини точок за якими буде побудовано графік), startExperiment() (проведення декількох серій експериментів з параметрами).

На *п'ятому* етапі запускається проведення експериментів, в результаті чого будуються графіки залежності спостережуваної величини від змінюваного параметра. В результаті даного експерименту можна аналізувати вплив двох параметрів на спостережувану величину.

На *шостому* етапі проводиться аналіз результатів експериментів. Якщо мінімальне значення спостережуваної величини не відстежується, варто змінити значення параметру і продовжити експерименти.

На *сьомому* етапі внаслідок проведення експериментів на попередніх етапах відбувається аналіз отриманих графіків та визначається оптимальне значення параметру, що відповідає мінімальному значенню спостережуваної величини.

У випадку, коли кількість параметрів більше або дорівнює 2, на *четвертому* етапі відбувається підготовка до застосування еволюційного алгоритм. Встановлюється кількість параметрів та їх область варіювання, кількість осіб в популяції.

На *п'ятому* етапі запускається еволюційний алгоритм та отримуються результати його виконання. Результатом даного етапу є такий набір параметрів з області варіювання , що гарантує мінімальний час виконання алгоритму.

3.5 Висновки до розділу 3

У розділі сформульована постановка задачі оптимізації та обґрунтовано існування оптимальних значень параметрів паралельних обчислень. Описані методи та засоби збору даних, які необхідні для дослідження впливу часових затримок у моделі на час виконання алгоритму. Наведено методи та засоби розробки моделі. Описано процес розробки Петрі-об'єктної моделі з використанням розробленого програмного забезпечення та наведено функції, які воно підтримує. Сформульовані основні етапи методу оптимізації параметрів паралельних обчислень та критерій оптимізації.

4 ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ ПАРАМЕТРІВ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

4.1 Паралельний алгоритм імітації

Розглянемо приклад застосування методу оптимізації параметрів паралельних обчислень на прикладі паралельного алгоритму імітації дискретно-подійної системи. Результати розробки моделі паралельного алгоритму та її дослідження опубліковані в роботі [7]. Звичайний алгоритм імітації відтворює впорядковану в часі послідовність подій. Тому його паралельна реалізація є нетривіальним завданням.

4.1.1 Теоретичне оцінювання складності паралельного алгоритму імітації

У загальному вигляді алгоритм моделювання складається з кроків, кожен з яких включає пошук моменту найближчої події та реалізацію події. Реалізація події – це зміна стану моделі. У послідовному алгоритмі події відтворюватимуться одна за одною відповідно до моментів часу. Кількість імітованих подій можна оцінити добутком інтенсивності подій та часу моделювання. Підсумовуючи інтенсивність усіх подій, можна отримати максимально можливе значення інтенсивності подій, що реалізується, якщо умова для настання події завжди виконана. Пошук найближчої події також сильно залежить від кількості подій, оскільки він знаходиться шляхом перевірки стану всіх подій моделі. Таким чином, обчислювальну складність послідовного алгоритму імітації можна оцінити таким виразом:

$$v \cdot k \cdot t \cdot (r \cdot k + c), \tag{4.1}$$

де k – кількість подій,

v – інтенсивність однієї події,

t – час моделювання,

r – обчислювальна складність пошуку найближчої події на одну подію,

c – обчислювальна складність зміни стану моделі.

Якщо різні частини моделі можуть працювати одночасно, час виконання буде визначатися складністю однієї частини моделі. Поділ моделі на n частин зменшить кількість подій, що обробляються в кожній частині. Отже, складність паралельного алгоритму імітації можна оцінити таким виразом:

$$v \cdot \frac{k}{n} \cdot t \cdot \left(r \cdot \frac{k}{n} + c\right) + n \cdot d, \tag{4.2}$$

де d – обчислювальна складність використання одного додаткового обчислювального ресурсу.

Вираз (4.2) вказує на квадратичну залежність між складністю паралельного алгоритму імітації та складністю кожної його частини. На рисунку 4.1 зображено зростання складності алгоритму (4.2) залежно від кількості подій у окремому випадку параметрів (v = 1, t = 1, r = 100, c = 100, d = 1000).

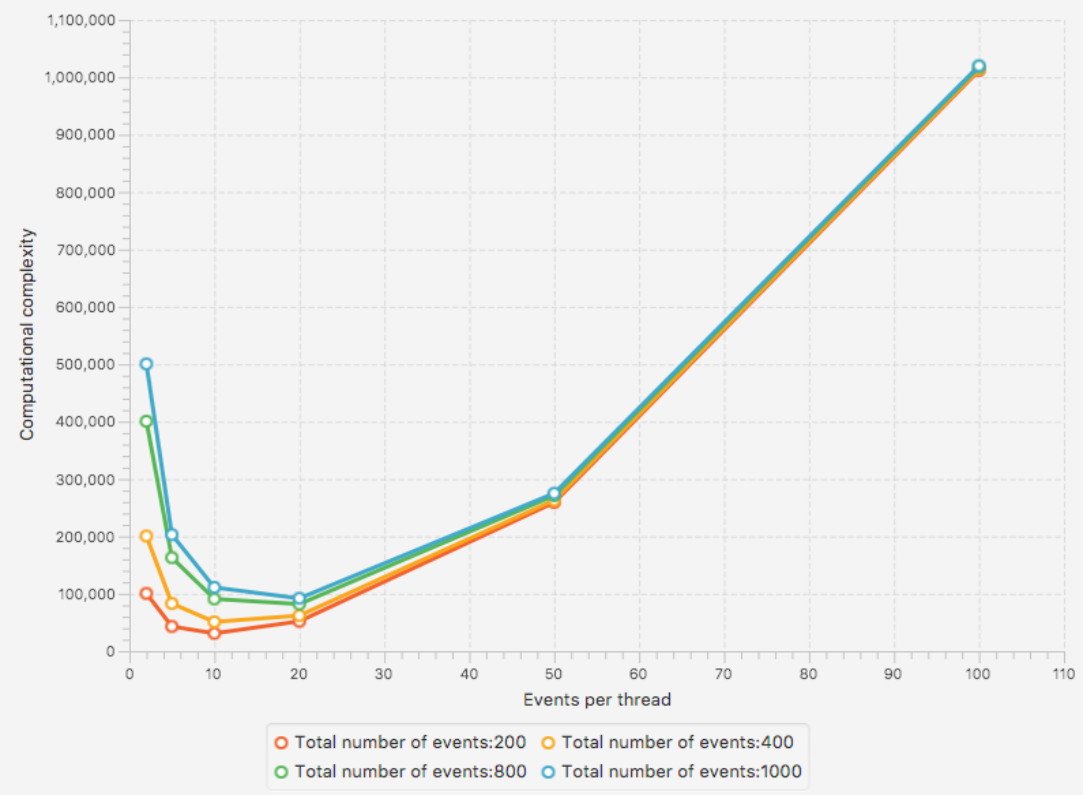


Рисунок 4.1 – Математична оцінка складності паралельного алгоритму імітації в залежності від кількості подій в частині, що запускається на паралельне виконання (v = 1, t = 1, r = 100, c = 100, d = 1000).

Найменша складність паралельного алгоритму забезпечує його найбільше прискорення у порівнянні з послідовним алгоритмом. Відповідно до аналітичних виразів складності (4.1) і (4.2) прискорення досягне 1000 разів для алгоритму імітації 1000 подій, що паралельно обчислює частини з 20 подій:

$$\frac{v \cdot k \cdot t \cdot (r \cdot k + c)}{v \cdot \frac{k}{n} \cdot t \cdot \left(r \cdot \frac{k}{n} + c\right) + n \cdot d} = \frac{1000 \cdot (100 \cdot 1000 + 100)}{20 \cdot (100 \cdot 20 + 100) + 50 \cdot 1000} = 1088{,}04.$$

Однак на практиці прискорення буде значно меншим через взаємодію потоків, яку неможливо просто представити в аналітичному виразі (4.2).

Таким чином, паралельна реалізація алгоритму імітації є дуже бажаною, особливо у випадку складної моделі. Однак це нетривіальне завдання, оскільки відтворення будь-якої частини моделі базується на моменті часу. Використання спільного значення часу для всіх частин викликає сповільнення, що знищує переваги використання паралельних обчислень (ефективність паралельного алгоритму менша за 1). Тільки коли частини використовують локальне значення часу, моделювання цих частин може виконуватися незалежно в межах інтервалу, в якому не потрібно обробляти інформацію про події в інших частинах. Іншими словами, кожна частина відтворює своє функціонування з урахуванням подій в інших частинах. Очевидно, що цей підхід має обмеження: частина повинна мати попередню частину, для якої ця не є попередньою або не має жодної попередньої.

4.1.2 Паралельна реалізація алгоритму імітації мовою Java та дослідження його швидкодії

Реалізація в Java паралельного алгоритму імітації включає потоки, що відтворюють частини системи, що моделюються, переміщуючи локальний час *tLoc* до найближчої події. Взаємодія між потоками реалізується двома парами умов wait/notify: перша контролює порожній стан буферу зовнішніх подій, а друга - досягнення максимального значення стану буфера (рис. 4.2).

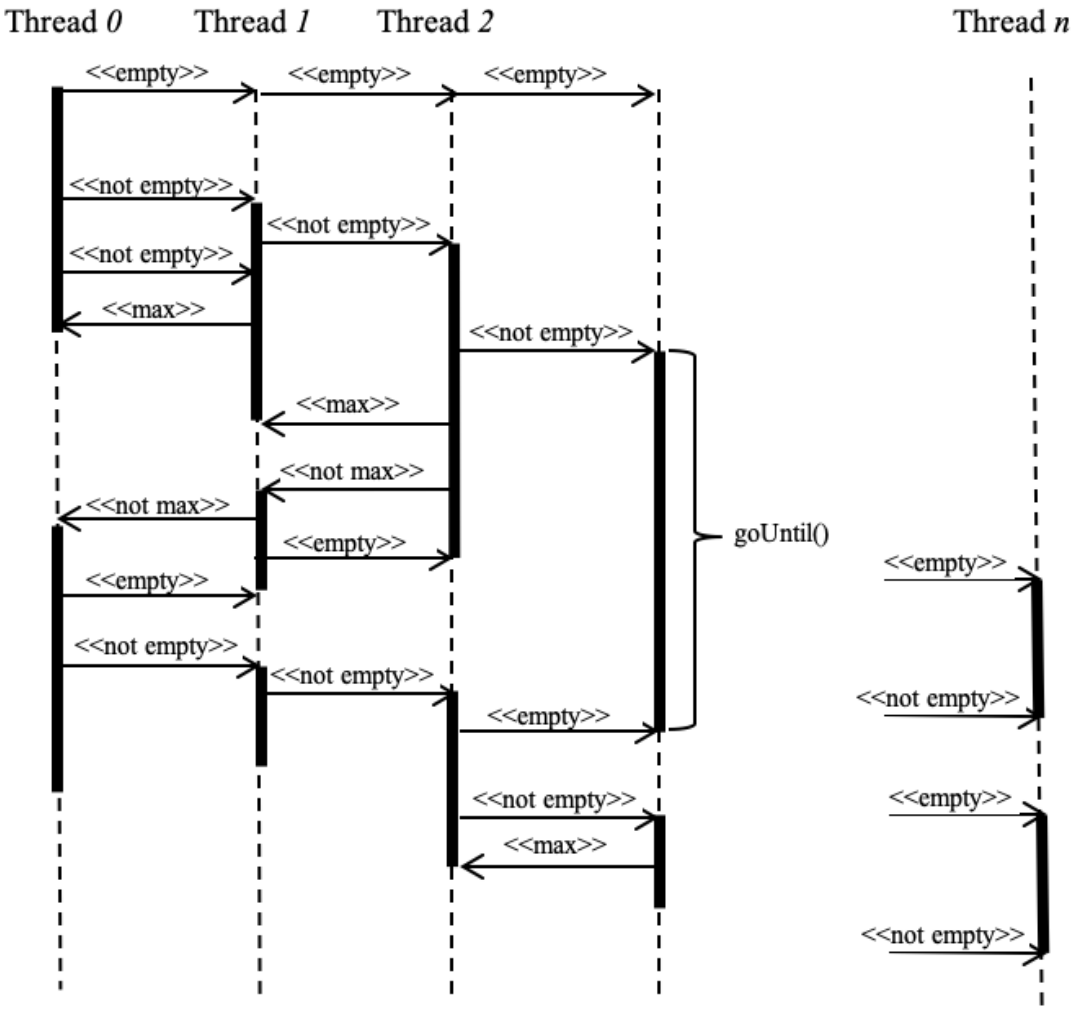


Рисунок 4.2 – Діаграма взаємодії потоків.

Основний потік створює потоки sim, імітуючи частини системи, поки не буде досягнуто час моделювання `tMod`. Кожен потік `sim` чекає, поки його буфер зовнішніх подій не буде порожнім, а потім запускає імітацію, доки не буде досягнуто найближчого моменту його зовнішньої події `tLim`. Коли зовнішня подія виконується, потік може продовжувати імітацію, переміщаючи час до наступної найближчої зовнішньої події таким же чином. Щоб уникнути ситуації, коли один потік `sim` займає ресурс комп'ютера, потік призупиняє моделювання, коли достатньо зовнішніх подій надано для іншого потоку.

Для дослідження прискорення алгоритму створено модель, що складається з груп послідовних подій. Оскільки послідовні події набагато важче розпаралелювати, оцінене прискорення буде найменшим з усіх, що можна

отримати в загальному випадку. Модель можна розділити на групи залежно від того, скільки потоків буде використано для обчислення імітації. Таким чином, кількість потоків і складність обчислень, що виконуються кожним потоком, можуть змінюватися простим способом.

Експериментальні результати дослідження впливу параметрів багатопотокового алгоритму на продуктивність часу представлені на рисунку 4.3. Для проведення експерименту використовувався комп'ютер MacOS, оснащений двоядерним процесором Intel Core i5. Експеримент повторювався зі збільшенням складності моделі. Виявлено, що найменший час роботи алгоритму забезпечується, якщо складність завдання, що виконується одним потоком, наближається до десяти незалежно від загальної кількості подій в імітаційній моделі. Цей результат корелює з математичною оцінкою складності алгоритму (4.2).

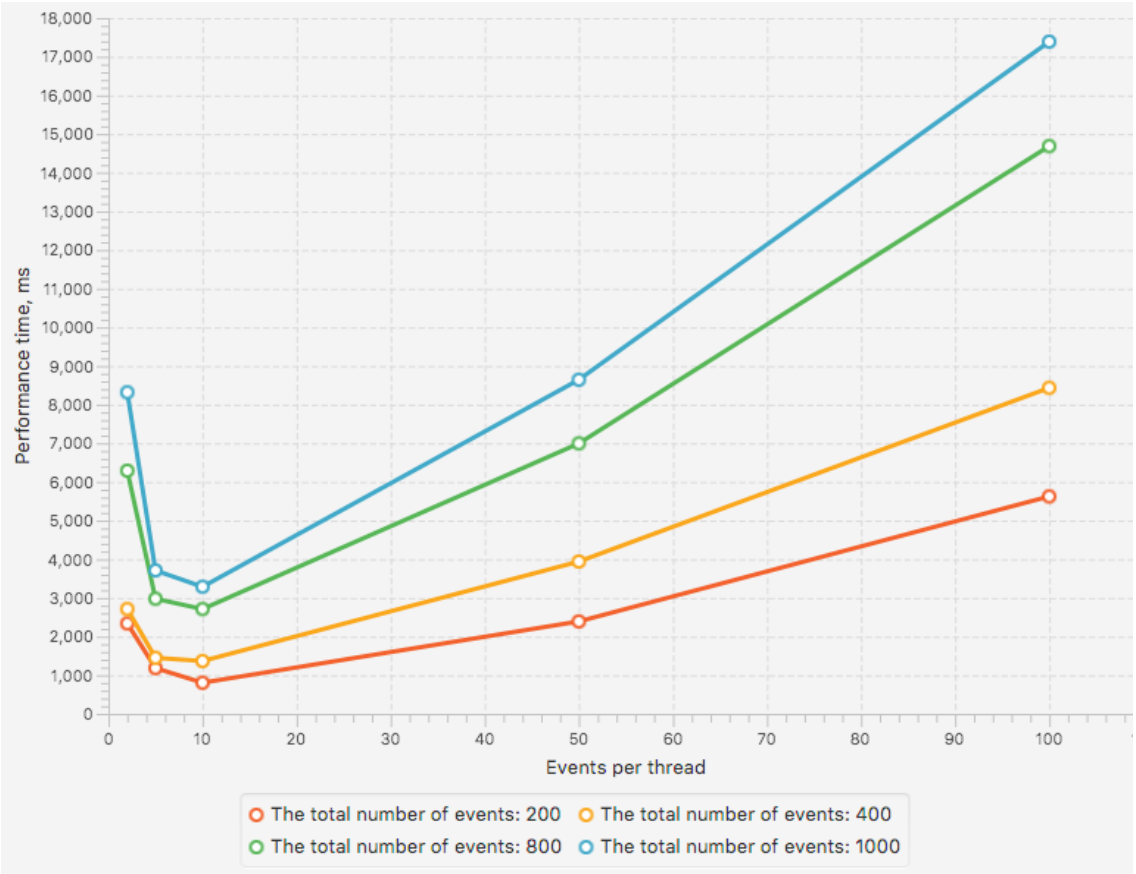


Рисунок 4.3 – Вплив параметрів багатопотокового алгоритму на час виконання.

4.1.3 Оптимізація параметрів паралельного алгоритму імітації на основі Петрі-об’єктної моделі

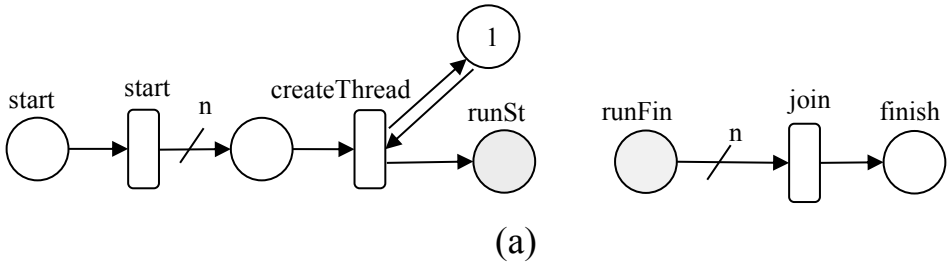
Побудова моделі паралельного алгоритму, яка необхідна для дослідження та оптимізації його параметрів може виглядати наступним чином. По-перше, для кожного об'єкта `Runnable` слід скласти список основних програмних інструкцій, які виконуються. Переходи, узгоджені з інструкціями, створять мережу Петрі-об’єкта, що імітує потік. По-друге, слід описати взаємодію між потоками. Найбільш типовими взаємодіями є очікування сигналу, очікування приєднання, очікування блокування, створення нового потоку та очікування завершення потоку. Нарешті, для кожного переходу повинна бути встановлена затримка часу, середнє значення якої повинно відповідати відповідному значенню продуктивності програмних інструкцій.

Для тестування методу було обрано паралельний алгоритм імітації. Коротко кажучи, паралельний алгоритм імітації реалізується такими методами:

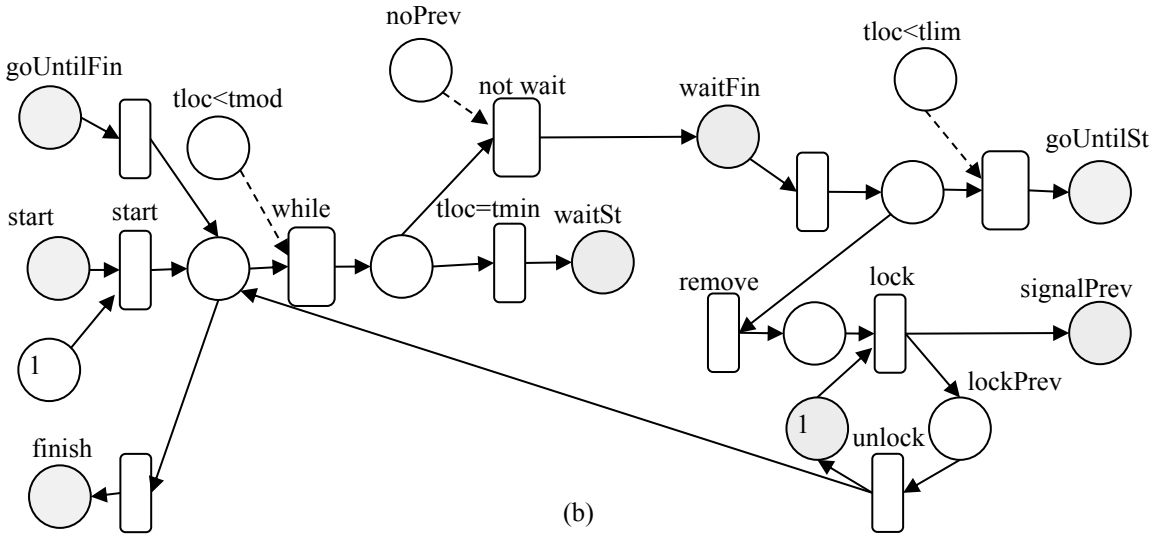
- Метод `main`, який встановлює значення часу моделювання `tMod`, створює потоки, об’єднує потоки та друкує результати.
- Метод `run`, який очікує на зовнішню подію, поки `tLoc<tMod`, потім встановлює `tLim` і запускає метод `goUntil`, якщо `tLim< tMod`.
- Метод `goUntil`, який поки `tLoc<tLim` чекає, якщо буфер зовнішніх подій досяг максимального розміру або цей буфер порожній, потім переміщує `tLoc` до найближчої події та виконує подію; після цього досягається момент `tLim` і повинна бути виконана зовнішня подія.
- Метод `output`, який змінює стан моделі відповідно до події та за потреби додає дані про зовнішні події до буферів інших частин моделі.

Метод `main` викликає метод `run` для кожного потоку та чекає його завершення. Метод `run` викликає метод `goUntil`, який, у свою чергу, викликає метод `output`.

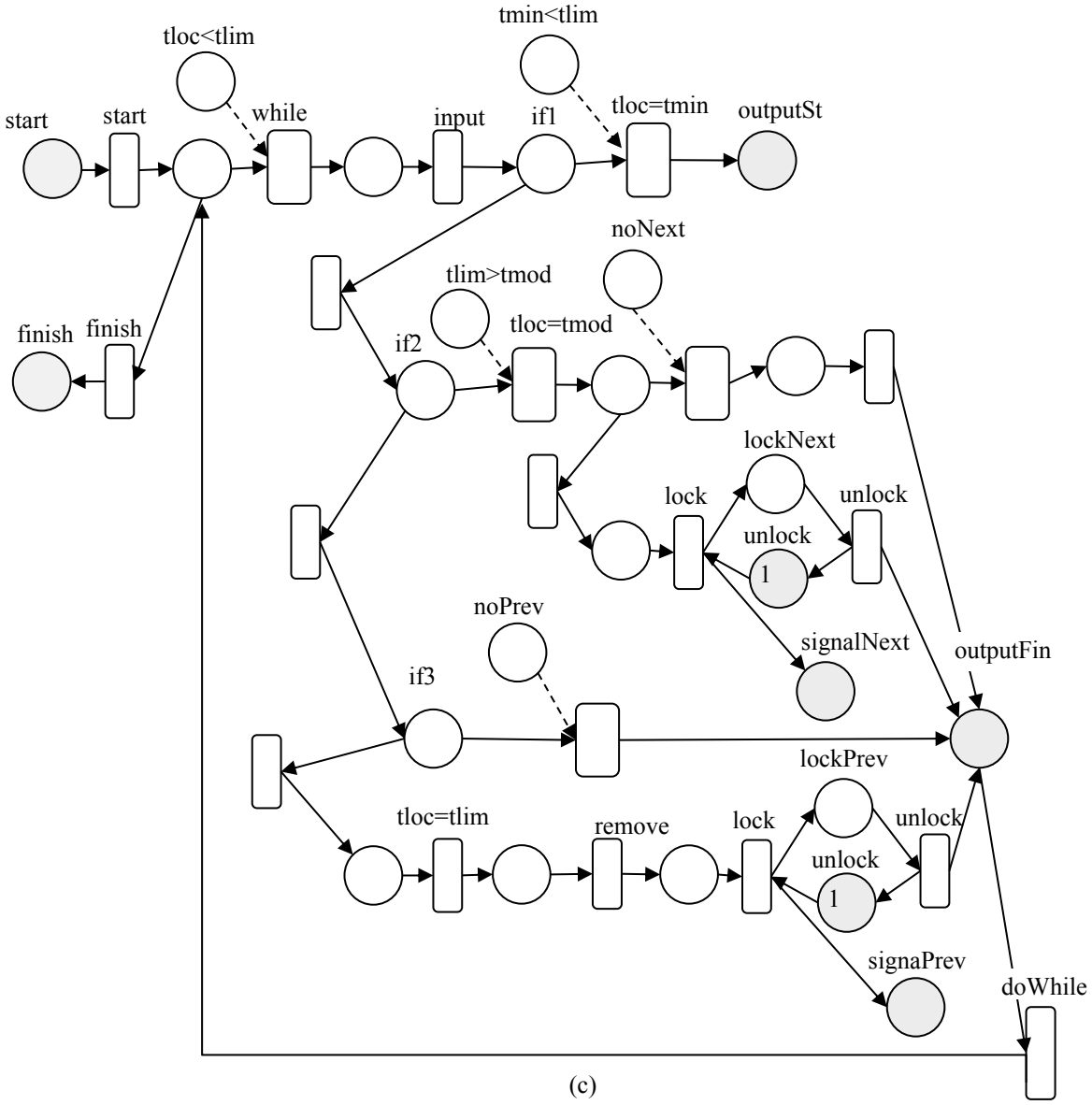
Структура моделі відтворює структуру програми. Класами, з яких побудована модель алгоритму, є `Main` і `ThreadModel`. Клас `ThreadModel` агрегує класи `RunSim`, `GoUntilSim`, `OutputSim`, `WaitSim`, які імітують методи "run", "goUntil", "output" і блок синхронізації дій (guarded block) відповідно, та клас `Control`, який зберігає значення моментів часу `tLoc`, `tLim`, `tMin`. Класи `Main`, `RunSim`, `GoUntilSim`, `OutputSim`, `WaitSim` є підкласами класу `PetriSim` і мають в своєму описі мережу Петрі. Клас `WaitSim` визначається мережею Петрі блоку синхронізації дій (див. рис. 2.8). Мережі Петрі інших класів зображені на рисунку 4.4. Кількість потоків позначається значенням n, загальна кількість подій позначається значенням k, кількість подій, імітованих одним потоком, відповідно позначається значенням k/n.



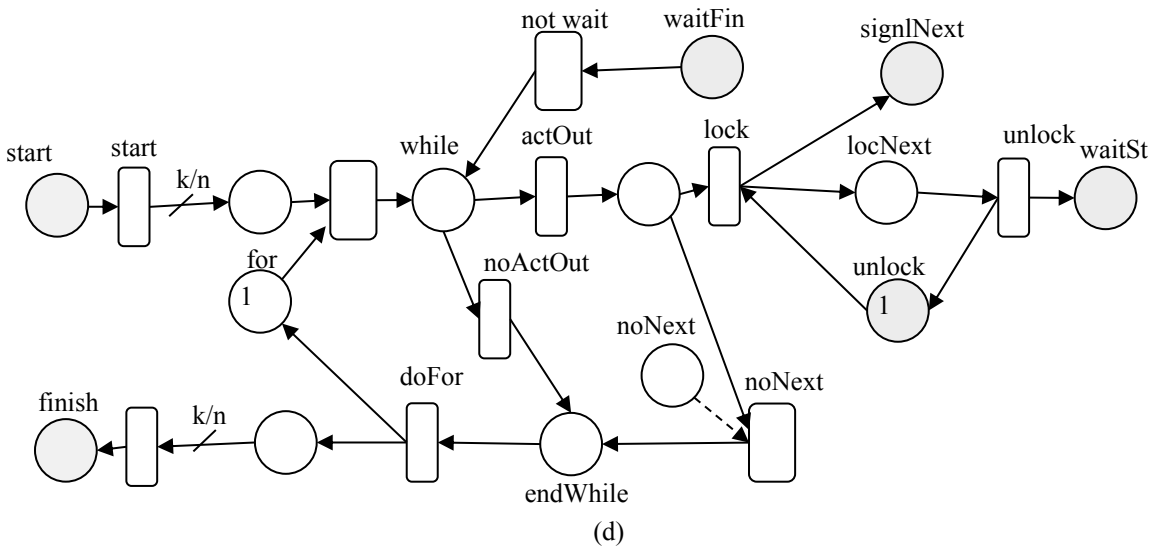
(a)


goUntilFin
tloc<tmod
noPrev
not wait
waitFin
tloc<tlim
goUntilSt
start
start
while
tloc=tmin
waitSt
remove
lock
signalPrev
1
lockPrev
1
unlock
finish

(b)


tloc<tlim
tmin<tlim
start
start
while
input
if1
tloc=tmin
outputSt
noNext
tlim>tmod
finish
finish
tloc=tmod
if2
lockNext
lock
unlock
unlock
1
noPrev
signalNext
outputFin
if3
lockPrev
tloc=tlim
remove
lock
unlock
unlock
1
signaPrev
doWhile

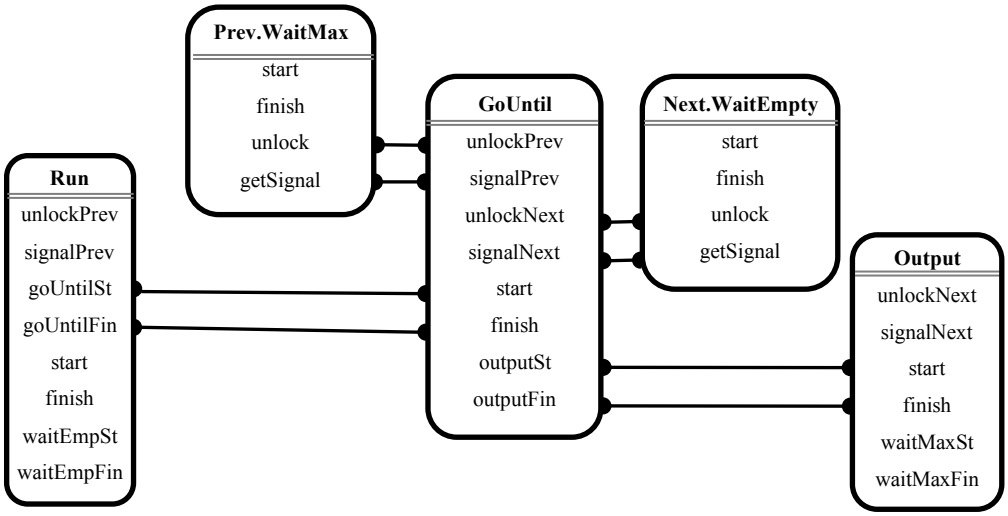
(c)



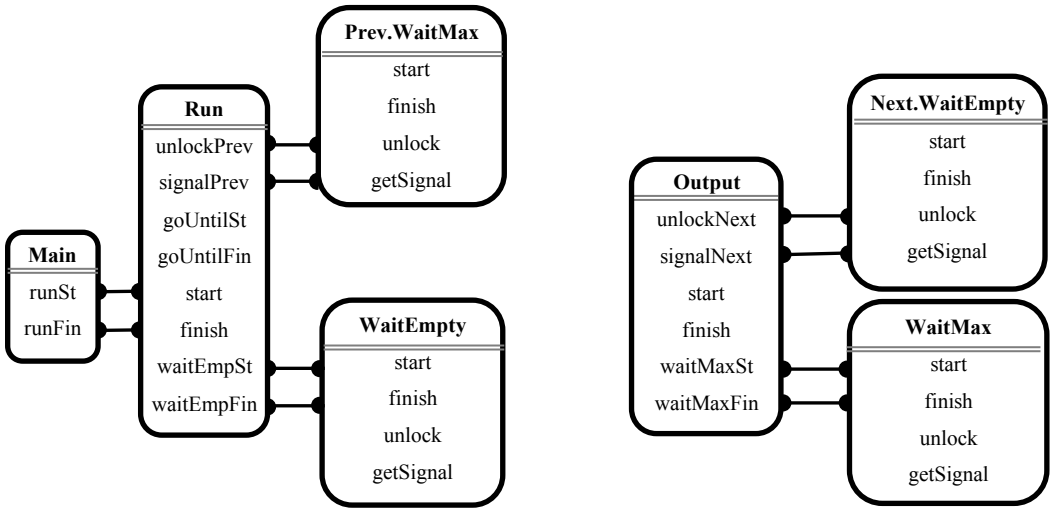
(d)

Рисунок 4.4 – Мережі Петрі-об’єктів:

(a) `Main`, (b) `RunSim`, (c) `OutputSim`, (d) `GoUntilSim`.



(a)



(б) (в)

Рисунок 4.5 – Ототожнення позицій Петрі-об’єктів: (а) Петрі-об’єкта GoUntil, (б) Петрі-об’єкта Run, (в) Петрі-об’єкта Output.

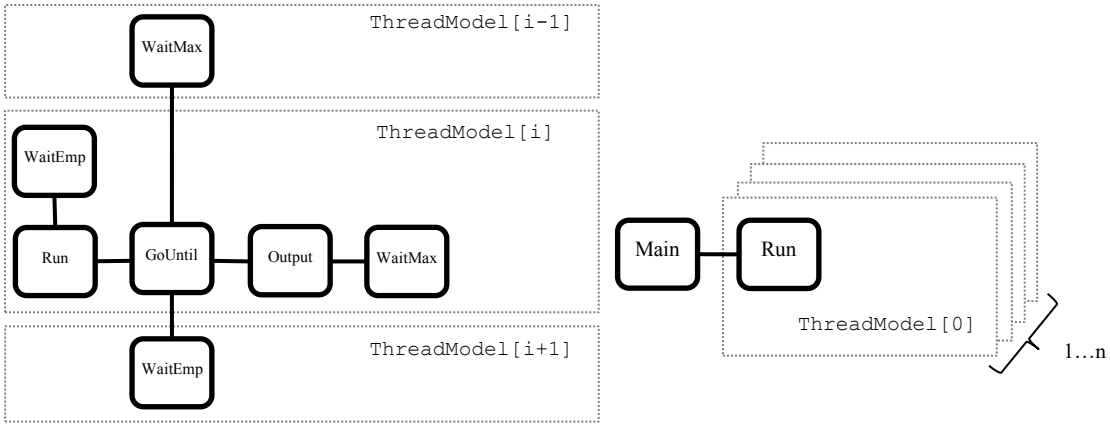


Рисунок 4.6 – Конструювання моделі з Петрі-об’єктів.

Після побудови моделі були заміряні та встановлені параметри моделі, близькі до спостережуваних у експерименті з реальним комп’ютерним ресурсом. Дані параметри представлені у таблиці 4.1.

Таблиця 4.1 – Основні параметри налаштування моделі в експерименті ($d = 0{,}001$ мс).

Зміст параметра	Реалізація параметра у моделі	Значення параметра
Кількість ядер	Кількість маркерів у позиції "cores"	2, 4, 8 або 16
Кількість подій у частині моделі, що виконується одним потоком	Кратність дуги рівна k/n	5, 10, 20, 50 або 100
Кількість потоків	Кратність дуги рівна n	$k/(k/n)$
Час виконання методу `input()` на одну подію	Часова затримка переходу "input"	$d \cdot 3 \cdot k/n$, мс
Час створення та запуску одного потоку	Часова затримка переходу "createThread"	$d \cdot 500 \cdot n$, мс
Час блокування/ розблокування монітору	Часова затримка переходу "lock" / "unlock"	$d \cdot 10$, мс
Ймовірність настання події output в поточний момент	Ймовірність переходу "actOut"	$\frac{1}{k/n} = n/k$
Ймовірність не настання події output в поточний момент	Ймовірність переходу "noActOut"	$1 - n/k$
Ліміт буфера зовнішніх подій	Поле "maxsize" класу `Control`	3, 10, 20, 50 або 100

Набір параметрів, що впливає на ефективність алгоритму складається зі складності підзадач в одному потоці, кількості ядер процесора та ліміту буфера

зовнішніх подій. Вплив цих параметрів досліджується при різній складності моделі.

Для дослідження впливу складності підзадач у потоці на час виконання було проведено експеримент із зазначеними параметрами моделі. Результати моделювання, отримані у випадку використання 2 ядер для пошуку оптимальної складності однієї підзадачі, представлені на рисунку 4.7. Найменший час виконання досягається, якщо кількість подій, оброблених одним потоком, дорівнює 10 при збільшенні складності моделі від 200 до 1000 подій, що відповідає реальній роботі на ресурсах комп'ютера (див. рис. 4.3). Таким чином, модель може бути використана для пошуку найкращих параметрів для запуску паралельного алгоритму. Абсолютні значення часу виконання відрізняються через недостатньо точну оцінку часових затримок на окремих інструкціях алгоритму.

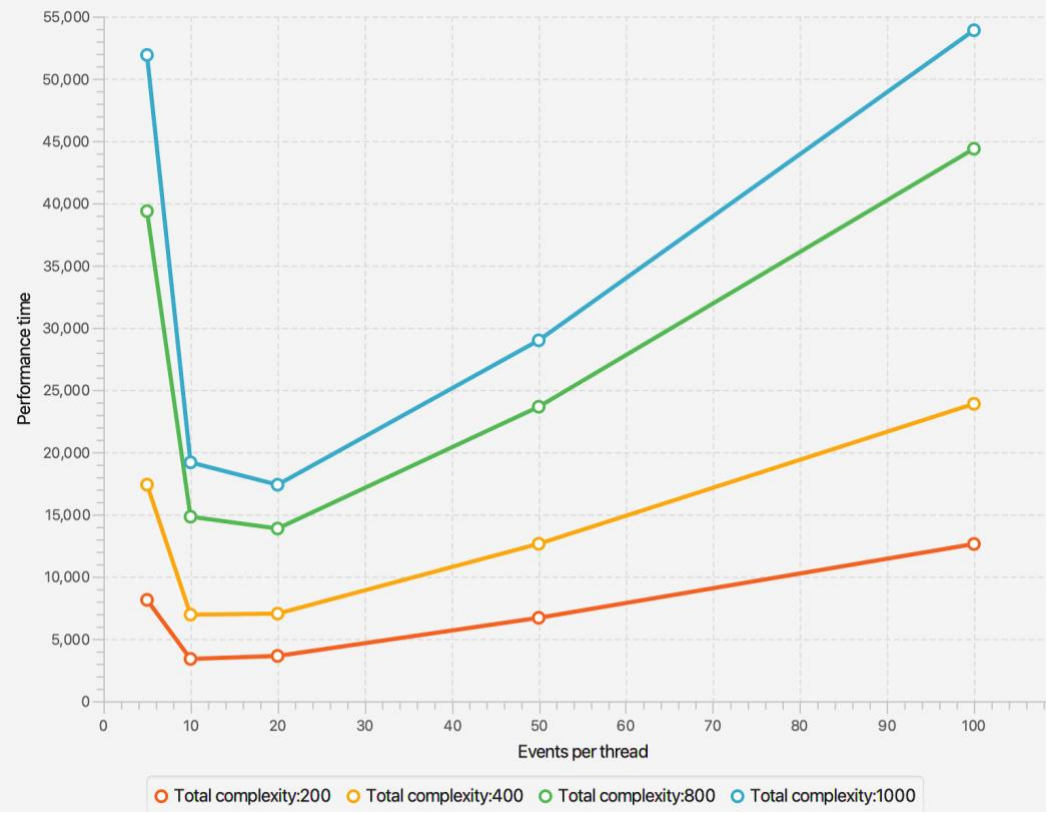


Рисунок 4.7 – Вплив складності підзадач на час виконання паралельного алгоритму.

Дослідження впливу кількості ядер та складності підзадач на час виконання при загальній кількості подій, що дорівнює 400, представлено на рисунку 4.8. При збільшенні кількості ядер до 4 ядер час виконання значно зменшується. Наступне збільшення ядер призводить до зміни оптимальної складності потоку до 20.

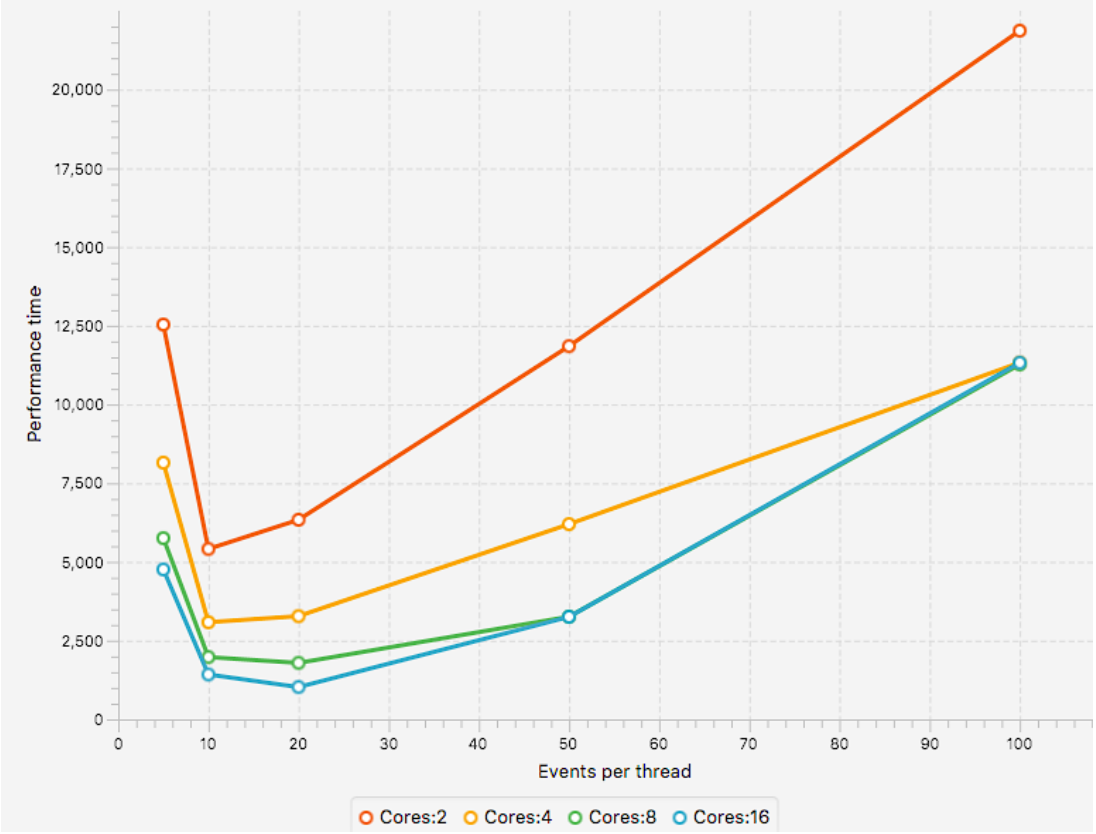
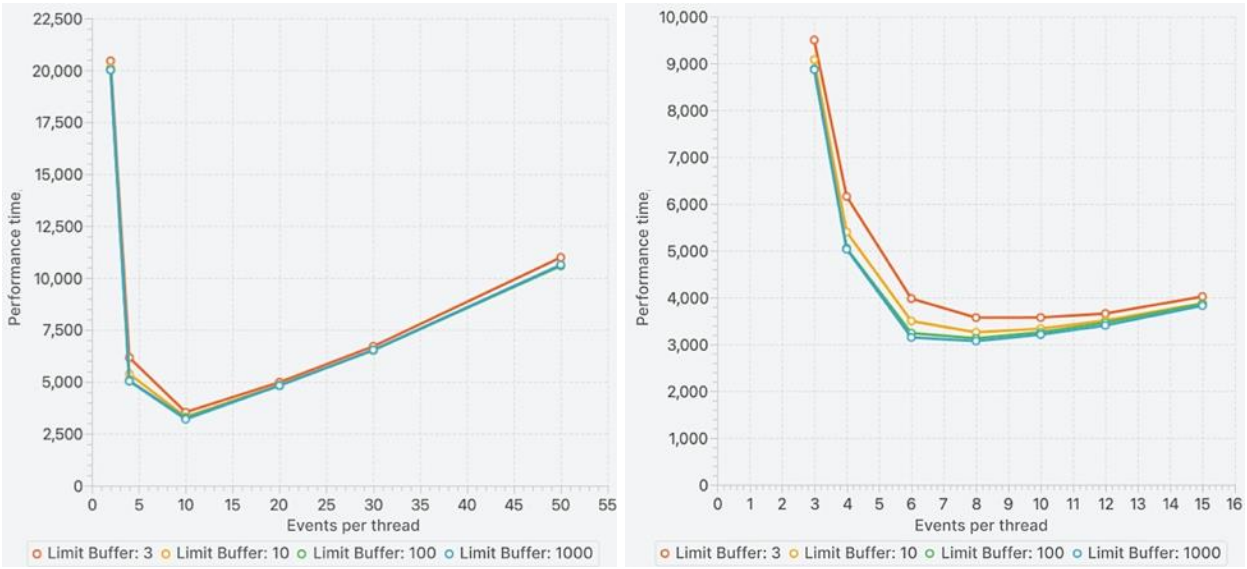


Рисунок 4.8 – Вплив кількості ядер на час виконання паралельного алгоритму.

Детальне дослідження впливу обмеження (ліміту) буфера зовнішніх подій та складності підзадач у потоці на час виконання паралельного алгоритму імітації представлено у роботах [67, 68]. У відповідності до покрокового алгоритму оптимізації спершу визначалось оптимальне значення ліміту буфера зовнішніх подій. Результати моделювання, отримані при ресурсі обсягом в 2 ядра та при складності моделі у 400 подій, представлені на рисунку 4.9. Можна помітити, що збільшення значення ліміту від 100 не спричиняє значного зменшення часу, адже криві для 100 і 1000 подій накладаються. З точки зору завантаженості обчислювального ресурсу, оптимальним значенням ліміту буфера зовнішніх подій є 100 подій.

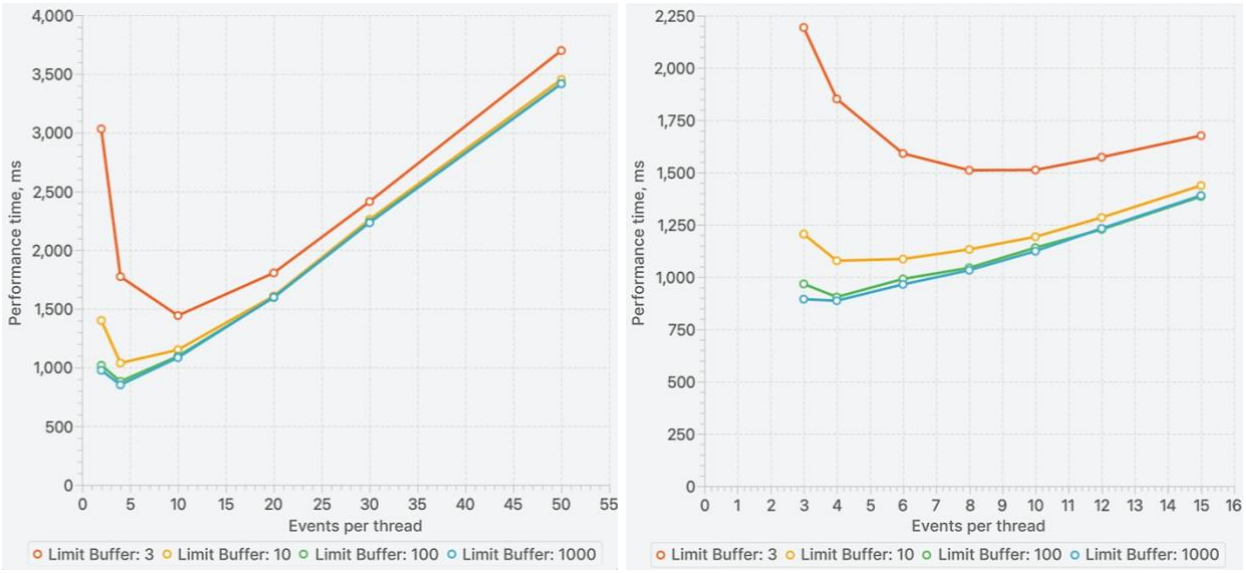
Після визначення оптимального значення першого параметра можливим став пошук оптимального значення складності підзадачі у потоці. На рисунку 4.9(а) видно, що мінімальний час виконання спостерігається, коли складність підзадачі складає 10 подій. Унаслідок деталізації дослідження в околі оптимуму (рис. 4.9(б)), вивлено, що найменше значення часу виконання алгоритму досягається при 8 подіях у потоці.



(а) (б)

Рисунок 4.9 – Вплив ліміту буфера зовнішніх подій та складності підзадачі у потоці на час виконання алгоритму в моделі: (а)загальний графік, (б) деталізований графік.

Результати аналогічного дослідження, але вже з реальною програмою, наведені на рисунку 4.10. Встановлено, що оптимальний час виконання алгоритму досягається, коли складність підзадачі складає 4 події, а обмеження буфера зовнішніх подій становить 100. Тож значення отримані з моделі та з програми досить близькі.



(а) (б)

Рисунок 4.10 – Вплив ліміту буфера зовнішніх подій та складності підзадачі у потоці на час виконання алгоритму в реальній програмі: (а)загальний графік, (б) деталізований графік.

Щоб оцінити наскільки різниця в результатах дослідження на моделі та реальній програмі впливає на час виконання алгоритму, було здійснено додаткові прогони програми із встановленим значенням параметрів: складність підзадачі – 4 події та складність підзадачі – 8 подій. Ліміт приймав значення у 100 подій. У таблиці 4.2 представлені усереднені по 4 прогонам результати обох експериментів. Помітно, що різниця в часі виконання невелика. Вплив похибки при визначенні оптимального значення на ефективність алгоритму складає лише 6%. При необхідності для точного визначення значень слід провести уточнюючий експеримент на реальній програмі, оскільки модель є спрощеним представленням програми, і оцінка часових затримок була достатньо грубою. Проте враховуючи, що модель дозволяє проводити безліч експериментів і тим самим обмежити область пошуку оптимальних значень, можна стверджувати про полегшення та прискорення процесу налаштування параметрів паралельних обчислень.

Таблиця 4.2 – Швидкодія алгоритму при оптимальних параметрах, знайдених експериментально та за допомогою методу.

Складність підзадачі	Ліміт буфера	Час виконання, мс
4	100	1025,625
8	100	1086,5

Таким чином, модель забезпечує комплексний аналіз параметрів алгоритму. Крім того, візуальне представлення паралельного алгоритму за допомогою Петрі-об’єктної моделі сприяє глибокому розумінню коду. У конкретному прикладі алгоритму імітації дискретно-подійної системи під час побудови моделі було виявлено два надлишкові фрагменти коду.

4.2 Пул потоків

Еволюційний алгоритм оптимізації параметрів паралельних обчислень був апробований на моделі пулу потоків. Опис моделі пулу потоків був наведений раніше у підрозділі 2.5.3.

Спершу проведено експеримент з заміром часу виконання реальної програми, що реалізує пул потоків з встановленими параметрами. Обчислювальна складаність загальної задачі в даному експерименті склала $2 \cdot 10^9$ елементарних операцій. Значення параметрів пулу потоків, а саме кількість потоків та підзадач, встановлювались такими, якими міг би задати їх програміст у реальному житті. У першому експерименті встановлено 2 потоки, виходячи з кількості ядер процесору комп’ютера, на якому проводились заміри, та 10000 підзадач у кожному потоці. Таким чином, обчислювальна складність кожної підзадачі склала 100000 елементарних операцій. У другому і третьому експерименті обчислювальна складність однієї підзадачі склала відповідно 8000 і 20000 елементарних операцій. Результати дослідження наведені у таблиці 4.3.

Таблиця 4.3 – Швидкодія пулу потоків

Кількість потоків	Кількість підзадач у потоці	Час виконання, мс
2	10000	769,645
50	5000	696,604
100	1000	713,229

Після проведених експериментів з реальною програмою було застосовано еволюційний алгоритм визначення оптимальних параметрів на Петрі-об'єктній моделі пулу потоків для обчислення задачі обсягом у $2 \cdot 10^9$ елементарних операцій. Область допустимих значень для потоків встановлена від 2 до 26 потоків, а параметр кількості підзадач набував значень: $\{2^0, 2^1, 2^2, \dots, 2^8\}$. У результаті роботи еволюційного алгоритму визначено, що оптимальними параметрами пулу потоків для заданої задачі є 13 потоків та 2^8 підзадач. Проміжні результати роботи алгоритму наведені в додатку Г. З отриманими значеннями параметрів було проведено експеримент на реальній програмі, внаслідок чого обчислено наскільки зменшився час виконання алгоритму відносно початкових значень (табл. 4.4).

Таблиця 4.4 – Оцінка зменшення часу виконання алгоритму після застосування еволюційного методу.

Час виконання алгоритму, мс		Зменшення часу виконання алгоритму
З параметрами, заданими програмістом	З параметрами, отриманими еволюційним метод	
769,645	633,254	18%
696,604		9%
713,229		11%

Отже, внаслідок проведеного дослідження, встановлено, що в середньому еволюційний алгоритм поліпшив час виконання алгоритму на 12,7%, що підтверджує його коректність та ефективність.

4.3 Висновки до розділу 4

У розділі наведено приклад пошуку оптимальних параметрів методом оптимізації, описаним у розділі 3. Паралельний алгоритм імітації, взятий для прикладу, є складним як за кількістю обчислювальних дій, так і за механізмами взаємодії підзадач, які забезпечують злагоджені дії усіх частин моделі.

Теоретично та експериментально доведено наявність впливу параметрів паралельного алгоритму на швидкодію його виконання. Встановлено, що параметром, від якого у найбільшій мірі залежить час виконання алгоритму – це складність одного фрагменту моделі, який запускається на одночасне виконання. Виконано дослідження оптимального значення параметру при зростанні складності моделі.

Побудована Петрі-об’єктна модель паралельного алгоритму та виконано пошук оптимальних параметрів на моделі експериментально. Знайдені оптимальні значення достатньо точно відповідають таким, що були виявлені при експериментуванні з паралельним алгоритмом в реальних умовах. Отримані результати свідчать про коректність пошуку оптимальних параметрів на моделі.

Виконано пошук оптимальних параметрів пулу потоків на моделі за допомогою еволюційного методу. У ході дослідження виявлено, що знайдені за допомогою еволюційного методу параметри пулу потоків сприяли зменшенню часу виконання алгоритму у порівнянні із параметрами, встановленими до застосування методу.

ВИСНОВКИ

У дисертації вирішено важливе для розвитку інформаційних технологій наукове завдання підвищення ефективності використання паралельних обчислень в інформаційній технології за рахунок моделювання паралельних обчислень та оптимізації їх параметрів. Петрі-об’єктні моделі дають змогу відтворити структуру програми та її поведінку.

Результатами дисертаційного дослідження є такі наукові та практичні результати:

1. Розроблено технологію моделювання паралельних обчислень на основі Петрі-об’єктного підходу, що надає можливість скоротити ресурсні витрати при розробці паралельних алгоритмів, і, на відміну від існуючих, дає змогу відтворити деталізовано структуру паралельної програми та механізми взаємодії одночасно виконуваних частин програми з урахуванням часових затримок на виконання обчислювальних дій та стохастичності захоплення обчислювального ресурсу і спрощує процес побудови моделі за рахунок тиражування фрагментів програми зі схожою функціональністю.

2. Удосконалено моделі базових механізмів синхронізації паралельних обчислень за рахунок підвищення точності відтворення, що забезпечує достатньо високу точність результатів моделювання.

3. Розроблено типові фрагменти мереж Петрі, що реалізують механізми багатопотокової технології Java, використання яких прискорює розробку моделі паралельного алгоритму за рахунок зменшення кількості помилок та зменшення загальної кількості елементів, необхідних для розробки моделі.

4. Запропоновано метод оптимізації параметрів паралельних обчислень на основі експериментального дослідження Петрі-об’єктної моделі обчислень, що забезпечує ефективне використання обчислювальних ресурсів і, на відміну від існуючих підходів, дає змогу проводити експериментальне дослідження ефективності паралельних обчислень на моделі замість експериментування на реальній програмі.

5. Досліджено точність моделей, побудованих розробленим методом, та доведено їх високу якість (похибка до 8%) .

6. Розроблено програмне забезпечення Parallel Program Simulation (PPS) для моделювання та оптимізації параметрів паралельних обчислень на основі Петрі-об’єктного моделювання.

7. Розроблено модель обчислень паралельного алгоритму імітації та виконано його дослідження з метою пошуку оптимальних параметрів паралельних обчислень. За результатами порівняння отриманих результатів з результатами експериментального дослідження програми обчислень в реальних умовах виявлено високу точність (похибка 6%) визначення оптимальних значень параметрів алгоритму, які забезпечують його найбільшу швидкодію.

Отже, якісні моделі паралельних обчислень можуть вирішити багато проблем, які виникають при розробці паралельних обчислень: визначити кількість процесів для досягнення найбільшої швидкодії, порівняти альтернативні способи взаємодії процесів, дослідити вплив параметрів обчислень на їх швидкодію, а також виявити в коді помилки та вдосконалити код.

Метод оптимізації, який розроблено в дисертації, та технологія моделювання можуть бути в подальшому розвинуті в середовище проєктування та аналізу багатопотокових програм.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mena A. S. Formal Methods for Concurrent Systems. Available at: https://xebia.com/blog/formal-methods-for-concurrent-systems/ (Accessed 06 August 2023).
2. Intel Guide for Developing Multithreaded Application (2011) https://www.intel.com/content/dam/develop/external/us/en/documents/gdma-2-165938.pdf (Accessed 03 August 2023).
3. Gustafson, J.L. (2011) Amdahl's Law. In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. Available at: https://doi.org/10.1007/978-0-387-09766-4_77 (Accessed 03 August 2023).
4. Gustafson, J.L. (2011) Gustafson's Law. In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. Available at: https://doi.org/10.1007/978-0-387-09766-4_78 (Accessed 03 August 2023).
5. Popov, G. (2010). Calculation of the acceleration of parallel programs as a function of the number of threads. ICCOMP'10: Proceedings of the 14th WSEAS international conference on Computers: part of the 14th WSEAS CSCC multiconference - Volume II, pp 411–414.
6. Jenkov.com. Jenkov J. (2021) Multithreading Costs. http://tutorials.jenkov.com/java-concurrency/costs.html (Accessed 03 August 2023).
7. Stetsenko I.V., Pavlov O.A., Dyfuchyna O. (2021) Parallel algorithm development and testing using Petri-object simulation. *International Journal of Parallel, Emergent and Distributed Systems*, 36(6), 549-564. Taylor and Francis Ltd. ISSN 1744-5779. https://doi.org/10.1080/17445760.2021.1955113
8. Stetsenko I.V., Dyfuchyna O. (2020) Thread Pool parameters tuning using simulation. *Advances in Intelligent Systems and Computing*, 938, 78-89. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-030-16621-2_8

9. De Boer, F.S., Grabe, I., Jaghoori, M.M., Stam, A., Yi, W. (2009) Modeling and analysis of thread-pools in an industrial communication platform. In: Breitman K., Cavalcanti A. (eds) International Conference on Formal Engineering Methods ICFEM 2009: Formal Methods and Software Engineering, pp.367-386. Springer-Verlag Berlin Heidelberg.
10. Renew. Available at: http://www.renew.de/ (Accessed 29 November 2023)
11. CPNTools. Available at: http://cpntools.org/ (Accessed 29 November 2023).
12. CPN IDE. Available at: https://cpnide.org/ (Accessed 8 December 2023).
13. Peterson J. (1981) Petri Nets Theory and the Modelling of Systems. Prentice-Hall, New Jer-sey.
14. Kavi, K., Moshtaghi, A., Chen, Dj. (2002) Modeling Multithreaded Applications Using Petri Nets. *International Journal of Parallel Programming,* 30(5), 353–371.
15. Katayama, T., Nakamura, H., Kita, Y. (2014) Proposal of a Supporting Method for Debugging to Reproduce Java Multi-threaded Programs by Petri-net. *Journal of Robotics, Networking and Artificial Life* 1(3), 207-211.
16. Giebas D., Wojszczyk R. (2020) Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis. *Applied Sciences* 10 (2), 532 DOI: 10.3390/app10020532
17. Ferscha A., Haring G. (1991) Petri net based modelling of parallel programs executing on distributed memory multiprocessor systems. *Periodica Polytechnica Electrical Engeneering*, 95(9), 199-219.
18. CAPSE. Computer Aided Parallel Software Engineering. Available at: https://informatik.univie.ac.at/forschung/projekte/projekt/232/ (Accessed 03 August 2023).
19. Ferscha A. (1992) A Petri net approach for performance oriented parallel program design. *Journal of Parallel and Distributed Computing*, 15 (3), 188-206.
20. Погорілий С.Д., Вітель Д.Ю. (2013) Використання мереж Петрі для проектування паралельних застосувань. *Проблеми програмування*, 2, 32-40.

21. Wolfmann, A. G. H., & Giusti, A. E. D. (2015) The PN-PEM framework: A Petri Net based parallel execution model. *Journal of Computer Science & Technology, 15* (2).
22. van der Werf J.M.E.M., Polyvyanyy A. (2020) The Information Systems Modeling Suite. In: Janicki R., Sidorova N., Chatain T. (eds) Application and Theory of Petri Nets and Concurrency. PETRI NETS 2020. *Lecture Notes in Computer Science*, vol. 12152. Springer, Cham.
23. Gold, R. (2004) Petri Nets in Software Engineering. *Arbeitsberichte – Working Papers* 5, Technische Hochschule Ingolstadt (THI).
24. ISO/IEC 15909-1:2004 Systems and software engineering — High-level Petri nets — Part 1: Concepts, definitions and graphical notation. Available at: https://www.iso.org/standard/38225.html (Accessed 03 August 2023).
25. Owe O., Yu I.C. (2014) Deadlock detection of active objects with synchronous and asynchronous method calls. In: Norsk Informatikkonferanse (NIK) OPJ/PKP, Halden, Norway.
26. Software Verify LTD. Smarter tools for better software, https://www.softwareverify.com/products/#threads (Accessed 03 August 2023)
27. Chen, Z. *et el*. (2015) MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications. SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. DOI: 10.1109/SC.2014.46
28. Elnashar, A. I., El-Zoghdy, S. F. (2015) An Algorithm for Static Tracing of Message Passing Interface Programs Using Data Flow Analysis. *International Journal of Computer Network and Information Security* 7(1), 1-8.
29. Malinowski, A. (2015) Modern platform for parallel algorithm testing: Java on Intel Xeon Phi. *International Journal Technology and Computer Science* 9, 8-14.
30. Ivanenko P., Doroshenko A. (2014) Method of Authomated Generation of Autotuners for Parallel Programs. *Cybernetics and Systems Analysis*, 50(3), 465-475. http://dx.doi.org/10.1007/s10559-014-9635-3

31. Іваненко П.А. (2018) Методи автоматизації створення автотюнерів для паралельних програм: дисертація.
32. Стеценко, И.В. (2011) Формальное описание систем средствами Петри-объектных моделей. *Вісник Національного технічного університету України «Київський політехнічний інститут». Інформатика, управління та обчислювальна техніка* 53, 74-81.
33. Стеценко, И.В. (2011) Теоретические основы Петри-объектного моделирования систем. *Математичні машини і системи* 4, 136-148.
34. Дифучин, А.Ю., Стеценко, І.В., Жаріков, Е.В. (2021) Граматика мови візуального програмування Петрі-об'єктних моделей. *Проблеми програмування* 4, 82-94. https://doi.org/10.15407pp2021.04.082
35. Petri C.A. (1962). Kommunikation mit Automaten. Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik 2. Universität Bonn. 128 Seiten.
36. Stetsenko I.V. (2012) State equations of stochastic timed Petri nets with informational relations. *Cybernetics and Systems Analysis* 48 (5), 784–797.
37. Stetsenko I.V., Dyfuchyn A. (2020) Petri-object Simulation: Technique and Software. *Information, Computing and Intelligent Systems* 1, 51-59 (2020). ISSN 2708-4930 https://doi.org/10.20535/2708-4930.1.2020.216057
38. ISO/IEC 19501:2004 Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2. Available at: https://www.iso.org/standard/32620.html (Accessed 06 August 2023).
39. Семеренко, В. П. (2018) Технології паралельних обчислень: навчальний посібник. Вінниця : ВНТУ. 104 с.
40. Rauber T., Runger G. (2012) *Parallel Programming: for multicore and Clusters Systems: Second edition.* Springer.
41. Holub A. (2000) *Taming Java Threads*. Apress, Berkeley, CA

42. von Praun, C. (2004) Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs, PhD thesis, Swiss Federal Institute of Technology, Zurich.
43. Liao H. (2013) Concurrency Bugs in Multithreaded Software: Modeling and Analyzing Using Petri Nets. *Discrete Event Dynamic Systems* 23(2), 157–195.
44. Xiang D. (2017) Detecting Data Inconsistency Based on the Un-folding Technique of Petri Nets. *IEEE Transactions on Industrial Informatics* 13(6), 2995 - 3005.
45. Xu Z. (2020). PVcon: Localizing Hidden Concurrency Errors With Prediction and Verification, *IEEE Access* 8, 165373-165386.
46. Westergaard M. (2012) Verifying Parallel Algorithms and Programs Using Coloured Petri Nets. *Lecture Notes in Computer Science* 7400, 146-168. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35179-2_7
47. José C. Cunha, João Lourenço, Vitor Duarte. (April 2001) Debugging of parallel and distributed programs. Conference: Parallel program development for cluster computing
48. Стеценко І.В., Дифучина О.Ю. (2017) Моделювання паралельних обчислень стохастичними мережами Петрі. *Вісник Національного технічного університету України «Київський політехнічний інститут». Інформатика, управління та обчислювальна техніка*, 66, 27-31.
49. Дифучина О.Ю. (2018) Тестування паралельних програм на моделях. Математичне та імітаційне моделювання систем. МОДС 2018: тези доповідей Тринадцятої міжнародної науково-практичної конференції. Чернігів: ЧНТУ. С.231-234.
50. Стеценко І.В., Дифучина О.Ю. (2018) Програмне забезпечення моделювання дискретно-подійних систем. Тези доповідей п’ятої міжнародної науково-практичної конференції «Управління розвитком технологій». К.: КНУБА. С.97-98.

51. Stetsenko I.V., Dyfuchyna O. (2019) Simulation of multithreaded algorithms using Petri-object models. *Advances in Intelligent Systems and Computing*, 754, 391- 401. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-319-91008-6_39
52. Дифучина, О. Ю. (2019) Аналіз ефективності використання паралельних обчислень в інформаційній технології : магістерська дис. : 126 Інформаційні системи та технології. Київ, 2019. 90 с.
53. Дифучина О.Ю., І.В.Стеценко. (2019) Критерій ефективності використання паралельних обчислень в інформаційній технології. Матеріали III всеукраїнської науково-практичної конференції молодих вчених та студентів «Інформаційні системи та технології управління» (ІСТУ-2019). Київ.: НТУУ «КПІ ім. Ігоря Сікорського», 20-22 листопада 2019 р. С.6-8.
54. Java™ Platform, Standard Edition 8. API Specification. Interface Lock. Available at: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html (Accessed 08 August 2023).
55. Jenkov.com. Jenkov J. (2020) Race Conditions and Critical Sections. Available at: https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html (Accessed 03 August 2023).
56. The Java Tutorials. Guarded Blocks. Available at: https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html (Accessed 09 August 2023).
57. Java™ Platform, Standard Edition 8. API Specification. Class Object. Available at: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-- (Accessed 09 August 2023).
58. The Java Tutorials. Deadlock. Available at: https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html (Accessed 03 August 2023).

59. The Java Tutorials. Lock Objects. Available at: https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html (Accessed 03 August 2023).
60. Jenkov.com. Jenkov J. (2023) JMH - Java Microbenchmark Harness. https://jenkov.com/tutorials/java-performance/jmh.html (Accessed 29 November 2023).
61. Стеценко І.В. (2010) Моделювання систем: навч. посібник. Черкаси : ЧДТУ, 2010. 399 с.
62. Law A. M. (2014) Simulation Modeling and Analysis. 5th Edition. McGraw-Hill Inc. New York. 800 pages.
63. Stochastic Simulation in Java (SSJ). Available at: https://github.com/umontreal-simul/ssj/tree/master (Accessed 22 August 2023).
64. Parallel Program Simulation (PPS) Available at: https://github.com/sashadif/PPS (Accessed 26 December 2023).
65. Стеценко И.В. (2012) Алгоритм имитации Петри-объектной модели *Математичні машини і системи*, 1, 154-165.
66. GitHub. PetriObjLib. Available at: https://github.com/StetsenkoInna/PetriObjLib (Accessed 03 August 2023).
67. Дифучина О.Ю. (2023) Метод оптимізації параметрів паралельних обчислень. *Технічні науки та технології,* 3(33), 130-140. (Фахове видання, «Б»). ISSN 2411-5363. https://doi.org/10.25140/2411-5363-2023-3(33)-130-14
68. Дифучина О.Ю. (2023) Метод оптимізації параметрів паралельних обчислень на основі Петрі-об'єктного моделювання. МОДС 2023: тези доповідей Вісімнадцятої міжнародної конференції (13 – 15 листопада 2023 р., м. Чернігів). Чернігів : НУ «Чернігівська політехніка», 2023. – 74 с.

ДОДАТОК А. МЕТОДИ-КРІЕЙТОРИ МЕРЕЖ ПЕТРІ ШАБЛОНІВ МОДЕЛЮВАННЯ БАГАТОПОТОКОВИХ ОБЧИСЛЕНЬ

Метод-кріейтор фрагменту мережі Петрі, що моделює цикл for:

```
public static PetriNet CreateNetFor(int m) throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("numIteration",m));
     d_P.add(new PetriP("computaionStart",0));
     d_P.add(new PetriP("computationEnd",0));
     d_P.add(new PetriP("P4",0));
     d_P.add(new PetriP("P5",0));
     d_P.add(new PetriP("for",1));
     d_T.add(new PetriT("forStart",0.0));
     d_T.add(new PetriT("T2",0.0));
     d_T.add(new PetriT("forFinish",0.0));
     d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(2),d_T.get(1),1));
     d_In.add(new ArcIn(d_P.get(3),d_T.get(2),m));
     d_In.add(new ArcIn(d_P.get(5),d_T.get(0),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
     d_Out.add(new ArcOut(d_T.get(2),d_P.get(4),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(5),1));
     PetriNet d_Net = new PetriNet("For",d_P,d_T,d_In,d_Out);
     PetriP.initNext();
     PetriT.initNext();
     ArcIn.initNext();
     ArcOut.initNext();

     return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює цикл while:

```
public static PetriNet CreateNetWhile() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("whileStart",0));
     d_P.add(new PetriP("conditiion",1));
     d_P.add(new PetriP("computationStart",0));
     d_P.add(new PetriP("computationEnd",0));
```

```
        d_P.add(new PetriP("whileIsFinished",0));
        d_T.add(new PetriT("whileStart",0.0));
        d_T.get(0).setPriority(9);
        d_T.add(new PetriT("whileFinish",0.0));
        d_T.add(new PetriT("T3",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(0),1));
        d_In.get(2).setInf(true);
        d_In.add(new ArcIn(d_P.get(3),d_T.get(2),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(0),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(4),1));
        PetriNet d_Net = new PetriNet("While",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює оператор if-then-else:

```
public static PetriNet CreateNetIf() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("condition",1));
        d_P.add(new PetriP("ifStart",0));
        d_P.add(new PetriP("",0));
        d_P.add(new PetriP("",0));
        d_T.add(new PetriT("If",0.0));
        d_T.get(0).setPriority(5);
        d_T.add(new PetriT("else",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.get(0).setInf(true);
        d_In.add(new ArcIn(d_P.get(1),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(1),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(4),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(4),1));
        PetriNet d_Net = new PetriNet("If",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
```

```
     ArcOut.initNext();

     return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює створення, початок та завершення роботи потоку:

```
public static PetriNet CreateNetCreateThread() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("P1",0));
     d_P.add(new PetriP("P2",0));
     d_P.add(new PetriP("P3",0));
     d_P.add(new PetriP("runStart",0));
     d_P.add(new PetriP("runEnd",0));
     d_P.add(new PetriP("P6",0));
     d_P.add(new PetriP("P7",0));
     d_T.add(new PetriT("createThread",0.0));
     d_T.add(new PetriT("start",0.0));
     d_T.add(new PetriT("end",0.0));
     d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
     d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(4),d_T.get(2),1));
     d_In.add(new ArcIn(d_P.get(5),d_T.get(2),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
     d_Out.add(new ArcOut(d_T.get(2),d_P.get(6),1));
     PetriNet d_Net = new
PetriNet("CreateThread",d_P,d_T,d_In,d_Out);
     PetriP.initNext();
     PetriT.initNext();
     ArcIn.initNext();
     ArcOut.initNext();

     return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює призупинку потоку та очікування на завершення виконання дій потоку:

```
public static PetriNet CreateNetSleep() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("P1",0));
```

```
        d_P.add(new PetriP("P2",0));
        d_P.add(new PetriP("P3",0));
        d_P.add(new PetriP("P4",0));
        d_P.add(new PetriP("P5",0));
        d_P.add(new PetriP("threadB.runEnd",0));
        d_T.add(new PetriT("sleepStart",0.0));
        d_T.add(new PetriT("sleepDelay",0.0));
        d_T.add(new PetriT("sleepEnd",0.0));
        d_T.add(new PetriT("threadB.join",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(3),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(4),1));
        PetriNet d_Net = new PetriNet("Sleep",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює захоплення монітора об’єкта, для якого викликано синхронізований метод:

```
public static PetriNet CreateNetSyncMethod() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("P1",0));
        d_P.add(new PetriP("P2",0));
        d_P.add(new PetriP("monitor",1));
        d_P.add(new PetriP("P4",0));
        d_P.add(new PetriP("P5",0));
        d_P.add(new PetriP("owner",0));
        d_T.add(new PetriT("syncMethodStart",0.0));
        d_T.add(new PetriT("syncMethodEnd",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(1),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(5),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(4),1));
```

```
     PetriNet d_Net = new
PetriNet("SyncMethod",d_P,d_T,d_In,d_Out);
     PetriP.initNext();
     PetriT.initNext();
     ArcIn.initNext();
     ArcOut.initNext();

     return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює захоплення локера об’єкта з урахуванням можливості багаторазового захоплення:

```
public static PetriNet CreateNetReentrantLock() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
     ArrayList<PetriP> d_P = new ArrayList<>();
     ArrayList<PetriT> d_T = new ArrayList<>();
     ArrayList<ArcIn> d_In = new ArrayList<>();
     ArrayList<ArcOut> d_Out = new ArrayList<>();
     d_P.add(new PetriP("lockingStart",0));
     d_P.add(new PetriP("P2",0));
     d_P.add(new PetriP("owner",0));
     d_P.add(new PetriP("holdCount",0));
     d_P.add(new PetriP("lock",1));
     d_P.add(new PetriP("lockingEnd",0));
     d_P.add(new PetriP("P7",0));
     d_T.add(new PetriT("lock",0.0));
     d_T.add(new PetriT("asOwner",0.0));
     d_T.add(new PetriT("release",0.0));
     d_T.get(2).setPriority(9);
     d_T.add(new PetriT("unlock",0.0));
     d_In.add(new ArcIn(d_P.get(3),d_T.get(2),1));
     d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(0),d_T.get(1),1));
     d_In.add(new ArcIn(d_P.get(6),d_T.get(3),1));
     d_In.add(new ArcIn(d_P.get(6),d_T.get(2),1));
     d_In.add(new ArcIn(d_P.get(4),d_T.get(0),1));
     d_In.add(new ArcIn(d_P.get(2),d_T.get(3),1));
     d_In.add(new ArcIn(d_P.get(2),d_T.get(1),1));
     d_In.get(7).setInf(true);
     d_Out.add(new ArcOut(d_T.get(3),d_P.get(5),1));
     d_Out.add(new ArcOut(d_T.get(2),d_P.get(5),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
     d_Out.add(new ArcOut(d_T.get(3),d_P.get(4),1));
     d_Out.add(new ArcOut(d_T.get(0),d_P.get(2),1));
     d_Out.add(new ArcOut(d_T.get(1),d_P.get(1),1));
     PetriNet d_Net = new
PetriNet("ReentrantLock",d_P,d_T,d_In,d_Out);
     PetriP.initNext();
     PetriT.initNext();
     ArcIn.initNext();
```

```
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює захоплення локера без очікування його звільнення:

```
public static PetriNet CreateNetTryLock() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("lockingStart",0));
        d_P.add(new PetriP("P2",0));
        d_P.add(new PetriP("P3",0));
        d_P.add(new PetriP("lockingEnd",0));
        d_P.add(new PetriP("lock",1));
        d_P.add(new PetriP("owner",0));
        d_T.add(new PetriT("tryLockTrue",0.0));
        d_T.get(0).setPriority(9);
        d_T.add(new PetriT("tryLockFalse",0.0));
        d_T.add(new PetriT("unlock",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(2),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(5),1));
        PetriNet d_Net = new PetriNet("TryLock",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює негайне (без очікування звільнення) захоплення локера з урахуванням можливості багаторазового захоплення:

```
public static PetriNet CreateNetTryReentrantLock() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
```

```
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("lockingStart",0));
        d_P.add(new PetriP("P2",0));
        d_P.add(new PetriP("P3",0));
        d_P.add(new PetriP("lockingEnd",0));
        d_P.add(new PetriP("lock",1));
        d_P.add(new PetriP("owner",0));
        d_P.add(new PetriP("holdCount",0));
        d_T.add(new PetriT("tryLockTrue",0.0));
        d_T.get(0).setPriority(9);
        d_T.add(new PetriT("tryLockFalse",0.0));
        d_T.add(new PetriT("unlock",0.0));
        d_T.add(new PetriT("tryAsOwner",0.0));
        d_T.get(3).setPriority(5);
        d_T.add(new PetriT("release",0.0));
        d_T.get(4).setPriority(9);
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(6),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(3),1));
        d_In.get(9).setInf(true);
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(5),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(6),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(3),1));
        PetriNet d_Net = new
PetriNet("TryReentrantLock",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює синхронізований доступ потоку до спільного значення:

```
public static PetriNet CreateNetSharedAccess() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
```

```
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("P1",0));
        d_P.add(new PetriP("P2",0));
        d_P.add(new PetriP("P3",0));
        d_P.add(new PetriP("P4",0));
        d_P.add(new PetriP("P5",0));
        d_P.add(new PetriP("P6",0));
        d_P.add(new PetriP("lock",1));
        d_P.add(new PetriP("owner",0));
        d_T.add(new PetriT("lock",0.0));
        d_T.add(new PetriT("read",0.0));
        d_T.add(new PetriT("modify",0.0));
        d_T.add(new PetriT("write",0.0));
        d_T.add(new PetriT("unlock",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(6),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(7),d_T.get(4),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(4),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(5),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(6),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(7),1));
        PetriNet d_Net = new
PetriNet("SharedAccess",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює очікування за умовою:

```
public static PetriNet CreateNetWaitNotify() throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("",0));
        d_P.add(new PetriP("condCheck",0));
        d_P.add(new PetriP("",0));
        d_P.add(new PetriP("",0));
        d_P.add(new PetriP("",0));
```

```
        d_P.add(new PetriP("condition",0));
        d_P.add(new PetriP("signalTo",0));
        d_P.add(new PetriP("monitor",1));
        d_P.add(new PetriP("",0));
        d_P.add(new PetriP("owner",0));
        d_P.add(new PetriP("waiting",0));
        d_P.add(new PetriP("signalFrom",0));
        d_P.add(new PetriP("",0));
        d_T.add(new PetriT("syncStart",0.0));
        d_T.add(new PetriT("check",0.0));
        d_T.get(1).setPriority(9);
        d_T.add(new PetriT("notifyAll",0.0));
        d_T.add(new PetriT("syncEnd",0.0));
        d_T.add(new PetriT("catchMonitor",0.0));
        d_T.add(new PetriT("wait",0.0));
        d_T.add(new PetriT("getSignal",0.0));
        d_T.get(6).setPriority(9);
        d_T.add(new PetriT("isNotWaiting",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(5),d_T.get(1),1));
        d_In.get(4).setInf(true);
        d_In.add(new ArcIn(d_P.get(7),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(8),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(7),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(5),1));
        d_In.add(new ArcIn(d_P.get(10),d_T.get(6),1));
        d_In.add(new ArcIn(d_P.get(11),d_T.get(7),1));
        d_In.add(new ArcIn(d_P.get(11),d_T.get(6),1));
        d_In.add(new ArcIn(d_P.get(9),d_T.get(5),1));
        d_In.add(new ArcIn(d_P.get(9),d_T.get(3),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(3),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(4),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(6),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(9),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(9),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(5),d_P.get(7),1));
        d_Out.add(new ArcOut(d_T.get(5),d_P.get(10),1));
        d_Out.add(new ArcOut(d_T.get(6),d_P.get(8),1));
        d_Out.add(new ArcOut(d_T.get(7),d_P.get(12),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(7),1));
        PetriNet d_Net = new
PetriNet("WaitNotify",d_P,d_T,d_In,d_Out);
        PetriP.initNext();
        PetriT.initNext();
        ArcIn.initNext();
```

```
        ArcOut.initNext();

        return d_Net;
}
```

Метод-кріейтор фрагменту мережі Петрі, що моделює пул потоків:

```
public static PetriNet CreateNetThreadPool(int w, int k) throws
ExceptionInvalidNetStructure, ExceptionInvalidTimeDelay {
        ArrayList<PetriP> d_P = new ArrayList<>();
        ArrayList<PetriT> d_T = new ArrayList<>();
        ArrayList<ArcIn> d_In = new ArrayList<>();
        ArrayList<ArcOut> d_Out = new ArrayList<>();
        d_P.add(new PetriP("poolSt",0));
        d_P.add(new PetriP("numThreads",0));
        d_P.add(new PetriP("P3",0));
        d_P.add(new PetriP("tasks",w));
        d_P.add(new PetriP("P5",0));
        d_P.add(new PetriP("runSt",0));
        d_P.add(new PetriP("runEnd",0));
        d_P.add(new PetriP("P8",0));
        d_P.add(new PetriP("P9",0));
        d_P.add(new PetriP("poolEnd",0));
        d_P.add(new PetriP("cores",0));
        d_T.add(new PetriT("poolCreate",0.0));
        d_T.add(new PetriT("execute",0.0));
        d_T.get(1).setPriority(9);
        d_T.add(new PetriT("shutdown",0.0));
        d_T.add(new PetriT("runSt",0.0));
        d_T.add(new PetriT("runEnd",0.0));
        d_T.add(new PetriT("awaitTermination",0.0));
        d_In.add(new ArcIn(d_P.get(0),d_T.get(0),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(3),d_T.get(1),1));
        d_In.add(new ArcIn(d_P.get(4),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(1),d_T.get(3),1));
        d_In.add(new ArcIn(d_P.get(6),d_T.get(4),1));
        d_In.add(new ArcIn(d_P.get(7),d_T.get(5),w));
        d_In.add(new ArcIn(d_P.get(8),d_T.get(5),1));
        d_In.add(new ArcIn(d_P.get(2),d_T.get(2),1));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(1),k));
        d_Out.add(new ArcOut(d_T.get(0),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(2),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(1),1));
        d_Out.add(new ArcOut(d_T.get(1),d_P.get(4),1));
        d_Out.add(new ArcOut(d_T.get(3),d_P.get(5),1));
        d_Out.add(new ArcOut(d_T.get(4),d_P.get(7),1));
        d_Out.add(new ArcOut(d_T.get(5),d_P.get(9),1));
        d_Out.add(new ArcOut(d_T.get(2),d_P.get(8),1));
           for (PetriT tr: d_T){
               d_In.add(new ArcIn(d_P.get(d_P.size()-1),tr,1));
               d_Out.add(new ArcOut(tr, d_P.get(d_P.size()-1),1));
```

```
        }
     PetriNet d_Net = new
PetriNet("ThreadPool",d_P,d_T,d_In,d_Out);
     PetriP.initNext();
     PetriT.initNext();
     ArcIn.initNext();
     ArcOut.initNext();

     return d_Net;
    }
```

ДОДАТОК Б. ЛІСТИНГ КОДУ ПЕТРІ-ОБ'ЄКТНОЇ МОДЕЛІ ПОТОКІВ, ЩО КОНФЛІКТУЮТЬ ЗА ЗАХОПЛЕННЯ ЛОКЕРІВ

```
package Friends;

import PetriObj.ExceptionInvalidNetStructure;
import PetriObj.ExceptionInvalidTimeDelay;
import PetriObj.PetriP;
import PetriObj.PetriSim;
public class FriendPetri extends PetriSim {

    public FriendPetri(String name, int loop,
                    int cores, double delay, double ratio)
                        throws ExceptionInvalidNetStructure,
                                            ExceptionInvalidTimeDelay {
      super(Nets.CreateNetFriend (name, loop, cores, delay, ratio));
    }
    public FriendPetri(String name)
                        throws ExceptionInvalidNetStructure,
                                    ExceptionInvalidTimeDelay {
        super(Nets.CreateNetFriend (name, 1000, 2, 100, 0.01));
    }

    public FriendPetri(String name, int loop,
                     double delay, double ratio)
                        throws ExceptionInvalidNetStructure,
                                            ExceptionInvalidTimeDelay {
        super(Nets.CreateNetFriend(name, loop, delay, ratio));
    }

    public FriendPetri(String name, int loop)
           throws ExceptionInvalidNetStructure,
                            ExceptionInvalidTimeDelay {
        super(Nets.CreateNetFriendUsingCores(name, loop,
                           2, 100, 0.8)); // 2 cores, delay=100,x=0.8
    }


    public int getFailures(){
        return this.getNet().getListP()[6].getMark() ;
    }

    public int getBows(){
        return this.getNet().getListP()[8].getMark();
    }


    public void addFriend(FriendPetri other) {
        this.getNet().getListP()[7] = other.getNet().getListP()[2];
//lockOther = lock
        this.getNet().getListP()[15] = other.getNet().getListP()[15]; //
coresOther = cores

    }
```

```
    public PetriP getLock(){
        return this.getNet().getListP()[15];
    }
    public void addFriendWithoutCores(FriendPetri other) {
        //lockOther = lock
        this.getNet().getListP()[7] = other.getNet().getListP()[2];
    }

    public double getResult() {
      return (double) getFailures() / ((double) getFailures() + (double)
getBows());
    }

     public void print() {
            System.out.println(this.getName() + " ,failures " +
getFailures() + ", bows " + getBows());
     }
}

package Friends;

import java.util.logging.Level;
import java.util.logging.Logger;

public class BowLoop implements Runnable {

    private Friend a;
    private Friend b;
    private long delay;

    public BowLoop(Friend bower, Friend bowee, long delaySleep) {
        a = bower;
        b = bowee;
        delay = delaySleep;
    }

    @Override
    public void run() {
        for (int j = 0; j < 100; j++) {
            try {
                Thread.sleep(delay);
            } catch (InterruptedException ex) {

Logger.getLogger(TestTryLockFriends.class.getName()).log(Level.SEVERE,
null, ex);
            }
            a.bow(b);
        }
    }
}

package Friends;

import PetriObj.*;
import java.util.ArrayList;
```

```
public class TestFriendsModel {

    public static void main(String[] args)
           throws ExceptionInvalidNetStructure,
                ExceptionInvalidTimeDelay, InterruptedException {

       PetriObjModel model = createModel(16, 20, 100, 0.01);
       model.setIsProtokol(true);
       System.out.println("freq of failures =
                           "+getResult (model, 1000000));

       for(PetriSim sim: model.getListObj()){
            ((FriendPetri)sim).print();
       }

    }

    public static double getResult(PetriObjModel model,
                                                    double time){
        model.go(time);
        double res=0.0;
        for(PetriSim sim: model.getListObj()){
            res+=((FriendPetri)sim).getResult();
        }
        return res/model.getListObj().size();
    }

    public static PetriObjModel createModel(int numFriends, int cores,
double delay, double ratio) throws ExceptionInvalidNetStructure,
ExceptionInvalidTimeDelay {
        ArrayList<PetriSim> list = new ArrayList<>();

        int num = numFriends;
        FriendPetri[] friends = new FriendPetri[num];
        for (int j = 0; j < num; j++) {
            friends[j] = new FriendPetri("Friend_" + j,
                                         1000, cores, delay,
                                    ratio);
        }
        for (int j = 0; j < num; j++) {
            for (int i = 0; i < num; i++) {
                if (i != j) {
                    friends[j].addFriend(friends[i]);
                }
            }
        }

        for (int j = 0; j < num; j++) {
            list.add(friends[j]);
        }

        return new PetriObjModel(list);
    }
}
```

ДОДАТОК В. ЛІСТИНГ КОДУ ЕВОЛЮЦІЙНОГО АЛГОРИТМУ ПОШУКУ ОПТИМАЛЬНИХ ПАРАМЕТРІВ

```
import PetriObj.ExceptionInvalidNetStructure;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class EvolutionAlg {

    private int numInPopulation;
    private final int progon;
    private final Random random;
    private int numRepeat;
    private ArrayList<Individual> searchHistory;

    public EvolutionAlg() {
        this(20, 1);
    }

    public EvolutionAlg(int numIndividuals, int progon) {
        this.numInPopulation = numIndividuals;
        this.numRepeat = 10*this.numInPopulation;
        this.progon = progon;
        random = new Random();
        searchHistory = new ArrayList<>();
    }

    public Individual[] createPopulationByCombinatorial() { // обидва
параметри рівномірно розкидуються і варіюються кожний з кожним
        // 0 - 01234, 1-01234,2-01234, 3-01234, 4-01234
        System.out.println("--------START population---------");
        Individual[] population = new Individual[numInPopulation];
        int num = (int) Math.sqrt(numInPopulation) + 1; // кількість
значень від min до max що варіюються

        int next = 0;
        for (int i = 0; i < num; i++) {
            for (int k = 0; k < num; k++) {
                population[next] = new Individual(progon, i, k, num -
1);
                population[next].calcFit(progon);
                if (next == 0) {
                    getSearchHistory().add(population[next]);
                } else {
                    if (!population[next].
                            isIdentical(searchHistory)) {
                        getSearchHistory().add(population[next]);
                    }
                }
                next++;
                if (next >= numInPopulation) {
                    break;
                }
```

```
            }
            if (next >= numInPopulation) {
                break;
            }
        }
       // System.out.println("Sorting by fit ... ");
       sortIndividuals(population);
        return population;
    }

    public Individual[] createNextPopulation(Individual[]
previousPopulation, double s, int variation) {
        Individual[] newPopul = new Individual[numInPopulation]; //
популяція завжди відсортована за зростанням фітнес-функції
        System.out.println("searching the best elements  ...");
        newPopul[0] = previousPopulation[0];
        newPopul[0].print();
        int k = 1;
        // відкидаємо за умовою "набагато гірший", а не за кількістю

        while (previousPopulation[k].getFit() -
previousPopulation[0].getFit() < s) {
            newPopul[k] = previousPopulation[k];
            previousPopulation[k].print();
            k++;
            if (k == numInPopulation) {
                break;
            }
        }
        int bests = k;// динамічно змінюється кількість найкращих

        System.out.println(bests + " elements have been identified as
the best ");
        // створюємо нові, щоб набрати популяцію
        if(bests==numInPopulation){
            System.out.println("All elements are the best ");
            for(int j=0; j<numInPopulation; j++){
                 newPopul[j] = previousPopulation[j];
            }
        }  else if(bests == 1){
            System.out.println("Only one element is the best");
            for (int j = k; j < numInPopulation; j++) {
                   newPopul[j] = previousPopulation[0].
                           childIndividual(previousPopulation[1],
progon, variation); //з наступним схрещуємо
                        if (!newPopul[j].
                                isIdentical(searchHistory)) {
               //             System.out.println(" unique has been found
by mutation ");
                            getSearchHistory().add(newPopul[j]);
                            newPopul[j].calcFit(progon);
                        } else{
              //              System.out.println("the same.....,
best=1");
                           newPopul[j] = previousPopulation[j]; //
залишаємо старий варіант, оскільки нового не знайдено
```

```
                        }
                }
            } else
            //if (k < numInPopulation)
             {
                int half = (bests) / 2;
                int one, other;
               boolean isBreak = false;
                for (int j = k; j < numInPopulation; j++) {
                    isBreak = false;
                    for(int i=0; i<numRepeat; i++){ // шукаємо унікальний
елемент

                        // щоб різні числа були згенеровані одне шукаємо
серед парних, а інше - серед непарних
                        one = 2 * random.nextInt(half + 1);
                        other = 1 + 2 * (random.nextInt(half + 1));
                        if (other >= bests) {
                            other = 1;
                        }
                        newPopul[j] = previousPopulation[one].
childIndividual(previousPopulation[other], progon, variation); //
схрещування

                        if (!newPopul[j].isIdentical(searchHistory)) {
                //          System.out.println("unique has been found ");
                            newPopul[j].calcFit(progon);
                            getSearchHistory().add(newPopul[j]);
                //          System.out.println("repeat has been break
"+i);
                            isBreak = true;
                            break;
                        }
                    }
                    if (!isBreak) { // якщо не вдалось відшукати унікальний
                        newPopul[j] =
previousPopulation[j].trivialMutation(); // +- 1 обов'язково
                       if (!newPopul[j].isIdentical(searchHistory)) {
                //          System.out.println(" unique has been found by
mutation ");
                            getSearchHistory().add(newPopul[j]);
                            newPopul[j].calcFit(progon);
                        } else{
                     //        System.out.println("the same....., repeat =
"+(numRepeat-1));
                           newPopul[j] = previousPopulation[j]; // залишаємо
старий варіант, оскільки нового не знайдено
//                          if(numRepeat>2)
//                           numRepeat=numRepeat/2; // скорочуємо кількість
спроб для пошуку унікального
                       }
                    }
                }
            }
//позбавитись від однакових в популяції
        if(bests==numInPopulation && bests>2){
```

```
            numInPopulation/=2;
        }
        sortIndividuals(newPopul);
        return newPopul;
    }

    public boolean isParametersIdentical(int[] parameters){
        boolean s;
        for(Individual ind: searchHistory){
            s=true;
            for(int j=0; j<parameters.length; j++){
                    if(parameters[j]!=ind.getParams()[j]){
                        s=false;
                        break;
                    }
            }
            if(s){
//                System.out.print("Indiidiual
"+this.getParams()[0]+"\t"+this.getParams()[1]+"\t "+
//                        "is identical to the "+
ind.getParams()[0]+"\t"+ind.getParams()[1]);
                return true;
            }
        }
        return false;
    }

    public static Individual[] sortIndividualsByParameters(Individual[]
population) {
        Arrays.sort(population, (o1, o2) -> {
            if (o1.getParams()[0] > o2.getParams()[0]) {

                return 1; //сортування у зростаючому порядку
            } else if (o1.getParams()[0] < o2.getParams()[0]) {
                return -1;
            } else {
                 if(o1.getParams()[1] > o2.getParams()[1]){
                     return 1;
                 } else if(o1.getParams()[1] < o2.getParams()[1]){
                     return -1;
                 }
                return 0;
            }
        });
        return population;
    }

    public static Individual[] sortIndividuals(Individual[] population)
{
        Arrays.sort(population, (o1, o2) -> {
            if (o1.getFit() > o2.getFit()) {
                return 1; //сортування у зростаючому порядку
            } else if (o1.getFit() < o2.getFit()) {
                return -1;
            } else {
                return 0;
```

```
            }
        });
        return population;
    }

    public Individual evolution() {
        int steps=20; // максимальна кількість кроків (популяцій)
еволюційного алгоритму
        ArrayList<Double> fitValues = new ArrayList<>();
        Individual[] popul = this.createPopulationByCombinatorial();
        this.print(popul);
        int counter = 0;
        fitValues.add(popul[0].getFit());

        for (int n = 0; n < steps; n++) {
            System.out.println("----------NEXT----------");
            popul = this.createNextPopulation(popul, 0.1 *
fitValues.get(n) / (n + 1), 2); // поріг для відкидання з кожним
поколінням зменшується
//  (n + 1) delta для мутації - зменшується
            this.print(popul);
            fitValues.add(popul[0].getFit());  // n+1 значення у списку

            System.out.println("Fitness value improvement    " +
(fitValues.get(n + 1) - fitValues.get(n)));

            if ((n > 0) && (fitValues.get(n + 1) - fitValues.get(n) <
0.1 * fitValues.get(0) / (n + 1))) { // має бути підряд кілька разів, а
не взагалі в усьому пошуку
                counter++;
            } else {
                counter = 0; // тому починаємо відлік наново
            }

            if (counter > steps/2) {
                System.out.println("----------THE BEST HAS BEEN FOUND---
-------");

                break;
            }
        }
        if (counter <= steps/2) {
            System.out.println("----------THE BEST IN THE LAST
POPULATION----------");
        }

        return popul[0];
    }

    public static void warmedUp(int n) {
        for (int j = 0; j < n; j++) {
            Math.random(); // warmed up
        }
    }
```

```
    public static void main(String[] args) throws
ExceptionInvalidNetStructure {
        int[] mins = {2, 0}; //[0]-threads, [1]-tasks
        int[] maxs = {26, 8};
        Individual.setMaxs(maxs); //ці значення використовуватимуться
для створення нових елементів та мутації
        Individual.setMins(mins);
        EvolutionAlg ev = new EvolutionAlg(20, 1); // 1 progon
        //warmedUp(1000000); // розігрів для моделі не потрібний, але
потрібний при експериментуванні з реальною програмою

        Individual best = ev.evolution();
        System.out.println("----------THE BEST----------");
        best.print();
        System.out.println("----------SEARCH HISTORY----------
"+ev.getSearchHistory().size());
        ev.getSearchHistory().sort((o1, o2) -> {
            if (o1.getParams()[0] > o2.getParams()[0]) {

                return 1; //сортування у зростаючому порядку
            } else if (o1.getParams()[0] < o2.getParams()[0]) {
                return -1;
            } else {
                 if(o1.getParams()[1] > o2.getParams()[1]){
                     return 1;
                 } else if(o1.getParams()[1] < o2.getParams()[1]){
                     return -1;
                 }
                return 0;
            }
        });
        ev.printHistory();

    }

    public void print(Individual[] popul) {
        Individual[] printedArr = Arrays.copyOf(popul, popul.length);
        sortIndividualsByParameters(printedArr); // для зручності
сприйняття інформації
        for (int j = 0; j < numInPopulation; j++) {
            printedArr[j].print();
        }
    }

     public void printHistory() {
        for (int j = 0; j < searchHistory.size(); j++) {
            searchHistory.get(j).print();
        }
    }

    public ArrayList<Individual> getSearchHistory() {
        return searchHistory;
    }

    public void setSearchHistory(ArrayList<Individual> searchHistory) {
        this.searchHistory = searchHistory;
```

```
    }
}

import PetriObj.ExceptionInvalidNetStructure;
import ThreadPoolTest.PoolModel;
import java.util.ArrayList;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Individual {

    private static int numParams;
    private final int[] params;
    private static int[] mins;
    private static int[] maxs;
    private double fit;
    private final Random random=new Random();

    public Individual() { // БЕЗ розрахунку фіт-значення
        numParams = mins.length;
        params = new int[numParams];
        for (int i = 0; i < numParams; i++) {
            if (maxs[i] == mins[i]) {
                this.params[i] = mins[i];
            } else {
                this.params[i] = mins[i] + random.nextInt(maxs[i] -
mins[i]);
            }
        }
        fit = Double.MAX_VALUE;
    }

    public Individual(int progon) { // З розрахунком фіт-значення
        numParams = mins.length;
        params = new int[numParams];
        for (int i = 0; i < numParams; i++) {
            if (maxs[i] == mins[i]) {
                this.params[i] = mins[i];
            } else {
                this.params[i] = mins[i] + random.nextInt(maxs[i] -
mins[i]);
            }
        }
        calcFit(progon);
    }
    // цей метод створює за зростанням одного індексу j
    public Individual(int progon, int j, int numInPopulation) { //
рівномірно розподіляємо
        numParams = mins.length;
        params = new int[numParams];
        for (int i = 0; i < numParams; i++) {
            if (maxs[i] == mins[i]) {
                this.params[i] = mins[i];
            } else {
```

```
                this.params[i] = mins[i] + (int) Math.rint(j * (maxs[i]
- mins[i]) / numInPopulation);
            }
        }
    //    calcFit(progon);
//        System.out.print("[ " + j + " ]");
//        this.print();
    }

    // цей метод для випадку двох параметрів і варіюються два індекси j
k
    public Individual(int progon, int j, int k, int num) { // рівномірно
розподіляємо
        numParams = mins.length;
        params = new int[numParams];
        if (maxs[0] == mins[0]) {
            this.params[0] = mins[0];
        } else {
            this.params[0] = mins[0] + (int) Math.rint(j * (maxs[0] -
mins[0]) / num);
        }
        if (maxs[1] == mins[1]) {
            this.params[1] = mins[1];
        } else {

            this.params[1] = mins[1] + (int) Math.rint(k * (maxs[1] -
mins[1]) / num);
        }
  //      calcFit(progon);

//        System.out.print("[ "+j+","+k+" ]");
       // this.print();
    }

     public Individual childIndividual(Individual other, int progon, int
variation) {
        Individual child = new Individual(progon);

        for (int j = 0; j < getNumParams(); j++) {
                child.params[j] = (this.getParams()[j] +
other.getParams()[j]) / 2;
                child.params[j] = generateMutationValue(child.params[j],
                        Math.min(this.getParams()[j],
other.getParams()[j]),
                        Math.max(this.getParams()[j],
other.getParams()[j]),
                        variation);
        }

     //   child.calcFit(progon);
//        System.out.print("child "+child.isIdentical(history));
//       child.printParams();
        return child;
    }
```

```
         public int generateMutationValue(int value, int min, int max,
int variation) {

            double rr = random.nextDouble();
            int delta = Math.max(1, (max - min) / variation); //
величина варіації при мутації 1...10
            int d = random.nextInt(delta + 1); // величина мутації
            if (rr < 0.33) {
                    value += d;
                    if (value > max) {
                        value = max;
                    }
                } else {
                    if (rr < 0.66) {
                        value -= d;
                        if (value < min) {
                            value = min;
                        }

                    }
                }
        return value;
    }
    public Individual trivialMutation() { // величина мутації = 1
        Individual child = new Individual();
        double rr;
        for (int j = 0; j < getNumParams(); j++) {
            rr = random.nextDouble();

            if (rr < 0.5) {
                child.getParams()[j] =this.getParams()[j] + 1;
            } else {
//                if (rr < 0.66) {
                 child.getParams()[j] =this.getParams()[j] - 1;
                  if (child.getParams()[j] < 1) {
                      child.getParams()[j] = 1;
                  }

//                }
            }
        }
        return child;
    }

     public final void calcFit(int progon) {
        double f = 0;
        int totalComplexity = 2000000000;
        int cores = 4;
        double tMod = 100;

        for (int i = 0; i < progon; i++) {
            try {
                f += PoolModel.getTimePerformance(cores, tMod,
this.params[0], (int)Math.pow(2, this.params[1]),
totalComplexity);//[0]-threads , [1]-tasks//
```

```
            } catch (ExceptionInvalidNetStructure ex) {
Logger.getLogger(Individual.class.getName()).log(Level.SEVERE, null,
ex);
            }
        }
        fit = f / progon;
    //    fit = f / progon / 1000000000; // time in seconds

    }

    public double getFit() {

        return fit;
    }

    public int[] getParams(){
        return params;
    }
    public void printParams(){

System.out.println(this.getParams()[0]+"\t"+this.getParams()[1]);
    }

    public void print(){

System.out.println(this.getParams()[0]+"\t"+this.getParams()[1]+"\t
"+this.getFit());
    }

    public boolean isIdentical(ArrayList<Individual> hystoric){
        boolean s;
        for(Individual ind: hystoric){
            s=true;
            for(int j=0; j<params.length; j++){
                    if(this.params[j]!=ind.params[j]){
                        s=false;
                        break;
                    }
            }
            if(s){
//                System.out.print("Individiual
"+this.getParams()[0]+"\t"+this.getParams()[1]+"\t "+
//                        "is identical to the "+
ind.getParams()[0]+"\t"+ind.getParams()[1]);
                return true;
            }
        }
        return false;
    }

    public static int[] getMins() {
        return mins;
    }
    public static void setMins(int[] aMins) {
        mins = aMins;
    }
```

```
    public static int[] getMaxs() {
        return maxs;
    }

    public static void setMaxs(int[] aMaxs) {
        maxs = aMaxs;
    }

    public static int getNumParams() {
        return numParams;
    }
}
```

ДОДАТОК Г. РЕЗУЛЬТАТИ РОБОТИ ЕВОЛЮЦІЙНОГО АЛГОРИТМУ ДЛЯ МОДЕЛІ ПУЛУ ПОТОКІВ

```
population: 40
run:
--------START population-------
--
2    0     686.1298960070574
2    1     661.4586215910435
2    2     609.6298886999609
2    4     651.6055536698145
2    5     638.8038674734757
2    6     651.2449468014713
2    8     642.751519861041
7    0     684.861974272231
7    1     689.112606158478
7    2     627.3077651274008
7    4     612.0973658822252
7    5     630.18473749323
7    6     641.4029434402859
7    8     631.493173672972
13   0     708.2269202327616
13   1     693.4948511019868
13   2     657.6127403936098
13   4     668.951541093724
13   5     637.0961186092292
13   6     637.0193991770866
----------NEXT----------
searching the best elements  ...
13   8     606.8775354296436
31   4     608.1630287898122
2    2     609.6298886999609
25   8     611.2493575027647
7    4     612.0973658822252
19   8     612.78453955436
31   0     623.8181674484796
7    2     627.3077651274008
25   4     628.6678208431117
7    5     630.18473749323
7    8     631.493173672972
19   1     633.261994211638
13   6     637.0193991770866
13   5     637.0961186092292
2    5     638.8038674734757
19   6     639.5265913179935
7    6     641.4029434402859
2    8     642.751519861041
19   2     643.5786740601409
31   1     643.9456714076102
20    elements    have    been
identified as the best
All elements are the best
2    2     609.6298886999609
2    5     638.8038674734757
2    8     642.751519861041
7    2     627.3077651274008
7    4     612.0973658822252
7    5     630.18473749323
7    6     641.4029434402859
7    8     631.493173672972
13   5     637.0961186092292
13   6     637.0193991770866
Fitness value improvement   0.0
----------NEXT----------
searching the best elements  ...
13   8     606.8775354296436
31   4     608.1630287898122
2    2     609.6298886999609
25   8     611.2493575027647
7    4     612.0973658822252
19   8     612.78453955436
31   0     623.8181674484796
7    2     627.3077651274008
25   4     628.6678208431117
7    5     630.18473749323
10    elements    have    been
identified as the best
All elements are the best
2    2     609.6298886999609
7    2     627.3077651274008
7    4     612.0973658822252
7    5     630.18473749323
13   8     606.8775354296436
Fitness value improvement   0.0
----------NEXT----------
searching the best elements  ...
13   8     606.8775354296436
31   4     608.1630287898122
2    2     609.6298886999609
25   8     611.2493575027647
7    4     612.0973658822252
5 elements have been identified
as the best
All elements are the best
2    2     609.6298886999609
7    4     612.0973658822252
```

```
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------NEXT----------
searching the best elements ...
13   8     606.8775354296436
31   4     608.1630287898122
2 elements have been identified
as the best
All elements are the best
13   8     606.8775354296436
31   4     608.1630287898122
Fitness value improvement   0.0
----------THE   BEST   HAS   BEEN
FOUND----------
----------THE BEST----------
13   8     606.8775354296436
```

ДОДАТОК Ґ. ПУБЛІКАЦІЇ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

1. Дифучина О.Ю. (2023) Метод оптимізації параметрів паралельних обчислень. *Технічні науки та технології,* 3(33), 130-140. (Фахове видання, «Б»). ISSN 2411-5363. https://doi.org/10.25140/2411-5363-2023-3(33)

2. Stetsenko I.V., Pavlov O.A., Dyfuchyna O. (2021) Parallel algorithm development and testing using Petri-object simulation. *International Journal of Parallel, Emergent and Distributed Systems*, 36(6), 549-564. Taylor and Francis Ltd. ISSN 1744-5779. https://doi.org/10.1080/17445760.2021.1955113

3. Stetsenko I.V., Dyfuchyna O. (2019) Simulation of multithreaded algorithms using Petri-object models. *Advances in Intelligent Systems and Computing*, 754, 391-401. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-319-91008-6_39

4. Stetsenko I.V., Dyfuchyna O. (2020) Thread Pool parameters tuning using simulation. *Advances in Intelligent Systems and Computing*, 938, 78-89. Springer, Cham. ISSN 2194-5365. https://doi.org/10.1007/978-3-030-16621-2_8

5. Стеценко І.В., Дифучина О.Ю. (2017) Моделювання паралельних обчислень стохастичними мережами Петрі. *Вісник Національного технічного університету України «Київський політехнічний інститут». Інформатика, управління та обчислювальна техніка*, 66, 27-31. (Фахове видання, «В»)

6. Дифучина О.Ю. (2023) Метод оптимізації параметрів паралельних обчислень на основі Петрі-об’єктного моделювання. МОДС 2023: тези доповідей Вісімнадцятої міжнародної конференції (13 – 15 листопада 2023 р., м. Чернігів). М-во освіти і науки України; Нац. Акад. наук України; Академія технологічних наук України; Інженерна академія України та ін. Чернігів: НУ «Чернігівська політехніка», 2023. С.25-28. http://ir.stu.cn.ua/123456789/29144

7. Дифучина О.Ю. (2018) Тестування паралельних програм на моделях. Математичне та імітаційне моделювання систем. МОДС 2018: тези доповідей Тринадцятої міжнародної науково-практичної конференції (м. Київ – с. Жукін, 25 червня – 29 червня 2018 р.). М-во осв. і наук. України, Нац. Акад. наук України, Академія технологічних наук України, Інженерна академія України та

ін. Чернігів: ЧНТУ, 2018. С.231-234. https://stu.cn.ua/wp-content/uploads/2021/04/mods18-p.pdf

8. Стеценко І.В., Дифучина О.Ю. (2018) Програмне забезпечення моделювання дискретно-подійних систем. Тези доповідей п’ятої міжнародної науково-практичної конференції «Управління розвитком технологій». К.: КНУБА, 2018. С.97-98. https://www.knuba.edu.ua/wp-content/uploads/2022/10/%D0%A2%D0%B5%D0%B7%D0%B8-2018.pdf

9. Дифучина О.Ю., Стеценко І.В. (2019) Критерій ефективності використання паралельних обчислень в інформаційній технології. Матеріали III всеукраїнської науково-практичної конференції молодих вчених та студентів «Інформаційні системи та технології управління» (ІСТУ-2019). Київ.: НТУУ «КПІ ім. Ігоря Сікорського», 20-22 листопада 2019 р. С.6-8.