

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

ГОНЧАРЕНКО ОЛЕКСАНДР ОЛЕКСІЙОВИЧ

УДК 004.72

ДИСЕРТАЦІЯ

МЕТОДИ ТА ЗАСОБИ ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ ТА
ЕФЕКТИВНОСТІ ТОПОЛОГІЙ КОМП'ЮТЕРНИХ СИСТЕМ

123 Комп'ютерна інженерія

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 _____

Науковий керівник: Волокита Артем Миколайович, к.т.н., доцент

Київ – 2025

ПОДЯКА

Перш ніж приступити до викладення матеріалу своєї дисертаційної роботи я хотів би подякувати всім, хто так чи інакше допомогли мені в її написанні.

По-перше, я хотів би подякувати своєму попередньому керівникові, д.т.н., професору Луцькому Георгію Михайловичу. З ним я починав свою роботу як науковець і саме завдяки його настановам дійшов туди, де є зараз. Він завжди умів підтримати в ці тяжкі часи і як наставник і як людина, надихнути на пошук нових ідей, допомогти з правильною розстановкою акцентів. Дякую вам за все, учителю! Спочивайте з миром.

По-друге, хочу подякувати керівнику нинішньому, к.т.н., доценту Волокиті Артему Миколайовичу. Скільки часу було проведено у розумових пошуках по безкрайнім полям науки, що навчило мене думати по-науковому, шукати нові ідеї, робити та оформлювати свої дослідження. Дякую вам за нашу спільну роботу і сподіваюсь, що в подальшому ми зробимо ще багато наукових відкриттів.

По-третє, хотів би подякувати рідній кафедрі Обчислювальної техніки та рідному Київському Політехнічному Інституту імені Ігоря Сікорського. Дякую вам за навчання, дякую вам за знання, що ви давали мені на бакалавраті, в магістратурі та в аспірантурі. Вже скоро я сподіваюсь приєднатись до вас не як здобувач, а як повноправний науковець, і разом із вами розвивати нашу рідну кафедру та наш університет.

І, звісно, хотів би подякувати рідному українському народу та нашій неньці Україні за можливість отримати безкоштовну вищу освіту, що дало мені змогу дійти до написання даної кваліфікаційної роботи. Як науковець я робитиму все що від мене залежить для розвитку української науки як в ключі мирному, щоб наступні покоління теж могли отримувати цей шанс і користуватись благами цивілізації, так і в ключі воєнному, для покращення нашої обороноздатності в умовах сучасної високотехнологічної війни.

Дякую вам усім!

АНОТАЦІЯ

Гончаренко О.О. Методи та засоби підвищення відмовостійкості та ефективності топологій комп'ютерних систем. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп'ютерна інженерія з галузі знань 12 – Інформаційні технології. – Національний Технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», Київ, 2025.

Робота присвячена розробці методу синтезу топологій на основі надлишкових кодів та методу синтезу ієрархічних топологій, що дозволяють підвищити відмовостійкість масштабованих високопродуктивних систем, а також покращити їх ефективність.

Розроблено нову математичну модель топології на основі надлишкового коду, що відрізняється від існуючих моделей використанням алфавіту, основи числення та довжини коду для визначення кількості альтернативних представлень довільного числа в заданій системі числення, та дозволяє прогнозувати максимальну кількість вершин з однаковим номером у графі, кількість вершин з унікальними (не-надлишковими) номерами.

Запропоновано новий спосіб формування імпліцитних кластерів в надлишкових топологіях, що відрізняється від існуючих використанням спеціальної багатовимірної матриці надлишкових представлень та кодування індексів в спеціальній системі числення та дозволяє формувати ребра між такими вершинами для топологій на основі кодів із певними співвідношеннями потужності алфавіту та основи числення.

Набув розвитку метод синтезу відмовостійких топологій на основі надлишкового коду, що відрізняється від існуючих використанням кодових перетворень, в тому числі послідовностей де Бруїна, в надлишкових системах числення та створенням нових зв'язків у таких топологіях за допомогою перетворень заміщення над кодами, які описують індекс альтернативного представлення в

багатовимірній матриці надлишкових представлень, що дозволяє синтезувати відмовостійкі топології заданого порядку, в тому числі з імпліцитними кластерами.

Запропоновано новий метод масштабування ієрархічних топологій, що відрізняється від існуючих використанням декартового добутку, деревовидних структур та рекурентного вкладення кластерів, що дозволяє поєднати відмовостійкі топології, синтезовані на основі надлишкового коду, із класичними топологіями, такими як гіперкуб та dragonfly.

Запропоновано новий спосіб моделювання відмов в топологіях, що відрізняється від існуючих використанням різних підходів до випадкового формування черги відмов, в тому числі з урахуванням коефіцієнту посередництва, та дозволяє при заданій кількості відмов вузлів аналізувати імовірність розриву зв'язності графа, підрахувати топологічні характеристики та їх зміну відносно початкового (безвідмовного) стану топології.

Для експериментального дослідження запропонованих методів було розроблено ряд інструментальних засобів з використанням мови *Python* та бібліотеки *NetworkX*.

Розроблено інструментальний засіб для моделювання характеристик топологій на основі бібліотеки *NetworkX*, який за рахунок запропонованого способу формування імпліцитних кластерів в надлишкових топологіях дозволяє дослідити топологічні характеристики графів, отриманих з використання запропонованих методів, та виконати їх порівняння із класичними топологіями, такими як гіперкуб, жирне дерево, dragonfly та dragonfly+, а також багатовимірні тори, включаючи топологію суперкомп'ютерна Fugaku.

Розроблено інструментальний засіб для моделювання відмов в топологіях, який є реалізацією запропонованого способу моделювання відмов в топологіях та дозволяє дослідити поведінку топологій в умовах наростаючого числа відмов і таким чином порівняти відмовостійкість запропонованих та існуючих графів.

Проведено експериментальне дослідження запропонованих методів, що включає в себе аналіз топологічних характеристик та аналіз відмовостійкості окремо для безпосередньо-зв'язаних (на основі надлишкового коду) та комутованих (ієрархічних) мереж. Розроблені методи продемонстрували суттєве підвищення

відмовостійкості та ефективності, дозволяючи покращити загальні топологічні характеристики, такі як мультиплікативна характеристика ступеня та діаметра (SD), для якої продемонстровано покращення в діапазоні 6.7-69.2%. Запропоновані рішення продемонстрували на 26,3% вищу відмовостійкість при 50% відмов для графів безпосередно-зв'язаних мереж і на 15,7% вищу при 40% відмов для комутованих мереж.

Розглянуті графи є конкурентоспроможними у порівнянні із популярними рішеннями в предметній сфері, такими як топологія жирного дерева та dragonfly. Порівняння із топологією суперкомп'ютера Fugaku (найефективніший на сьогодні суперкомп'ютер з точки зору тесту HPCG) показало значну перевагу розроблених рішень (покращення SD на 69,2%, краща топологічна ефективність на 103,4%).

Розроблені топологічні рішення можуть бути застосовані при розробці комп'ютерних систем з масовим паралелізмом, кластерних систем та датацентрів, а також комп'ютерних мереж, включаючи мережі, що керуються програмним забезпеченням.

Ключові слова: відмовостійкість, ефективність, топологія, мережа, математичне моделювання, ієрархічні структури, мережеві топологічні організації, живучість, програмний засіб, алгоритм, багаторівневі структури, високопродуктивні системи, паралельні обчислення, розподілені обчислення, масштабування, послідовності де Бруйна, надлишковий код.

ABSTRACT

Honcharenko O.O. Methods and tools of increasing the efficiency of scalable high-performance computing systems. - Qualified scientific work on the rights of the manuscript.

Dissertation for the degree of Doctor of Philosophy in the specialty 123 - Computer Engineering and 12 - Information Technologies. - National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2025.

The work is devoted to the development of a method for synthesizing topological organizations based on redundant codes and a method for synthesizing hierarchical topological organizations, which allow increasing the efficiency of scalable high-performance systems, as well as ensuring their fault tolerance.

A new mathematical model of topology based on redundant numeral code has been developed, which differs from existing models by using the alphabet, base of number, and code length to determine the number of alternative representations of an arbitrary number in a given numeral system, and allows predicting the maximum number of vertices with the same number in a graph, the number of vertices with unique (non-redundant) numbers

A new method of forming implicit clusters in redundant topologies is proposed, which differs from existing ones by using a special multidimensional matrix of redundant representations and encoding indices in a special number system and allows forming edges between such vertices for topologies based on codes with certain ratios of the power of the alphabet and the base of the number.

A method for synthesizing fault-tolerant topologies based on redundant numeral code has been developed, which differs from existing ones by using code transformations, including de Bruijn sequences, in redundant numeral systems and creating new connections in such topologies using exchange transformations over codes that describe the index of an alternative representation in a multidimensional matrix of redundant representations, which allows synthesizing fault-tolerant topologies of a given rank, including those with implicit clusters.

A new method for scaling hierarchical topologies is proposed, which differs from existing ones by using the Cartesian product, tree structures, and recurrent cluster nesting,

which allows combining fault-tolerant topologies synthesized based on redundant numeral code with classical topologies such as hypercube and dragonfly.

A new method of modeling failures in topological organizations is proposed, which differs from existing ones by using different approaches to random formation of a failure queue, including taking into account the betweenness centrality coefficient, and allows, for a given number of node failures, to analyze the probability of a graph disconnection, calculate topological characteristics and their change relative to the initial (fault-free) state of the topology.

For experimental research of the proposed methods, a number of tools were developed using the *Python* programming language and the *NetworkX* library.

A tool for modeling the characteristics of topologies based on the *NetworkX* library has been developed, which, due to the proposed method of forming implicit clusters in redundant topologies, allows us to investigate the topological characteristics of graphs obtained using the proposed methods and compare them with classical topologies, such as hypercube, fat tree, dragonfly and dragonfly+, as well as multidimensional tori, including the Fugaku supercomputer topology.

A tool for modeling failures in topological organizations has been developed, which is an implementation of the proposed method for modeling failures in topological organizations and allows you to study the behavior of topologies under conditions of an increasing number of failures and thus compare the fault tolerance of the proposed and existing graphs.

An experimental study of the proposed methods was conducted, which includes an analysis of topological characteristics and a fault tolerance analysis separately for directly connected (based on redundant numeral code) and switched (hierarchical) networks. The developed methods demonstrated a significant increase in fault tolerance and efficiency, allowing to improve general topological characteristics, such as the multiplicative characteristic of degree and diameter (SD), for which an improvement in the range of 6.7-69.2% was demonstrated. The proposed solutions demonstrated a 26.3% higher fault tolerance at 50% failures for graphs of directly connected networks and a 15.7% higher at 40% failures for switched networks.

The considered graphs are competitive in comparison with popular solutions in the subject area, such as fat tree topology and dragonfly. Comparison with the topology of the Fugaku supercomputer (the most efficient supercomputer today in terms of HPCG test) showed a significant advantage of the developed solutions (*SD* improvement by 69.2%, better topological efficiency by 103.4%).

The developed topological solutions can be applied in the development of massively parallel computing systems, cluster systems and data centers, as well as computer networks, including software-controlled networks.

Keywords: fault tolerance, efficiency, topology, network, mathematical simulation, hierarchical structures, network topological organizations, survivability, software tool, algorithm, multilevel structures, high performance computing, parallel computing, distributed computing, scaling, de Bruijn sequences, redundant numeral code.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковано основні наукові результати дисертації.

1. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2022). Extended DragonDeBruijn topology synthesis method. *International Journal of Computer Network and Information Security*, 9(6), 23. DOI: [10.5815/ijcnis.2022.06.03](https://doi.org/10.5815/ijcnis.2022.06.03) (Scopus) Q3
2. Volokyta, A., Loutskii, H., Honcharenko, O., Cherevatenko, O., Rusinov, V., Kulakov, Y., & Tsybulia, S. (2023). Fault Tolerance Exploration and SDN Implementation for de Bruijn Topology based on betweenness Coefficient. *Computer Network and Information Security*, 5 (pp. 1-17). DOI: [10.5815/ijcnis.2024.01.08](https://doi.org/10.5815/ijcnis.2024.01.08) (Scopus) Q3
3. Гончаренко, О., & Череватенко, О. (2021). СПОСОБИ МУЛЬТИКАНАЛЬНОЇ МАРШРУТИЗАЦІЇ В МЕРЕЖАХ НАДЛИШКОВОГО ДЕ БРУЙНА. *Технічні науки та технології*, (2 (24)), 123-130. DOI: [https://doi.org/10.25140/2411-5363-2021-2\(24\)-123-130](https://doi.org/10.25140/2411-5363-2021-2(24)-123-130) (Фахове видання) категорія Б
4. Гончаренко, О., & Волокита, А. (2024). МЕТОД СИНТЕЗУ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ З ІМПЛІЦИТНИМИ КЛАСТЕРАМИ НА ОСНОВІ ПЕРЕТВОРЕНЬ ДЕ БРУЙНА В НАДЛИШКОВИХ СИСТЕМАХ ЧИСЛЕННЯ. *Проблеми інформатизації та управління*, (4 (80)), 20-27. DOI: <https://doi.org/10.18372/2073-4751.80.19784> (Фахове видання) категорія Б

Апробація наукових результатів дисертації:

5. Rusinov, V., Honcharenko, O., Volokyta, A., Loutskii, H., Pustovit, O., & Kyrianov, A. (2023, March). Methods of Topological Organization Synthesis Based on Tree and Dragonfly Combinations. In *International Conference on Computer Science, Engineering and Education Applications* (pp. 472-485). Cham: Springer Nature Switzerland. DOI: 10.1007/978-3-031-36118-0_43 (Scopus) Q3
6. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2021). Topology synthesis method based on excess de Bruijn and dragonfly. In *Advances in Computer Science for Engineering and Education IV* (pp.

- 315-325). Springer International Publishing. DOI: [10.1007/978-3-030-80472-5_27](https://doi.org/10.1007/978-3-030-80472-5_27) (Scopus) Q4
7. Loutskii, H., Volokyta, A., Rehida, P., Honcharenko, O., & Thinh, V. D. (2021). Method for synthesis scalable fault-tolerant multi-level topological organizations based on excess code. In *Advances in Computer Science for Engineering and Education III 3* (pp. 350-362). Springer International Publishing. DOI: [10.1007/978-3-030-55506-1_32](https://doi.org/10.1007/978-3-030-55506-1_32)
 8. Loutskii, H., Volokyta, A., Rehida, P., Honcharenko, O., Ivanishchev, B., & Kaplunov, A. (2019, December). Increasing the fault tolerance of distributed systems for the Hyper de Bruijn topology with excess code. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)* (pp. 1-6). IEEE. DOI: [10.1109/ATIT49449.2019.9030487](https://doi.org/10.1109/ATIT49449.2019.9030487)
 9. Honcharenko, O., Volokyta, A., & Loutskii, H. (2024, August). Method of fault tolerant routing in distributed systems based on non-binary de brujin topology. In *The International Conference on Security, Fault Tolerance, Intelligence*. [Online] Available at: <https://icsfti-proc.kpi.ua/article/view/307020>

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	16
ВСТУП.....	17
РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ МЕТОДІВ ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ ТА ЕФЕКТИВНОСТІ КОМП'ЮТЕРНИХ СИСТЕМ	24
1.1. Визначення основних понять та аспектів високопродуктивних КС	24
1.2. Аналіз проблематики ефективності масштабованих КС	27
1.2.1. Обґрунтування проблематики масштабованих обчислень	27
1.2.2. Обґрунтування критеріїв ефективності КС.....	30
1.2.3. Аналіз існуючих рішень для підвищення ефективності КС.....	31
1.3. Аналіз деяких методів підвищення відмовостійкості КС	33
1.3.1. Класифікація аномальних станів.....	33
1.3.2. Надлишковість як основа відмовостійкості	34
1.3.3. Забезпечення відмовостійкості на рівні мережі	37
1.3.4. Високорівневі методи забезпечення відмовостійкості	41
1.3.5. Аналіз існуючих методів підвищення відмовостійкості	44
1.4. Огляд деяких методів підвищення ефективності КС.....	45
1.4.1. Причини низької ефективності паралельних КС.....	45
1.4.2. Огляд КС на основі квантового паралелізму.....	47
1.4.3. Огляд КС, які керуються потоком даних	48
1.4.4. Підвищення ефективності на основі натхнення природою обчислень ...	50
1.4.5. Аналіз розглянутих методів підвищення ефективності КС.....	51
1.5. Топологічні рішення сучасних високопродуктивних КС.....	52
1.5.1. Топологічні характеристики та критерії ефективності топологій.....	53
1.5.2. Топології сучасних суперкомп'ютерів.....	55

1.5.3. Деревовидні топології	56
1.5.4. Тороїдальні та гіперкубічні топології.....	57
1.5.5. Частково-повнозв'язні топології	59
1.6. Узагальнення проблематики та постановка задач дисертаційної роботи	61
Висновок до розділу 1	62
РОЗДІЛ 2. МАТЕМАТИЧНА МОДЕЛЬ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ НА ОСНОВІ НАДЛИШКОВОГО КОДУ	64
2.1. Взаємозв'язок відмовостійкості та ефективності топологій.....	64
2.2. Дослідження надлишкових кодів для синтезу топологій.....	66
2.2.1. Основа теорії надлишкового кодування	66
2.2.2. Шаблон як основа властивостей коду	70
2.3. Формалізація теорії надлишкового коду	77
2.3.1. Рекурентне обчислення $\alpha(v)$ для кодування розрядності r	82
2.3.2. Доведення незалежності α -розподілу від вмісту алфавіту.	84
2.3.3. Властивості кодів типу (tb, b) . Поняття багатовимірної форми.	87
2.4. Математична модель топологій на основі надлишкових кодів	93
2.4.1. Використання багатовимірної форми для обчислення $\alpha_r(V)$	93
2.4.2. Імпліцитні кластери на основі багатовимірної форми	97
2.4.3. Спосіб формування імпліцитних кластерів в надлишкових топологіях..	101
2.4.4. Порядок алфавіту та надлишкова самоподібність.....	102
2.4.5. Формальний опис математичної моделі та доведення її адекватності	104
Висновок до розділу 2	107
РОЗДІЛ 3. МЕТОД СИНТЕЗУ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ НА ОСНОВІ НАДЛИШКОВОГО КОДУ	108

3.1. Класичний метод синтезу топологій на основі кодових перетворень.....	108
3.2. Метод синтезу топологій на основі надлишкового коду.....	111
3.2.1. Синтез на основі надлишкового бінарного представлення	111
3.2.2. Надлишкова топологія де Бруйна.....	113
3.2.3. Практичний сенс надлишкового представлення в графі де Бруйна	116
3.2.4. Особливості методу синтезу на основі надлишкових кодів	120
3.3. Розвиток методу синтезу на основі надлишкового коду	122
3.3.1. Створення імпліцитних кластерів на основі багатовимірної форми.....	122
3.3.2. Декомпозиція на основі самоподібності	125
Висновок до розділу 3	127
РОЗДІЛ 4. МЕТОД МАСШТАБУВАННЯ ІЄРАРХІЧНИХ ТОПОЛОГІЙ.....	128
4.1 Опис методу масштабування ієрархічних топологій.....	128
4.2. Синтез графів зі вкладеною ієрархічністю	129
4.2.1. Синтез на основі декартового добутку.....	129
4.2.2. Синтез на основі рекурентних вкладень	130
4.2.3. Простий рекурентний граф	131
4.2.4. Насичення топології.....	133
4.2.5. Редукція топології.....	136
4.2.6. Надлишковий рекурентний синтез. DDB топологія.....	137
4.3. Синтез ієрархічних топологій на основі поєднання вкладеності та деревовидної ієрархії.....	138
4.3.1 Синтез деревовидних мереж з використанням топологічної інтеграції...	139
4.3.2. Опис методу. Топологія Tree-Dragonfly.....	140
4.3.3. Деревовидна мережа з неоднорідною ієрархією.....	145

Висновок до розділу 4.....	149
РОЗДІЛ 5. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ НА ОСНОВІ РОЗРОБЛЕНИХ ЗАСОБІВ.....	150
5.1. Опис засобів оцінки ефективності та відмовостійкості топологій.....	150
5.2. Засіб для моделювання топологій.....	150
5.2.1. Опис моделі.....	150
5.2.2. Відбір графів БЗМ на основі класичних характеристик.....	153
5.2.3. Аналіз відмовостійкості та топологічної ефективності	156
5.2.4. Аналіз впливу імпліцитних кластерів	157
5.2.5. Порівняння результатів із класичними рішеннями	158
5.2.6. Порівняння із популярними сучасними рішеннями.....	159
5.2.7. Кількісна оцінка ефективності запропонованих рішень	161
5.3. Засіб моделювання відмов у топологіях	163
5.3.1. Спосіб моделювання відмов у топологіях.....	163
5.3.2. Потік відмов на основі посередництва.....	165
5.3.3. Моделювання відмов для топологій БЗМ.....	166
5.3.4. Моделювання відмов для графів комутованих мереж.....	167
5.3.5. Порівняння на основі моделювання відмов	168
Висновки до розділу 5.....	170
ВИСНОВКИ.....	172
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	175
Додаток А.....	201
Додаток Б.....	222
Додаток В.....	224

Додаток Г	230
Додаток Г.....	249
Додаток Д.....	257

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

- КС – комп’ютерна система (комп’ютерні системи)
- HPCG – High Performance Conjugate Gradient (високопродуктивний спряжений градієнт)
- MTBF – Mean Time Before Failure (середній час між відмовами)
- CPU – Central Processing Unit (центральний процесор)
- GPU – Graphics Processing Unit (графічний процесор)
- FPGA – Field Programmable Gate Arrays (програмовані вентильні матриці)
- TMR – Triple modular redundancy (потрійне модульне резервування)
- NMR – N-modular redundancy (N-кратне модульне резервування)
- ПЗ – програмне забезпечення
- RAS – Reliability, Availability, Security (надійність, доступність, безпека)
- ABFT - Algorithm-based fault tolerance (відмовостійкість на основі алгоритму)
- ШІ – штучний інтелект
- ГА – генетичний алгоритм
- RISC – Reduced Instruction Set Computing (скорочений набір інструкцій)
- CGRA - Coarse Grain Reconfigurable Architecture (крупнозерниста реконфігурована архітектура)
- RBR – Redundant binary representation (надлишкове двійкове представлення)
- СЧ – система числення
- ІСЧ – індексна система числення
- HDB – Hyper de Bruijn (гіпер де Бруйн)
- DDb - Dragon De Bruijn (Драгон де Бруйн)

ВСТУП

Актуальність. Важливим аспектом, що характеризує сучасні найпродуктивніші комп'ютерні системи (КС), є використання надвеликої кількості ядер – від сотні тисяч до десяти мільйонів. Таке велике число обчислювальних елементів дає змогу отримати широкі можливості для паралельної обробки, але побічним ефектом цього є наявність в системі великої кількості потенційних точок відмов. Відомі методи забезпечення відмовостійкості, такі як резервування, дозволяють вирішити цю проблему, втім їх використання в загальному вигляді веде до суттєвого удорожчання системи, що для суперкомп'ютера є критичним.

Не менш важливим питанням є і питання ефективності, оскільки не дивлячись на високу номінальну продуктивність, сучасні суперкомп'ютери демонструють відносно посередню ефективність. Згідно до даних рейтингу TOP 500 на момент листопада 2024 року середнє значення ефективності на тестах LINPACK становило 62,94%. Окремо варто зазначити, що ці тести вимірюють реальну системну продуктивність і не враховують багатьох користувацьких аспектів, таких як залежності по даним між підзадачами і пов'язані із цим пересилки.

Одним із можливих рішень поставленої задачі є використання специфічних топологій, що дозволяють закласти в систему надлишковість ще на етапі конструювання, керуючи при цьому топологічними параметрами і досягаючи компромісу між відстанню до найвіддаленіших вузлів та щільністю комутацій. Втім варто зазначити, що ідея керування характеристиками мережі через топологію не є новою. Існує багато наукових праць, присвячених синтезу топологій для задач обчислювальної техніки і, зокрема, високопродуктивних обчислень. В той же час кожен із існуючих методів має свої особливості, переваги і недоліки. Окремою проблемою є те, що, не дивлячись на актуальність питання відмовостійкості, саме їй в існуючих розробках присвячено досить небагато уваги. Це породжує потребу в нових методах та актуалізує пошук таких комбінацій рішень, які б дозволили досягти компромісу між ефективністю, вартістю та стійкістю до відмов.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота входить в план наукової роботи кафедри обчислювальної техніки КПІ ім. Ігоря Сікорського і виконана в рамках наступних пошукових досліджень (ініціативних тематик): «Високопродуктивні комп'ютерні системи та мережі: теорія, методи і засоби апаратної та програмної реалізації» (факультет інформатики та обчислювальної техніки – керівник: доц. А. М. Волокита), № договору: Д/р №0121U108261, дата реєстрації: 11.02.2021.

Мета та завдання дослідження. Метою дисертаційної роботи є побудова топологій комп'ютерних систем, які забезпечують високу відмовостійкість та ефективність їх функціонування, за рахунок покращення мультиплікативного критерію ступеня та діаметру та забезпечення вищої імовірності збереження графом зв'язності в умовах зростаючого числа відмов елементів.

Для її досягнення необхідно вирішити наступні завдання:

1. Виконати аналіз відомих методів забезпечення відмовостійкості та ефективності комп'ютерних систем, з метою визначення проблематики та постановки задач в області розроблення ефективних та відмовостійких топологій паралельних комп'ютерних систем.
2. Розробити математичну модель топології на основі надлишкового коду, яка дозволяє на основі характеристик коду, таких як алфавіт та основа числення, визначити число альтернативних представлень довільного числа в цьому коді, що в контексті синтезу топологій дає змогу прогнозувати максимальне число вершин з однаковими номерами, кількість вершин з унікальними номерами.
3. Розробити спосіб формування імпліцитних кластерів в надлишкових топологіях, що з використанням спеціальної багатовимірної матриці надлишкових представлень та кодування індексів в спеціальній системі числення дозволяє формувати ребра між вершинами з однаковим номером для топологій на основі кодів із певними співвідношеннями потужності алфавіту та основи числення.

4. Удосконалити метод синтезу відмовостійких топологій на основі надлишкового коду, який дає змогу створити нові імпліцитні кластери в топології на основі довільного надлишкового коду, а також виконувати декомпозицію надлишкового графа на підграфи наперед відомої форми.
5. Розробити метод масштабування ієрархічних топологій, що дозволяє поєднати відмовостійкі топології, синтезовані на основі надлишкового коду, із класичними топологіями, такими як гіперкуб та dragonfly.
6. Розробити спосіб моделювання відмов в топологіях для аналізу імовірності розриву зв'язності графа при заданій кількості відмов вузлів, підрахунку топологічних характеристик та їх зміни відносно безвідмовного стану топології.
7. Розробити інструментальний засіб для моделювання характеристик топологій, яке дозволяє експериментально дослідити топологічні характеристики графів, отриманих з використання запропонованих методів, та довести їх вищу ефективність у порівнянні із класичними топологіями.
8. Розробити інструментальний засіб для моделювання відмов в топологіях, яке дозволяє експериментально дослідити поведінку запропонованих топологій в умовах наростаючого числа відмов і таким чином довести їх вищу відмовостійкість у порівнянні з гіперкубом, тором та dragonfly.

Об'єкт дослідження – процеси синтезу топологій комп'ютерних систем що породжують проблему побудови ефективних та відмовостійких топологій комп'ютерних систем.

Предмет дослідження – методи та засоби синтезу топологій паралельних та розподілених комп'ютерних систем та методи підвищення відмовостійкості та ефективності топологій комп'ютерних систем.

Методи дослідження. Методичною основою дисертаційного дослідження є системне опрацювання та аналіз теоретичного матеріалу, присвяченого підвищенню відмовостійкості та ефективності комп'ютерних систем (особливо – суперкомп'ютерів). В процесі дослідження було використано методи теорії графів,

перетворення в надлишкових системах числення, а також програмні засоби для моделювання топологій.

Наукова новизна отриманих результатів:

1. ***Розроблено нову математичну модель*** топології на основі надлишкового коду, що ***відрізняється від існуючих*** моделей використанням алфавіту, основи числення та довжини коду для визначення кількості альтернативних представлень довільного числа в заданій системі числення, та ***дозволяє*** прогнозувати максимальну кількість вершин з однаковим номером у графі, кількість вершин з унікальними (не-надлишковими) номерами.
2. ***Запропоновано новий спосіб*** формування імпліцитних кластерів в надлишкових топологіях, що ***відрізняється від існуючих*** використанням спеціальної багатовимірної матриці надлишкових представлень та кодування індексів в спеціальній системі числення та ***дозволяє*** формувати ребра між такими вершинами для топологій на основі кодів із певними співвідношеннями потужності алфавіту та основи числення.
3. ***Набув розвитку метод*** синтезу відмовостійких топологій на основі надлишкового коду, що ***відрізняється від існуючих*** використанням кодових перетворень, в тому числі послідовностей де Бруїна, в надлишкових системах числення та створенням нових зв'язків у таких топологіях за допомогою перетворень заміщення над кодами, які описують індекс альтернативного представлення в багатовимірній матриці надлишкових представлень, що ***дозволяє*** синтезувати відмовостійкі топології заданого порядку, в тому числі з імпліцитними кластерами.
4. ***Запропоновано новий метод*** масштабування ієрархічних топологій, що ***відрізняється від існуючих*** використанням декартового добутку, деревовидних структур та рекурентного вкладення кластерів, що ***дозволяє*** поєднати відмовостійкі топології, синтезовані на основі надлишкового коду, із класичними топологіями, такими як гіперкуб та dragonfly.
5. ***Запропоновано новий спосіб*** моделювання відмов в топологіях, що ***відрізняється від існуючих*** використанням різних підходів до випадкового

формування черги відмов, в тому числі з урахуванням коефіцієнту посередництва, та *дозволяє* при заданій кількості відмов вузлів аналізувати імовірність розриву зв'язності графа, підрахувати топологічні характеристики та їх зміну відносно початкового (безвідмовного) стану топології.

Практичне значення отриманих результатів. Запропоновані методи та одержані результати дозволяють синтезувати нові топології для комп'ютерних систем, які дозволяють підвищити їх відмовостійкість та ефективність. Для дослідження запропонованих методів було розроблено 2 засоби, а саме:

1. Інструментальний засіб для моделювання характеристик топологій на основі бібліотеки NetworkX, який *за рахунок* запропонованого способу формування імпліцитних кластерів в надлишкових топологіях *дозволяє* дослідити топологічні характеристики графів, отриманих з використання запропонованих методів, та виконати їх порівняння із класичними топологіями, такими як гіперкуб, жирне дерево, dragonfly та dragonfly+, а також багатовимірні тори, включаючи топологію суперкомп'ютерна Fugaku.
2. Інструментальний засіб для моделювання відмов в топологіях, який *є реалізацією* запропонованого способу моделювання відмов в топологіях та *дозволяє* дослідити поведінку топологій в умовах наростаючого числа відмов і таким чином порівняти відмовостійкість запропонованих та існуючих графів.

Було виконано експериментальне дослідження з використанням розроблених програмних засобів, яке показало, що запропоновані методи дозволяють отримати нові топології для комп'ютерних систем, які забезпечують кращий мультиплікативний критерій ступеня та діаметру, а також меншу імовірність розриву зв'язності графа при заданій кількості відмов вузлів порівняно з існуючими топологіями, і таким чином дозволяють досягти поставленої мети підвищення відмовостійкості та ефективності комп'ютерних систем.

Особистий внесок здобувача. Дисертація є результатом самостійних наукових досліджень, в яких вкладено авторський підхід до вирішення проблеми

відмовостійкості кооп'ютерних систем. Наукові положення та основні результати, що містяться в дисертації, отримані здобувачем самостійно у процесі науково-дослідницької роботи. В роботах, опублікованих у співавторстві, дисертанту належать: [1] – механізми обмеження масштабування та насичення топології Dragon de Bruijn, [2] – метод синтезу відмовостійких топологій на основі надлишкових кодів, а також спосіб моделювання відмов в топологіях з урахуванням коефіцієнту посередництва, [3] – алгоритм багатоканальної маршрутизації на основі кодових перетворень, [4] – спосіб формування імпліцитних кластерів в надлишкових топологіях та опис синтезу топологій на основі перетворень в надлишковому коді, [5] – спосіб синтезу деревовидних топологій на основі інтеграції графів та топології Dragonfly, [6] – метод ієрархічного синтезу топологій на основі рекурентного синтезу, [7] – спосіб синтезу багаторівневої надлишкової топології на основі декартового добутку з топологією на основі надлишкового коду, [8] – механізм відмовостійкої маршрутизації в мережах на основі надлишкового двійкового коду, що базується на 2-знакових «шаблонах», [9] - алгоритми пошуку альтернативних маршрутів в топології надлишкового де Бруйна на основі властивостей надлишкових кодів.

Апробація результатів дисертації. Основні результати роботи опубліковано та обговорено на всеукраїнських та міжнародних конференціях, зокрема:

1. 2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT) (18-20 December 2019, Ukraine);
2. ICCSEEA2020: The Third International Conference on Computer Science, Engineering and Education Applications (21-22 January 2020, Kiev, Ukraine);
3. ICCSEEA2021: The Fourth International Conference on Computer Science, Engineering and Education Applications (January 23 - January 24 , 2021 , Kyiv, Ukraine);
4. ICCSEEA2023: The 6th International Conference on Computer Science, Engineering and Education Applications (March 17 - March 19, 2023, Warsaw, Poland);
5. ICSFTI 2024: International Conference on Security, Fault Tolerance, Intelligence (June 07, 2024, Kyiv, Ukraine).

Публікації. За результатами дисертаційних досліджень опубліковано 5 наукових публікацій, що входять до наступних наукометричних баз даних з міжнародним індексом цитування: Scopus – 4, фахові видання України категорії Б – 1.

Структура та обсяг роботи. Дисертаційна робота складається зі вступу, п'яти розділів, загальних висновків, списку використаних джерел із 252 найменувань та додатків. Загальний обсяг дисертації становить 276 сторінок, з яких 151 сторінок основного тексту, 7 додатків на 75 сторінках. Дисертація містить 41 рисунок, 56 формул, 12 таблиць.

РОЗДІЛ 1

ОГЛЯД ІСНУЮЧИХ МЕТОДІВ ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ ТА ЕФЕКТИВНОСТІ КОМП'ЮТЕРНИХ СИСТЕМ

1.1. Визначення основних понять та аспектів високопродуктивних КС

Сучасні комп'ютерні системи досягли вражаючих показників, і найпоказовішим прикладом є стан сучасних суперкомп'ютерів. Пікова продуктивність 1679.82 петафлопс і продуктивність 1194 петафлопс на тестах LINPACK, – такими є показники суперкомп'ютера Frontier, що на листопад 2023 року займав 1 місце в рейтингу TOP 500 [10]. Втім, інша сторона цих досягнень є не менш вражаючою. Маючи в своєму складі 8 699 904 ядра, дана система має користувацьку продуктивність 14 054 терафлопс [11] – і займає 2 місце в рейтингу тестування High Performance Conjugate Gradient (HPCG). Таким чином, можна виділити 2 ключові проблеми, від вирішення яких залежить подальший розвиток даної галузі.

Однією із ключових є проблема відмовостійкості. Мільйони ядер і тисячі вузлів – це мільйони і тисячі потенційних точок відмов. Щоб яскравіше висвітлити даний аспект, звернемось до теорії надійності. Ключовим показником, що характеризує надійність системи, є напрацювання на відмову (англ. Mean Time Between Failure, MTBF) [12]. Цей параметр показує час, що в середньому проходить між двома відмовами в системі [13]. Інше визначення – імовірність, протягом якої система працюватиме без відхилень від узгодженої поведінки протягом визначеного часу. У випадку, коли компоненти системи є однотипними і надійність кожного R_1 з них не залежить від надійності інших, надійність всієї системи з N компонент близька до обернено пропорційної [14]:

Звісно, варто розуміти, що така залежність між надійністю вузла та надійністю системи передбачає незалежність надійності кожного компонента від інших, що є не цілком коректно через властивість відмов поширюватись. Втім дана наближена оцінка ілюструє основну суть проблеми: зростання кількості компонентів веде до падіння надійності системи як цілого.

Підтвердження даної тези пропонується в наукових роботах, присвячених темі відмовостійкості високопродуктивних обчислень. Досліджено, що навіть якщо індивідуальне напрацювання на відмову елементів становить одне століття, система, що складається зі 100 000 подібних елементів, матиме MTBF приблизно 9 годин. Щодо систем з 1 000 000 компонент, то для них час роботи між відмовами становитиме в середньому 53 хвилини [15, 16]. Ці дані підтверджуються розрахунками для системи IBM Blue Gene/L, яка має у складі кластеру 131 000 процесорів з середнім напрацюванням на відмову 876 000 годин (приблизно 100 років). Після обрахунку MTBF кластеру було отримано значення 6.68 годин [14], що відповідає даним, наведеним вище.

Іншою проблемою є проблема низької ефективності суперкомп'ютерів, коли мова йде про реальну користувацьку задачу. Під ефективністю *суперкомп'ютерів* в даному випадку мається на увазі відношення продуктивності (R , у Flop/s) на певній задачі до максимальної теоретично можливої (пікової, R_{peak}) продуктивності. Гарним прикладом цього є вищезгадане тестування HPCG, яке є альтернативою класичного тесту LINPACK і застосовує обчислення над тривимірними розрідженими матрицями в якості задачі [17]. В табл. 1.1 приведено ефективність деяких суперкомп'ютерів, що посідали перші місця за рейтингом TOP 500 (станом на червень 2024 року).

Табл. 1.1

Ефективність суперкомп'ютерів за даними TOP 500 на червень 2024 року [10] згідно до результатів тесту HPCG (повна версія – табл. Д.1)

Рейтинг TOP 500	Рейтинг HPCG	Суперкомп'ютер	Пікова продуктивність (R_{peak})	Результат HPCG (R)	Ефективність за HPCG
1	2	Frontier	1679818.75	14054	0.84 %
4	1	Supercomputer Fugaku	537212	16004.5	2.98 %
5	3	LUMI	531505.15	4586.95	0.86 %
6	4	Leonardo	304465.92	3113.94	1.02 %

Як продемонстровано в таблиці 1.1, найпотужніші сучасні суперкомп'ютери мають ефективність на рівні 1–3%. Таким чином, актуальним питанням є не лише забезпечення відмовостійкості, а й пошук шляхів для підвищення ефективності.

Втім, перш ніж переходити до постановки задачі, варто виділити один важливий аспект, пов'язаний з поняттями «суперкомп'ютер» та «високопродуктивна система». Згідно із визначенням, приведеним у статті [18], суперкомп'ютером прийнято називати найпродуктивніший тип комп'ютера, що використовується для спеціалізованих додатків з великою кількістю математичних обчислень, в той час як галузь високопродуктивних обчислень передбачає й інші форми систем високої продуктивності, такі як кластери, хмари, розподілені системи, тощо [19]. Втім, варто розуміти що таке співвідношення цих визначень не є безальтернативним: так, сучасне визначення суперкомп'ютерів значною мірою базується на рейтингу TOP 500 високопродуктивних систем [20], крім того, в рамках сучасного наукового дискурсу виділяють різні види суперкомп'ютерів, такі як:

- Класичні суперкомп'ютери на основі мультикомп'ютерної чи кластерної архітектури [21, 22]
- Опортуністичні суперкомп'ютери [23–25], які базуються на розподілених обчисленнях і не гарантують доступності ресурсів. При цьому така система може як мати внутрішню структуру, так і не мати її взагалі. У такому випадку можна казати про повністю розподілений суперкомп'ютер, вузли якого взаємодіють між собою через Інтернет / локальну мережу з використанням класичних мережевих протоколів.
- Квазіопортуністичні суперкомп'ютери [26, 27], що базуються на технології GRID, хмарних (cloud) та туманних (fog) обчислень. Даний підхід передбачає наявність певного проміжного програмного забезпечення, що гарантує певну якість обслуговування. Як правило, передбачається, що така система має певну (приховану від користувачів) внутрішню мережу для міжвузлової комунікації і взаємодіє з Інтернетом лише для надання сервісу.

Таким чином, для забезпечення однозначності термінів необхідно дати визначення поняттю «суперкомп'ютер».

Під суперкомп'ютером розумітимемо паралельну комп'ютерну систему, що (1) має продуктивність на порядки вищу, ніж у звичайних користувацьких систем та (2) має певну внутрішню організацію (комунікативну мережу), що дозволяє вузлам / процесорам системи комунікувати між собою. Таке визначення узгоджується із термінологією, що використовується в роботах [18] та [20–22].

Оскільки проблематика роботи пов'язана із суперкомп'ютерами, далі під комп'ютерними системами в першу чергу розумітимемо саме їх.

1.2. Аналіз проблематики ефективності масштабованих КС

1.2.1. Обґрунтування проблематики масштабованих обчислень.

Ключовою опорою пета- та ексафлопсних обчислень сьогодення є масштабування. Із року в рік людство намагається побити власні рекорди, розвиваючи старі системи та розробляючи нові, створюючи нові підходи і тим самим сприяючи розвитку сфери високопродуктивних обчислень. Саме розвиток галузі, – а не її поточний стан як такий, – має ключове значення. Таким чином, поняття *масштабованої системи* в контексті суперкомп'ютерів може бути визначено не лише як система, ресурси якої динамічно змінюються протягом роботи (таку систему варто назвати динамічно масштабованою), проте і як система, що з певними інтервалами часу нарощує свою потужність – тобто, масштабується статично.

Загальновідомою є класифікація методів масштабування на 2 основних класи: вертикальне масштабування, тобто, заміщення елементної бази більш сучасними рішеннями, та горизонтальне масштабування, що передбачає нарощування продуктивності через збільшення елементів [28, 29]. Буде помилкою сказати, що нинішня галузь використовує лише на щось одне, – обидва ці напрями застосовуються і розвиваються. Втім принципи та масштаби їх застосування принципово відрізняються.

Так, вертикальне масштабування передбачає оновлення апаратури, часткову чи повну заміну старих елементів на більш сучасні. Прикладом є суперкомп'ютер Jaguar, що проходив кілька етапів такого масштабування. Так, в рамках даного комп'ютера в різні періоди часу використовувались такі процесори як Cray X1, XT3, XT4 та XT5,

які також містили в собі спочатку чотирьох, а потім і шестиядерні AMD Opteron. Загалом за 7 років існування суперкомп'ютера Jaguar було зроблено 6 оновлень [30]. В 2011 році суперкомп'ютер було списано і на його основі розроблено іншу систему – Titan [31–33].

Інший факт полягає в тому, що з часом з'являються нові системи, що використовують новішу елементну базу і за рахунок цього досягають вищих показників номінальної продуктивності. Це не є в чистому вигляді масштабуванням, оскільки розробляється повністю нова система, проте з точки зору галузі це можна вважати певним оновленням, яке дозволяє ввести в предметну область нову елементну базу та пов'язані із нею технології.

Втім основою високої продуктивності сучасних систем є горизонтальне масштабування та пов'язаний із ним паралелізм. Кожен сучасний суперкомп'ютер є паралельною системою тієї чи іншої архітектури, причому число паралельно працюючих процесорних елементів (ядер) в таких системах може сягати 10 мільйонів. Окрім того, більшість суперкомп'ютерів є гетерогенними, тобто, можуть містити не лише центральні процесори (CPU), але і графічні прискорювачі (GPU), й інші додаткові функціональні елементи, на кшталт програмованих вентильних матриць (field-programmable gate arrays, FPGA).

Аналізуючи підхід розробників до вибору елементної бази, варто зазначити, що більшість використовують вже готові апаратні рішення, а не розробляють свої. Хоча є і виключення: так, компанія Fujitsu, створила власну мікроархітектуру в співробітництві з компанією ARM і розробила процесор A64FX для суперкомп'ютера Fugaku [34, 35]. Проте така ситуація не є типовою. Іншим важливим трендом останніх років є використання процесорів з великою (до 96) кількістю ядер, що дозволяє значно зменшити число вузлів в системі і тим самим мінімізувати звертання до міжпроцесорної мережі.

На рис. 1.1 представлено дані для найпродуктивніших згідно тесту HPCG суперкомп'ютерів, починаючи з листопада 2017 року і завершуючи листопадом 2023 року. Графік зліва показує, як зростало число ядер в цих системах. Загалом, найбільш виражений ріст проявлявся при передачі лідерства від одного суперкомп'ютера

іншому, проте було декілька моментів, пов'язаних із масштабуванням вже існуючої системи. Так, Summit наращував число ядер 2 рази: в листопаді 2018 та в червні 2019. Число ядер в Fugaku змінювалось 1 раз: в листопаді 2020 року. Відповідно, графік справа демонструє реальну користувачську продуктивність відповідних систем.

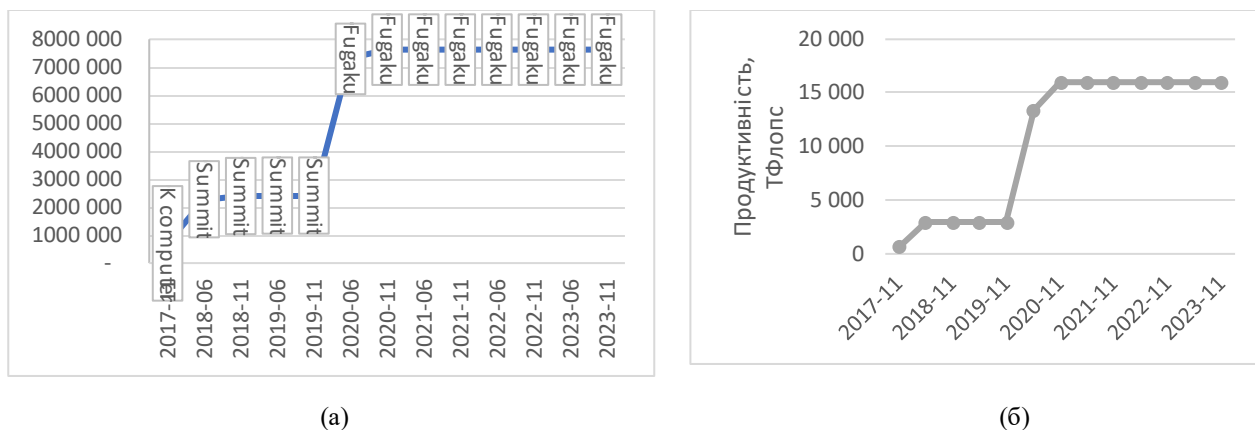


Рис.1.1. Характеристики суперкомп'ютерів з найвищою користувачською продуктивністю з листопада 2017 по листопад 2023: (а) – число ядер, (б) – реальна користувачська продуктивність.

Дані рис. 1.1. наочно демонструють два факти: по перше, ріст продуктивності безпосередньо зав'язаний на число ядер, зростання одного параметру призводить до зростання іншого. По-друге, завжди є певна межа реальної продуктивності, що пов'язана не лише із числом ядер, але і з іншими параметрами, такими як структура системи, принципи організації обчислень та технології, що для цього використовуються. Суть цієї межі можна виразити наступним чином: протягом певного часу незалежно від того, як зростає номінальна продуктивність, максимум реальної продуктивності не змінюється. Гарним прикладом є суперкомп'ютер Frontier, що переважає Fugaku і за числом ядер, і за поколінням елементної бази, і за результатами тесту LINPACK, проте має реальну (за тестом HPCG) продуктивність меншу приблизно на 2 петафлопси (див. табл. 1.1).

Гіпотетично, це обмеження не є критичним. Врешті решт, історично реальна продуктивність поступово зростає, хоч і набагато повільніше, ніж номінальна. Ціною збільшення числа ядер, через покращення елементної бази та скорочення міжпроцесорної взаємодії за рахунок укрупнення вузлів є можливість збільшити реальну продуктивність, не виходячи за межі існуючих підходів. Проте існують й інші

проблеми, такі як енергоспоживання, витрати площі, складнощі охолодження, тощо, які зі свого боку унеможливають нескінченне використання такого підходу.

Таким чином, необхідним кроком є підвищення ефективності комп'ютерних систем. Оскільки в даному випадку мова йде про продуктивність та ефективність поза контекстом конкретної задачі, під *ефективністю комп'ютерної системи* чи суперкомп'ютера розумітимемо комплексну характеристику, що включає в себе такі аспекти як ефективність паралельних обчислень, ефективність (швидкість) обміну даними між вузлами та ядрами системи, вартість системи, а також *відмовостійкість*, яку розглядатимемо в якості окремого критерію, забезпеченню якого було приділено окрему увагу.

1.2.2. Обґрунтування критеріїв ефективності КС

Проблема підвищення ефективності комп'ютерних систем є вкрай широкою і включає в себе проблеми таких предметних областей як паралельні та розподілені обчислення, масштабування обчислень, планування обчислень, комп'ютерні мережі, надійність та відмовостійкість, тощо. Розгляд всіх конкретних проблем та їх аналіз є нетривіальною задачею, втім, в дисертаційній роботі основними обмеженнями які впливають на ефективність визначені наступні:

- *Обмеження паралельних обчислень* [36]. В першу чергу це обмеження на розпаралелювання задачі, верхню границю якого визначає закон Амдала. Крім того, існує обмеження паралельної архітектури, пов'язане із обробкою паралельних частин і взаємодією між процесором та пам'яттю. Також сюди можна віднести проблему залежності по даним, що виливається у додаткові затримки, та проблему планування, що полягає в пошуку якнайкращого розподілення підзадач між процесорами з огляду на різні параметри.
- *Проблема структурної організації* [37]. Оскільки сучасні системи працюють з паралелізмом на рівні мільйонів елементів, виникає питання щодо компоновки такої системи. Одним із ключових підходів сучасної галузі є використання багаторівневого підходу, коли процесори з

десятками (як правило, 32–64) ядер із власною паралельною архітектурою розміщуються на одному чіпі, вони об'єднуються в групи, із груп утворюються кластери, а з них складається сама система. Втім, важливими проблемами все одно залишаються дистанція між вузлами, обмежена пропускна здатність, затримка та якість маршрутизації, затримки при передачі даних, що безпосередньо впливає на ефективність міжпроцесорної взаємодії.

- *Проблема відмовостійкості* [16]. В контексті мільйонів ядер важливою проблемою стає вихід елементів з ладу, що може призвести як мінімум до суттєвої деградації продуктивності, як максимуму – до відмови самої системи. Оскільки для більшості великих систем зупинка та перезапуск є тривалими операціями, а напрацювання на відмову, як правило, є невисоким, існує проблема надійності обчислень та сервісів, а також надійного зберігання та передачі даних. Крім того, виявлення відмови потребує швидкої та автоматичної реакції з боку системи, що породжує питання маскування та виправлення помилок. Від вирішення цього питання залежить ефективність роботи системи в процесі експлуатації.

Ці проблеми визначені, як основні обмеження, що впливають на ефективність. Таким чином, в дисертаційній роботі визначені наступні три критерії ефективності комп'ютерних систем : ефективність паралельної обробки даних, ефективність топологічної структури системи та відмовостійкість.

1.2.3. Аналіз існуючих рішень для підвищення ефективності КС

Загалом, кожна із зазначених основних проблем є добре описаною в наукових публікаціях, більш того – кожна із них є основою окремої предметної області. Так, велика кількість робіт присвячена проблемам паралелізму [38–40], відмовостійкості [41–43], синтезу структурних організацій для високопродуктивних систем [44–46]. Кожна з цих сфер пропонує свій набір рішень означених проблем.

Однією із найбільш досліджених є проблема відмовостійкості. Існує великий спектр методів, що можуть бути застосовані для її вирішення, втім, опираючись на класифікацію рівнів, можна виділити наступні основні групи:

1. Надлишковість та резервування [47], що є одним із ключових методів вирішення проблеми і застосовується на всіх рівнях.
2. Відновлення стану та корекція помилок [48–50]. Задача даної групи методів – усунення помилки, яка вже сталась, втім їх рівні дещо різняться. Відновлення стану є методом, переважно, системного та програмного рівня, в той час як коригуючі коди частіше застосовуються на рівні апаратури та мережі.
3. Тестування та виявлення помилок [41]. Мета даного класу методів – виявити помилку чи відмову. Як правило, кожен рівень системи має свої методи для тестування, такі як апаратні тести, приймальні тести, візантійське узгодження та ін. Також виділяють самотестування та взаємотестування.
4. Маскування відмов [51, 52]. Ідея даного класу полягає в прихованні несправної одиниці від користувача та інших складових частин системи. Свої методи маскування існують майже на кожному рівні.
5. Передбачення відмов та міграція даних [53, 54] є методами системного рівня і застосовуються для превентивного реагування на відмову, що очікується найближчим часом.
6. Відмовостійкість на рівні алгоритмів [55] є методом, що дозволяє закласти методи виявлення та корекції помилок в сам алгоритм і використовується для розробки відмовостійкого програмного забезпечення.

На противагу проблемі вище, задача оптимального топологічного синтезу не має однозначного вирішення, тож в рамках сучасних високопродуктивних обчислень розглядається велика кількість різних підходів та їх варіацій. Загалом, варто казати не про конкретні топології, а про сімейства топологій, що об'єднані спільними чи схожими методами синтезу. Список нижче не є вичерпним і приводить лише деякі методи, що використовуються в сфері петафлопсних обчислень:

1. К-арне N-дерево [56, 57], також відоме як жирне дерево, є популярною мережевою організацією для датацентрів, а різні її варіації використовуються в сфері високопродуктивних обчислень
2. Dragonfly [58, 59], популярна топологія суперкомп'ютерів, основною перевагою якої є виконання пересилок рівно за 3 кроки.
3. nD тор [60–62], що включає в себе такі варіації як 3D, 5D, 6D тор і використовується в таких системах як K Computer та Supercomputer Fugaku.
4. HyperX [63, 64], що базується на поєднанні решітки та кубічно-зв'язаних циклів, пропонує кращі характеристики ніж жирне дерево.

Третя предметна область є найбільш різноманітною, оскільки проблеми паралелізму є вкрай обширними і включають в себе і декомпозицію задачі, і планування, і міжпроцесорну комунікацію, і методи синхронізації. Загалом її рішення можна поділити на декілька основних груп:

1. Перехід на іншу обчислювальну модель чи архітектуру, що включає в себе, але не вичерпується такими рішеннями як квантові обчислення [65, 66], сучасні варіації моделі потоку даних [67, 68] та їх поєднання із суперскалярною архітектурою [69, 70], гібридні обчислення [71, 72], тощо.
2. Використання новітніх методів планування [73, 74]
3. Методи автоматичного розпаралелювання [75, 76]
4. Методи обчислення з високим ступенем внутрішнього паралелізму та широкою сферою застосування [77–79], такі як машинне навчання та еволюційні обчислення.

1.3. Аналіз деяких методів підвищення відмовостійкості КС

1.3.1. Класифікація аномальних станів

В сучасній науковій літературі розглядають ряд несправних станів – це несправність / збій (fault), відмова (failure) та помилка (error) [80–82]. Під несправністю варто розуміти безпосередній стан компонента або програми:

наприклад, фізичні пошкодження, помилки в коді, збої апаратного забезпечення внаслідок дії зовнішнього середовища, тощо. Проявом несправності є помилка – спостережуване користувачем відхилення в роботі системи. Відмова ж визначається як нездатність компонента виконувати свої функції чи надавати коректний сервіс [83].

Несправні стани класифікуються за рядом характеристик: за тривалістю (постійні, регулярні та тимчасові несправності), за зловмисністю (випадкові та зловмисні), за місцем та часом виникнення помилки (некоректна специфікація, помилки в імплементації, проблеми в програмному забезпеченні, тощо) [80].

Ще одним виміром, за яким виконується розрізнення аномальних станів, є наслідки, до яких призводить несправність: це може бути або повна зупинка роботи компонента, що вийшов із ладу (fail-stop error) – або його робота з некоректними результатами (fail-continue error) [83]. Також до даної класифікації можна віднести і так звані «тихі помилки» (silent errors), що не проявляють себе безпосередньо, а лише опосередковано (наприклад, через пошкодження певних даних) та/або через певний проміжок часу [16].

Більш докладний аналіз причин відмов у суперкомп'ютерах на основі джерел [84–88] приведено у Додатку Д.1.

1.3.2. Надлишковість як основа відмовостійкості

Вкрай небезпечною властивістю відмов і помилок є їх схильність до розповсюдження. Це ставить перед розробниками відмовостійких систем задачу – (а) своєчасного виявлення відмови (тестування) і (б) її ізоляції від працездатних частин системи (маскування). Проблемою є те, що це має бути зроблено максимально швидко – в іншому випадку «тиха» відмова призведе до розповсюдження помилок по системі і зробить пошук джерела помилки задачею нетривіальною. Іншою проблемою залишається той факт, що несправний елемент до відмови мав певну функцію, яка має виконуватись попри відмову – таким чином, до приведених вище задач має бути додано (с) якомога швидше заміщення несправного компоненту справним.

Така постановка проблеми створює необхідність в резерві – додаткових елементах, вузлах чи програмних модулях, які б узяли на себе роль несправного пристрою і тим самим приховали факт відмови від стороннього користувача. Це може

бути зроблено на багатьох рівнях: від апаратних модулів та логічних вентилів до програмного забезпечення та сервісів [83, 89]. Таким чином, надлишковість є ключовим елементом будь-якої відмовостійкої системи [90].

Одним із основних підходів до надлишковості є резервування, яке включає в себе такі архітектури як дуплексна та триплексна [91, 92]. Також виділяють симплексну та N-плексну архітектуру [93, 94].

Така архітектура передбачає порівняння результатів всіх модулів, що дозволяє визначити факт наявності помилки, проте залишає невирішеним питання про її локалізацію та маскування [95, 96] і потребує додаткових засобів – таких як апаратне тестування, приймальні тести та пряме відновлення з використанням додаткового перевіряючого елемента [80]. Розширенням даної архітектури є пара-резерв [84, 97, 98] та дуплекс-триплексна [99] архітектура, що використовує дуплекси в якості структурних елементів для реалізації більш відмовостійких конструкцій [80].

Таким чином, актуальним залишається питання автоматизації тестування та маскування відмов. Розповсюдженим апаратним підходом є голосування. Даний підхід застосовують у випадку $n > 2$ [100], таким чином, мінімальним числом модулів для реалізації схеми голосування є 3. Таку схему називають потрійним модульним резервуванням (Triple Modular Redundancy, TMR) [101]. При використанні мажоритарного голосування дана схема передбачає наявність в системі $n = 2m - 1$ модулів і дозволяє виявити до m помилок [102]. Тоді кажуть про N-кратне модульне резервування (N modular redundancy, NMR). Слабким місцем даної схеми є модуль голосування, тому розрізняють модульне резервування з одинарною та кратною схемами голосування [103], що продемонстровано на рис. Д.1 в Додатку Д на прикладі TMR.

Також NMR поділяють на статичний (приведений вище), динамічний та гібридний типи. В основі динамічного лежить теза, що невеликі помилки є прийнятними, поки система здатна замінити несправний модуль резервом [104]. Згідно до визначення, приведенного в роботі [80], його схема складається з 1 основного та $n-1$ резервних елементів, а також – схеми виявлення відмов та реконфігурації. Гібридний підхід є поєднанням вищеописаних і передбачає наявність N основних та

К резервних елементів. Рис. Д.2 в Додатку Д наочно показує різницю між цими двома підходами.

Важливим недоліком, який стосується всіх схем з мажоритарним голосуванням, є потреба в наявності працездатної більшості. Обмеженою альтернативою є використання не-мажоритарних принципів голосування, таких як зважений підхід, використання k -більшості або наближеного порівняння [80].

Іншим рішенням є використання відсіювання. Така схема дозволяє забезпечити роботу системи поки хоча б 2 елемента із N досягають згоди щодо результату [105, 106]. Її ключовим недоліком є чутливість відсіювання. Крім того, ризики несе невизначеність, що може виникнути, коли кілька елементів мають однакову несправність і генерують однакові результати.

Головною проблемою всіх раніше розглянутих методів є високі апаратні витрати і, як наслідок, висока вартість системи. Це робить актуальним питання не лише резервування апаратури, а і резервування процесів та / або програмного забезпечення. Методи забезпечення такого роду відмовостійкості були описані авторами робіт [14, 80, 107–109]. Оскільки ці аспекти не мають безпосереднього відношення до тематики дослідження, але є важливими для розуміння предметної області, їх огляд було винесено в Додаток Д.2.

Необхідно зазначити, що методи забезпечення відмовостійкості через резервування постійно розвиваються. На основі публікацій [89, 92, 110–112] було виконано огляд розвитку існуючих методів резервування, який також приведено в Додатку Д.2.

Окремим видом надлишковості є інформаційна надлишковість. Її суть полягає в додаванні до інформації певного обсягу додаткових даних, які допомагають виявити і, – іноді, – виправити помилку. Сфера застосування інформаційної надлишковості – це і виявлення помилок при обчисленнях, і зберігання даних, і обмін інформацією на різних рівнях. Було виконано короткий огляд основних підходів до інформаційної надлишковості на основі робіт [80] та [113]. Матеріали цього огляду також представлені у Додатку Д.2.

1.3.3. Забезпечення відмовостійкості на рівні мережі

Однією із ключових частин будь-якої високопродуктивної системи є мережа, що з'єднує її компоненти між собою. За рівнем можна виділити різні типи мереж, від внутрішньої мережі вузла, що з'єднує його внутрішні складові (процесори, пам'ять та ін.), до загальної мережі, що поєднує вузли, групи вузлів та кластери. За типами мережі розділяють на комутовані та безпосередньо зв'язані.

Проблема відмови в мережі відрізняється від аналогічної апаратної проблеми, оскільки мало просто замаскувати елемент – необхідно забезпечити відновлення функціонування мережі, а головне – відновити коректну маршрутизацію. Таким чином, можна виділити 3 методи для виконання цього. Грубий підхід полягає в редукції рангу топології. Це дозволяє зберегти маршрутизацію, проте видаляє частину працездатних вузлів, тож цей метод не є придатним. Тонкий підхід полягає у видаленні лише того вузла, що відмовив. При цьому алгоритмом маршрутизації має бути реалізовано обхід несправності. Третій метод – заміщення вузла резервом. Рис. 1.2 наочно ілюструє описані 3 методи.

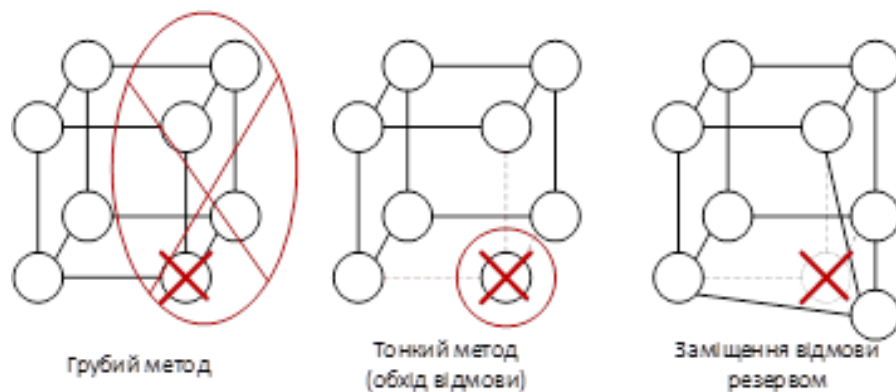


Рис. 1.2. Три базові методи усунення відмови в мережі

Таким чином, коли мова йде про відмовостійкість мережевого рівня, виділяють декілька основних задач. Перша задача – це відмовостійка маршрутизація. Її постановка наступна: є вузол-відправник повідомлення та вузол-приймач. Є маршрут, по якому прямує (чи має прямувати) повідомлення і на цьому маршруті є відмова (одна чи декілька), причому точкою відмови може бути як вузол так і зв'язок. Питання ставиться наступним чином: як, знаючи точки відмови, обійти їх і доставити

повідомлення, витративши якнайменший час на модифікацію алгоритму маршрутизації?

Одне із можливих рішень полягає у використанні властивостей конкретної топології для обходу відмови. В якості прикладів можна навести сегментно-базований алгоритм маршрутизації для решітки та тору [114], кластерний метод маршрутизації для тору [115], алгоритм на основі трьохкомпонентної зв'язності для гіперкуба [116] та ін. Як правило, такі алгоритми опираються на певну топологію, що і є їх ключовим обмеженням.

Інший варіант вирішення даної проблеми – це використання універсальних методів маршрутизації, на кшталт табличних, доповнених алгоритмом обходу небезпечних частин топології. Прикладами такого рішення є адаптивна маршрутизація [117], виділення кістякових дерев [118] та використання випадкових графів [119]. Очевидним недоліком таких методів є те, що вони, як правило, створюють більшу затримку ніж топологічно-орієнтовані методи та/або потребують більших обсягів пам'яті.

Методи відмовостійкої маршрутизації є одним із невід'ємних аспектів мережевої відмовостійкості, втім загальною проблемою алгоритмічних методів залишається їх накладні витрати. Час пошуку маршруту є вкрай важливим показником, оскільки він безпосередньо включається в час передачі повідомлення. З іншого боку, витрати пам'яті алгоритму маршрутизації впливають на ціну мережевих пристроїв. Ці аспекти додатково ускладнюють пошук оптимального рішення в кожному конкретному випадку і породжують потребу в (а) максимально простих і (б) максимально відмовостійких методах.

Третій варіант вирішення проблеми базується на використанні специфічних протоколів та/або властивостей мережі. Наприклад, протокол маршрутизації для сенсорної мережі, продемонстрований в роботі [120], передбачає регулярний пошук альтернативних маршрутів, що може бути прийнятним для сенсорних мереж, але при екстраполяції на комунікаційні мережі суперкомп'ютерів не є застосовним через їх великі розміри.

Втім, хоча відмовостійка маршрутизація і вирішує частину проблем, її можливості є обмеженими. Таким чином, актуальним питанням є не лише обхід, а й маскуванню відмов, а також резервування вузлів. В контексті мережі задача резервування ускладнюється тим, що як основні так і резервні вузли поєднані мережею, і коли резервний вузол бере на себе роль основного – він має взяти не лише обчислення, а і зв'язки, якими основний елемент пов'язаний із сусідами.

Класичний підхід зводиться до використання особливостей топології та додавання до неї нових елементів з метою відновити її структурні властивості. Давно відомими є методи такого резервування для гіперкубічної [121, 122], решітчастої [80, 123] та тороїдальної топологій [124, 125]. Важливим аспектом тут є те, що саме топологія визначає конкретну стратегію розміщення резервних вузлів та план дій щодо заміщення при виявленні відмови. Звідси також формуються і накладні витрати.

Ще один цікавий підхід полягає у самовідновленні мережі за рахунок додаткових ребер, без додавання вузлів. Існує ряд алгоритмів такого самовідновлення. Наприклад, авторами робіт [126, 127] запропоновано алгоритми Forgiving Tree та Forgiving Graph, які передбачають, що при відмові вершини її сусіди заміщують її певним підграфом. При цьому вершини цього відновлюючого графа є віртуальними і симулюються сусідами несправної вершини – фактично додаються лише ребра. Більш наочно роботу алгоритму Forgiving Graph продемонстровано на рис. 1.3.



Рис. 1.3. Принцип роботи алгоритму Forgiving Graph

В статті [128] також розглядаються інші подібні алгоритми – xheal та xheal+. Їх особливістю є те, що на відміну від попередніх алгоритмів, що при певних обставинах допускають не лише додавання нових, але і видалення існуючих ребер, дані

алгоритми передбачають реберну монотонність, що дозволяє скоротити накладні витрати, пов'язані із видаленням існуючих ребер.

Загальною проблемою класу самовідновлюючих алгоритмів є потреба в реконфігурованій мережі, оскільки зміна топології залежить від точки відмови і, як наслідок, є складно передбачуваною. Інша проблема – це накладні витрати, що виникають при реконфігурації. Оскільки заміщуючим об'єктом є граф, а заміщуваним – одна вершина, це породжує зміну в топологічних характеристиках і, як наслідок, впливає на такі параметри як пропускна здатність та час передачі повідомлення. Ще більш суттєвим є вплив відновлення на ступінь: наприклад, для алгоритму Forgiving Graph показано, що у максимальному теоретичному випадку цей топологічний показник може потроїтись відносно його початкового значення [127].

Окремо тут варто виділити алгоритм самовідновлення на основі кодів корекції помилок [129]. Його особливістю є досягнення високої відмовостійкості без зростання ступеня порівняно із не-відмовостійкою системою. Крім того, він є детерміністичним і замість повністю реконфігурованої мережі передбачає лише часткову реконфігурованість. В той же час ключовими його недоліками є обмеженість в числі відмов, що можуть бути усунені, і надлишковість по комутаторам, які і забезпечують вищезгадану незмінність ступеня.

Таким чином, поєднуючи техніки обходу та заміщення відмов можна досягти досить високих показників мережевої відмовостійкості. Втім, узагальнюючи дані щодо мережевої надлишковості, варто зазначити той факт, що її ключовим елементом є топологія мережі. Від неї залежать і алгоритми обходу відмов і доступні можливості щодо їх заміщення. Частково це стосується навіть топологічно-незалежних методів. Так, адаптивна маршрутизація опирається на альтернативні маршрути, число яких залежить від графу системи. Аналогічно, методи самовідновлення, які загалом не передбачають прив'язки до конкретної структурної організації, пов'язані із нею через накладні витрати, що безпосередньо залежать від вихідного графу.

Окремим аспектом мережевої відмовостійкості є захист від візантійських помилок, описаний авторами робіт [41, 80, 102], короткий огляд яких приведено в Додатку Д.3.

1.3.4. Високорівневі методи забезпечення відмовостійкості

Під високорівневими методами тут розумітимемо такі, що не маскують відмову, а намагаються вирішити проблему іншими способами: відновити робочий стан системи, захистити сервіси / обчислення / дані від потенційної відмови, яка ще не відбулась, або зробити сам алгоритм стійким до помилок та відмов.

Відновлення після відмови

Складно переоцінити важливість надлишковості в контексті відмовостійкості, проте ефективна боротьба з відмовами передбачає не лише їх маскування за допомогою резервних елементів, але і виправлення ситуації. Доведеним фактом є те, що можливість ремонту складових частин після відмов суттєво, – на декілька порядків, – подовжує напрацювання на відмову системи як такої [92]. Таким чином, відновлення після відмови може бути не менш важливим, ніж її заміщення.

На жаль, проблема апаратного відновлення є досить складною і, як правило, потребує втручання людини. Щодо самовідновлюючих алгоритмів, то вони існують, проте зазвичай є секретом виробника і не є доступними широкому загалу. Втім, доступним є часткове вирішення проблеми через використання польових змінних блоків (field replaceable units) [130–132]. Ідея таких блоків полягає у наявності вбудованих тестів, що дозволяють швидко виявити несправність, замаскувати і повідомити про неї обслуговуючий персонал. Таким чином, відмова оперативно ідентифікується, а несправний блок – вручну замінюється точно таким же резервним.

Іншим варіантом роботи з помилками є їх обробка на програмному рівні. Загалом тут можна виділити декілька основних методів: це пряме відновлення (forward recovery) [133–135], відкат (rollback) [136–138] та семантики відмовостійкості [139, 140].

Ідея прямого відновлення полягає у можливості виправити помилку, що вже сталася, без повторних обчислень та повернення до попереднього стану. Даний метод є актуальним, коли пріоритетом є негайне відновлення роботи – наприклад, в системах реального часу. Втім даний метод є складним у реалізації, а також не дає гарантій відновлення функціонування системи.

Таким чином, в системах, де час відновлення не є критичним, застосовується інший метод, що носить назву зворотного відновлення або відкату. Розділяють відновлення на основі журналу та на основі контрольних точок [14]. В свою чергу, виділяють велику кількість підходів до створення контрольних точок: скоординовані та нескоординовані, блокуючі та неблокуючі, на рівні системи, на рівні користувача та на рівні додатків, ієрархічні контрольні точки, контрольні точки в пам'яті, тощо [16, 83, 141]. Даний метод є значно дешевшим і простішим в реалізації ніж пряме відновлення, проте він не захищає від помилок проектування, які можуть відправити систему у нескінченний цикл «помилка-відновлення». Також існують ситуації, коли зворотне відновлення стану неможливо. Таким чином, використовуваним є поєднання різних методів прямого та зворотного відновлення.

Семантики відмовостійкості, в свою чергу, дозволяють визначити реакцію на той чи інший очікуваний несправний стан. Такими станами можуть бути, наприклад, повна відмова вузла, втрата повідомлення, пошкодження даних та ін. Система, виявляючи проблему, здатна відреагувати визначеним чином і використати методи відновлення, що якнайкраще підходять конкретній ситуації. Втім недоліком цього підходу є його прив'язка до здатності розробників прогнозувати проблему. При виникненні несправності, що не була заздалегідь передбачена у відповідній семантиці, система не зможе коректно відновити своє функціонування.

Передбачення та уникнення відмов

Основними методами передбачення помилок, що ще не відбулися, є статистичний метод [142–144], метод кореляції [145–147] та використання методів машинного навчання [148].

Статистичний метод використовує датчики та моніторинг для виявлення аномальних станів, які підвищують імовірність відмови того чи іншого компоненту. Кореляційний – використовує журнал для збору даних і припускає, що в залежності від типу несправності та розташування вузла існує кореляція між подіями виходу із ладу.

Використання машинного навчання для виявлення відмов поєднує інформацію попередніх методів і передбачає використання таких інструментів як кластеризація k-

середніх [149], машини опорних векторів [150, 151], логістична регресія та дерева рішень [152]. Крім того, в публікаціях розглядається використання нейронних мереж з довгою короткостроковою пам'яттю (LSTM) [153] та штучного інтелекту AIOps [83].

Можливість прогнозування дає змогу відреагувати на відмову, що ще не відбулась, і тим самим скоротити шкоду від неї до мінімуму. Втім загальною проблемою всіх методів прогнозування є їх неточність, до того ж, створення ефективної системи передбачення є непростю задачею коли мова йде про масштаби сучасних суперкомп'ютерів [154, 155].

Основним методом реакції на відмову є міграція даних [85]. Розрізняють швидку міграцію, міграцію з попереднім копіюванням та посткопіюванням, статичну (з очисткою) та динамічну (живу) міграцію [14, 156]. Також виділяють міграцію на рівні процесів та на рівні віртуальних машин. Головним недоліком міграції є її залежність від правильності передбачення відмови, а отже, при її використанні необхідними є додаткові методи відмовостійкості.

Відмовостійкість на рівні алгоритмів

Ідея алгоритмічно-базованої відмовостійкості (algorithm-based fault tolerance, ABFT) полягає в тому, що для деяких алгоритмів існує взаємозв'язок між результатами виконання та контрольними сумами. Це дозволяє реалізувати відмовостійкість з мінімальними витратами на рівні самої задачі і тим самим подолати відмову попри недоліки попередніх методів.

Розділяють офлайн та онлайн підходи ABFT. Перший полягає в перевірці контрольних сум після повного завершення обчислень, що створює складнощі, коли кількість помилок перевищує можливості коригуючих кодів. Онлайн підхід передбачає перевірку в процесі обчислення, проте його недоліком є висока складність розробки таких алгоритмів.

Головною і ключовою проблемою ABFT як методу відмовостійкості є той факт, що існує обмежене число алгоритмів, які можуть бути виконані з використанням даного методу.

1.3.5. Аналіз існуючих методів підвищення відмовостійкості

Розглядаючи зазначені методи, варто зазначити певні їх недоліки. По-перше, якщо мова йде про методи, орієнтовані на апаратне забезпечення, такі як TMR/NMR, варто розуміти, що як правило при розробці суперкомп'ютерів використовується стандартна елементна база. Таким чином, забезпечення такого роду відмовостійкості потребує розробки власної елементної бази і пов'язано із відповідними складнощами та витратами.

Іншими є методи системного рівня, такі як відновлення, передбачення відмов та міграція, а також ABFT. Кожен із них є потужним, особливо в поєднанні із іншими, проте і тут є ряд важливих недоліків. Наприклад, ABFT є одним із найкращих методів в контексті співвідношення ефективності та накладних витрат [16, 80, 83]. В той же час його обмеженням є прив'язка до конкретних алгоритмів. З іншої сторони, міграція даних дозволяє нівелювати негативний вплив відмови на систему – проте цей метод є неточним. Відновлення, в свою чергу, є необхідною складовою відмовостійкості, втім тут теж є свої проблеми. Навіть не беручи до уваги відсутність гарантій щодо відновлення стану, цей метод направлений на виправлення *помилки*, а не *відмов*. В ситуації, коли вузол фізично недоступний чи знаходиться в стані візантійської помилки, відновлення може стати непростою задачею.

Програмні методи відмовостійкості, такі як N-версійне програмування, теж мають свої проблеми. Коли питання стоїть про використання максимуму обчислювальних потужностей системи для вирішення масштабної прикладної задачі (наукові та дослідницькі проекти), розробка паралельного ПЗ потребує серйозних капіталовкладень. Таким чином, резервування програмного рівня стикається з проблемою стрімкого збільшення вартості.

Окремо варто зазначити, що жоден із вищерозглянутих методів ніяк не впливає на комунікаційну мережу системи, відмова якої призведе до колапсу незалежно від характеристик апаратних, системних та програмних засобів відмовостійкості.

Щодо відмовостійкості мережі, то існуючі методи, в більшості своїй, опираються на її топологію. Це певною мірою стосується навіть топологічно-незалежних підходів. Якщо казати про питання обходу відмов, то як правило такі

алгоритми є складнішими ніж їх не-відмовостійкі аналоги, що поглиблює проблему ефективності: затримка на маршрутизацію пакета безпосередньо входить в час його передачі. Таким чином, чим складніший алгоритм – тим повільніше маршрутизація, тим меншою є продуктивність системи. З іншого боку, питання заміщення відмов також не є простим, адже резервування кожного вузла є дорогим, а відновлення на основі ребер потребує реконфігурованої мережі, що неможливо при нинішній кількості вузлів у суперкомп'ютерах.

Таким чином, вразливим місцем комп'ютерної системи є її комунікаційна мережа, а відмовостійкість мережевого рівня опирається на топологію цієї мережі.

1.4. Огляд деяких методів підвищення ефективності КС

1.4.1. Причини низької ефективності паралельних КС

Основою високої продуктивності сучасних суперкомп'ютерів є широкомасштабне розпаралелювання, яке дозволяє вирішувати задачі з надвеликою розмірністю за прийнятний час. Таким чином, мільйони ядер дають змогу розбити вихідну задачу на мільйони частин і тим самим досягти прискорення, недосяжного для звичайних паралельних систем. Втім, використання паралельної обробки в якості «основи» породжує проблеми, які суттєво поглиблюються, коли мова йде про системи надвеликого масштабу.

Загалом, в літературі розглядають ряд проблем, пов'язаних із паралелізмом. Коротко узагальнивши їх, можна виділити наступні:

1. *Проблема декомпозиції задачі* [157], яка полягає у розділенні задачі на паралельні підзадачі. З одного боку, існує верхня межа ефективності розпаралелювання, продиктована законом Амдала. З іншого боку, максимальне розпаралелювання не завжди є оптимальним з практичної точки зору. Ще одним аспектом є той факт, що для деяких задач існує більше одного алгоритму вирішення, тож, варіювання алгоритмом в сторону варіанту з кращими паралельними характеристиками також може розглядатись як можливий підхід до вирішення проблеми.

2. *Проблема залежності по даним* [158, 159], яка пов'язана з тим, що паралельні підзадачі, як правило, залежать від результатів виконання інших підзадач. Це призводить до необхідності передачі даних від потоку-джерела до потоку-споживача, а також потреби в синхронізації.
3. *Проблема багатозадачності та багатопроцесорного виконання* [160]. Дана проблема полягає в накладних витратах, пов'язаних із перемиканням контексту між паралельними процесами в рамках одного обчислювального модуля. Загалом, сучасні процесори, як правило, вирішують цю проблему через скорочення контексту, що дозволяє виконувати перемикання досить швидко.
4. *Проблема планування* [161], пов'язана із розподіленням паралельних підзадач між обчислювальними модулями системи у відповідності із чергою пріоритету. Варто зазначити, що на цей процес суттєво впливає розмір та складність системи, оскільки мільйони та десятки мільйонів паралельних потоків мають бути розподілені між мільйонами ядер. Це породжує похідну проблему *балансування навантаження*, що полягає у тому, що для великої системи досить складно забезпечити рівномірне навантаження на вузли та ядра. Як наслідок, виникають такі явища як простої, черги, конфлікти доступу до різноманітних ресурсів та мережі.
5. *Проблема міжпроцесорної комунікації* [162], яка полягає в складнощах при паралельній передачі даних по мережі, таких як конфлікти пересилки, обмеження пропускної здатності, потреба в обході відмов, проблема відстані між джерелом та приймачем, тощо.

Одним із рішень, що активно використовуються в сучасних системах, є гібридні обчислення. Їх суть полягає в об'єднанні CPU, GPU, FPGA та інших функціональних модулів в рамках однієї системи [72]. Особливою популярністю у розробників користуються графічні прискорювачі, оскільки вони забезпечують високу пікову продуктивність при ціні, нижчій ніж у кластерів CPU [163]. Також перспективним напрямом вважається використання FPGA, що дозволяють завдяки спеціалізації обчислень досягти високої продуктивності на конкретних задачах. Хоча в контексті

швидкості обчислень та роботи з пам'яттю реконфігуровані пристрої поступаються графічним процесорам, втім їх енергоефективність загалом знаходиться на тому ж рівні [164]. Проте, на відміну від гібридності CPU-GPU, даний підхід все ще має значну кількість проблем [165].

В контексті проблеми ефективності гібридизація є гарним рішенням, особливо коли мова йде про матричні/тензорні обчислення та спеціалізовані алгоритми. Втім, проблемою є ускладнення планування, пов'язане з поєднанням різних моделей обчислення. Виникає необхідність розрізняти різні типи задач, що призводить до ускладнення прикладної програми. Це розрізнення відображається на системі, і потребує відповідного ускладнення планувальника, а також імплементації апаратних механізмів швидкої передачі даних між CPU та додатковими модулями. Також структура задачі не гарантує наявності потрібної кількості відповідних обчислень для GPU та FPGA. Таким чином, гібридність може не лише не підвищувати ефективність, а й навпаки, породжувати прості чи вузькі місця.

Окремо варто зазначити, що суперкомп'ютер Fugaku, що є одним з найкращих за реальною продуктивністю на сьогоднішній час, не використовує ні GPU, ні FPGA.

1.4.2. Огляд КС на основі квантового паралелізму

Як було згадано вище, основною причиною низької ефективності сучасних систем є проблеми паралелізму, більша частина яких мають непереборний характер. Таким чином, єдиний спосіб повністю їх подолати – це відмовитись від паралелізму в класичному розумінні і перейти до такої моделі обчислень, яка з одного боку дасть можливість виконувати швидку обробку надвеликих даних, а з іншої – дозволить радикально зменшити число елементів, які працюють паралельно.

Таким рішенням є квантові обчислення. Їх ідея полягає у використанні квантових об'єктів в якості носіїв інформації. Це дозволяє працювати не над окремими даними, а над всім простором значень. Таким чином, з'являється так званий квантовий паралелізм. Проте варто зазначити, що такі обчислення мають строго імовірнісний характер.

На основі робіт [166–190] було виконано огляд сучасних систем та елементів для квантових обчислень, включаючи такі класи як універсальні квантові комп'ютери,

малі квантові пристрої та спеціалізовані квантові обчислювачі. Виділено їх ключові проблеми. Цей огляд приведено в Додатку Д.4.

Узагальнюючи результати цього огляду, можна виділити наступне. З одного боку, існує великий потенціал використання (спеціалізованих) квантових систем як прискорювачів в рамках парадигми гібридних обчислень, що дозволить суттєво підвищити ефективність в рамках обмеженого спектру задач. З іншого боку, рано казати про те, що квантовий паралелізм може замінити класичні паралельні обчислення. Наявність технічних обмежень, висока вартість, складність практичного застосування – ці проблеми суттєво обмежують реальне використання квантових систем.

1.4.3. Огляд КС, які керуються потоком даних

Потенційним рішенням проблеми ефективності паралельних комп'ютерних систем є архітектура, що керується потоком даних (dataflow), завдяки своїй здатності динамічно керувати паралельними обчисленнями без необхідності застосування додаткових інструментів синхронізації потоків. Її ранні реалізації мали масу недоліків, в той же час основні її аспекти були інтерпретовані та інтегровані в таких сферах як програмування [191], оптимізація компіляторів [192], автоматичне розпаралелювання [193], Big Data [194], тощо.

В якості основи парадигми dataflow можна виділити наступні аспекти:

1. Представлення програми у вигляді, що передбачає доповнення операцій (команд) їх залежностями по даним.
2. Позачергове виконання (по готовності даних).
3. Автоматичне призначення задач обчислювальним ресурсам.

В сучасному світі існує багато рішень, які частково чи повністю реалізують дану модель з певними доповненнями / модифікаціями, використовуючи її безпосередньо для обчислень. Загалом їх можна розділити на наступні основні категорії:

1. Архітектурні рішення, що передбачають наявність фізичного процесора, який містить в собі елементи dataflow. Прикладами цього є сучасні суперскалярні процесори [195], векторні dataflow процесори [196, 197], гібридні системи [198], тощо.

2. Обчислювальні моделі, що можуть допускати апаратну реалізацію, проте не прив'язані до неї. Яскравим прикладом моделі, заснованої на dataflow, є подійно-орієнтовані обчислення, – наприклад, Codelet [199].
3. Розподілені dataflow системи [200], що базуються на програмній реалізації парадигми dataflow і (досить часто) використовують її для задач обробки великих даних та машинного навчання [201].

Розглядаючи апаратні реалізації, не можна не виділити RISC та суперскалярну архітектуру, що фактично є прямими наступниками класичної парадигми dataflow [202]. Їх основою є конвеєризація на обмеженому наборі команд [203], що загалом мало перетинається із концепцією dataflow, втім в нинішній час існує тенденція до поєднання набору команд RISC [204] та суперскалярності [195] з аспектами керування на основі потоку даних. Важливою особливістю даної інтеграції, яку варто відмітити, є той факт, що визначення залежності по даним в цих системах відбувається апаратно, – тобто, програма автоматично перетворюється процесором із control-flow в dataflow. Втім, наслідком цього є суттєве зростання апаратної складності. Крім того, ще одне обмеження накладається конвеєром команд та проблемою переходів, що потребує окремої схеми-оракула для передбачення розгалужень [205] і в кінцевому результаті також збільшує апаратні витрати.

З іншого боку, програмна реалізація цих аспектів передбачає спрощення апаратури, проте її проблемою є той факт, що апаратне прискорення системи зберігає обмеження, пов'язані із фізичною архітектурою. Причому це стосується як програмування, так і автоматичного розпаралелювання та керування конвеєрами виконання.

Також не можна не згадати і про інший напрямок – dataflow пристрої на базі FPGA. Загалом існує великий спектр такого роду пристроїв, що включає в себе пристрої компанії Maxeler [206, 207], крупнозернисті реконфігуровані масиви (Coarse Grain Reconfigurable Architecture, CGRA) [208–210], систолічні масиви [211]. Важливо відмітити, що такого роду рішення принципово відрізняються від класичної dataflow архітектури і передбачають, по суті, створення апаратного забезпечення під кожну конкретну програму чи задачу. Відповідно, програмою тут є не граф задачі, а код на

мові програмування схеми (VHDL, Verilog, тощо), що реконфігурує схему під чергову задачу.

З одного боку, такий підхід дійсно може дати прискорення, адже система стає ідеально відповідною до конкретної задачі. Також зникають проблеми планування та накладні витрати, оскільки всю роботу щодо обробки залежностей по даним бере на себе апаратура, що виконує обчислення. Звісно, це призводить до виникнення іншої проблеми: в кожен момент часу пристрій може виконувати лише обмежений набір задач. Використання динамічної часткової реконфігурації [212] може зменшити вплив даного недоліку, проте такий пристрій все одно втрачатиме продуктивність у випадках, коли виникатиме потреба у зміні задачі. Іншим аспектом, що обмежує використання FPGA в сучасних суперкомп'ютерах є проблеми недосконалості самих FPGA у порівнянні з іншими альтернативними рішеннями. Мова йде про нижчу тактову частоту, проблеми з енергоефективністю та затримки роботи з пам'яттю, які роблять ефективність подібних пристроїв дискусійною [164, 165].

1.4.4. Підвищення ефективності на основі натхнених природою обчислень

Існує спектр задач, для яких ідеальним рішенням є використання квантового алгоритму, проте сучасні квантові комп'ютери мають суттєві обмеження. В той же час обчислення на основі потоку даних в різних їх варіаціях дозволяють автоматизувати керування паралелізмом – проте вони нездатні подолати фундаментального обмеження, яке лежить в основі проблематики. Таким чином, якщо зміна апаратної основи не може вирішити проблему, альтернативою стає зміна методу вирішення задачі.

Одним із підходів є використання натхнених природою обчислень, що мають широку сферу застосування, і в той же час мають високу ступінь внутрішнього паралелізму. До них відносяться генетичні алгоритми (ГА), а також штучний інтелект (ШІ). В роботах [172, 213–222] описано застосування даних методів для рішення класичних задач, що дозволяє розширити паралельні можливості задачі. Іншим підходом є використання ГА та ШІ для рішення задач розпаралелювання та

планування, що було розглянуто авторами праць [223–230] Детальніший огляд цих робіт приведено у Додатку Д.5.

1.4.5. Аналіз розглянутих методів підвищення ефективності КС

Перш ніж підбивати підсумок розглянутим методам, варто зазначити, що методи підвищення ефективності не обмежуються описаним переліком. Розглядаються різні підходи, що включають в себе і принципово нову елементну базу, і принципово нові методи взаємодії, і багато інших речей. Деякі із них є досить перспективними, інші поки що ближче до наукової фантастики, але загалом їх можна розбити на наступні категорії:

1. *Апаратні, що передбачають нову елементну базу.* Сюди можна віднести квантові, оптичні [231], нейронні комп'ютери [232], хімічні [233] та генетичні [234] пристрої для зберігання даних.
2. *Апаратні, що використовують вже існуючу елементну базу.* Сюди можна віднести більшість методів, що вже використовуються: гібридні обчислення на основі GPU та FPGA, різноманітні паралельні (мікро)архітектури, які є розвитком таких класичних рішень як dataflow (суперскалярна архітектура), VLIW (архітектура EPIC), NUMA (мікроархітектура AMD Zen) та ін.
3. *Мережеві рішення, що направлені на прискорення пересилки, мінімізацію міжвузлових відстаней та максимізацію пропускну здатності.*
4. *Програмні рішення, що включають в себе різноманітні методи автоматизації паралелізму та оптимізації планування, а також орієнтовані на паралелізм обчислювальні моделі.*

Аналізуючи приведені категорії, варто відмітити, що нова елементна база передбачає потребу в розробці цілого спектру нового системного на програмного забезпечення, складної інтеграції нових та старих систем, розробки нових протоколів та засобів взаємодії. Вкрай багатообіцяючими серед нових елементів є квантові, втім вони мають купу недоліків, що було показано в параграфі 1.3.2. Щодо інших наведених рішень, вони можуть при певних обставинах давати гарні показники продуктивності та енергоефективності, втім їх вплив на ефективність обчислень в

комп'ютерних системах дуже далекий від того, що пропонується квантовими комп'ютерами.

Існуюча елементна база на основі давно відомих паралельних архітектур, в свою чергу, активно використовується в сучасних системах. Досить цікавим рішенням в контексті ефективності паралелізму є парадигма dataflow, втім її використання на апаратному рівні веде до зростання апаратної складності. Загалом дана категорія рішень вже застосовується в сучасних суперкомп'ютерах, отже, приведені раніше дані TOP 500 вже враховують більшу частину ефекту від них. Тож їх подальша інтеграція може дати деякий ефект, але не вирішити проблему загалом.

Багатообіцяючими є підходи автоматичного розпаралелювання. Виділити тут можна три майже рівноцінні «інтелектуальні» методи: штучний інтелект, еволюційні обчислення та нечітку логіку, та великий спектр класичних рішень, що включає в себе аналіз потоків даних, аналіз залежностей, аналіз циклів, тощо [235, 236]. Суттєвим обмеженням кожного із приведених методів є те, що отримання оптимального (чи хоча б субоптимального) результату не гарантується.

Рішенням, яке на сьогоднішній час широко використовується та розвивається, є планування обчислень в комп'ютерних системах, проте їх ефект значною мірою залежить від структури системи – у випадку суперкомп'ютера ця структура описується комунікаційною мережею.

1.5. Топологічні рішення сучасних високопродуктивних КС

Розглянуті в параграфах 1.3 та 1.4 рішення здаються привабливими на перший погляд, проте мають ряд суттєвих недоліків, таких як вразливість комунікаційної мережі в контексті відмовостійкості та відсутність дієвих рішень в контексті ефективності обчислень. Втім, точкою перетину цих двох предметних областей є комунікаційна мережа комп'ютерної системи, від якої залежить і обчислювальна ефективність, і вартість, і стійкість системи як цілого до відмов окремих елементів.

Характеристики такої мережі залежать від багатьох факторів, таких як мережеві протоколи, обладнання, технології передачі даних, тощо. Втім, ці речі, як правило, є стандартними, тож їх вплив на відмовостійкість та ефективність системи є

обмеженим. З іншого боку, такий фактор як топологія комунікаційної мережі комп'ютерної системи, навпроти, є визначальними для характеристик мережі незалежно від конкретного обладнання (хоча обладнання та технології накладають певні обмеження та вимоги до мережевої топології).

Задача побудови оптимальної топології не є новою, втім відомим фактом є відсутність ідеального для всіх випадків рішення. Але можливими є часткові рішення, направлені на покращення окремих характеристик чи досягнення компромісу між різними параметрами. За рахунок цього синтез нових топологій дозволяє підвищити ефективність та відмовостійкість комп'ютерних систем.

1.5.1. Топологічні характеристики та критерії ефективності топологій

Основними топологічними характеристиками графа є число вузлів N , число ребер R , ступінь S (іноді також позначається як Δ), діаметр D , середній діаметр \bar{D} , топологічний трафік T , локальна зв'язність L та глобальна зв'язність G .

В літературі, присвяченій відмовостійким мережам, найбільш актуальними характеристиками вважаються мультиплікативний критерій ступеня та діаметру (SD), який базується на проблемі (Δ, D) [237] та діаметростабільність DS [80], детальніший опис яких наведено в Додатку Д.6. В той же час, задача ефективності включає в себе питання відмовостійкості і пов'язані із ним параметри, проте додає свої акценти, а саме:

- *Пропускна здатність*, що в контексті топології визначається діаметром, кратністю зв'язків та кількістю альтернативних маршрутів між вузлами.
- *Затримка маршрутизації*, яка залежить від обраного методу та складності алгоритму і в контексті топології визначається методом її синтезу.

Як було згадано раніше, *ефективність комп'ютерних систем* поділяється на *ефективність обчислень* (тобто, відношення прискорення паралельних обчислень до числа ядер), *ефективність передачі даних* (яка визначається пропускнуою здатністю мережі), *вартість системи* (що залежить від характеристик мережевого обладнання) та *відмовостійкість*. В контексті топології неможливо казати про обчислення,

оскільки цей параметр залежить від обладнання, втім з характеристиками топології можна зв'язати інші параметри. Розглянемо їх детальніше:

- *Ефективність передачі даних* залежить від відстаней між вершинами у графі (топології). Ключовими топологічними характеристиками для нього є діаметр D та середній діаметр \bar{D} . Також сюди можна віднести метрику топологічної ефективності E , що визначається як середня мультиплікативно обернена відстань між всіма вершинами графа.
- *Вартість системи* залежить від вартості мережевого обладнання, для якої ключовою є ступінь S , а також (меншою мірою) число ребер R (які визначають ціну лінків і підлягають мінімізації).
- *Відмовостійкість* є комплексною характеристикою, що потребує окремого аналізу та дослідження, втім однозначно залежить від локальної зв'язності L та числа ребер R (які показують резервні канали і підлягають максимізації).

Щоб вирішити проблему неоднозначності трактування характеристики R , варто виділити аспект *ефективності використання ребер*, тобто, потенційні можливості системи до завантаження наявних зв'язків. В контексті топологій цей параметр може вважатись еквівалентним характеристиці топологічного трафіку T .

На підставі вищенаписаного, виділимо наступні критерії *ефективності топологій* (графів, топологічних організацій, топологічних рішень):

- *Мультиплікативна характеристика ступені та діаметра (SD)*, що дозволяє визначити ефективність графа з точки зору проблеми (Δ, D) . Даний параметр поєднує в собі вартість і максимальну затримку передачі, а отже, підлягає мінімізації.
- *Топологічний трафік T* , що показує повноту використання системою зв'язків і дозволяє комплексно аналізувати параметри вартості та надлишковості, а також визначати потенційне виникнення вузьких місць. Передбачається, що ідеальним значенням T є 1, втім, враховуючи потреби відмовостійкості, прийнятними можна вважати також значення в діапазоні 0,5 – 1,0. Значення нижче 0,5 показують завищеної надлишковості, в той час як значення вище 1,0

свідчать про ризик зниження ефективності передачі даних через перевантаження зв'язків (що, потенційно, збільшує і ризик відмови).

- *Топологічна ефективність E* , яка може використовуватись для грубого теоретичного визначення пропускної здатності мережі КС і підлягає максимізації. Втім даний параметр є суто теоретичним і потребує врахування можливостей системи щодо використання цієї пропускної здатності (тобто, має аналізуватись сукупно із топологічним трафіком).

1.5.2. Топології сучасних суперкомп'ютерів

В сучасному науковому дискурсі розглядається дуже велика кількість різноманітних топологій та методів синтезу. Повний огляд існуючих рішень не є можливим, тож є сенс обмежити його, зробивши припущення: найактуальнішими для огляду є ті топологічні рішення, що використовуються в найпродуктивніших суперкомп'ютерах. Список цих рішень продемонстровано в таблиці 1.2.

Табл. 1.2.

Топологічні рішення сучасних суперкомп'ютерів TOP 10

№	Назва	Топологія	№	Назва	Топологія
1	Frontier	Dragonfly [238]	6	Leonardo	Dragonfly+ [243]
2	Aurora	Dragonfly [239]	7	Summit	Non-blocking fat tree [244]
3	Eagle	Гіперкуб [240]	8	MareNostrum 5 ACC	Fat tree [245]
4	Supercomputer Fugaku	6D top [241]	9	Eos NVIDIA DGX SuperPOD	Leaf/spine [246]
5	LUMI	Dragonfly [242]	10	Sierra	Fat tree [244]

Важливо відмітити, що приведені рішення в більшості своїй є варіаціями певних стандартних мережевих організацій. Так, і жирне дерево, і топологія leaf/spine є деревовидними мережами. Цей факт дозволяє виконати агрегацію існуючих рішень, тим самим спростивши їх огляд.

1.5.3. Деревовидні топології

Жирне дерево, також відоме як k -арне n -дерево, базується на дуже простому принципі: ближче до кореня – вище пропускна здатність. В цій топології обчислювальні вузли агреговані в листі, а вершини вище реалізовані комутаторами, де між рівнями присутні надлишкові зв'язки [189].

Розглядаються різні варіанти побудови жирних дерев в залежності від того, який аспект є більш пріоритетним: кількість вузлів на комутатор чи пропускна здатність. На рис. 1.4 продемонстровано 4 приклади жирних дерев, що включають в себе повне, кінцеве, однопланове та багатопланове жирне дерево.

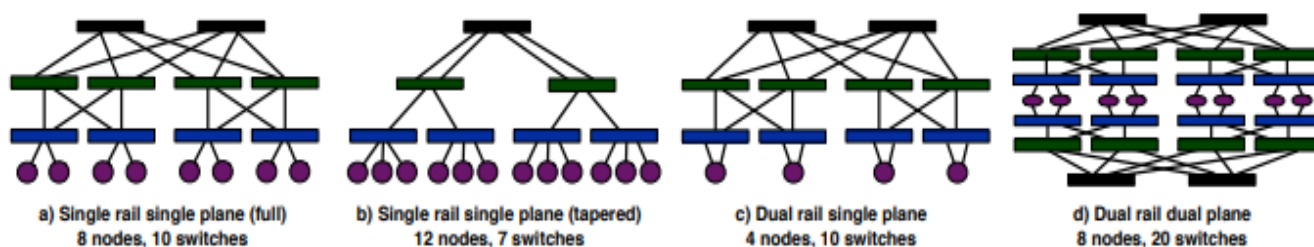


Рис. 1.4. Варіації жирних дерев за структурою [248]

В той же час існує проблема, пов'язана зі зміною пропускної здатності між рівнями дерева. Таким чином, окрім класичного (повного) жирного дерева виділяють узагальнене жирне дерево, де замість зміни пропускної здатності використовується багатокореневість. Втім така заміна породжує можливість блокування. Його підтипами є розширене узагальнене жирне дерево, яке дозволяє різну кількість з'єднань між рівнями [249], та паралельно-портове узагальнене жирне дерево, що використовується в реальних системах та застосовує кратні зв'язки для забезпечення необхідної зміни пропускної здатності [250].

Основною проблемою жирних дерев є блокування, які вирішуються або через ускладнення мережевої структури, або через маршрутизацію. Тут варто зазначити, що для повного (неблокованого) жирного дерева пропускна здатність має зростати в K разів з кожним підйомом на рівень, що суттєво ускладнює досягнення неблокованості через кратні зв'язки. Іншою проблемою є досить високий діаметр – близький до діаметра звичайного дерева. Таким чином, з масштабуванням складність такої мережі неухильно зростатиме, а характеристики – погіршуватимуться.

Дещо іншу концепцію пропонує leaf/spine. Це дерево, обмежене 3 рівнями, де є обчислювальні вузли, комутатори листя та корені, причому кожен комутатор листя зв'язаний з усіма кореневими, а кожен кореневий – з усіма комутаторами листя (рис. 1.5).

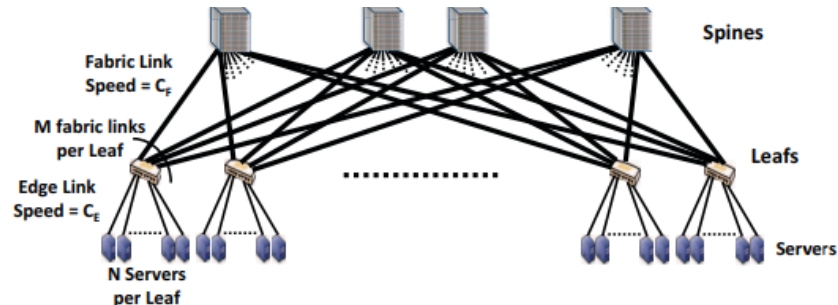


Рис. 1.5. Топологія leaf/spine [251]

Очевидною проблемою даної мережевої організації є велика ступінь, що виникає внаслідок міжрівневої повнозв'язності. Крім того, оскільки принцип зміни пропускної здатності тут той же, що і в жирному дереві, це породжує ті самі проблеми із блокуваннями та способами їх уникнення.

1.5.4. Тороїдальні та гіперкубічні топології

В даному класі варто виділити 2 основні топології – це 3D та 6D тор. Перша мережева організація є досить простою: на відміну від класичного (2D) тору, вона має 3 виміри і масштабується по осям x , y , z .

Втім, коли мова йде про 6D тор, який використовується в суперкомп'ютері Fugaku, варто розуміти, що це багаторівнева мережа, що поділяється на зовнішній 3D тор та групу з 12 вузлів, поєднаних у несиметричний тор розмірності $2 \times 3 \times 2$ [241]. Таким чином, коли мова йде про масштабування, його параметри відрізнятимуться від передбачуваних для «нормального» 6D тора.

Гарно відомими є і гіперкубічні мережі. Класичний гіперкуб є ідеальним з точки зору топологічного трафіку, проте має проблему, пов'язану із відмовостійкістю, що потребує введення в систему додаткових вузлів. Таким чином, розглядається покращений гіперкуб, що має додаткові з'єднання між найвіддаленішими вузлами кожного тривимірного підкуба (рис. 1.6).

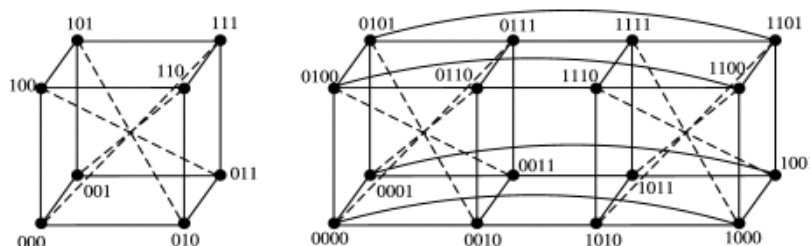


Рис. 1.6. Покращений гіперкуб та його масштабування [252]

Таблиця 1.3 демонструє основні параметри описаних топологічних організацій відносно рангу масштабування r .

Табл. 1.3

Основні характеристики тороїдальних мереж

Характеристика	Тор				Гіперкуб	
	2D	3D	6D	6D (Fugaku)	Класичний	Покращений
Число вузлів N	r^2	r^3	r^6	$12r^3$	2^r	2^r
Ступінь S	4	6	12	12	r	$r+1$
Діаметр D	r	r	r	$r+3$	r	$r-1$

Важливим недоліком обох видів мереж є їх топологічні характеристики. З одного боку, тор має незмінну ступінь, що гарно для обладнання, оскільки масштабування такої системи не потребуватиме заміни комутаційної елементної бази. В той же час ступінь 6, фактично, є ідеальною для безпосередньо зв'язаних мереж, так само як і ступінь 12, для якої достатньо дуже простих комутаторів та/або двоканальних зв'язків. Крім того, тороїдальна мережа досить просто масштабується: немає необхідності додавати елементи по всім осям – за необхідності це можна зробити для лише однієї осі. З іншого боку, проблемою є зростання цих характеристик з масштабуванням: навіть часткове розширення мережі (по одній осі) призводить до того, що діаметр росте так, ніби масштабування було повним. В свою чергу, швидкість повного масштабування є кубічною, що досить повільно для системи, яка має включати в себе мільйони вузлів. Така система матиме високий ранг, а це приведе до досить високого діаметру.

Інша топологія, гіперкуб, має високу швидкість масштабування, проте зміна ступеня створює проблему: щоб розширити таку систему, необхідно змінити

мережеве обладнання, що породжує витрати. Крім того, при високих рангах ступінь гіперкуба буде занадто високим для безпосередньо-зв'язаних та слабо-комуртованих мереж, але в той же час за критерієм (Δ, D) така топологія поступатиметься повноцінним комуртованим мережам.

1.5.5. Частково-повнозв'язні топології

Концепція часткової повнозв'язності полягає в дуже простому факті: ідеальна з точки зору діаметру топологія – повнозв'язна. Вона ж є ідеальною і з точки зору відмовостійкості, оскільки число альтернативних шляхів (потенційних обходів) в ній прямо пропорційне числу вершин. Але така мережа є утопічною в контексті сучасних систем.

Втім, альтернативою повнозв'язності справжньої є повнозв'язність часткова: коли лише деякі частини системи (групи) пов'язані повною системою зв'язків і між цими групами також існує повна система зв'язків. Прикладом такої мережі є бабка (dragonfly).

Її схему зв'язків можна описати наступним чином: кожен комутатор в групі зв'язаний з кожним іншим комутатором в цій групі. При цьому кожен комутатор має один чи декілька зовнішніх зв'язків, так що кожна група поєднана з будь-якою іншою групою. При цьому до кожного комутатора підключені декілька обчислювальних вузлів (так само як це відбувається у жирному дереві). Dragonfly є варіативною топологією, структура якої визначається четвіркою параметрів (p, a, g, h) . При цьому p – кількість термінальних підключень вузлів до комутатора, a – це кількість комутаторів в кожній групі, g – число груп, а h – число зовнішніх зв'язків між групами [58, 187].

Втім навіть часткова повнозв'язність є складною для реалізації, тож для підвищення параметру p було запропоновано використання leaf/spine топології на рівні групи. Така модифікація носить назву dragonfly+. В ній комунікацією з p вузлами займаються l листових роутерів, що мають повну систему зв'язків із s корневими роутерами. Ті, в свою чергу, комунікують із h корневими роутерами інших груп. Ні l ні s роутери не мають зв'язків із роутерами «свого» типу. На рис. 1.7 представлено порівняння структури групи dragonfly та dragonfly+.

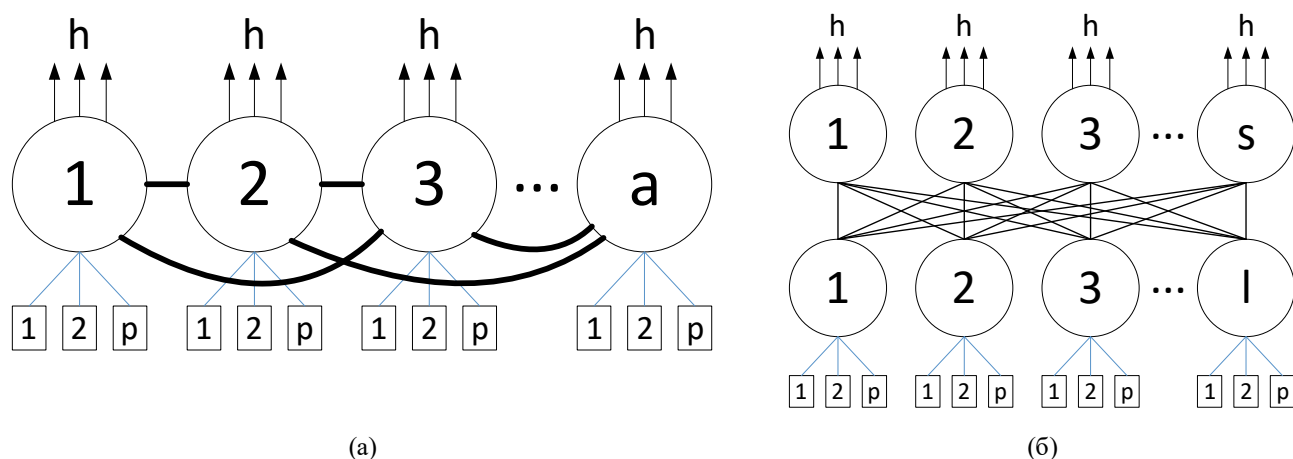


Рис. 1.7. Група в топології (а) dragonfly, (б) dragonfly+ [58, 253]

З точки зору діаметру обидві топології є майже ідеальними, оскільки забезпечують статичний діаметр 3 та 5 відповідно (якщо рахувати від комутаторів листя). При цьому версія Dragonfly+ забезпечує більші розміри груп, і як наслідок – менший середній діаметр. Проте ступінь топології в обох випадках є дуже високим, що потребує тонкого балансування між параметрами топології та можливостями апаратури. Хоча теоретично масштабування є простим, проте з огляду на цей факт реальне додавання вузлів може потребувати заміни як мінімум частини мережевого обладнання.

Ще однією топологією, яка також може бути віднесена до класу псевдо-повнозв'язних, є HyperX. Вона передбачає наявність декількох вимірів, в рамках яких існує повна система зв'язків. При цьому, як і в топології dragonfly, вузлами HyperX є комутатори, до яких приєднуються обчислювальні вузли. Приклади даної топології в різних її варіаціях продемонстровано на рис. 1.8.

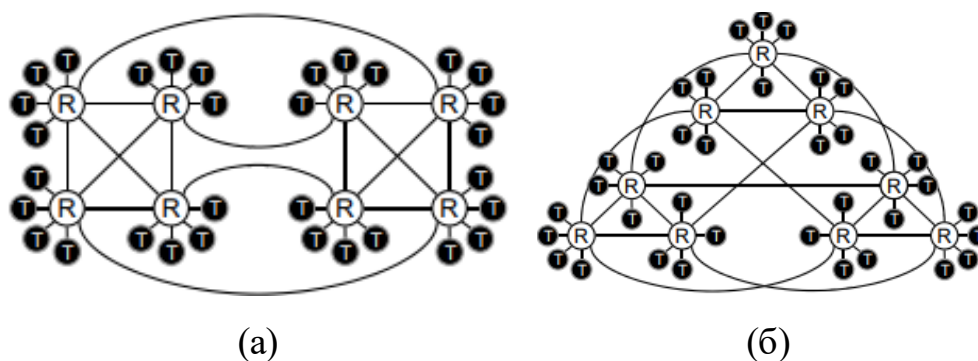


Рис. 1.8. Двовимірна топологія HyperX: (а) 4x2, (б) 3x3 [44]

У порівнянні з dragonfly така мережева організація може мати менший діаметр. Крім того, її маршрутизація простіша: достатньо простої операції заміщення над кодом вершини. Проте її ефективність також упирається в проблему (Δ, D) , оскільки гіперкуб та тор, які лежать в основі, мають посередні характеристики.

1.6. Узагальнення проблематики та постановка задач дисертаційної роботи

В даному розділі було розглянуто 2 проблеми: проблема відмовостійкості та проблема ефективності. Кожна із них передбачає свої підходи до вирішення, проте існує і спільний аспект, пов'язаний із топологією КС. Було виконано огляд існуючих топологічних рішень, такий як Fat tree та Dragonfly, що наразі використовуються в суперкомп'ютерах TOP 10, а також альтернативних варіантів, таких як HyperX.

Існуючі рішення, особливо частково-повнозв'язні, здаються привабливими, проте мають різноманітні недоліки, такі як висока вартість та зависока надлишковість. Це в результаті призводить до зниження ефективності.

Таким чином, актуальною є розробка методів підвищення відмовостійкості та ефективності комп'ютерних систем на основі синтезу нових топологій для цих систем. Можна виділити 2 ключові види топологій:

- Топології для безпосередньо-зв'язаних мереж (БЗМ), такі як гіперкубічні та тороїдальні графи
- Топології для комутованих мереж, такі як Fat Tree, Dragonfly, HyperX, тощо.

В контексті дисертаційного дослідження доцільно буде розглянути обидва варіанти, оскільки вони мають свої переваги, свої сфери застосування, а також свої обмеження.

Висновок до розділу 1

Аналіз можливостей сучасних комп'ютерних систем показав, що на сьогодні актуальною є розробка методів та засобів підвищення ефективності та відмовостійкості топологій комп'ютерних систем, які за рахунок синтезу нових топологічних організацій дозволяють підвищити ефективність комп'ютерних систем та забезпечити високий рівень відмовостійкості без надмірної надлишковості.

В процесі огляду методів підвищення відмовостійкості та методів підвищення ефективності комп'ютерних систем було виділено, що характеристики даних методів залежать від характеристик комунікаційної мережі комп'ютерної системи, які визначаються топологією даної мережі. Таким чином, синтез нових топологій дозволить підвищити ефективність комп'ютерних систем, до якої входять ефективність передачі даних, вартість та відмовостійкість.

Аналіз літературних джерел показав, що хоча топології сучасних комп'ютерних систем забезпечують низький діаметр, високу топологічну ефективність та передбачають велику кількість альтернативних маршрутів. Одним із таких популярних рішень є жирне дерево та dragonfly. Проте їх суттєвими недоліками є висока вартість, зміна ступеня з масштабуванням, що породжує потребу в оновленні обладнання при розширенні системи, а також складність ефективного використання наявних надлишкових зв'язків. Інші рішення, такі як гіперкуб та тороїдальні графи (включаючи топологію суперкомп'ютера Fugaku) мають кращі показники ступеня та топологічного трафіку – проте мають проблеми із топологічною ефективністю. Аналіз рішень на основі наукових публікацій показав, що з усіх розглянутих топологій немає таких, які б дозволили одночасно забезпечити низькі значення мультиплікативного критерія ступеня та діаметра, помірно високі показники топологічної ефективності та прийнятні показники топологічного трафіку, при цьому забезпечуючи високу відмовостійкість.

Таким чином, на основі оглянутого матеріалу було поставлено наступні задачі:

- Розробити математичну модель топології на основі надлишкового коду, яка дозволяє на основі характеристик коду прогнозувати максимальне

число вершин з однаковими номерами та кількість вершин з унікальними номерами.

- Розробити спосіб формування імпліцитних кластерів в надлишкових топологіях, що дозволяє формувати ребра між вершинами з однаковим номером для топологій на основі деяких надлишкових кодів.
- Удосконалити метод синтезу відмовостійких топологій на основі надлишкового коду, який дає змогу створити нові імпліцитні кластери в надлишковій топології, а також розкласти надлишковий граф на підграфи наперед відомої форми.
- Розробити метод масштабування ієрархічних топологій що дозволяє поєднати відмовостійкі топології, синтезовані на основі надлишкового коду, із класичними топологіями, такими як гіперкуб та dragonfly.
- Розробити спосіб моделювання відмов в топологіях для аналізу імовірності розриву зв'язності графа при заданій кількості відмов вузлів, підрахунку топологічних характеристик та їх зміни відносно безвідмовного стану топології.
- Розробити програмне забезпечення для моделювання характеристик топологій з метою експериментально дослідити топологічні характеристики графів та довести вищу ефективність запропонованих топологій у порівнянні із класичними.
- Розробити програмне забезпечення для моделювання відмов в топологіях з метою експериментально дослідити поведінку запропонованих топологій в умовах наростаючого числа відмов і таким чином довести їх вищу відмовостійкість у порівнянні з класичними топологіями

РОЗДІЛ 2

МАТЕМАТИЧНА МОДЕЛЬ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ НА ОСНОВІ НАДЛИШКОВОГО КОДУ

2.1. Взаємозв'язок відмовостійкості та ефективності топологій

Як було продемонстровано в Розділі 1, топологія є основою і для відмовостійкості, і для ефективності передачі даних в КС. В контексті відмовостійкості важливими є альтернативні маршрути, локальна/глобальна зв'язність та можливість зручного виконання обходу/заміщення відмов. В контексті ефективності передачі даних ключовим є діаметр, простота маршрутизації та висока пропускна здатність. Втім, не можна забувати і про інший аспект – вартість. Проблема не лише в тому, щоб спроектувати відмовостійку та продуктивну систему, але і у тому, щоб зробити її економічно доцільною. І це має бути закладено ще на етапі проектування топології. На рис. 2.1 наочно продемонстровано різні характеристики топологій та властивості пов'язаних із ними аспектів, що мають важливість для кожної із описаних проблем.



Рис. 2.1. Аспекти ефективності топологій

Загалом вартість є дуже специфічною характеристикою і в першу чергу залежить від обладнання (системний рівень). Втім, деякі топологічні характеристики безпосередньо визначають вимоги до цього обладнання, як-от ступінь, від якої залежить число портів комутаторів. Крім того, варто розуміти, що будь-яка надлишковість (чи то вузли, чи то ребра) в контексті вартості – це додаткові апаратні витрати. Те ж можна сказати і про маршрутизацію: чим складнішим є алгоритм – тим більшою має бути складність маршрутизаторів для підтримки прийнятної пропускної здатності. Це приводить до потреби пошуку золотієї середини: максимальний обхід, проте мінімальна складність алгоритму; мінімальна ступінь та мінімальний діаметр (SD) для максимального числа вузлів; суттєва надлишковість, проте не зовелика. Загалом ці параметри є взаємовиключними, проте в контексті того, що мова йде про топологію КС, є одна точка, де можливий майже ідеальний компроміс – це маршрутизація. Відмовостійка маршрутизація на основі елементарних кодових перетворень. А це, в свою чергу, підводить до іншої задачі – топологічного синтезу на основі цих самих перетворень.

Таким чином, завданням даного розділу є синтез топологічних організацій з використанням (повністю чи на певних рівнях) методу кодових перетворень з метою отримання рішень із мінімальною SD -характеристикою та якнайпростішою відмовостійкою маршрутизацією.

Існує багато методів синтезу графів. Одним із найпростіших і в той же час доволі ефективних є синтез на основі кодів. Прикладами таких топологій є гіперкуб та граф зсувів-вставок, що синтезується на основі послідовностей де Бруїна (де Брейна, де Брюїна, de Bruijn). Оскільки ребра графа формуються через перетворення над певним n -розрядним кодом, кожен вузол може запросто визначити своїх сусідів та сформував шлях, знаючи лише власний код і код вузла призначення. Це означає, що для маршрутизації не потрібні ні громіздкі таблиці, ні складне налаштування – лише елементарні перетворення над кодом певної системи числення.

Звісно, найпопулярнішим у такому синтезі є бінарний код, втім інші види кодів та систем числення (трійкова, четвіркова, b -кова) також розглядаються науковцями. Проте класичний метод передбачає використання лише «чистих» систем, тобто, з

позитивною основою b і алфавітом від 0 до $b-1$. В той же час теорія систем числення знає й інші варіанти – такі як нега-позиційна система чи-от надлишкова бінарна, на якій варто зупинитись детальніше.

Специфікою надлишкового бінарного представлення (RBR) та інших подібних систем є той факт, що один і той же номер в ній може мати багато кодових представлень. Це дозволяє створити певне підґрунтя для відмовостійкості, адже різні вершини топології можуть репрезентувати одну і ту саму сутність, а це дає змогу без змін на рівні маршрутизації чи системного представлення робити такі дії як маскування, обхід та заміщення відмови, балансування навантаження, відновлення після помилки та ситуативне резервування, тощо.

Тож, коди, що мають надлишковість на рівні представлення, це малодосліджена предметна область, яка в поєднанні з відомими методами синтезу має надати досить просте та елегантне рішення існуючої проблеми.

2.2. Дослідження надлишкових кодів для синтезу топологій

2.2.1. Основа теорії надлишкового кодування

Надлишкове бінарне представлення є досить простим, проте не єдиним можливим. В той же час саме код визначає, які представлення є альтернативними, а які – ні, що фактично є основною суттю та відмінністю методу надлишкового топологічного синтезу. Таким чином, розширення методу потребує детальнішого розгляду теорії систем числення.

Будь-яка позиційна система числення (СЧ) A_b може бути описана двома параметрами – алфавітом $A = \{k, k+1, \dots, m-1, m\}$, що містить цифри системи, де $k < m$, та основою числення b , яка визначає вагу розрядів. Відповідно, оскільки мова йде про систему числення, логічно, що для будь-якого n -розрядного коду $D_n = d_{n-1}d_{n-2} \dots d_1d_0$ (де кожне $d_i \in A$) існує значення V , таке, що:

$$V = v(D_n) = \sum_{i=0}^{n-1} d_i b^i \quad (2.1)$$

Тож, існує чітка відповідність між множиною кодів та множиною значень, що можуть кодуватись n розрядами. В залежності від значень b та складу A виділяють різні види систем числення. Наприклад:

- *Класичні системи числення*, де $A = \{0, \dots, k-1\}$, $b \geq 2$ і $k = b$. Типовим прикладом таких систем є двійкова, трійкова та десяткова системи числення. Характерною їх особливістю є унікальність кожного числового представлення, тобто, $\forall V > 0 \exists! D_n \in A_b$, де $n = \log_b(V + 1)$.
- *Нега-позиційні системи числення*, де $A = \{0, \dots, k-1\}$, $b \leq -2$ і $k = -b$. Особливістю таких систем числення є можливість простого представлення від'ємних чисел. При цьому додатні числа мають парне число цифр, а від'ємні – непарне. Кожен код в цій системі представляє унікальне число, причому $n = \log_b(V + 1) + 1$.
- *Знако-розрядні системи числення*, де $A = \{l, \dots, m-1\}$, де $b \geq 2$ і $k = m-1$. Специфікою систем даного класу також є можливість представлення негативних чисел, але, на відміну від нега-позиційного представлення, немає потреби робити це за рахунок довжини коду: від'ємне число формується виключно за рахунок значення цифр. Крім того, закодувати чи розкодувати число в таких системах значно простіше.

Самі по собі ці класи систем не є надлишковими чи не-надлишковими, оскільки просто описують можливі співвідношення основи та алфавіту і визначають можливий склад останнього. Крім того, ці класи не є і вичерпними: так, передбачається, що алфавіт містить всі цифри від деякого $l \in \mathbb{Z} \setminus \mathbb{N}$ до $m \in \mathbb{N}$ рівно в одному екземплярі. Цю властивість ми назвемо *нормальністю* алфавіту / коду, а при невиконанні вищевказаних умов казатимемо про аномальність.

Тож, логічною є інша постановка питання: як отримати надлишкову систему? Відповідь насправді досить проста: алфавіт системи має містити більше цифр, ніж передбачено основою [254]. Так, найпростішою такою системою є раніше згадане надлишкове двійкове представлення, що має алфавіт $\{0, 1, \bar{1}\}$ і основу 2. В той же час

збалансована трійкова система має той же алфавіт і основу 3, проте надлишковою не є.

Відповідно, для будь-якого нормального алфавіту надлишковість може бути досягнута простим зменшенням основи. Істинно і зворотно: надлишкова система може бути сформована на основі класичної через розширення алфавіту в додатній чи від’ємній площині.

Ключовим питанням тут є властивості системи, що отримуються внаслідок такої трансформації. В першу чергу мова йде про ті із них, що впливають на топологічний синтез. Нескладно визначити, що єдиним кодовим параметром, важливим для топологічних характеристик (числа вершин, ступеня та діаметра) є розмір алфавіту k . Це впливає із самих властивостей перетворень, що працюють ідентично як для звичайних, так і для надлишкових кодів.

Втім, є річ, властива лише надлишковим кодам – це альтернативні представлення одних і тих же чисел. І першою характеристикою, – мабуть, основною, – є розподіл альтернативних представлень $\alpha(V)$. Справа в тому, що не всі цифри мають однакове число представлень. Наприклад, для чотирьохрозрядного RBR число 1 репрезентується кодами 0001, 001T, 01TT, 1TTT, а число 8 має лише один варіант представлення – 1000. Таким чином, $\alpha_2^{T01}(1)_4 = 4$, а $\alpha_2^{T01}(8)_4 = 1$.

Існує два способи отримання даної характеристики. Перший полягає у генерації всіх можливих r -розрядних кодів для заданої системи числення з подальшим визначенням чисел, що ними репрезентуються. Це є затратним, коли задача полягає у визначенні конкретного $\alpha(V)$, але доцільно, коли необхідно визначити α на всій області визначення.

Другий підхід базується на теорії шаблонів і передбачає, що для певного V генерується базовий код. Для системи T01 базовими є коди, що включають в себе лише цифри 0 і 1 (для $V > 0$) або цифри 0 і T (для $V < 0$). Далі виконується пошук всіх наявних шаблонів та генеруються похідні коди через їх альтернацію (тобто, одна форма шаблону заміщається на іншу). Для похідних кодів виконується аналогічний аналіз, і пошук триває доти, доки всі форми представлення V не будуть знайдені.

На основі розподілу можна виділити ще два параметри – це максимальне число альтернативних представлень α_{max} та кількість унікальних кодів для r -розрядної системи $n_{\alpha=1}$. Якщо розподіл задає число представлень для конкретного V і має практичне значення для відмовостійкості кожного конкретного логічного вузла, то α_{max} визначає максимальну кількість представлень одного числа і дозволяє визначити складність створення імпліцитних кластерів для системи загалом. Другий параметр, число унікальних представлень, позначає число вразливих вершин, які не можуть бути захищені зазначеним методом. Варто зазначити, що так само як розподіл змінюється з ростом r , так і ці два параметри зростають: чим довшим є код – тим більшу «жирність вузлів» він передбачає і тим більше незахищених вершин зазвичай породжує. Таким чином, саме розподіл, його форма та екстремуми є ключовими для опису коду.

Проте, оскільки мова йде про синтез топології, можна виділити ще декілька властивостей, які можуть бути корисними, а саме:

- $\alpha 1$ -стабільність, яка полягає в тому, що незалежно від довжини коду СЧ передбачає константне число унікальних представлень. Тобто, $n_{\alpha=1}(r) = const$. Практичне значення даної властивості в тому, що з масштабуванням кількість вразливих вузлів не росте. Таким чином, ці вузли можуть бути захищені класичними методами (наприклад, TMR), а при масштабуванні – перейменовані.
- Симетричність, що передбачає симетрію системи відносно нуля. Така властивість гарантує, що знаючи шлях із А в В завжди можна визначити шлях із \bar{A} в \bar{B} . Це може бути корисним для визначення альтернативних шляхів, а також для організації обчислень (топологія завжди може бути декомпонована на частини з наперед відомою формою). Крім того, завдяки цій властивості центр розподілу не зміщуватиметься з масштабуванням, що суттєво спрощує сам його процес.
- Рівномірність, що передбачає кусково монотонну форму розподілу, де при $V \in \left[V_{min}, \frac{V_{max}-V_{min}}{2} \right)$ $\alpha(V)$ зростає, а на іншому відрізку – навпаки,

спадає. На практиці це означає, що у топології є фактичне «ядро» з потужними кластерами і «периферія» з вузлами невеликої кратності. Це також спрощує організацію обчислень.

Так, код $01T_2$ є симетричним, але нестабільним і нерівномірним. Код $012T_2$ – $\alpha 1$ -стабільний і рівномірний, але не симетричний. $012TZ_2$ – $\alpha 1$ -стабільний і симетричний, але не рівномірний.

2.2.2. Шаблон як основа властивостей коду

Звісно, постає питання: чому коди мають властивості, які вони мають? Звідки ці властивості беруться і чи можна їх цілеспрямовано сформулювати?

Очевидно, що основою симетричності є алфавіт. Оскільки в такій системі $\forall d \in A \exists \bar{d} \in A \rightarrow \forall c = d_{r-1}d_{r-2} \dots d_0 \exists! \bar{c} = \overline{d_{r-1}}\overline{d_{r-2}} \dots \overline{d_0}$. Отже, якщо всі альтернативні представлення числа V подати у вигляді множини $W_V = \{c_1, c_2, \dots, c_n\}$, справедливо, що $\forall V: W_V = \{c_1, c_2, \dots, c_n\} \exists! -V: W_{-V} = \{\bar{c}_1, \bar{c}_2, \dots, \bar{c}_n\}$. Як наслідок, $\alpha(V) = \alpha(-V) = n$.

Втім, інші характеристики, такі як рівномірність та $\alpha 1$ -стабільність, залежать від структури розподілу $\alpha(V)$, а також від принципу його масштабування. Попередні дослідження [254] показали, що для кодів з однаковим розміром алфавіту k і однаковою основою b $\alpha(V)_r$ відрізняються лише зміщенням вздовж осі абсцис. Таким чином, була сформована наступна гіпотеза:

Гіпотеза 2.1. Для будь-якого кодування із не-аномальним алфавітом форма $\alpha(V)$, а також α_{max} і $n_{\alpha=1}$ залежать лише від трьох параметрів: розміру алфавіту k , основи числення b і довжини (рангу) коду r . Вміст алфавіту не впливає на форму розподілу, проте зсув алфавітної послідовності приводить до зміщення так, що новий розподіл $\alpha'(V) = \alpha(V + t)$.

Для перевірки даного припущення було виконано дослідження, яке передбачало генерацію розподілів для надлишкових кодів з розміром алфавіту від 3 до 7 включно. Такі параметри були обрані з двох причин. По-перше, ефективність методу синтезу на основі коду залежить від простоти бінарного представлення цифр, тож використання великих алфавітів є недоцільним (у таких випадках бітність кодування буде

завеликою). По-друге, топологічні параметри залежать від розміру алфавіту. Для топології де Бруйна семизначний алфавіт дає ступінь 14, що виходить за рамки «вікна ефективності» для безпосередньо-зв'язаних мереж. Таким чином, експеримент хоч і не є повним, проте покриває більшість потенційних ситуацій використання. Рис. 2.2 і 2.3 ілюструють частину експериментальних результатів. Інші дані приведено в Додатку В.

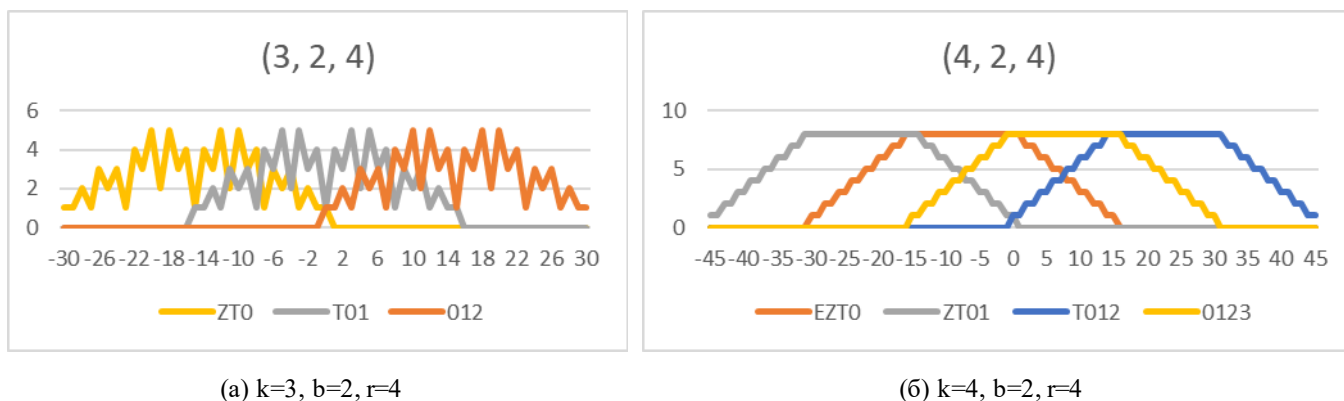


Рис. 2.2. Розподіл представлень для RBR з різними алфавітами, $r=4$

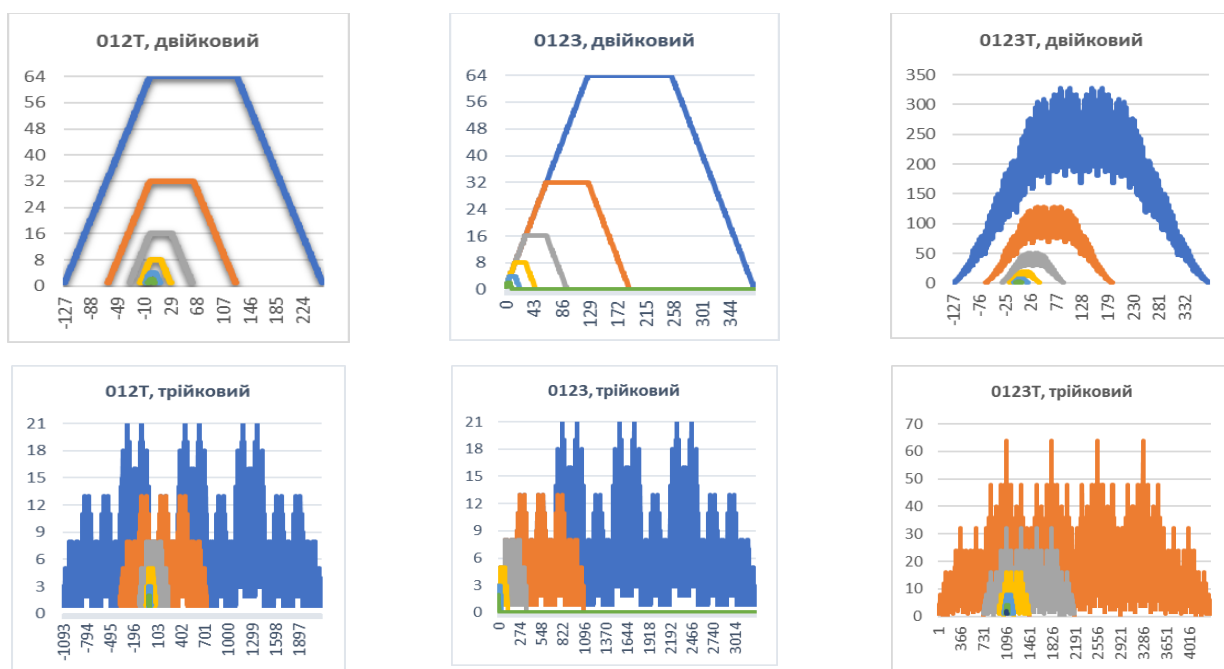


Рис. 2.3. Розподіл числа представлень для кодів 012T, 0123 та 0123T з двійковою та трійковою основою, $r \in [2, 8]$.

Цей експеримент (з певними обмеженнями) доводить, що дійсно, для розподілу $\alpha(V)$ вміст алфавіту не принциповий. Таким чином, коли є необхідність абстрагуватись від конкретного набору цифр коду, є сенс казати про його тип, представлений через пару параметрів (k, b) .

Проте невирішеним залишається головне питання – чому надлишкові коди мають ті властивості, які вони мають?

Відомо, що найменшою одиницею, що може мати надлишковість, є шаблон – дворозрядна підпоследовність. Логічно, що будь-який код складається із ряду таких шаблонів, причому для коду довжини r існує рівно $r-1$ шаблон, що може або реалізовувати надлишковість (мати кілька представлень) або ж бути унікальним.

Нехай дано код $a_{r-1} a_{r-2} \dots a_0$. Позначимо кожен дворозрядну последовність як T_i , де i – номер молодшого розряду последовності. Тоді будь-який код може бути розкладений на 2 «спектри», що в залежності від парності чи непарності r прийматимуть наступний вигляд (крапками позначені «точки розриву», по яким розділяється код):

$$\text{Для парних } r: \begin{cases} a_{r-1} a_{r-2} \cdot a_{r-3} a_{r-4} \cdot a_{r-5} \dots a_3 a_2 \cdot a_1 a_0 \rightarrow T_{r-2} T_{r-4} T_{r-6} \dots T_2 T_0 \\ a_{r-1} \cdot a_{r-2} a_{r-3} \cdot a_{r-4} a_{r-5} \cdot \dots a_3 \cdot a_2 a_1 \cdot a_0 \rightarrow a_{r-1} \cdot T_{r-3} T_{r-5} T_{r-7} \dots T_3 T_1 \cdot a_0 \end{cases}$$

$$\text{Для непарних } r: \begin{cases} a_{r-1} a_{r-2} \cdot a_{r-3} a_{r-4} \cdot a_{r-5} \dots a_3 \cdot a_2 a_1 \cdot a_0 \rightarrow T_{r-2} T_{r-4} T_{r-6} \dots T_3 T_1 \cdot a_0 \\ a_{r-1} \cdot a_{r-2} a_{r-3} \cdot a_{r-4} a_{r-5} \cdot \dots a_3 a_2 \cdot a_1 a_0 \rightarrow a_{r-1} \cdot T_{r-3} T_{r-5} T_{r-7} \dots T_2 T_0 \end{cases}$$

Таким чином, можна казати про те, що кожен код має рівно $r-1$ шаблонів, де можуть сформуватись альтернативні представлення. Якщо відомо, що деякий код має шаблони з максимальною кількістю представлень $\alpha_{max,2} = \beta$, це означає, що максимальна теоретична межа кількості представлень $A_{max,r} = \beta^{r-1}$.

Наступним кроком, що дозволить отримати відповідь на поставлене запитання, є дослідження розподілів α_2 . Таблиця із шаблонами для деяких надлишкових та не-надлишкових кодів приведена в Додатку Д (табл. Д.2). Так, було встановлено α_1 -стабільні коди мають лише по 4 вразливі шаблони. Як приклад, для коду 0123 це шаблони 00, 01, 32 і 33. Більш того: ці шаблони розташовані по краям розподілу, 2 з них мають однакові цифри (00 і 33), а 2 інші є їх сусідами. Це дозволяє пояснити описану властивість. Так, 2 коди (00..0 та 33..3) ніколи не міститимуть жодного надлишкового шаблону. Поява ж у коді 1 чи 2 породить надлишковість у будь-якому випадку, крім двох: якщо цифра 1 чи 2 стоятиме у кінці, відповідно, последовності 00..0 чи 33..3.

Ще одне цікаве спостереження пов'язане із рівномірністю коду: так, коди типу (4, 2) мають трапецієвидний розподіл, а їх шаблони також формують трапецію, з 4 одинарними шаблонами по краях і подвійними шаблонами в центрі. В той же час інший α_1 -стабільний тип коду (5, 2) також має «закритий центр», проте в розподілі наявні «стрибки»: деякі шаблони мають по 3 альтернативи, в той час як у більшості їх строго 2. При цьому жоден код, що має «прогалину» серед центральних значень (тобто, не-надлишковий шаблон), α_1 -стабільним також не є.

Таким чином, доцільно припустити, що «перекриття» і «прогалини» в розподілі α_2 впливають на форму розподілу α_r . Втім, подальше продовження аналізу потребує введення додаткового описового апарату для представлення інформації.

Погляньмо уважніше на те, як, власне, надлишкові шаблони з'являються в коді. В не-надлишковій системі числення коди слідуєть одне за одним. Друга колонка таблиці Д.2 ілюструє таке кодування – збалансовану трійкову систему числення. Проте у надлишковому коді певні коди «наповзають» одне на одне, породжуючи надлишковість. При цьому точками перетину завжди є місця, де старша цифра змінюється. Тож, є сенс ввести таке поняття як ланцюг. Ланцюгом назовемо впорядковану підпоследовність дворозрядних кодів ax , де a – наперед відома цифра, а x – змінна, що послідовно змінює значення у відповідності до алфавіту [4].

Маючи цей описовий механізм, варто звернути увагу на код 0123_2 , що є і рівномірним, і α_1 -стабільним. Оскільки його алфавіт містить 4 цифри, в ньому існуватиме рівно 4 ланцюги довжиною 4. При цьому основа коду – 2. Таким чином, різниця між розміром алфавіту та основою дозволяє визначити глибину накладання, що витікає із властивостей позиційних систем числення. Більш наочним тут є схематичне представлення, що дано на рис. 2.4.

Відповідно, співвідношення довжини ланцюгів та глибини накладання гарантує, що кожне значення крім крайніх буде «покрито» за рахунок перетину значенням із сусіднього ланцюга, причому рівно 1 раз. Узагальнюючи цей факт, можна висунути наступну гіпотезу:

Гіпотеза 2.2. Будь який не-аномальний код є $\alpha 1$ -стабільним і рівномірним тоді, коли в ньому виконується деяке співвідношення між основою числення b і розміром алфавіту k .

Постає питання – яким має бути це співвідношення? Код 0123_2 відноситься до типу $(4, 2)$, тож, доцільно припустити, що інші коди виду $(2b, b)$ також матимуть схожі властивості. Кодування $0123TZ_3$ відноситься до типу $(6, 3)$ і також відповідає означеним параметрам. Таким чином, є сенс дослідити його ланцюгову схему також.



Рис. 2.4. Ланцюгова схема для кодів $01T_2$, $012T_2$, $012TZ_2$, $0123TZ_3$.

Як видно зі схеми, обидва коди мають трапецієвидний розподіл. Це дозволяє припустити, що і розподіл α_r в них буде мати схожу властивість. Рис. 2.5 ілюструє розподіли для різних кодів зі співвідношенням $(2b, b)$.

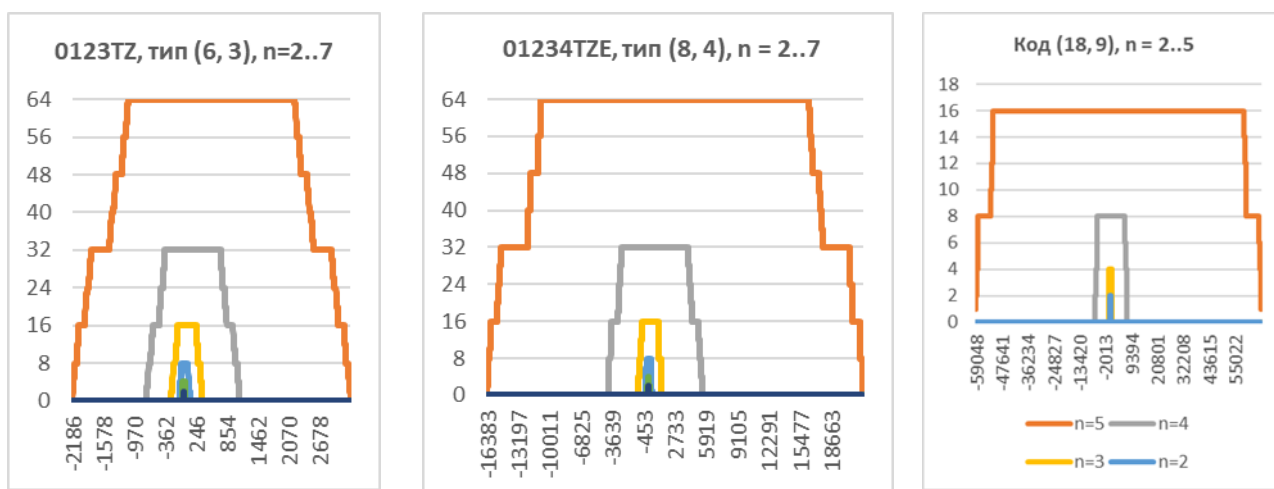


Рис. 2.5. Розподіли кодів $(6, 3)$, $(8, 4)$ та $(18, 9)$

І дійсно, форма розподілу є трапецієвидною. Цікавим аспектом, який варто також відмітити, є те, що число унікальних представлень завжди є рівним кількості

унікальних дворозрядних кодів (яке завжди є рівним $2b$), а число представлень – завжди рівним 2^{r-1} незалежно від системи числення. Тобто, на центральних точках розподілу для таких кодів досягається максимальне теоретично можливе число представлень.

Звісно, залишається питання: як щодо інших співвідношень k та b ? Чи матимуть вони подібні властивості? На рис. 2.6 представлено розподіл для коду типу $(6, 2)$ в класичній та логарифмічній формі за основою 3.

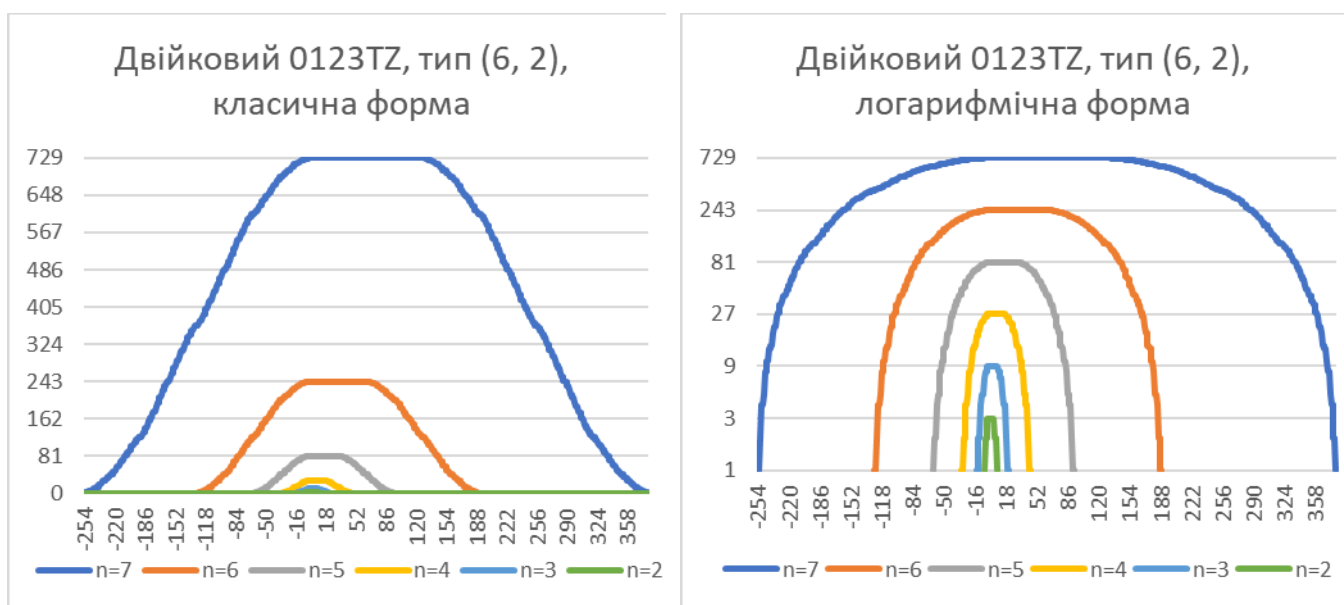


Рис. 2.6. Розподіл коду $(6, 2)$ в звичайній та логарифмічній формі

Аналогічно, цей код є $\alpha 1$ -стабільним і рівномірним, а максимальна кількість його представлень $\alpha_{max}(r) = 3^r$. Аналогічно, розглядаючи ланцюгову схему коду (рис. 2.7), маємо ситуацію накладання, причому форма накладання є ступінчастою, число представлень зростає до центру розподілу, а потім спадає.

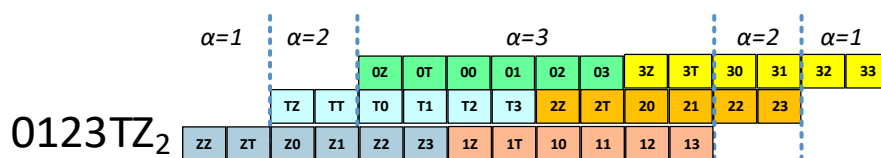


Рис. 2.7. Ланцюгова схема коду $(6, 3)$

Таким чином, логічним питанням буде – чи можна поєднати властивості $\alpha 1$ -стабільності, рівномірності та симетричності в рамках одного коду, забезпечивши при цьому число представлень рівне степені деякого цілого числа? Дослідимо код типу $(9, 3)$. Цей код має непарне число цифр – таким чином, алфавіт із центром в точці 0 буде

симетричним, а отже, і система числення буде симетричною. При цьому співвідношення кодів дає шанс досягти рівномірності і $\alpha 1$ -стабільності. На рис. 2.8 представлена ланцюгова схема такого коду.

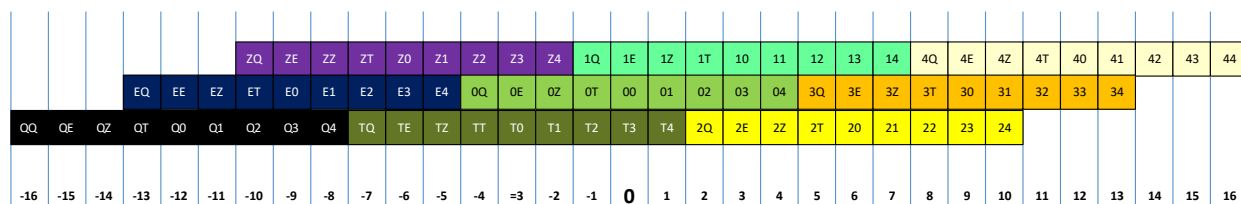


Рис. 2.8. Ланцюгова схема кодування QEZT01234₃, тип (9, 3)

Подібно до (6, 3), бачимо, що структура схеми є ступінчастою і має максимальну кратність 3. Таким чином, можна теоретично передбачити наступні параметри такого кодування: максимальну кратність представлень 3^{r-1} , число унікальних вузлів 6 незалежно від рангу.

На рис. 2.9 представлено графік, побудований на основі емпіричних даних. Як видно, характеристики дійсно відповідають теоретичному передбаченню, що вказує на правильність гіпотези.

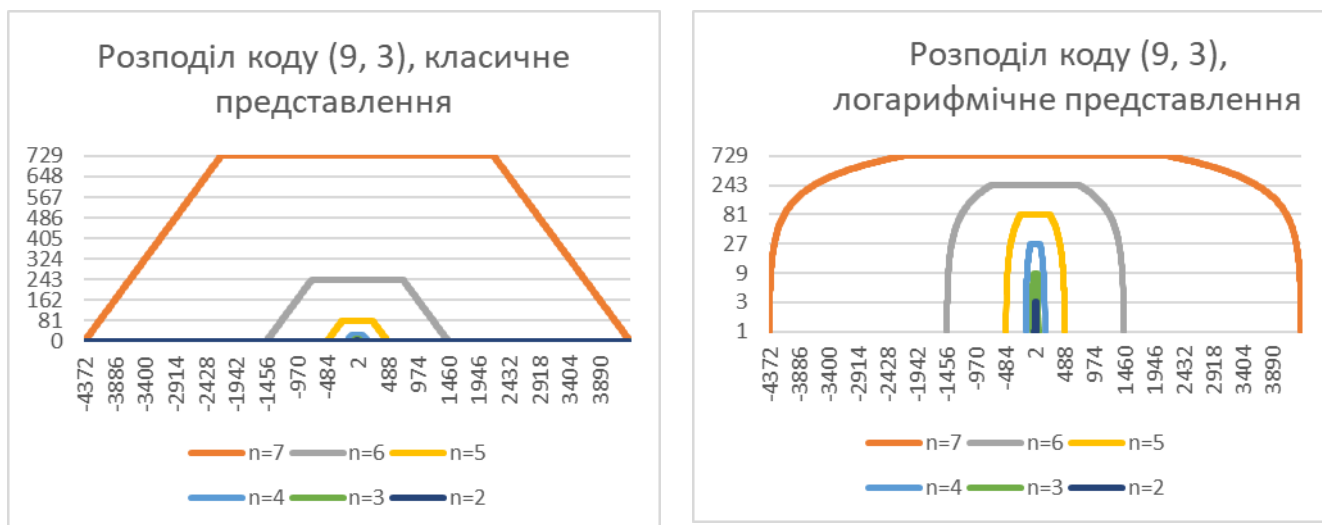


Рис. 2.9. Розподіл кодування QEZT01234₃, тип (9, 3)

Крім того, отриманий результат має деякий практичний сенс: оскільки надлишкові властивості топології залежать від властивостей коду, а властивості, в свою чергу, опираються на певні співвідношення параметрів системи числення, це дає змогу закладати в код певні властивості, що в подальшому матимуть вплив і на систему, спроектовану на основі надлишкової топології.

2.3. Формалізація теорії надлишкового коду

Дані, приведені вище, отримані емпірично на основі ряду гіпотез, з використанням різноманітних евристик та індукції на основі перебору варіантів. Звісно, такий підхід хоч і дає результат, проте це не є повноцінним доведенням. З іншого боку, емпіричне виявлення чіткого спостережуваного зв'язку між кодом і параметрами дає змогу побудувати математичну модель і формально довести гіпотези, що були висунуті в ході попередніх підготовчих досліджень. Таким чином, визначимо наступні поняття:

Основоположна послідовність – це послідовність цифр від деякого мінімального значення 1 до максимального значення m , що описує нормальний алфавіт.

Базовий розподіл α_2 – найпростіший розподіл, властивий системі числення.

Ланцюг L_a – це послідовність кодів виду ax , де a є наперед відомою цифрою, а x змінюється відповідно до основоположної послідовності. Накладення ланцюгів одне на одне породжує базовий розподіл.

Шаблон – деякий дворозрядний (рідше – n -розрядний) код в системі числення. Передбачається, що шаблон може мати декілька альтернативних форм, що можуть бути реалізовані.

Ключовою властивістю шаблону є той факт, що при його реалізації (заміщенні однієї кодової послідовності іншою) змінам підлягає лише частина коду, що покривається шаблоном. Інші частини коду не міняються. Цей факт є багатократно перевіреним емпірично, проте потребує математичної формалізації. Таким чином, наступна теорема може бути сформульована:

Теорема 2.1 (про заміщення). Для будь-якого надлишкового коду виду $\lambda T \mu$, де λ і μ є довільними кодовими послідовностями, а T – шаблоном, що має альтернативні представлення, властиво, що заміщення T на будь-яку його альтернативну форму T^* дасть код $\lambda T^* \mu$, такий, що λ і μ залишаться незмінними і числові значення $v(\lambda T \mu) = v(\lambda T^* \mu)$.

Довести її нескладно. Загальновідомо, що числове значення будь-якого коду позиційної СЧ формується за формулою $v(a_{r-1}a_{r-2} \dots a_0) = \sum_{i=0}^{r-1} a_i b^i$. Враховуючи властивості суми, якщо замінити деяку частину коду невідомою p -розрядною послідовністю T так, що $a_{r-1}a_{r-2} \dots a_{j+1}a_j a_{j-1} \dots a_{j-p+1}a_{j-p}a_{j-p-1} \dots a_0 = a_{r-1}a_{r-2} \dots a_j T a_{j-p-1} \dots a_0 = \lambda T \mu$, де $j \geq p$, функція значень теж може бути перетворена наступним чином: $v(a_{r-1}a_{r-2} \dots a_0) = \sum_{i=0}^{r-1} a_i b^i = \sum_{i=0}^{j-p-1} a_i b^i + \sum_{i=j-p}^j a_i b^i + \sum_{i=j+1}^{r-1} a_i b^i = \sum_{i=0}^{j-p-1} a_i b^i + b^{j-p} \sum_{i=0}^{p-1} a_i b^i + b^j \sum_{i=0}^{r-j-1} a_i b^i = v(a_{j-p-1}a_{j-p-2} \dots a_0) + b^{j-p} v(a_{j-1}a_{j-2} \dots a_{j-p}) + b^j (a_{r-1}a_{r-2} \dots a_j) = v(\mu) + b^{j-p} v(T) + b^j v(\lambda)$.

З даної формули слідує наступне твердження: якщо замінити послідовність T будь-якою іншою послідовністю T^* , код матиме те саме значення за умови, що їх довжина є однаковою і $v(T^*) = v(T)$. За визначенням, шаблони є n -розрядними послідовностями, тож альтернація $T \rightarrow T^*$ завжди передбачає однакові довжини T і T^* . Аналогічно, за визначенням заміна по шаблону гарантує, що числові значення T і T^* однакові. Тож теорему можна вважати доведеною.

Крім того, дана теорема має наслідок: якщо шаблон T має q представлень, то і $\lambda T \mu$ має не менше ніж q представлень, таких, що якщо $T \rightarrow T^1, T^2, \dots, T^q$, то $\lambda T \mu \rightarrow \lambda T^1 \mu, \lambda T^2 \mu, \dots, \lambda T^q \mu$.

В якості наступного кроку необхідно більш детально проаналізувати властивості кодів. З емпіричних даних відомо, що існує певна залежність між кодами порядку r та $r-1$. Крім того, такі властивості як рівномірність та $\alpha 1$ -стабільність також належать до таких, що передаються між розподілами різних порядків. Тож нагальним питанням є формалізація та математичне доведення даних залежностей.

Почати варто із наступних лем:

Лема 2.1. Базовий розподіл будь-якої нормальної системи числення типу (k, b) містить рівно k ланцюгів довжиною k , а розмір їх області накладання ξ визначає характеристики цього розподілу

Доведення даної лема полягає у визначенні ланцюга $L_a = ax$. Оскільки a є цифрою алфавіту, в системі числення існує рівно k ланцюгів. Відповідно, x – також цифра алфавіту, яка при сталому a приймає всі значення, властиві алфавіту. За визначенням нормальності алфавіт є нормальним, коли містить всі цифри зі значеннями від l до m рівно в одному екземплярі. Таким чином, $x \in [l, m]$; $l, m \in \mathbb{Z}$; $m - l + 1 = k$. Тож, дійсно, довжина кожного L_a завжди буде рівною k незалежно від a .

Тепер поговоримо про характеристики розподілу. За визначенням, базовий розподіл представляє собою функцію, де коду або значенню ставиться у відповідність число його альтернативних представлень в заданій СЧ. Очевидно, що кожен ланцюг сам по собі не містить жодної надлишковості, оскільки $x \in [l, m]$, а отже, в нормальному алфавіті сусідні елементи ланцюга завжди матимуть числову різницю в 1. Таким чином, саме стик ланцюгів формує надлишковість. Коли в одній точці накладаються два сусідні ланцюга $\alpha_2=2$, коли перетинаються три – то $\alpha_2=3$ і т.д.

Відомо, що початкові точки всіх ланцюгів мають вигляд al , причому між ланцюгами $a \in [l, m]$ – таким чином, для двох ланцюгів числова різниця між початковими елементами ($x_1=x_2=l$) є рівною $(a+1)l-al$, що в числовому вигляді представляється як $(a+1)b+l-ab-l=b$. Таким чином $\forall k > b \exists \xi = k - b: \forall L_a, L_{a+1}, a \in [l, m-1] \exists v_1 \in [v(al), v(al+b-1)], v_2 \in [v(al+b), v([a+1]m-b)], v_3 \in [v([a+1]m-b+1), v([a+1]m)]: \alpha_2^{a,a+1}(v_1) = 1, \alpha_2^{a,a+1}(v_2) = 2, \alpha_2^{a,a+1}(v_3) = 1, \alpha_2^{a,a+1}(V \notin [v(al), v([a+1]m)]) = 0$, де $\alpha_2^{a,a+1}$ представляє собою розподіл лише для двох окремо взятих сусідніх ланцюгів. Очевидно, що загальний розподіл $\alpha_2(V) = \sum_{i=0}^{k/2} \alpha_2^{2i,2i+1}(V)$. Таким чином, залишається довести, що $v([a+1]m-b) - v(al+b) + 1 = k - b = \xi$. Розкриємо ліву частину виразу. Маємо $v([a+1]m-b) - v(al+b) = (a+1)b + m - b - ab - l - b + 1 = ab + b + m - ab - b - l - b + 1 = m - l - b + 1 = |m - l + 1 = k| = k - b$.

Таким чином, лему 2.1 можна вважати доведеною.

Лема 2.2. Для будь-якої пари цифр ax кількість альтернативних представлень може бути визначена за формулою:

$$\alpha(ax) = \min\left(m - a, \frac{x-l}{b}\right) + \min\left(a - l, \frac{m-x}{b}\right) \quad (2.1)$$

Почати варто із того, як змінюється код при реалізації шаблону. Оскільки альтернативні шаблони мають представляти однакові числові значення, то, згідно властивостей систем числення, при зміні старшої цифри на деяке $+p$ молодша має змінитись на $-bp$. Справедливо і зворотнє: зміна a в бік зменшення має вести до аналогічної зміни x в більшу сторону з кроком b .

Розглянемо 2 ситуації. Нехай рух по старшій цифрі відбувається в сторону збільшення. Очевидно, що $a \leq m$, таким чином, $p = 1 \dots (m-a)$. Проте існує обмеження, що накладається молодшою цифрою: при зменшенні на bp вона не може стати менше ніж l . Таким чином, можна вивести наступну нерівність: $x - bp \geq l \rightarrow bp \leq x - l \rightarrow p \leq \frac{x-l}{b} \rightarrow p = 1 \dots \left[\frac{x-l}{b}\right]$.

Таким чином, число представлень буде відповідати максимальній межі p , що рівна $\alpha_{\uparrow} = \min\left(m - a, \frac{x-l}{b}\right)$. Аналогічно, при русі в бік зменшення старшої цифри існує еквівалентне обмеження $x + bp \leq m$ і, як наслідок, межа p при такому напрямі руху буде $\alpha_{\downarrow} = \min\left(a - l, \frac{m-x}{b}\right)$.

Відповідно, загальне число представлень враховує обидва можливі напрями руху і є рівним $\alpha(ax) = \alpha_{\uparrow} + \alpha_{\downarrow}$.

Лема 2.3. Для будь-якого коду uax число альтернативних представлень $\alpha_3(uax)$ може бути отримано за формулою:

$$\alpha_3(uax) = \sum_{i=l-y}^{i=m-y} \alpha_2(v([a - ib]x)) \quad (2.2)$$

Де $v(ax)$ – функція значення коду і $\alpha_2 = 0$ для $v(ax) \notin [v(ll), v(mm)]$.

Для доведення виконаємо розклад коду uax на 2 спектри yT та θx . Згідно наслідку теореми 2.1, якщо T має $p = \alpha_2(T)$ представлень T^1, T^2, \dots, T^p , то і uax також має представлення yT^1, yT^2, \dots, yT^p .

Тепер зосередимось на непарній частині спектру θx . Нехай існує $q = \alpha_2(\theta)$ альтернативних представлень $\theta^1, \theta^2, \dots, \theta^q$. Згідно леми 2.2, можна визначити ці представлення через формулу $\theta^i = [y + i][a - ib], i = l - y \dots m - y$.

Таким чином, перехід по шаблону θ породжує нові шаблони $y^i T_i$, де $\alpha_2(T_i) = p_i$. Логічно, що код включатиме і ці представлення також. Таким чином, $\alpha(yax) = \sum_{i=l-y}^{i=m-y} \alpha(T_i)$, яке включає в себе і початковий варіант yT .

Втім, така формула не є повністю коректною. Справа в тому, що при переході $a' = a - ib$ можлива ситуація, що a вийде за межі діапазону $[l, m]$. Проте, при $b(x - l) \geq l - a \rightarrow x \geq \frac{l-a}{b} - l$ чи $b(m - x) \geq a - m \rightarrow x \leq \frac{a-m}{b} + m$ можлива ситуація, що початково неправильний код матиме в системі правильне представлення, що призведе до розходження між теоретичним і практичним результатом. Таким чином, для корекції варто використати функцію значення $v(T_i)$ замість самого T_i .

Розглянемо приклад. Нехай дано систему числення $0123T_2$. Відомо, що код 01 в ній має 3 альтернативні представлення: 01, T_3 і $1T$. Нехай дано код 001, для якого потрібно отримати всі його альтернативні представлення. Тоді, згідно звичайної формули:

1) $001 = \theta x \rightarrow \theta = 00$. 00 має 2 представлення: 00 і T_2 . Таким чином, отримуємо коди $yT = 001$ і $y^1 T^1 = T_2 1$.

2) аналізуємо T і T^1 . $\alpha(T) = 3$, $\alpha(21) = 3$. При цьому T^1 має такі представлення як 21, 13 і $3T$. Таким чином маємо $\alpha(001) = \alpha(01) + \alpha(21) = 6$.

Згідно емпіричних даних $\alpha(001) = 7$. Де помилка? Проаналізуємо коди, які було знайдено теоретично та емпірично. Теоретично були визначені 6 кодів: 001, $0T_3$, $01T$, $T_2 1$, $T_1 3$ і $T_3 T$. Втім, емпіричні дані вказують на ще один варіант $1TT$, який не було виявлено при аналізі. Чому? Справа в тому, що перехід по представленням 0 не дає жодного варіанту, де б старшою цифрою була 1. Втім, якщо припустити, що ми можемо порушувати правила кодування при переході по θ , ситуація змінюється. Таким чином, з'являється наступна послідовність дій:

1) $\theta=00$. Виконаємо перебір всіх варіантів по старшій цифрі 0, припускаючи, що молодша цифра може набувати при цьому некоректних значень. Маємо:

$$\theta^{-1}=T2, \theta^0=00, \theta^1=1[-2], \theta^2=2[-4], \theta^3=3[-6].$$

2) Аналізуємо отримані T^i . $T^{-1}=21, T^0=01, T^1=[-2]1, T^2=[-4]1, T^3=[-6]1$. Очевидно, що T^1, T^2 і T^3 не так просто проаналізувати, тому скористаємось функцією значень для коригування. Маємо: $v(T^{-1}) = 2 * 2 + 1 = 5$; $v(T^0) = 0 * 2 + 1 = 1$; $v(T^1) = -2 * 2 + 1 = -3$; $v(T^2) = -4 * 2 + 1 = -7$; $v(T^3) = -6 * 2 + 1 = -11 \rightarrow \alpha(T^{-1}) = \alpha_2(5) = 3$; $\alpha(T^0) = \alpha_2(1) = 3$; $\alpha(T^1) = \alpha_2(-3) = 1$; $\alpha(T^2) = \alpha_2(-7) = 0$; $\alpha(T^3) = \alpha_2(-11) = 0 \rightarrow \alpha_3(001) = \alpha(v(T^{-1})) + \alpha(v(T^0)) + \alpha(v(T^1)) = 3 + 3 + 1 = 7$.

2.3.1. Рекурентне обчислення $\alpha(v)$ для кодування розрядності r

Таким чином, дана лема приводить до наступної теореми:

Теорема 2.2 (про представлення). Число альтернативних представлень будь-якого r -розрядного коду може бути отримано з формули:

$$\alpha_r(V) = \sum_{i=l-\lfloor V/b^{r-1} \rfloor}^{m-\lfloor V/b^{r-1} \rfloor} \alpha_{r-1}(V + (i - \lfloor \frac{V}{b^{r-1}} \rfloor)b^{r-1}) \quad (2.3)$$

Доведення даного факту для $r=3$ представлено лемою 2.3. Втім, його розширення на загальний випадок потребує певного додаткового пояснення. По-перше, для будь-якого числа V , що може бути представлено системою числення, завжди існує базове представлення – таке, що може бути отримано через послідовне ділення числа на основу з записом остачі. Відповідно, якщо записати це представлення як yT , де T є кодом довжини $r-1$, старша цифра y може бути отримана через цілочисельне ділення на основу в степені $r-1$ з округленням вниз. Тобто, $y = \lfloor V / b^{r-1} \rfloor$. Відповідно, $v(T) = V - yb^{r-1} = V - \lfloor V / b^{r-1} \rfloor b^{r-1}$.

Тепер розглянемо уважніше доказ леми 2.3. В ній T і θ є кодами довжини 2. Втім, якщо припустити, що T має довжину $r-1$, маємо наступне:

1. Для будь-якого коду виду yT $\alpha(T) \in \alpha(yT)$. Це твердження слідує із наслідку теореми 2.1 і не залежить від довжини T .

2. Для коду виду θx справедливо, що реалізація шаблону θ призводить до появи нових шаблонів $y^i T_i$, де $y^i \in [l, m]$, $a^i \in T_i$, $a^i = a + b(y - y^i)$. Згідно теореми 2.1 довжина T тут також не має значення – єдиними умовами є еквівалентність довжини та значень, що гарантується визначенням шаблону.

3. У відповідності до п.1 і п.2: $\alpha(yT) \ni \{\alpha(T_1), \alpha(T_2), \dots, \alpha(T_q)\} \rightarrow \alpha(yT) = \sum_{i=1}^q \alpha(T_i)$. Припускаючи, що y^i приймає всі допустимі в рамках алфавіту значення, маємо: $\alpha(yT) = \sum_{i=l}^m \alpha([a + b(y - i)]x)$. Аналогічно, x може бути замінено на будь-яке μ і, у відповідності до п.1 і п.2, дана формула також буде справедлива.

Таким чином, представлена у числовому вигляді формула матиме вигляд $\alpha(V) = \sum_{i=l}^m \alpha(v([a + b(y - i)]\mu)) = \sum_{i=l-y}^{m-y} \alpha(v([a + bi]\mu)) = \sum_{i=l-y}^{m-y} \alpha(v(a\mu) + b^{r-2}v([bi])) = |a\mu = T| = \sum_{i=l-y}^{m-y} \alpha\left(V - \left\lfloor \frac{V}{b^{r-1}} \right\rfloor b^{r-1} + bi * b^{r-2}\right) = \sum_{i=l-\lfloor V/b^{r-1} \rfloor}^{m-\lfloor V/b^{r-1} \rfloor} \alpha\left(V + (i - \left\lfloor \frac{V}{b^{r-1}} \right\rfloor)b^{r-1}\right)$.

Аналогічно, ця формула може бути спрощеною, якщо працювати лише з базовим представленням виду yT . Тоді вона набуватиме наступного вигляду [4]:

$$\alpha_r(yT) = \sum_{i=l-y}^{m-y} \alpha_{r-1}(v(T) + ib^{r-1}) \quad (2.4)$$

Згідно до даної теореми, можна встановити рекурентний зв'язок між розподілами порядку r та $r-1$. Крім того, можна сформулювати наслідок даної теореми у наступному вигляді:

Наслідок (про накладання). Для всіх $r > 2$ розподіл α_r може бути представлений через послідовне накладення розподілів рангу $r-1$ із кроком b^{r-1} . Найкращим способом наочно це представити є ланцюгова схема. Так, на рис. 2.10 представлено схему коду 012 T_2 , де через ланцюги зображується розподіл α_2 . Таблиця показує те, як 4 таких розподіли накладаються одне на одне в залежності від старшої цифри, а нижче представлена та сама схема, де ланцюги замість кодів показують число альтернативних представлень, що забезпечуються зсунутим розподілом α_2 для

відповідного числа V . Останній ланцюг представляє собою розподіл α_3 , сформований в результаті цього накладання.

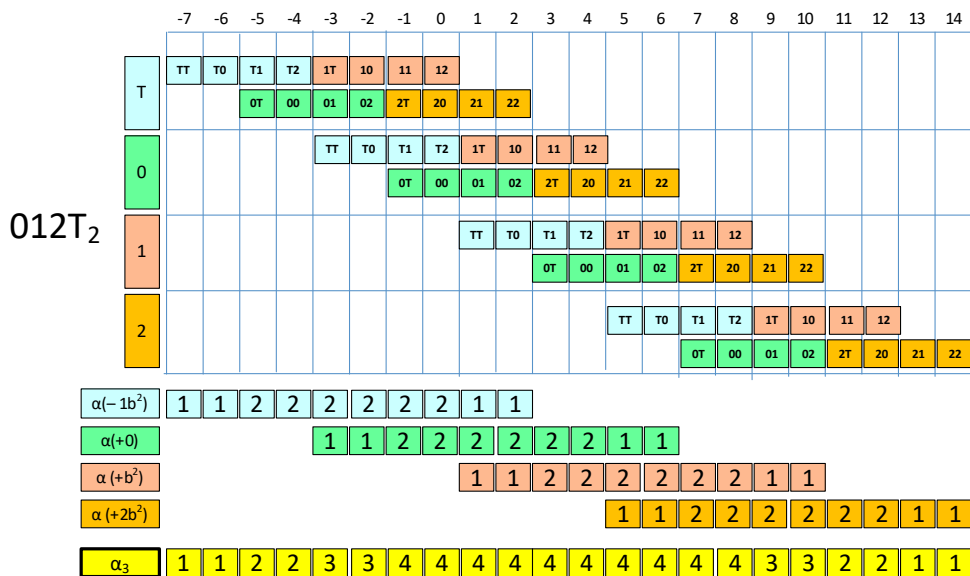


Рис. 2.10. Утворення розподілу α_3 через накладання α_2

Наявність чіткого співвідношення між розподілами різних рангів дає змогу перейти до аналізу гіпотез, що були висунуті раніше. І ключовою із них є гіпотеза 2.1.

2.3.2. Доведення незалежності α -розподілу від вмісту алфавіту.

Таким чином, наступна теорема може бути сформульована і доведена:

Теорема 2.3 (про алфавітний зсув). Для будь-якого нормального алфавіту зсув основоположної послідовності на p веде до зміщення розподілу за наступною формулою:

$$\alpha'_r(V) = \alpha_r(V + p(b^r - 1)) \quad (2.5)$$

Почнемо із властивостей базового розподілу. Нехай для основоположної послідовності $[l, m]$ дано розподіл α_2 . Згідно леми 2.1, цей базовий розподіл розкладається на ланцюги $L_l, L_{l+b}, \dots, L_{l+(k-1)b}$, а його характеристики визначаються довжиною накладання $\xi = k - b$. Оскільки лема 2.1 доводить, що характеристики базового розподілу опираються виключно на ξ , це дозволяє довести першу частину теореми: $\alpha'_2(V) = \alpha_2(V + \delta)$. Визначимо цей зсув δ . Відомо, що змінена основоположна послідовність задана на відрізку $[l+p, m+p]$. Відповідно, зміщення між двома розподілами можна визначити віднявши їх початкові коди $[l+p][l+p] - ll$, що в числовому вигляді представлятиметься як $(l+p)b + (l+p) - lb - l = pb + p$

Тепер розглянемо випадок довжини r . Згідно теореми 2.2 та її наслідку про накладання, результуючий розподіл може бути описаний як k розподілів α_{r-1} , накладених на одне одного із кроком b^{r-1} . Спробуємо перетворити формулу 2.4 у відповідності до зміненої основоположної послідовності. Маємо:

$$\begin{aligned}\alpha'_r(yT) &= \sum_{i=l+p-y}^{m+p-y} \alpha'_{r-1}(v(T) + ib^{r-1}) = \sum_{i=l-y}^{m-y} \alpha'_{r-1}(v(T) + (p+i)b^{r-1}) \\ &= \sum_{i=l-y}^{m-y} \alpha'_{r-1}(v(T) + ib^{r-1} + pb^{r-1})\end{aligned}\quad (2.6)$$

Таким чином, знаючи, що $\alpha'_2(V) = \alpha_r(V + pb + p)$, а $V=v(yT)$, можемо отримати формулу для $\alpha'_3(yT) = \sum_{i=l-y}^{m-y} \alpha'_2(v(T) + ib^2 + pb^2) = \sum_{i=l-y}^{m-y} \alpha_2(v(T) + ib^{r-1} + pb^2 + pb + p) = \sum_{i=l-y}^{m-y} \alpha_2(v(T) + ib^{r-1} + \sum_{j=0}^2 pb^j)$. Оскільки $\forall i$ при $r = \text{const}$: $\sum_{j=0}^2 pb^j = \text{const} \rightarrow \sum_{i=l-y}^{m-y} \alpha_2(v(T) + ib^{r-1} + \sum_{j=0}^2 pb^j) = \alpha_3(V + \sum_{j=0}^2 pb^j)$. Відповідно, можемо перетворити формулу 2.6 у наступний вигляд:

$$\alpha'_r(yT) = \sum_{i=l-y}^{m-y} \alpha_{r-1}\left(v(T) + ib^{r-1} + \sum_{j=0}^{r-1} pb^j\right)\quad (2.7)$$

Це еквівалентно зсуву складових частин розподілу α_r на одну і ту саму константу вздовж осі абсцис. При цьому відносна складова, що визначає крок накладання ib^{r-1} , залишається незмінною. Оскільки характеристики розподілу визначаються саме відносним накладанням складових частин, константний зсув не буде впливати на властивості розподілу.

Останнім, що варто зробити – це спростити формулу 2.7, замінивши суму степенів b сумою відповідного ряду. Тож, $\sum_{j=0}^{r-1} pb^j = p \sum_{j=0}^{r-1} b^j = p(b^r - 1)$. Після даної заміни представлена формула відповідатиме формулі 2.5, приведеній раніше.

Маючи ці доведення, можна перейти до гіпотез 2.2 та 2.3, об'єднавши їх воедино. Втім, перш ніж переходити до безпосередньої роботи із нею, є необхідність розглянути ряд проміжних тверджень, пов'язаних із властивостями коду.

Лема 2.4. При $k=tb$ формується рівномірний розподіл $\alpha_2(V)$. При цьому $\forall p = 1 \dots t-1: V \in [V_{min} + (p-1)b, V_{min} + pb) \cup (V_{max} - pb, V_{max} - (p-1)b] \rightarrow \alpha_2(V) = p$. Для $V \in [V_{min} + (t-1)b, V_{max} - (t-1)b] \rightarrow \alpha_2(V) = t$. Таким чином, існує рівно $n_{\alpha=1}(2) = 2b$ унікальних кодів.

Доведення даної лема також є доволі тривіальним. Згідно лема 2.1 система розкладається на k ланцюгів довжиною k . Причому ці ланцюги накладаються одне на одне із кроком b , як це було продемонстровано раніше, на рис. 2.4 та 2.10. Таким чином, $m-l+1 = tb \rightarrow (a+t)b + l = ab + tb + l = ab + m + 1$. Як наслідок, $\forall a \leq m-t \exists L_{a+t}: L_a \ni [ab+l, ab+m] \rightarrow L_{a+t} \ni [ab+m+1, ab+2m-l+1]$. Це приводить до такого поняття як металанцюг – послідовність кодів L_a^2 , що починається від деякого al , який є частиною ланцюга L_a , і включає в себе всі коди, що належать ланцюгам L_{a+p} , де $p=1 \dots t-1$. Як наслідок, перша група чисел $[V_{min}, V_{min} + b)$ «покривається» лише одним металанцюгом L_l^2 , група $[V_{min} + b, V_{min} + 2b)$ – двома металанцюгами L_l^2, L_{l+1}^2 і т.д. Логічно, що якщо в металанцюг входять ланцюги з кроком t , то існує рівно t металанцюгів, причому їх довжина є рівною $V_j = \frac{k}{t} k = \frac{tb}{t} tb = tb^2$. Відповідно, на кінцях металанцюгів ситуація повторюється: оскільки їх довжини є рівними, накладання має ступінчасту форму. Тож, загальне число чисел, що представляються системою $V_n = V_{max} - V_{min}$ може бути записано через формулу $V_n = V_{ladder} + V_{mchain} = b(t-1) + tb^2 = tb^2 + tb - b$. Таким чином, розподіл є рівномірним за умови, що $\frac{V_n}{2} \leq V_j \rightarrow tb^2 + tb - b \leq 2tb^2 \rightarrow t-1 \leq tb \rightarrow 1 - \frac{1}{t} \leq b$. Згадавши, що $t \in \mathbb{N}$, а $b \geq 2$ можемо помітити, що вищезазначена умова завжди буде істинною незалежно від конкретних значень t та b . Це гарантує, що $\alpha_2(V)$ є зростаючою при $V \in [V_{min}, \frac{(V_{max}-V_{min})}{2}]$. Інша частина функції, відповідно, симетрична відносно середини розподілу, отже, рівномірність α_2 доведено.

Число унікальних кодів довести ще простіше: оскільки $\alpha = 1$ лише на проміжках $[V_{min}, V_{min} + b)$ і симетричному йому $(V_{max} - b, V_{max}]$, звідси і $n_{\alpha=1}(2) = 2b$.

2.3.3. Властивості кодів типу (tb, b) . Поняття багатовимірної форми.

Таким чином, можемо сформулювати наступну теорему:

Теорема 2.4 (про ідеальне співвідношення). Код типу (tb, b) завжди є α_1 -стабільним і рівномірним $\forall t \in \mathbb{N}, t \geq 2$, причому його максимальне число представлень $\alpha_{\max}(r) = t^r$, а число унікальних представлень $n_{\alpha=1} = 2b$.

Для доведення варто звернутись до наслідку теореми 2.2 і представити розподіл у формі накладених ланцюгів. Відомо, що крок накладання на кожному етапі r є рівним b^{r-1} . Розглянемо α_3 . Очевидно, що умова рівномірності виконуватиметься, якщо металанцюги кожного α_2 можуть бути об'єднані в ланцюги більш високого порядку.

Тож, проаналізуємо наявні металанцюги. В накладенні беруть участь $k=tb$ розподілів, кожен із яких, згідно леми 2.4, розкладається на t металанцюгів довжиною tb^2 . Очевидно, що оскільки крок накладання є b^2 , а довжина кожного металанцюга – tb^2 , то існуватиме рівно t ланцюгів більш високого порядку, що включатимуть в себе по $k/t=b$ металанцюгів. Загалом, нескладно встановити, що таким чином довжина кожного високорівневого ланцюга стає рівною tb^3 , а їх число – відповідно, t^2 . Проте подальша робота з ланцюгами вищих порядків в такій формі є проблематичною, що приводить до потреби введення нової форми опису та представлення такого роду сутностей.

Тож, введемо таке поняття як порядок ланцюга. Звичайні ланцюги розглядатимемо як ланцюги порядку 1. Металанцюги, що включають в себе деяке число ланцюгів – як ланцюги порядку 2. Структури, що включають в себе ланцюги порядку $r-1$ – як ланцюги порядку r . Також призначатимемо їм індекси від 0, тим самим абстрагуючись від конкретного алфавіту і розглядаючи лише їх взаємне розташування. Таким чином, представимо розподіл порядку 2 в наступному вигляді:

$$\alpha_2 \ni \begin{pmatrix} L_0^2 \\ L_1^2 \\ \vdots \\ L_{t-1}^2 \end{pmatrix} \ni \begin{pmatrix} L_0 & L_t & \cdots & L_{(b-1)t} \\ L_1 & L_{t+1} & \cdots & L_{1+(b-1)t} \\ \vdots & \vdots & & \vdots \\ L_{t-1} & L_{2t-1} & \cdots & L_{bt-1} \end{pmatrix} \quad (2.8)$$

Тепер варто розглянути властивості даної матриці. Її рядки містять елементи, що входять до єдиного металанцюга. Як видно, їх індекси змінюються із кроком t , а загальна кількість є рівною bt , що повністю відповідає теорії. Звідси ж можна легко знайти і довжину кожного металанцюга: оскільки кожен ланцюг має довжину bt і ряд містить b ланцюгів – маємо довжину b^2t . Інші параметри, такі як кратність накладання на кожному із відрізків, можна легко виявити, якщо пам'ятати, що ланцюг з індексом на одиницю вищим є зсунутим на b .

Тепер використаємо це представлення із формули 2.8 і спробуємо представити розглянутий в лемі 2.4 розподіл порядку 3. Для цього розкладемо накладені розподіли α_2 на k матриць наступного вигляду (верхній індекс показує зміщення i відносно наймолодшого розподілу):

$$\begin{aligned} \alpha_2^0 \ni \begin{pmatrix} L_0^2 \\ L_1^2 \\ \vdots \\ L_{t-1}^2 \end{pmatrix} &= \begin{pmatrix} L_0 & L_t & \cdots & L_{bt-t} \\ L_1 & L_{1+t} & \cdots & L_{1+bt-t} \\ & \vdots & & \\ L_{t-1} & L_{2t-1} & \cdots & L_{bt-1} \end{pmatrix}, \\ \alpha_2^1 \ni \begin{pmatrix} L_b^2 \\ L_{b+1}^2 \\ \vdots \\ L_{b+t-1}^2 \end{pmatrix} &= \begin{pmatrix} L_b & L_{b+t} & \cdots & L_{b+bt-t} \\ L_{b+1} & L_{b+1+t} & \cdots & L_{b+1+bt-t} \\ & \vdots & & \\ L_{b+t-1} & L_{b+2t-1} & \cdots & L_{b+bt-1} \end{pmatrix}, \dots, \\ \alpha_2^{(k-1)} \ni \begin{pmatrix} L_{(k-1)b}^2 \\ L_{(k-1)b+1}^2 \\ \vdots \\ L_{(k-1)b+t-1}^2 \end{pmatrix} &= \begin{pmatrix} L_{(k-1)b} & L_{(k-1)b+t} & \cdots & L_{(k-1)b+bt-t} \\ L_{(k-1)b+1} & L_{(k-1)b+1+t} & \cdots & L_{(k-1)b+1+bt-t} \\ & \vdots & & \\ L_{(k-1)b+t-1} & L_{(k-1)b+2t-1} & \cdots & L_{(k-1)b+bt-1} \end{pmatrix} \end{aligned} \quad (2.9)$$

Очевидно, що оскільки $k=bt$, існує b розподілів, чий ланцюги відрізняються одне від одного на крок bt . Таким чином, можемо сформулювати матрицю, поставивши у один рядок ті розподіли, які ідеально продовжують ланцюги одне одного.

$$\alpha_3 \ni \begin{pmatrix} \alpha_2^0 & \alpha_2^t & \cdots & \alpha_2^{(b-1)t} \\ \alpha_2^1 & \alpha_2^{1+t} & \cdots & \alpha_2^{1+(b-1)t} \\ & \vdots & & \\ \alpha_2^{t-1} & \alpha_2^{2t-1} & \cdots & \alpha_2^{bt-1} \end{pmatrix} \quad (2.10)$$

Тепер замінімо розподіли їх металанцюгами. Отримуємо тривимірну матрицю наступного вигляду:

$$\alpha_3 \ni \left(\begin{array}{ccc} \begin{pmatrix} L_0^2 \\ L_1^2 \\ \vdots \\ L_{t-1}^2 \end{pmatrix} & \begin{pmatrix} L_t^2 \\ L_{t+1}^2 \\ \vdots \\ L_{2t-1}^2 \end{pmatrix} & \dots & \begin{pmatrix} L_{(b-1)t}^2 \\ L_{1+(b-1)t}^2 \\ \vdots \\ L_{bt-1}^2 \end{pmatrix} \\ \begin{pmatrix} L_b^2 \\ L_{b+1}^2 \\ \vdots \\ L_{b+t-1}^2 \end{pmatrix} & \begin{pmatrix} L_{b+t}^2 \\ L_{b+t+1}^2 \\ \vdots \\ L_{b+2t-1}^2 \end{pmatrix} & \dots & \begin{pmatrix} L_{b+(b-1)t}^2 \\ L_{b+(b-1)t+1}^2 \\ \vdots \\ L_{b+bt-1}^2 \end{pmatrix} \\ \vdots & \vdots & \ddots & \vdots \\ \begin{pmatrix} L_{b(t-1)}^2 \\ L_{b(t-1)+1}^2 \\ \vdots \\ L_{b(t-1)+t-1}^2 \end{pmatrix} & \begin{pmatrix} L_{b(2t-1)}^2 \\ L_{b(2t-1)+1}^2 \\ \vdots \\ L_{b(2t-1)+t-1}^2 \end{pmatrix} & \dots & \begin{pmatrix} L_{b(bt-1)}^2 \\ L_{b(bt-1)+1}^2 \\ \vdots \\ L_{b(bt-1)+t-1}^2 \end{pmatrix} & \dots \end{array} \right) = \left(\begin{array}{c} \begin{pmatrix} L_0^3 \\ L_1^3 \\ \vdots \\ L_{t-1}^3 \end{pmatrix} \\ \begin{pmatrix} L_b^3 \\ L_{b+1}^3 \\ \vdots \\ L_{b+t-1}^3 \end{pmatrix} \\ \vdots \\ \begin{pmatrix} L_{b(t-1)}^3 \\ L_{b(t-1)+1}^3 \\ \vdots \\ L_{(b+1)(t-1)}^3 \end{pmatrix} \end{array} \right) \quad (2.11)$$

Втім, дана форма представлення є занадто громіздкою. Тож логічним кроком буде транспонувати частини, що належать різним розподілам, для отримання наступної матриці:

$$\alpha_3 \ni A_3 = \left(\begin{array}{cccc} L_0^3 & L_1^3 & \dots & L_{t-1}^3 \\ L_b^3 & L_{b+1}^3 & \dots & L_{b+t-1}^3 \\ \vdots & \vdots & \vdots & \vdots \\ L_{b(t-1)}^3 & L_{b(t-1)+1}^3 & \dots & L_{(b+1)(t-1)}^3 \end{array} \right) \quad (2.12)$$

Тут металанцюги, зображені в одному рядку, не зливаються, а навпроти – існують паралельно одне одному. При цьому їх доповняють ланцюги в стовпцях, які також існують паралельно ланцюгам в рядках. При цьому, як можна бачити із розподілу, число елементів в рядках і стовпцях є еквівалентним і рівним t .

Таким чином, формується ключове питання: які частини A_3 взаємно накладаються, а які – ні? Оскільки елементи рядків належать одному розподілу, згідно леми 2.4, вони мають накладатись між собою. В той же час елементи одного стовпця також мають накладатись. Нехай дано початковий елемент стовпця i та кінцевий елемент $i+b(t-1)$. Тоді їх стартові точки рівні, відповідно, ib та $tb^2 - b^2 + ib$. Відомо, що довжина L_i^3 рівна tb^3 , тож його кінцева точка, відповідно, $ib + tb^3 - 1$. Таким чином, отримуємо нерівність $tb^2 - b^2 + ib \leq tb^3 + ib - 1 \rightarrow b(t-1) \leq tb^2 - 1$. 3

урахуванням того, що t і b не можуть бути менше 2, можемо напевно сказати, що дана умова завжди є істинною.

Найскладнішим питанням залишається діагональний перетин. Очевидно, що якщо найвіддаленіші ланцюги L_0^3 та $L_{(b+1)(t-1)}^3$ перетинаються, то і всі інші перетинаються також. Тож, підставивши у нерівність кінцеву точку першого ланцюга і стартову точку останнього, маємо:

$$\begin{aligned} (b+1)(t-1)b \leq tb^3 &\rightarrow tb - b + t - 1 \leq tb^2 \rightarrow b + 1 - \frac{b+1}{t} \leq b^2 \\ &\rightarrow (b+1)\left(1 - \frac{1}{t}\right) \leq b^2 \end{aligned} \quad (2.13)$$

Дана нерівність опирається на 2 параметри b і t . Втім, якщо підставити мінімальні значення, отримуємо $(2+1)(1 - 1/2) \leq 4 \rightarrow 1,5 \leq 4$. Тепер спробуємо оцінити поведінку нерівності 2.13 з ростом t і b . Знаючи, що стартове значення лівої частини менше ніж у правої, представимо обидві частини нерівності як деякі (псевдо)неперервні функції $f(b, t) = (b+1)(1 - 1/t)$; $\varphi(b, t) = b^2$. Продиференціювавши їх по b , маємо: $\frac{\partial f}{\partial b} \leq \frac{\partial \varphi}{\partial b} \rightarrow (1 - 1/t) \leq b$. Тут вже, загалом, очевидно, що ліва частина ростиме повільніше за праву незалежно від t , оскільки $\lim_{t \rightarrow \infty} 1 - 1/t = 1 \rightarrow (1 - 1/t) \in [1/2, 1]$. З іншого боку, $b \in [2, \infty)$. Таким чином, $\forall b, t: f(b, t) < \varphi(b, t)$.

Це доводить відразу декілька речей. Оскільки всі ланцюги порядку 3 накладаються між собою, то α_3 гарантовано є рівномірною, оскільки спершу відбувається накладання всіх існуючих ланцюгів порядку 3, а потім, коли ланцюги закінчуються – з'являється симетрична область спадання. Аналогічно, максимальне число її представлень рівне числу взаємно накладених ланцюгів – тобто, t^2 . Звідси ж і третя властивість – α_1 -стабільність: в α_2 лише кінцеві ланцюги L_0 і L_{bt-1} мають ненакладені частини. При переході до α_3 ненакладені частини існують лише в металанцюгах L_0^3 та $L_{(b+1)(t-1)}^3$. Проте, оскільки α_3 повністю включає в себе α_2 , то довжина цих ненакладених частин залишається сталою.

Дане доведення доводить майже ті самі твердження, що і лема 2.4, проте здається значно громіздкішим. Проте, на відміну від леми 2.4, воно може бути поширеним на випадок $r=n$. Так, об'єднаємо ідеально накладені розподіли $\alpha_2^0, \alpha_2^t, \dots, \alpha_2^{(b-1)t}$ в єдину ланцюгову групу λ_0^3 , де верхнім індексом позначатимемо порядок, а нижнім – індекс початкової точки. Тоді, перетворимо формули (2.10) і (2.11) в наступний вигляд:

$$\alpha_3 \ni \begin{pmatrix} \alpha_2^0 & \alpha_2^t & \dots & \alpha_2^{(b-1)t} \\ \alpha_2^1 & \alpha_2^{1+t} & \dots & \alpha_2^{1+(b-1)t} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_2^{t-1} & \alpha_2^{2t-1} & \dots & \alpha_2^{bt-1} \end{pmatrix} = \begin{pmatrix} \lambda_0^3 \\ \lambda_1^3 \\ \vdots \\ \lambda_{t-1}^3 \end{pmatrix} \quad (2.14)$$

$$\alpha_3 \ni \begin{pmatrix} \lambda_0^3 \\ \lambda_1^3 \\ \vdots \\ \lambda_{t-1}^3 \end{pmatrix} \ni \begin{pmatrix} L_0^3 & L_1^3 & \dots & L_{t-1}^3 \\ L_b^3 & L_{b+1}^3 & \dots & L_{b+t-1}^3 \\ \vdots & \vdots & \ddots & \vdots \\ L_{b(t-1)}^3 & L_{b(t-1)+1}^3 & \dots & L_{(b+1)(t-1)}^3 \end{pmatrix} \quad (2.15)$$

Узагальнимо ці формули для деякого рангу r . Згідно теореми 2.2, будь-який розподіл рангу r містить в собі $k = tb$ розподілів рангу $r-1$, накладених із кроком b^{r-1} . Очевидним є той факт, що якщо ці розподіли містять t^{r-2} ланцюгів рангу $r-1$, то їх довжини є рівними tb^{r-1} – значить, формула 2.14 прийме вигляд:

$$\alpha_r \ni \begin{pmatrix} \lambda_0^r \\ \lambda_1^r \\ \vdots \\ \lambda_{t-1}^r \end{pmatrix} = \begin{pmatrix} \alpha_{r-1}^0 & \alpha_{r-1}^t & \dots & \alpha_{r-1}^{(b-1)t} \\ \alpha_{r-1}^1 & \alpha_{r-1}^{1+t} & \dots & \alpha_{r-1}^{1+(b-1)t} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{r-1}^{t-1} & \alpha_{r-1}^{2t-1} & \dots & \alpha_{r-1}^{bt-1} \end{pmatrix} \quad (2.16)$$

Що ж відбуватиметься із ланцюговою матрицею A_n (формула 2.12)? При розширенні до A_4 матриця стане тривимірною, оскільки нові індекси, згідно формулі 2.11, не вливатимуться в уже існуючу систему індексування і потребуватимуть виділення додаткового простору. При цьому, оскільки крок накладення – b^3 , то їх індекси змінюватимуться пропорційно b^2 . Продемонструвавши тривимірну матрицю через декілька двовимірних, маємо:

$$A_4 = \begin{pmatrix} L_0^4 & L_1^4 & \cdots & L_{t-1}^4 \\ L_b^4 & L_{b+1}^4 & \cdots & L_{b+t-1}^4 \\ \vdots & \vdots & \ddots & \vdots \\ L_{b(t-1)}^4 & L_{b(t-1)+1}^4 & \cdots & L_{(b+1)(t-1)}^4 \end{pmatrix} \cdots \begin{pmatrix} L_{(t-1)b^2}^4 & L_{(t-1)b^2+1}^4 & \cdots & L_{(t-1)(b^2+1)}^4 \\ L_{(t-1)b^2+b}^4 & L_{(t-1)b^2+b+1}^4 & \cdots & L_{(t-1)(b^2+1)+b}^4 \\ \vdots & \vdots & \ddots & \vdots \\ L_{(t-1)(b^2+b)}^4 & L_{(t-1)(b^2+b)+1}^4 & \cdots & L_{(t-1)(b^2+b+1)}^4 \end{pmatrix} \quad (2.17)$$

За аналогією, можна представити і $(r-1)$ -вимірну матрицю у наступному вигляді (для зручності перенесемо порядок у верхній індекс):

$$A^r = \begin{pmatrix} A_0^{r-2} & A_{b^{r-3}}^{r-2} & \cdots & A_{(t-1)b^{r-3}}^{r-2} \\ A_{b^{r-2}}^{r-2} & A_{b^{r-3}(b+1)}^{r-2} & \cdots & A_{(b+t-1)b^{r-3}}^{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{b^{r-2}(t-1)}^{r-2} & A_{b^{r-2}(t-1)+b^{r-3}}^{r-2} & \cdots & A_{(t-1)\sum_{i=0}^{r-2} b^i}^{r-2} \end{pmatrix} \quad (2.18)$$

Тут A^{r-2} є $(r-3)$ -вимірними матрицями, кожна з яких містить t^{r-3} ланцюгів порядку r , утворених через злиття розподілів згідно до формули 2.16.

Тепер доведемо рівномірність. Знаючи довжину ланцюгів tb^r і вигляд останнього індексу $(t-1)\sum_{i=0}^{r-2} b^i$, підставимо їх у нерівність 2.13. Відповідно:

$$\left(1 - \frac{1}{t}\right) \sum_{i=0}^{r-2} b^i \leq b^{r-1} \quad (2.19)$$

Аналогічно, підставивши початкову точку $(2, 2)$, маємо:

$$\frac{1}{2} \sum_{i=0}^{r-2} 2^i \leq 2^{r-1} \rightarrow \frac{1}{2} (2^{r-1} - 1) \leq 2^{r-1} \quad (2.20)$$

Звідси робимо висновок, що початкове значення для лівої частини менше ніж для правої. Аналогічно, перетворивши частини нерівності на функції та продиференціювавши по b^{r-2} , отримуємо ту ж саму нерівність похідних $(1 - 1/t) \leq b$, що гарантує відсутність будь-яких перетинів між ними, а отже, неіснування такої пари (t, b) , що порушувала б зазначену нерівність. Звідси можемо зробити висновок, що всі ланцюги $L^r \in A^r$ перетинаються, отже, розподіл завжди є рівномірним, його максимум представлень $\alpha_{max}(r) = t^{r-1}$, а число унікальних кодів $n_{\alpha=1}(r) = n_{\alpha=1}(2) = 2b$.

Таким чином, висунуті гіпотези отримують математичне доведення.

2.4. Математична модель топологій на основі надлишкових кодів

Розглянуті теореми дозволяють розрахувати характеристики надлишковості топології ще до синтезу, опираючись на обраний код, проте залишаються ще декілька невирішених питань:

1. В доведенні леми 2.3 виникає ситуація, коли звичайний спектральний розклад не дає коректної відповіді на питання про число представлень, оскільки виникають ситуації, коли некоректні на перший погляд коди мають в системі коректні аналоги. Ця проблема вирішується через функцію значень і базові коди, проте дане явище ставить питання про зв'язок між різними алфавітами.

2. В теоремі 2.4 запропоновано механізм представлення розподілу через $(n-1)$ -вимірну матрицю. Це використовується лише як зручне представлення, проте такий підхід породжує питання про можливість використання такої структури для безпосереднього отримання значення розподілу та альтернативних представлень.

Отримання відповідей на ці питання дасть змогу не лише прогнозувати певні характеристики надлишкових графів, а й розробити підходи, які дозволили б удосконалити метод синтезу на основі надлишкового коду.

2.4.1. Використання багатовимірної форми для обчислення $\alpha_r(V)$

Почнемо із другого питання, оскільки воно має вищу практичну цінність. Для початку перетворимо матрицю A^r і введемо багатовимірну індексацію. Так, кожному ланцюгові порядку r ставитимемо у відповідність множину його індексів в матриці. Для прикладу, наступним чином виглядатиме матриця порядку 4:

$$A_4 = \begin{pmatrix} L_{(1,1,1)}^4 & L_{(1,1,2)}^4 & \cdots & L_{(1,1,t)}^4 \\ L_{(1,2,1)}^4 & L_{(1,2,2)}^4 & \cdots & L_{(1,2,t)}^4 \\ \vdots & \vdots & \vdots & \vdots \\ L_{(1,t,1)}^4 & L_{(1,t,2)}^4 & \cdots & L_{(1,t,t)}^4 \end{pmatrix} \cdots \begin{pmatrix} L_{(t,1,1)}^4 & L_{(t,1,2)}^4 & \cdots & L_{(t,1,t)}^4 \\ L_{(t,2,1)}^4 & L_{(t,2,2)}^4 & \cdots & L_{(t,2,t)}^4 \\ \vdots & \vdots & \vdots & \vdots \\ L_{(t,t,1)}^4 & L_{(t,t,2)}^4 & \cdots & L_{(t,t,t)}^4 \end{pmatrix} \quad (2.21)$$

Таким чином, маючи кортеж індексів $I = (i_{r-1}, i_{r-2}, \dots, i_1, i_0)$ можна однозначно встановити зміщення ланцюга відносно початку розподілу за формулою:

$$s_I = \sum_{j=0}^{r-2} (i-1)_j b^j \quad (2.22)$$

Як можна бачити, ця формула майже еквівалентна формулі 2.1, що описує отримання значення коду в позиційній СЧ. Таким чином, можна казати про існування окремої індексної системи числення (ІСЧ) типу (t, b) , кожен розряд $\sigma_j = i_j - 1$ якої описує місцеположення відповідного ланцюга в багатовимірній матриці, а його значення кодує зміщення даного ланцюга відносно початку розподілу. Це також означає, що при $t > b$ ІСЧ стане надлишковою, а значить, існуватимуть надлишкові ланцюги з тим самим зміщенням, що ідеально покриватимуть одне одного.

Втім, спробуємо використати властивості ланцюгів для вирішення проблеми пошуку представлень. Задаймо на ланцюгах операцію отримання коду $L_a(V)$, що повертає код значення якщо ланцюг його містить або порожню множину.

Логічно, що для звичайного ланцюга порядку 1, де нижнім індексом позначено першу цифру, дана операція дає код $a[V - ba]$. Відповідно, отримання некоректного коду свідчить про те, що число не може бути представлено заданим ланцюгом. Відповідно, зміна індексування на проміжок $1 \dots k$ чи $0 \dots (k-1)$ не є критичною: знаючи основоположну послідовність, завжди можна виконати приведення до потрібної форми.

Тепер розглянемо ланцюги порядку r . Дана структура складається з великої кількості звичайних ланцюгів, які слідують одне за одним з кроком tb . Таким чином, знаючи початкову старшу цифру a і маючи деяке V , необхідно спершу визначити номер ланцюга в металанцюзі, для чого віднімемо від V значення наймолодшого коду (приведемо його до проміжку $0 \dots tb^r$) і розділимо на довжину одного ланцюга. Таким чином, отримуємо:

$$p = \left\lfloor \frac{V - ba - l}{tb} \right\rfloor \rightarrow a' = a + tp \rightarrow L_a^r(V) = a'[V - ba'] \quad (2.23)$$

Визначити, що V не має представлення в ланцюгу, також нескладно: коректне значення існує при $p \in [0, b^{r-1})$ і не існує при інших значеннях. Важливо відмітити, що дана формула не потребує знати самі значення, що входять в ланцюг — достатньо лише характеристик системи та першу цифру металанцюга.

Втім, ще два питання залишаються невирішеними. По-перше, формула 2.23 дозволяє визначити 2 цифри основного коду, а не весь код. При цьому необхідно знати

першу цифру послідовності a . По-друге, невирішеним залишається питання корекції V з урахуванням того, що ланцюг, для якого проводиться аналіз, є зміщеним відносно початку розподілу.

Знову-таки, почнемо з другого питання. Припустимо, що основоположна послідовність існує для проміжку $[0, tb)$. Тоді розглядатимемо $A^r(V)$ як перетворення, аналогічне $L_a(V)$, що виконується над усіма елементами матриці. Знаючи, що старший (перший) індекс кодує зміщення на b^{r-2} , маємо:

$$R^r = A^r(V) = (A_1^{r-1}(V) \quad A_2^{r-1}(V - b^{r-2}) \quad \dots \quad A_t^{r-1}(V - (t-1)b^{r-2})) \quad (2.24)$$

Дана формула є рекурентною і описує послідовний порядок розрахунку зміщення. Втім, знаючи, що індексом матриці є код в індексній СЧ, можна значно спростити цю операцію. Нехай відомо, що L_l^r є елементом A^r , а ρ_l^r – відповідним їй елементом матриці R^r . Тоді:

$$\rho_l^r = L_l^r \left(V - \sum_{j=0}^{r-2} (i_j - 1)b^j \right) = L_l^r(V - v(\sigma_{r-2}\sigma_{r-3} \dots \sigma_0)) = L_l^r(V - s_l) \quad (2.25)$$

Наступна задача – встановити вигляд старших цифр, маючи код в індексній СЧ. Важливо помітити, що ІСЧ відноситься до типу (t, b) , а вихідна – до типу (tb, b) . Тож, виникає питання – як співвідносяться між собою коди цих двох систем?

Розглянемо випадок $r=2$. Формула 2.8 описує матрицю розмірності $b \times t$, де кожні b ланцюгів об'єднані в t ланцюгів порядку 2, а індекс (молодша цифра ІСЧ) вказує на модуль від ділення на t початкової цифри ланцюга, приведеної до основоположної послідовності $[0, tb)$. Тобто, якщо шуканий код представлено у виді $a_{r-1}a_{r-2} \dots a_0$, то $\sigma_0 = (a_1 - l) \bmod t$.

Тепер розглянемо $r=3$. Згідно формулам 2.10, 2.14 і 2.15 розподіл розкладається на менші розподіли, що зливаються між собою. Згідно теореми 2.2 загальне число базових розподілів рівне bt , а їх «зовнішнім індексом» можна вважати цифру a_2 , якій відповідає конкретний розподіл. Проте, оскільки крок накладання – t , то в ІСЧ це значення представлено все тією ж формулою $\sigma_1 = (a_2 - l) \bmod t$. Наступні кроки, аналогічно, розширюють систему на bt , але кожні b ланцюгів поглинаються ланцюгами вищого порядку. Таким чином:

$$\sigma_i = (a_{i+1} - l) \bmod t \quad (2.26)$$

Як же відновити код $a_{r-1} a_{r-2} \dots a_0$, маючи індексний код $I = \sigma_{r-2} \sigma_{r-3} \dots \sigma_0$ і скориговане значення $V' = V - s_I$? Знаючи, що $V' = 0$ вказує на початок конкретного ланцюга порядку r , а допустимий діапазон значень є рівним tb^r , можемо відразу перевірити коректність. Таким чином, $\forall V' \notin [0, tb^r) \rightarrow L(V') = \emptyset$. В іншому випадку щоб отримати V' -й елемент ланцюга, необхідно і достатньо знайти код $P = p_{r-2} p_{r-3} \dots p_0$ в системі типу (b, b) .

Розглянемо детальніше це кодування. Відомо, що ланцюг порядку r містить b ланцюгів порядку $r-1$. P -код вказує на номер кожного підланцюга в ланцюзі вищого порядку, тож, старші цифри шуканого коду можуть бути відновлені за формулою:

$$a_{i+1} = tp_i + \sigma_i + l \quad (2.27)$$

Відповідно, залишається лише визначити P -код. Знаючи сенс кодування, це можна легко зробити, використовуючи наступну формулу:

$$p_i = \left\lfloor \frac{V'_i}{tb^i} \right\rfloor = \left\lfloor \frac{V' - t \sum_{j=i+1}^r p_j b^j}{tb^i} \right\rfloor = \left\lfloor \frac{V'}{tb^i} \right\rfloor - \sum_{j=1}^{r-i} p_{j+i} b^j \quad (2.28)$$

Логіка даної формули в тому, щоб, рухаючись від старшої цифри до молодшої, знайти, на який підланцюг вказує V . Знаючи довжину кожного i -рангового ланцюга, можна точно визначити, в який діапазон потрапляє значення V , використовуючи ділення націло. Проте необхідно враховувати, що кожне V'_i має бути вирівняне по діапазону так, щоб вказувати на початок підланцюга порядку $i+1$ (що реалізується через віднімання вже знайдених частин). Інший варіант – знайти абсолютний номер ланцюга i -го порядку в ланцюзі порядку r , і вже потім виконати корекцію, віднявши число i -рангових ланцюгів між початком ланцюга порядку r і ланцюга порядку $i+1$.

Остання цифра a_0 , таким чином, може бути отримана наступним чином:

$$a_0 = V' - \sum_{j=0}^{r-2} p_j b^j - l = V - bv(a_{r-1} a_{r-2} \dots a_1) + l \quad (2.29)$$

Таким чином, дійсно, існує спосіб отримання $\alpha_r(V)$ через перетворення над $(r-1)$ -вимірною матрицею, проте цей спосіб не є вигідним з точки зору практичного застосування, оскільки передбачає велику кількість операцій множення та ділення.

2.4.2. Імпліцитні кластери на основі багатовимірної форми

Втім, є ще один аспект, який впливає із багатовимірної форми. Нехай для певного V дана матриця R^r , що містить всі його надлишкові представлення. Знаючи про метод шаблонної маршрутизації та розклад на спектри, можемо представити кожен елемент ρ_l^R як спектр шаблонів $(T_{r-2}, T_{r-3}, \dots, T_0)$, а перехід по кожному із цих шаблонів – як кодове перетворення, що служить основою для синтезу деякого графу Q_V^r , що співіснує із загальним графом G^r , отриманим в результаті загального методу синтезу на основі надлишкового коду.

Таким чином, 2 питання може бути поставлено:

1) який вигляд має граф Q_V^r ?

2) яким чином спектр $(T_{r-2}, T_{r-3}, \dots, T_0)$ співвідноситься з кодом $\sigma_{r-2} \sigma_{r-3} \dots \sigma_0$ в індексній СЧ і як перетворення над спектром впливають на індексне кодування?

Почнемо зі співвідношення. Відомо, що σ_i , фактично, кодує модуль цифри a_{i+1} . Шаблон, в свою чергу, є парою цифр, де старша цифра при реалізації змінюється з кроком 1, а молодша – із кроком b . Очевидно, що модуль старшої цифри при цьому змінюватиметься. Виникає питання: чи може виникнути ситуація, коли 2 форми одного шаблону матимуть однакову форму? Припустимо, що це можливо. Для цього старша цифра має змінитись на $\pm t$, а молодша – відповідно, на $\pm bt$. При такій зміні молодша цифра гарантовано вийде за межі основоположної послідовності, тож, можна казати про існування взаємно-однозначного зв'язку між шаблоном і модулем від ділення його старшої цифри на t . Таким чином, індексний код може слугувати в якості скороченої форми опису спектру.

Втім, наявність зв'язку між спектром та багатовимірною формою не гарантує однозначності перетворень. Хоча одну позицію у матриці гарантовано займає лише один код, існує рівно b значень, що можуть ідентифікуватись конкретною цифрою ІСЧ. Це означає, що два коди ІСЧ, що відрізняються на одну цифру, необов'язково відрізнятимуться на один шаблон в загальній СЧ. Вірно і зворотне: оскільки шаблонне заміщення передбачає заміну відразу двох цифр, відмінність на один шаблон гарантує хемінгову відстань 2 між кодами ІСЧ, що їх відображають.

Проте, сам факт використання заміщення в якості базового перетворення дозволяє зробити припущення щодо (суб)оптимальної структури «багатовимірного кластеру». Виглядатиме воно наступним чином:

Гіпотеза 2.4 (про багатовимірний кластер). Багатовимірний (імпліцитний) кластер для r -рангового коду (tb, b) , представленого у вигляді ІСЧ відповідної багатовимірної форми, може бути зведений до $(r-1)$ -вимірного гіперкуба з надлишковою основою t .

Для перевірки даної гіпотези звернемо увагу на шаблонне заміщення в ІСЧ. Оскільки індексна СЧ використовує арифметику по модулю t , це означає, що важливим для неї є не сам результат операції, а її модуль. Таким чином, $T_i \rightarrow T_i^{+p} \Rightarrow a_{i+1}a_i \rightarrow [a_{i+1} + p][a_i - bp]$, що еквівалентно $\sigma_i\sigma_{i-1} \rightarrow [(\sigma_i + p) \bmod t][(\sigma_{i-1} - bp) \bmod t]$.

Виділимо деяку двовимірну підплощину (підматрицю) $\Psi_{\lambda,\mu}^i$ багатовимірної форми, таку, що $\forall \lambda, \mu \in \lambda\sigma_i\sigma_{i-1}\mu \in I^r, r > 2 \exists \Psi_{\lambda,\mu}^{i,i-1}: \lambda\sigma_i\sigma_{i-1}\mu \in \Psi_{\lambda,\mu}^i \forall \sigma_i, \sigma_{i-1} \in I$. Таким чином, елементами даної скінченої площини є точки виду (σ_i, σ_{i-1}) , де $\sigma_i, \sigma_{i-1} \in \mathbb{Z}$ і цифри σ_i, σ_{i-1} є числами в скінченному полі залишків за модулем t . Тож $\forall x, y \in \mathbb{Z}: (x, y) \equiv (x \bmod t, y \bmod t)$.

Таким чином, $\forall (\sigma_i, \sigma_{i-1}) \in \Psi_{\lambda,\mu}^i \exists e(\sigma_i, \sigma_{i-1})(\sigma_i + p, \sigma_{i-1} - bp) \in E \in Q_V^r: (\sigma_i + p, \sigma_{i-1} - bp) \in \Psi_{\lambda,\mu}^i$. Знаючи, що перетворення над точками виконується в модульній арифметиці, можемо представити одиничний зсув молодшого розряду як $\delta = b \bmod t$. Тоді індексний код сусіднього вузла матиме вигляд $(\sigma_i + p, \sigma_{i-1} - \delta p)$. Звідси нескладно встановити, що при $\delta = 0$ перетворення шаблонного заміщення є тотожним класичному заміщенню в ІСЧ.

Виникає питання, що ж робити, коли $\delta \neq 0$. Тоді виникає необхідність в перетворенні $\Psi_{\lambda,\mu}^i$ так, щоб $(\sigma_i + p, \sigma_{i-1} - \delta p) \rightarrow (\sigma_i + p, \sigma_{i-1})$. Таким чином, отримуємо альтернативну $\Psi_{\lambda,\mu}'^i$, що має деяку нейтральну точку (o_i, o_{i-1}) , еквівалентну відповідній точці в $\Psi_{\lambda,\mu}^i$, а інші точки якої перетворюються за правилом:

$$\begin{cases} \psi'_i = \sigma_i = o_i + \xi \\ \psi'_{i-1} = \sigma_{i-1} + \delta\xi = \sigma_{i-1} + \delta(\sigma'_i - o_i) \end{cases} \quad (2.30)$$

З точки зору того, що $\Psi_{\lambda,\mu}^i$ може бути представлена як двовимірна матриця кодів, $\Psi_{\lambda,\mu}^i$ матиме вигляд схожої за своєю структурою матриці, де деякий рядок буде еквівалентний $\Psi_{\lambda,\mu}^i$, а всі наступні рядки з індексами $+\xi$ – циклічно зсунуті на $\delta\xi$ відносно даного. Відповідно, при шаблонному переході між сусідніми рядками перетворення в $\Psi_{\lambda,\mu}^i$ матиме вигляд:

$$\begin{cases} \psi'_i = \psi + 1 \\ \psi'_{i-1} = \psi_{i-1} - \delta + \delta(\psi_i - \psi'_i) = \psi_{i-1} \end{cases} \quad (2.31)$$

Відповідно, таким чином можна казати про можливість перетворення багатовимірної форми у такий вигляд, де перехід по шаблону в ІСЧ буде еквівалентним простому заміщенню.

Розглянемо приклад. Нехай дано систему 0123TZ типу (6, 2). Її параметри $t=3$, $b=2 \rightarrow \delta=2$. Нехай $r=4$, а досліджувана точка – 0. Вона має 27 представлень наступного виду: 0000, T200, 1Z00, T120, Z320, 0T20, 01Z0, T3Z0, 1TZ0, T112, Z312, 0T12, T032, Z232, 0Z32, 00T2, T2T2, 1ZT2, 001Z, T21Z, 1Z1Z, T13Z, Z33Z, 0T3Z, 01TZ, T3TZ, 1TTZ. Спробуємо присвоїти їм індекси згідно правилам перетворення ІСЧ (деталі див. у табл. Д.3). Тоді можемо сформуувати багатовимірну форму для неї, що матиме наступний вид:

$$R^r = \begin{pmatrix} 1Z1Z & 1ZT2 & 1Z00 \\ 1TZ0 & 1TTZ & Z232 \\ Z312 & Z320 & Z33Z \end{pmatrix} \begin{pmatrix} T112 & T120 & T13Z \\ T21Z & T2T2 & T200 \\ T3Z0 & T3TZ & T032 \end{pmatrix} \begin{pmatrix} 01Z0 & 01TZ & 0Z32 \\ 0T12 & 0T20 & 0T3Z \\ 001Z & 00T2 & 0000 \end{pmatrix} \quad (2.32)$$

Як видно із даної форми, заміщення молодшого шаблону завжди відповідає однорозрядному заміщенню по молодшому індексу (але не навпаки). Проте для інших індексів це не є правдою – для коректності необхідно змінювати 2 цифри. В той же час, використовуючи формули 2.30 і 2.31, можемо переіндексувати елементи форми, щоб отримати альтернативну форму. В якості нейтрального елементу оберемо точку (0, 0, 0). Таким чином, можемо визначити наступні правила даної переіндексації:

1. Старший індекс кожного елементу (матриця) не змінюється ($\psi_2 = \sigma_2$)

2. Середній індекс (рядок) змінюється в залежності від номеру матриці. Елементи матриці 0 не змінюються. В рамках матриці 1 рядки зсуваються вгору на 1, в рамках матриці 2 – вниз на 1 ($\psi_1 \equiv \sigma_1 + 2\sigma_2$)

3. Молодший індекс змінюється в залежності від номеру рядка в матриці. Рядок 0 не змінюється, рядок 1 зсувається вліво, рядок 2 – на 1 елемент вправо ($\psi_0 = \sigma_0 + 2\sigma_1$)

Таким чином, отримуємо наступну змінену багатовимірну форму:

$$R'^r = \begin{pmatrix} 1Z1Z & 1ZT2 & 1Z00 \\ 1TTZ & Z232 & 1TZ0 \\ Z33Z & Z312 & Z320 \end{pmatrix} \begin{pmatrix} T21Z & T2T2 & T200 \\ T3TZ & T032 & T3Z0 \\ T13Z & T112 & T120 \end{pmatrix} \begin{pmatrix} 001Z & 00T2 & 0000 \\ 01TZ & 0Z32 & 01Z0 \\ 0T3Z & 0T12 & 0T20 \end{pmatrix} \quad (2.33)$$

Ключовою властивістю даної форми є те, що заміщення шаблону веде до переходу в одному вимірі, але не в кількох одночасно. В такому разі казатимемо про відповідність по конкретному шаблону. Так, між матрицями еквівалентні елементи мають еквівалентну частину $a_1 a_0$ (T2-відповідність). В рамках стовпців, відповідно, еквівалентними є a_4 і a_0 (T1-відповідність), а в рамках рядків – $a_3 a_2$ (T0-відповідність). Звісно, це еквівалентність деколи порушується. Розглянемо ці виключення детальніше:

1. Всі елементи багатовимірної форми виконують умову T2-відповідності

2. T1-невідповідність властива наступним елементам:

- a. (020) Z33Z. Шаблон 33 не має інших представлень, тож не формуватиме зв'язок в класичному розкладі.
- b. (001) 1ZT2. Аналогічно, ZT не має альтернативних форм.
- c. (022) Z320. Послідовність 32 є унікальною.

3. T0-невідповідність характерна для наступних елементів:

- a. (011) Z232. Молодший шаблон 32 в ній є унікальним, тож розклад на спектри не дасть жодного зв'язку, що не передбачався би багатовимірною формою.
- b. (111) T032. Аналогічно, перехід по 32 є неможливим.
- c. (211) 0Z32. Ситуація аналогічна іншим.

Таким чином, можна казати про можливість приведення багатовимірної форми до такого вигляду, що для будь-якого можливого шаблонного переходу існуватиме еквівалентне йому однорозрядне заміщення в багатовимірній формі (але не навпаки!). Як наслідок, природньою топологією для кластеру, сформованого на основі такої форми, є гіперкуб із надлишковою основою t , де внутрішнім ідентифікатором вершини є її код в ІСЧ.

2.4.3. Спосіб формування імпліцитних кластерів в надлишкових топологіях

На основі запропонованої математичної моделі було розроблено спосіб [4] формування імпліцитних (прихованих) кластерів в надлишкових топологіях для кодів типу (tb, b) . Кожен такий кластер v включає в себе всі вузли, що кодуються альтернативними представленнями v . Запропонований спосіб включає в себе наступні кроки:

1. Формування множини всіх r -розрядних кодів S в деякій СЧ з алфавітом $A = \{l, l+1, \dots, t-1, t\}$ та основою числення b . Дана множина має взаємно-однозначну відповідність з елементами множини вузлів деякого графа G .
2. Для кожного $V \in [v(ll \dots l); v(tt \dots t)]$ виконується виділення всіх альтернативних представлень в множину альтернативних представлень S .
3. Формуємо множину індексів I , що містить індексні коди в ІСЧ відповідної СЧ, має взаємно-однозначну відповідність з елементами множини S і формується за правилом: $\forall s = a_{r-1} a_{r-2} \dots a_0 \in S \exists i = \sigma_{r-2} \sigma_{r-3} \dots \sigma_0: \sigma_j = (a_{j+1} - l) \bmod t \forall j = [0; r-2]$
4. Підраховуємо коефіцієнт невідповідності $\delta = b \bmod t$. Якщо $\delta = 0$ – цей крок пропускається ($\psi_j = \sigma_j \forall j$). Інакше виконується додаткове перетворення $\psi_{r-2} = \sigma_{r-2}; \psi_j = \sigma_j + \delta \sigma_{j+1} \forall j = [0, r-2]$.
5. У відповідності до отриманих кодів в ІСЧ виконується синтез гіперкуба з використанням методу синтезу на основі кодових перетворень (описаний в Розділі 3). Неіснуючі коди можуть як пропускатись (ці вершини зв'язки до них не додаються) так і додаватись в граф (при цьому взаємна однозначність індексних кодів та кодів основної СЧ буде порушена).

2.4.4. Порядок алфавіту та надлишкова самоподібність

Завершивши опис багатовимірної форми, можемо повернутись до останнього питання – питання алфавіту. Для алфавіту ключовими характеристиками є його розмір k та основоположна послідовність, яка визначається множиною цифр $A = \{l, l + 1, \dots, m - 1, m\}$. Нехай дано інший алфавіт $A' = \{l', l' + 1, \dots, m' - 1, m'\}$. Введемо між даними алфавітами відношення порядку, таке, що якщо $A \supset A'$, то A є алфавітом на $|A| - |A'| = p \in$ порядків вищим. Очевидно, абсолютний порядок алфавіту A є рівним його кардинальному числу $|A| = k$, проте якщо $A \not\supset A'$, то в такому разі казатимемо, що їх порядок є непорівнюваним.

Використовуючи ці алфавіти для синтезу топології, отримуємо два графи G та G' . Оскільки всі коди, що належать A' , містяться і в A , це означає, що всі вершини G' також належать і G . Що стосується зв'язків, то це залежить від перетворень синтезу. Введемо таку характеристику перетворення як самоподібність. Перетворення f є самоподібним тоді і тільки тоді, коли $\forall A' \subset A, C = a_{r-1}a_{r-2} \dots a_0 \in A': F_{A'}(C) = F_A(C)$. Виникає питання: чи є перетворення зсуву та заміщення, які розглядалися раніше, самоподібними?

Почнемо із перетворень зсуву. Нехай деякий код $C = a_{r-1}a_{r-2} \dots a_1a_0$ належить до алфавіту A' . Очевидно, що оскільки $A' \subset A$ цей код є коректним і в A . Припустимо, що перетворення зсуву не є самоподібним – це означає, що при виконанні перетворення f в A' можливо отримати інший результат, аніж в A . Позначимо вставку як d . Тоді $Sl_d(C) = a_{r-2} \dots a_0d, Sr_d(C) = da_{r-1}a_{r-2} \dots a_1$. Очевидно, $d \in A'$, а оскільки $A' \subset A$ то $d \in A$. Таким чином, виникає протиріччя: для будь-якого зсуву зі вставкою d в A' існує еквівалентне перетворення в A .

Те саме стосується і заміщення. Незалежно від місця заміщення, якщо $C \in A' \in A$, то $\forall d \in A' \exists E_A(C) = a_{r-1}a_{r-2} \dots a_{i+1}da_{i-1} \dots a_1a_0 = E_{A'}(C)$. Аналогічно, оскільки багаторозрядне заміщення можна розкласти на декілька порозрядних, будь-яке заміщення без додаткових умов є самоподібним.

Те саме стосується всіх перетворень, пов'язаних із перестановкою цифр коду (циклічний зсув, реверсивний код, тощо). Оскільки всі цифри коду одночасно

належать і A' і A , будь-яка їх комбінація також залишатиметься коректною для обох цих систем.

Втім, не-самоподібні перетворення також існують. Наприклад, заміщення по модулю k передбачає, що певна цифра коду заміщується не на випадкове d , а на конкретне $a_i + \Delta$ з урахуванням того, що основоположна послідовність утворює собою скінченне кільце залишків за модулем k . Відповідно, оскільки це кільце базується на основоположній послідовності в цілому, воно відрізнятиметься для A та A' , тож результат операції необов'язково буде еквівалентним.

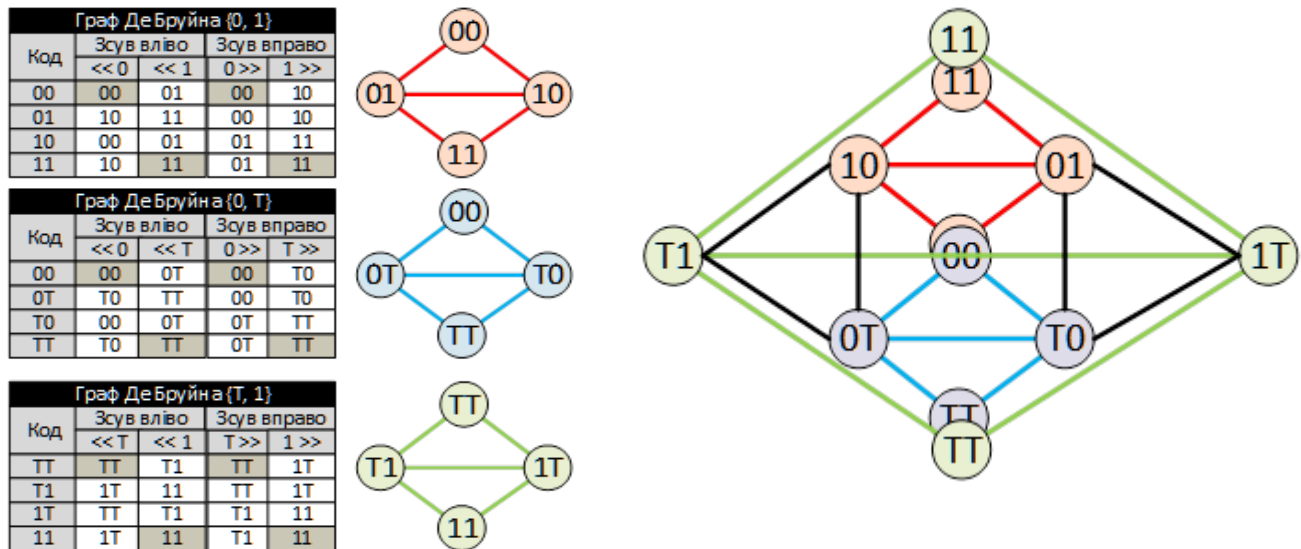
Відповідно, застосувавши ці факти до топологій, отримуємо наступну теорему:

Теорема 2.5 (про самоподібність графів). Для будь-якого графу G утвореного на основі алфавіту A і самоподібних кодових перетворень існує декомпозиція на k графів G' , що базуються на алфавітах A' , чий порядок на 1 нижче ніж у алфавіту A .

Доведення даної теореми є елементарним. Твердження про те, що граф $G'(V', E')$ є підграфом $G(V, E) \forall A' \subset A$ базується на двох інших твердженнях: $V' \subset V, E' \subset E$. Оскільки V' утворюється на основі алфавіту A' , всі можливі коди $C \in A'$ в той же час належать і A . Щодо зв'язків, їх відповідність доводиться визначенням самоподібності перетворень: оскільки кожне перетворення в A' має відповідну еквівалентну форму в A , це означає, що всі зв'язки G' існують і в G .

Відповідно, кількість G' слідує із того факту, що кількість підмножин, що містять на 1 елемент менше, є рівною числу елементів – тобто, k .

Розглянемо для прикладу граф де Бруїна та надлишкову бінарну систему. Її алфавіт $\{-1, 0, 1\}$ розкладається на 3 алфавіти $\{0, 1\}$, $\{-1, 0\}$, $\{-1, 1\}$, 2 з яких є нормальними, а 1 – аномальний. У відповідності до кодового синтезу, всі ці 3 алфавіти також можуть бути використані для синтезу топології, таким чином, маємо 3 псевдобінарні топології. В той же час ці топології, згідно самоподібності зсуву, можуть бути виділені в еквівалентній їм надлишковій топології. Рис. 2.11 демонструє подібну декомпозицію.



(а) 3 псевдобінарні графи

(б) самоподібна декомпозиція надлишкової топології де Бруйна

Рис. 2.11. Формування топології надлишкового де Бруйна рангу 2 із 3 топологій (псевдо)бінарного де Бруйна рангу 2.

Як видно із демонстрації, не всі зв'язки, присутні в цілісній топології, представлені у декомпозиції. Так само, в ній можуть бути представлені і не всі вершини. Візьмемо в якості прикладу вершину із кодом T01, що належатиме надлишковій топології де Бруйна рангу 3. Очевидно, що жоден із варіантів декомпозиції по підалфавітам $\{0,1\}$, $\{-1,0\}$, $\{-1,1\}$ не включатиме в себе такий код. Таким чином, кожен елемент (вершина чи зв'язок), присутній в декомпозиції, буде присутній і в цілій топології, проте ціле не буде ідеальною сумою своїх складових. Це означає, що декомпозиції дебруйнівських підграфів на дерева не будуть еквівалентні відповідній декомпозиції загального графу. Крім того, жодна вершина крім тих, що складаються з однакових цифр (00..0, 11..1, TT..T), не може належати двом підграфам одночасно – таким чином, декомпозиція кожного підграфу міститиме унікальні дерева, що не є тотожними ні одне одному, ні деревам основного графу.

2.4.5. Формальний опис математичної моделі та доведення її адекватності

Вхідними даними для моделі є:

- Надлишковий код, що має нормальний алфавіт A потужності k та основу числення b .

- Перетворення $f_1 \dots f_n$ в заданому надлишковому коді, за якими виконується синтез надлишкового графу.

Вихідними величинами є:

- Кількість альтернативних представлень $\alpha_r(v)$ деякого числа v в r -розрядному коді, що фактично показує розмір імпліцитного багатовимірного кластеру з номером v . Дане значення обчислюється рекурентно за формулою (2.4) як $\alpha_r(v(y\mu)) = \sum_{i=l-y}^{m-y} \alpha_{r-1}(v(\mu) + ib^{r-1})$, де μ – деякий код порядку $r-1$, а y – цифра, що додається для розширення μ до порядку r .
- Максимальне число альтернативних представлень $\alpha_{max} = \max(\alpha_r(v)) \forall v \in \mathbb{Z}$ та число унікальних представлень $n_{\alpha=1}$, яке для $\alpha 1$ -стабільних кодів вимірюється як $n_{\alpha=1} = 2b$.
- Основні характеристики топології на основі коду типу (tb, b) з урахуванням імпліцитних кластерів, такі як ступінь та діаметр.
 - $S \leq S_0 + \log_t(\alpha_{max})$, де S_0 – ступінь синтезованого графу без урахуванням імпліцитних кластерів.
 - $D \leq D_0$, де D_0 - діаметр синтезованого графу без урахуванням імпліцитних кластерів.
- Графи, на які може бути виконана декомпозиція заданого графу: k надлишкових підграфів того ж типу на основі алфавітів порядку $k-1$, спільними вершинами яких є вершини з однаковими цифрами (такі як 00..0) та кільцеві графи, що не входять до відповідних підграфів.

Довести адекватність даної моделі можливо, тому що її складові опираються на гарно відомі математичні теорії, такі як теорія систем числення, теорія множин, теорія графів. Оскільки сама математична модель була запропонована щоб пояснити та обґрунтувати деякі емпіричні результати через відому математичну теорію позиційних систем числення, а також використати її для розробки способу формування імпліцитних кластерів в надлишкових топологіях. Тож, необхідним і достатнім є доведення того, що запропонована модель дійсно коректно описує емпіричні дані. Це доведення є наступним:

- Властивості формули (2.4) підтверджуються експериментальними дослідженнями (див. Додаток В), що охоплювало всі коди із $k = 3 \dots 7, b = 2 \dots k, l_{min} = -9, m_{max} = 9$. Ідея даного дослідження полягала у використанні добре відомих властивостей позиційних СЧ для отримання α -розподілів для відповідних кодів. Отримані результати повністю відповідають запропонованій математичній моделі.
- Властивості $n_{\alpha=1}$ слідують із визначення $\alpha 1$ -стабільних кодів, отже, доведення не потребують.
- Ступінь графа визначає максимальне число ребер, інцидентних вершині. Відомо, що багатовимірний кластер має гіперкубічну форму з основою t , отже його ступінь є рівною $S_h = \log_t(\alpha_{max})$. Неповні гіперкубічні кластери мають менше вершин та ребер, але вершини, що вже мають ребра, нових ребер при усміченні не отримують, отже, $S_h^* \leq S_h$. Найгіршим для надлишкової топології випадком є ситуація, коли у вершину зі ступінню $S_i = S_0$ інтегрується відповідна вершина гіперкуба із $s_j = S_h$. Тоді загальна ступінь $S = S_0 + S_h = S_0 + \log_t(\alpha_{max})$. В інших випадках ступінь буде меншою.
- Аналогічно, для діаметру неможливим є зменшення, оскільки зв'язки додаються в граф, але не прибираються. Таким чином, гіпотетично допустимою є ситуація падіння діаметру, проте принципово неможливим є його зростання.
- Характеристики декомпозиції графа доводяться теоремою 2.5 на основі добре відомих математичних теорій, таких як теорія позиційних СЧ та теорія множин. Оскільки топологія (граф) сама по собі є добре відомою математичною моделлю деякої мережі, її адекватність не потребує доведення.

Таким чином, показано, що розроблена модель може бути застосована для досліджен відмовостійких топологій на основі надлишкових позиційних систем числення.

Висновок до розділу 2

В розділі представлено дослідження надлишкових кодів для синтезу топологій, виділено основні властивості таких кодів. Математично доведено, що для основних (в контексті синтезу топологій) властивостей кодів, а саме, для максимального числа альтернативних представлень α_{max} , числа унікальних (не-надлишкових) представлень $n_{\alpha=1}$ та форми розподілу числа альтернативних представлень $\alpha(v)$ деякого числа v , ключовими параметрами є потужність k нормального алфавіту A та основа числення b . Вміст самого алфавіту, мінімальні та максимальні значення цифр не впливають на ці характеристики і визначають лише зміщення α -розподілу відносно області визначення ($\alpha^{A1}(v) = \alpha^{A2}(v \pm d), k = |A1| = |A2|$).

Проаналізовано властивості надлишкових кодів типу (tb, b) , описано характеристики їх α -розподілів через концепцію багатовимірної форми. Виведено поняття індексної системи числення (ІСЧ), що дозволяє встановити взаємно-однозначну відповідність між альтернативним представленням деякого v та іншим кодом у відповідній ІСЧ, а отже, розглядати вузли з однаковими номерами як деяку топологію, отриману за допомогою методу синтезу на основі кодових перетворень.

Представлено математичну модель топологій на основі надлишкових кодів, що дає змогу розрахувати властивості графів на основі заданих надлишкових кодів, а також дозволяє декомпонувати довільний надлишковий граф на підграфи того ж типу на основі алфавіту меншого порядку.

Представлено спосіб створення багатовимірного імпліцитного кластеру в топологіях на основі надлишкових кодів типу (tb, b) , який дозволяє перетворити багатовимірну форму альтернативних представлень у такий вигляд, де перехід по шаблону відповідатиме однорозрядному заміщенню в ІСЧ, що еквівалентно кодовому перетворенню синтезу гіперкуба.

В якості перспективного напрямку дослідження необхідно звернути увагу на код $(4,2)$, який досить близький до $(3,2)$ і має кращі характеристики. Так, при $r=8$ його $\alpha_{max} = 128$ (проти 34), а $n_{\alpha=1} = 4$ (проти 17).

РОЗДІЛ 3

МЕТОД СИНТЕЗУ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ НА ОСНОВІ НАДЛИШКОВОГО КОДУ

3.1. Класичний метод синтезу топологій на основі кодових перетворень

Почати опис варто з короткого огляду класичного методу. Загалом, синтез на основі кодів передбачає наступні кроки:

1. Обрання довжини коду (рангу топології) $r = \log_n(N_{desired} - 1) + 1$, де $N_{desired}$ – бажане число вузлів системи, n – тип (основа) системи числення, що використовується.
2. Формування множини кодів C , що має потужність $N = n^r$ і містить всі можливі r -розрядні коди $a_{r-1}a_{r-2} \dots a_1a_0$ обраної системи числення.
3. Формування множини V вершин графа $G(V, E)$, елементи якої мають взаємно-однозначну відповідність із елементами множини C .
4. Обрання бажаних кодових перетворень f_1, f_2, \dots, f_m в заданій системі числення.
5. Формування множини E ребер графа $G(V, E)$ за правилом $\forall i = 1 \dots m \ \& \ \forall c \in C: f_i(c) = c'_i, c'_i \in C \rightarrow \exists e_{c \leftrightarrow c'_i} \in E$.

Відповідно, «ядром» методу є дві речі: обраний код і перетворення, взяті для синтезу. Список цих перетворень є досить містким і включає в себе велику кількість варіантів, втім, серед них можна виділити декілька основних:

- Заміщення, що полягає в заміні деяких цифр обраного коду за певним правилом. Наступні підвиди можуть бути виділені:
 - Порозрядне (класичне) заміщення
 $E_i(a_{r-1}a_{r-2} \dots a_{i+1}a_i a_{i-1} \dots a_1a_0) \rightarrow a_{r-1}a_{r-2} \dots a_{i+1}a'_i a_{i-1} \dots a_1a_0$
 - Групове заміщення
 $E_{i..j}(a_{r-1}a_{r-2} \dots a_{i+1}a_i a_{i-1} \dots a_{j+1}a_j a_{j-1} \dots a_1a_0) \rightarrow$
 $a_{r-1}a_{r-2} \dots a_{i+1}a'_i a'_{i-1} \dots a'_{j+1}a'_j a_{j-1} \dots a_1a_0$
 - Повне заміщення $E_f(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow a'_{r-1}a'_{r-2} \dots a'_1a'_0$.

- Тасування, що полягає у зсуванні коду чи його частини зі вставкою певного значення. Аналогічно, існує декілька підвидів даної операції, а саме:
 - Звичайне тасування $SL_d(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow a_{r-2} \dots a_1a_0d$ і $SR_d(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow da_{r-1}a_{r-2} \dots a_1$.
 - Ідеальне тасування $PS(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow a_{r-2} \dots a_1a_0a_{r-1}$.
 - Зворотне тасування $US(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow a_0a_{r-1}a_{r-2} \dots a_1$.
 - Підтасування $Sb_i(a_{r-1}a_{r-2} \dots a_{i+1}a_ia_{i-1} \dots a_1a_0) \rightarrow a_{r-1}a_{r-2} \dots a_{i+1}a_{i-1} \dots a_1a_0a_i$
 - Надтасування $Sp_i(a_{r-1}a_{r-2} \dots a_{i+1}a_ia_{i-1} \dots a_1a_0) \rightarrow a_ia_{r-1}a_{r-2} \dots a_{i+1}a_{i-1} \dots a_1a_0$.
- Перестановка кодів, що базується на перестановці обраних частин коду без зсування чи інших змін над іншими його частинами. Наступні типи варто розглянути уважніше:
 - Однорозрядна перестановка $B_{i,j}(a_{r-1}a_{r-2} \dots a_{i+1}a_ia_{i-1} \dots a_{j+1}a_ja_{j-1} \dots a_1a_0) \rightarrow a_{r-1}a_{r-2} \dots a_{i+1}a_ja_{i-1} \dots a_{j+1}a_ia_{j-1} \dots a_1a_0$
 - Групова перестановка $B_{i-j,k-m}(a_{r-1}a_{r-2} \dots a_{i+1}a_i \dots a_ja_{j-1} \dots a_{k+1}a_k \dots a_ma_{m-1} \dots a_1a_0) \rightarrow a_{r-1}a_{r-2} \dots a_{i+1}a_k \dots a_ma_{j-1} \dots a_{k+1}a_i \dots a_ja_{m-1} \dots a_1a_0$
 - Реверсування $R(a_{r-1}a_{r-2} \dots a_1a_0) \rightarrow a_0a_1 \dots a_{r-2}a_{r-1}$.

Із описаних перетворень в контексті найбільш цікавими є два – це порозрядне заміщення і тасування. Їх особливість полягає в тому, що вони є основою синтезу для двох топологій, а саме, гіперкуба та графу зсувів-вставок, він же топологія де Бруйна, названа на честь голандського математика Ніколаса Говерта де Бруйна (де Брейна, де Брюйна, англ. de Bruijn). Перший граф відомий своєю ідеальністю з точки зору топологічного трафіку, що передбачає мінімальну кількість зв'язків водночас із максимальним їх використанням, а також відсутність топологічно-зумовлених затримок. Особливістю другого є близькість до оптимального балансу з точки зору

проблематики (Δ , D) [237]. Тобто, з одного боку він передбачає ріст N як n^r , з іншого – статичну ступінь $S=2n$ при досить невеликому діаметрі $D=r$. Таким чином, забезпечується мінімальна ціна і простота масштабування, і в той же час досягається невисока відстань пересилок.

Синтез двох графів в бінарній системі проілюстровано на Рис. 3.1. В таблиці синтезу для графа де Бруїна темно-зеленим кольором позначені петлі (на графі не зображені), блакитним – кратні дуги.

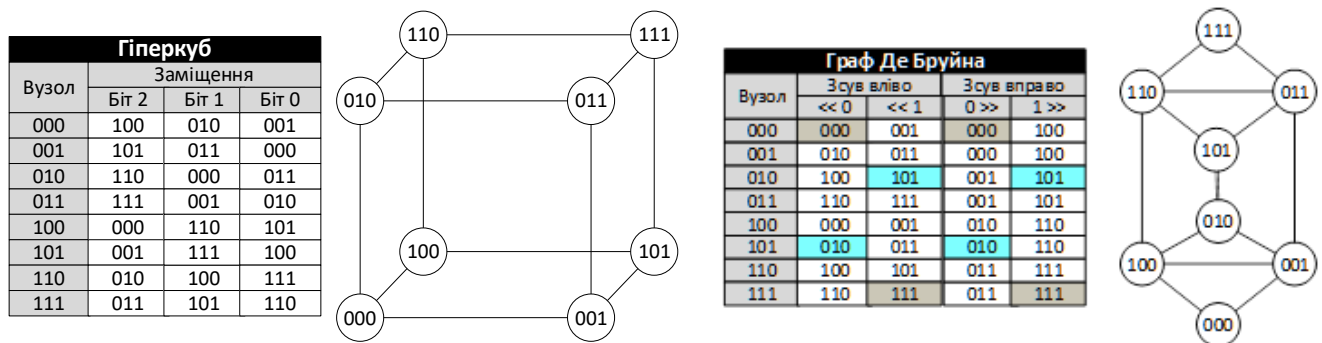


Рис. 3.1. Синтез топологій в бінарному коді

Важливою специфікою кодового синтезу є те, що маршрутизація може бути виконана за допомогою тих же перетворень, що використовувались при синтезі певного графа. Наслідком цього є можливість виконання маршрутизації за допомогою елементарних операцій – тобто, бітових інверсій та зсувів. На Рис. 3.2. наочно проілюстровано всі можливі варіанти (шляхи) маршрутизації між двома вершинами для цих топологій з використанням алгоритму на основі описаних перетворень.

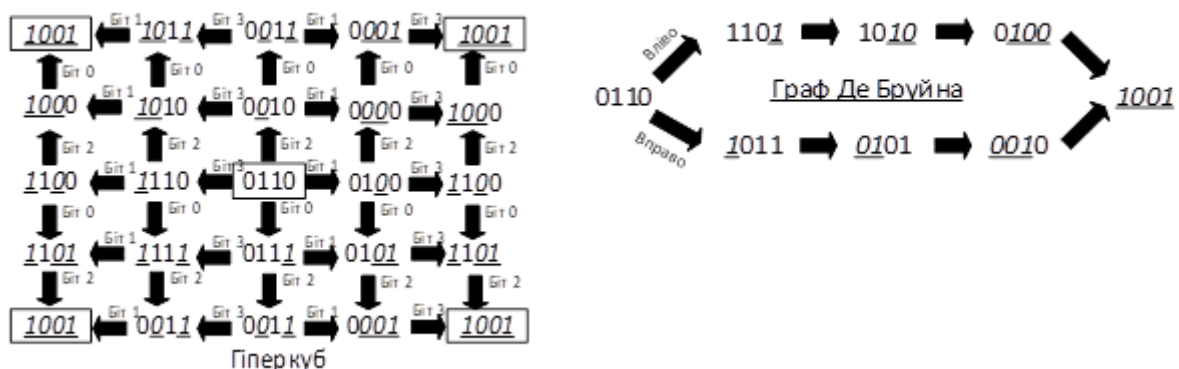


Рис. 3.2. Маршрутизація в гіперкубі та графі де Бруїна [3]

Що важливіше, оскільки в синтезі використовуються відразу декілька однотипних перетворень (побітові заміщення над кожним i -м бітом і 2 версії

тасування з різними напрямками та різними вставками) – це відразу ж закладає першу підвалину для відмовостійкості та багатоканальної передачі, що була описана в публікації [3].

3.2. Метод синтезу топологій на основі надлишкового коду

Модифікований метод синтезу на основі надлишкової системи числення передбачає схожу послідовність дій, проте додає нові кроки.

1. Обрання системи числення (СЧ) із множиною цифр (алфавітом) A , який має потужність k та основу числення b , причому для надлишкових СЧ $k > b$.

2. Обрання довжини коду (рангу топології) $r = \log_k(N_{desired} - 1) + 1$, де $N_{desired}$ – бажане число вузлів системи.

3. Формування множини кодів C , що має потужність $N = k^r$ і містить всі можливі r -розрядні коди $a_{r-1}a_{r-2} \dots a_1a_0$ обраної системи числення, та формування на її основі множини вершин графа $G(C, E)$.

4. Обрання бажаних кодових перетворень f_1, f_2, \dots, f_m в заданій системі числення.

5. Формування множини E ребер графа $G(C, E)$ за правилом $\forall i = 1 \dots m \ \& \ \forall c \in C: f_i(c) = c'_i, c'_i \in C \rightarrow \exists e_{c \leftrightarrow c'_i} \in E$.

6. Формування множини E' додаткових ребер графа, які дозволяють створити імпліцитні кластери в графі G , зв'язавши вершини з однаковими числовими значеннями.

3.2.1. Синтез на основі надлишкового бінарного представлення

RBR є однією із найпростіших і найбільш досліджених надлишкових систем, яка використовується для таких цілей як швидке додавання та множення. Тож першим кроком варто розглянути саме її використання в запропонованому методі. Дана система має основу 2, як і класична двійкова, проте на додачу до цифр 0 і 1 містить також цифру $\bar{1}$, що має від'ємне значення -1 і в кодовому записі може бути зручно позначена латинською літерою T.

При використанні RBR замість бінарного коду відбуваються деякі зміни. По-перше, в методі синтезу $n = k \neq b$ (де k позначає розмір алфавіту і дорівнює 3, b – основу числення, що складає 2). Відповідно, і приріст вершин, і структура кодування базуються на збалансованому тернарному алфавіті.

По-друге, змінюються перетворення, властиві для графів. Перетворення тасування включає в себе не 4, а 6 варіантів (в 2 напрями зі вставкою 3 цифр). Щодо заміщення, то воно ще більше ускладнюється: до очевидних варіантів додається можливість використання нетипових (наприклад, чистої інверсії, що зачіпає лише протилежні за знаком цифри, чи циклічного заміщення, де заміни відбуваються не на всі варіанти, а за конкретним правилом). Втім, навіть звичайне заміщення кожного i -го розряду на всі можливі альтернативи вносить свої особливості.

Як наслідок, структура топології набуває зовсім іншої форми. З одного боку, зберігаються всі зв'язки, що існували в не-надлишковій топології. З іншого, з'являються часткові повтори: надлишкова топологія включає в себе як мінімум 2 не-надлишкових версій себе самої, що частково перетинаються. Гарним прикладом є надлишковий гіперкуб, що внаслідок надлишковості набуває високу основу (рис. 3.3).

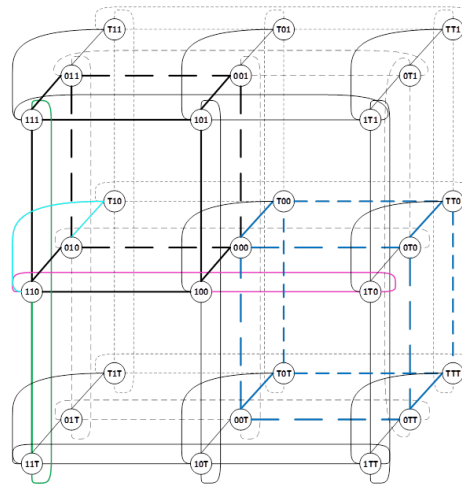


Рис. 3.3. Надлишковий гіперкуб (збільшена версія – на рис. Д.1)

Є декілька способів застосування цієї особливості. По-перше, висока основа породжує природний резерв: так, кожна площина такого гіперкубу є зарезервованою, причому кожна вершина має 3 резерви і кожен резерв захищає 2 вершини. Прикладом є зв'язки для вершини 110, яка резервується одночасно вершинами 11T, 1T0 та T10, що продемонстровано зеленими, фіолетовими та блакитними зв'язками.

По-друге, в топології існує ще один бінарний під-гіперкуб, що не задіюється в резервуванні основного і перетинається з ним лише у вершині 000 (позначений темно-синім кольором). Це дає можливість розподілити користувацькі задачі між цими частинами, уникаючи тим самим взаємних завад та великих черг виконання.

По-третє, ніщо не заважає використовувати резервні вершини такої топології в якості додаткових обчислювальних ресурсів, тим самим збільшуючи продуктивність ціною відмовостійкості. Причому в один момент часу додаткові вершини можуть слугувати резервами, в інший – використовуватись як основні обчислювачі, і це перемикання не потребує жодної реконфігурації.

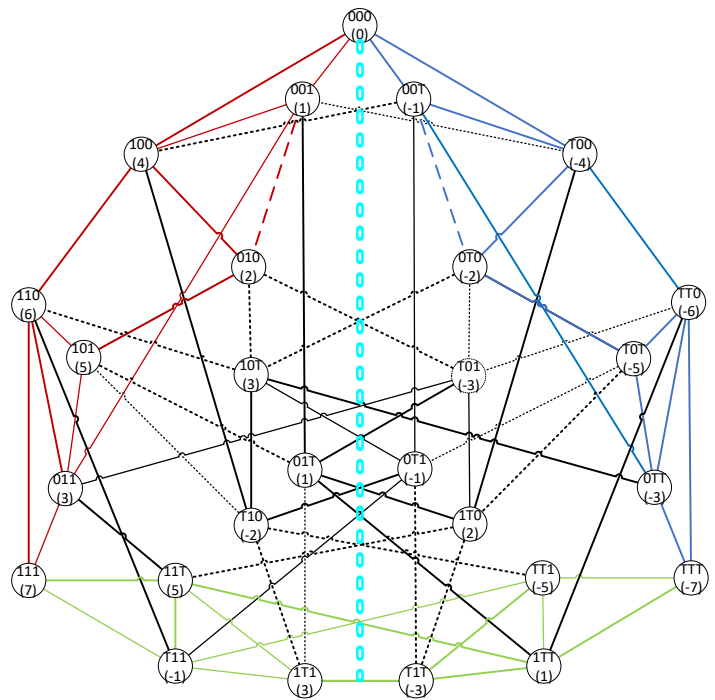
3.2.2. Надлишкова топологія де Бруйна

Те саме справедливо і для інших подібних топологій, таких як надлишковий граф де Бруйна [3]. На відміну від гіперкуба, чия ступінь з масштабуванням зростає, а в надлишковому варіанті – зростає дуже швидко, специфікою графа де Бруйна є незмінність ступеня з масштабуванням. З точки зору конструювання системи це означає, що для розширення не потрібно замінювати мережеве обладнання – достатньо просто модифікувати структуру мережі. Це дозволяє суттєво спростити подальшу підтримку і розвиток системи, побудованої на основі такої топології. Більш того, як показує досвід Fugaku, при ступені до 12 включно може бути використана безпосередньо-зв'язана мережа, що додатково здешевлює систему та прискорює взаємодію.

Рис. 3.4. демонструє метод синтезу надлишкового графа де Бруйна. Дана топологія включає в себе рівно 3 версії не-надлишкового графа де Бруйна, які виділені, відповідно, червоним, синім та зеленим кольорами.

Надлишковий граф Де Бруїна									
Вузол	Зсув вліво			Зсув вправо			Знач.		
	<< T	<< 0	<< 1	T >>	0 >>	1 >>			
TTT	TTT	TT0	TT1	TTT	0TT	1TT	-7		
TT0	T0T	T00	T01	TTT	0TT	1TT	-6		
TT1	T1T	T10	T11	TTT	0TT	1TT	-5		
T0T	0TT	0T0	0T1	TT0	0T0	1T0	-5		
T00	00T	000	001	TT0	0T0	1T0	-4		
T01	01T	010	011	TT0	0T0	1T0	-3		
T1T	1TT	1T0	1T1	TT1	0T1	1T1	-3		
T10	10T	100	101	TT1	0T1	1T1	-2		
T11	11T	110	111	TT1	0T1	1T1	-1		
0TT	TTT	TT0	TT1	T0T	00T	10T	-3		
0T0	T0T	T00	T01	T0T	00T	10T	-2		
0T1	T1T	T10	T11	T0T	00T	10T	-1		
00T	0TT	0T0	0T1	T00	000	100	-1		
000	00T	000	001	T00	000	100	0		
001	01T	010	011	T00	000	100	1		
01T	1TT	1T0	1T1	T01	001	101	1		
010	10T	100	101	T01	001	101	2		
011	11T	110	111	T01	001	101	3		
1TT	TTT	TT0	TT1	T1T	01T	11T	1		
1T0	T0T	T00	T01	T1T	01T	11T	2		
1T1	T1T	T10	T11	T1T	01T	11T	3		
10T	0TT	0T0	0T1	T10	010	110	3		
100	00T	000	001	T10	010	110	4		
101	01T	010	011	T10	010	110	5		
11T	1TT	1T0	1T1	T11	011	111	5		
110	10T	100	101	T11	011	111	6		
111	11T	110	111	T11	011	111	7		

а



б

Рис. 3.4. Надлишковий граф де Бруїна: (а) – метод синтезу та (б) – вигляд (форма «сапфір», збільшена версія – на рис. Д.3).

Ще одним фактом, як і у випадку гіперкуба, є повна симетричність топології відносно нуля. Що це дає? Загалом це значить, що для кожного шляху ($A \rightarrow B$) існує антишлях ($\bar{A} \rightarrow \bar{B}$), такий, що якщо вершина $C \in (A \rightarrow B)$ то вершина з оберненим до неї кодом $\bar{C} \in (\bar{A} \rightarrow \bar{B})$.

Цікавою особливістю такої топології є можливість її декомпозиції на 3 однакові дерева. Що це дає? Як було продемонстровано раніше, у порівнянні із гіперкубом граф зсувів та вставок надає значно менше можливостей для пошуку альтернативних шляхів. Стандартна елементарна маршрутизація дозволяє виявити рівно 2 таких маршрути, і це властиво всім видам дебруїнівських графів незалежно від кодування. Тобто, при виникненні відмови стандартна маршрутизація дозволяє виконати обхід лише однієї несправності.

На рис. 3.5 показано іншу форму тієї ж топології, що сформована із 3 різних дерев (показані червоним, зеленим та синім кольорами), побудованих через зсув вліво. Ключовою властивістю цих дерев є те, що множини їх нетермінальних вершин не

перетинаються. Таким чином, кожна вершина топології завжди знаходиться у листі двох дерев.

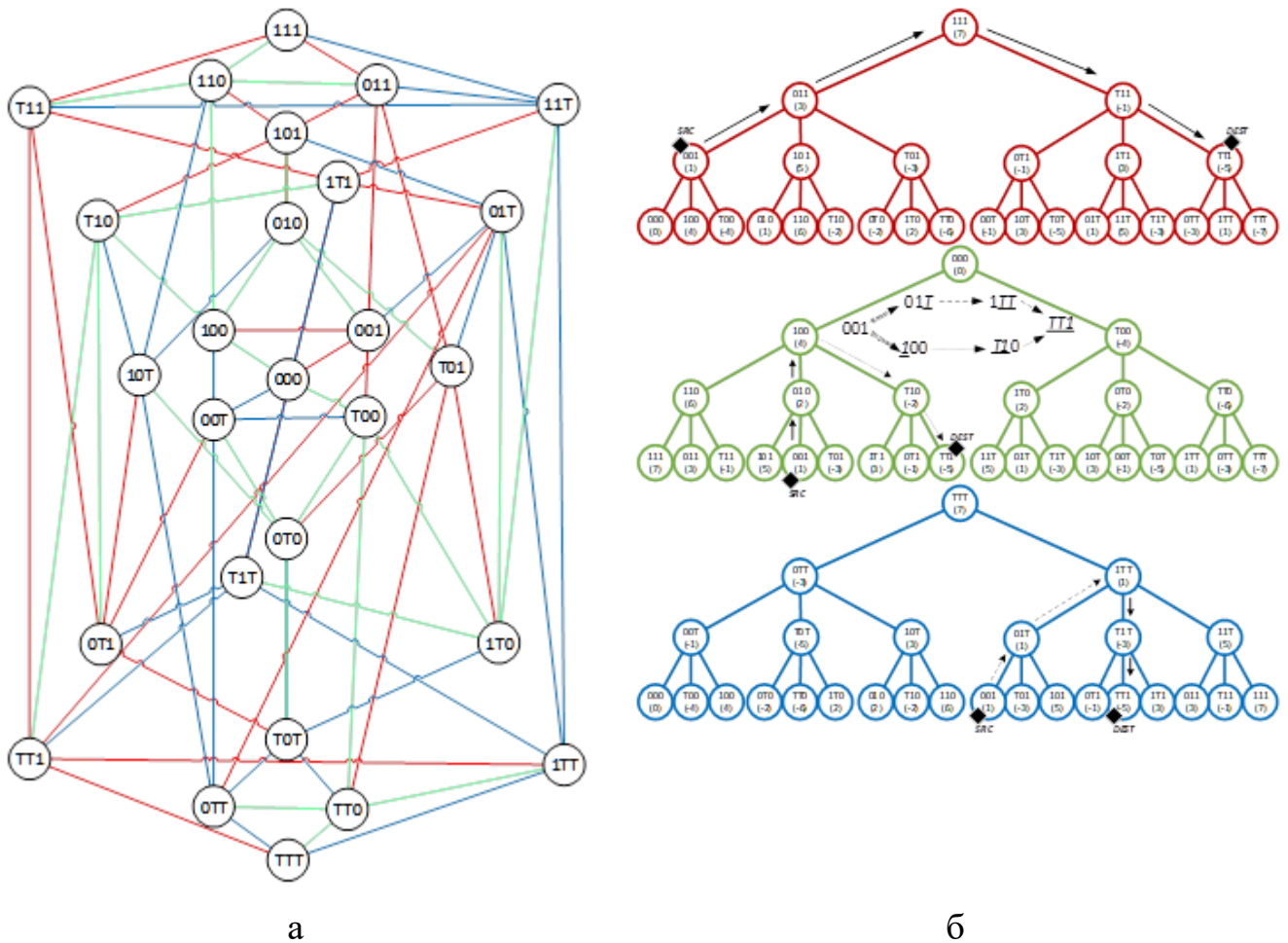


Рис. 3.5. Надлишковий граф де Бруйна. (а) – форма «смарагд» та (б) – декомпозиція на 3 дерева маршрутизації з прикладом (збільшена версія – на рис. Д.5).

Така властивість є цікавою з двох сторін. По-перше, це робить обхід відмови елементарною операцією: якщо десь на маршруті трапляється несправність, необхідно просто взяти маршрут з іншого дерева. Таким чином, для виходу системи із ладу необхідна поява як мінімум 3 відмов. Другий аспект не менш цікавий: наявність 3 дерев гарантує існування як мінімум 3 альтернативних маршрутів, які можуть бути використані для паралельної передачі даних. Крім того, як видно із представленого на рисунку прикладу, ці маршрути лише частково перетинаються із маршрутами, отриманими стандартним алгоритмом (перетини позначені штрихом та пунктиром відповідно для зсуву вправо та зсуву вліво). Це дозволяє використати їх

для розбиття чи агрегації трафіку, розподіляючи навантаження по мережі та мінімізуючи затримки [3].

3.2.3. Практичний сенс надлишкового представлення в графі де Бруйна

Втім, вищезазначені властивості характерні не лише надлишкової, а і будь-якій топології де Бруйна на основі небінарного кодування. Постає питання: чим надлишковий метод синтезу відрізняється від існуючого і чого дозволяє досягти?

Відповідь на це питання полягає у властивостях альтернативних представлень, характерних надлишковим кодам. Як було згадано раніше, передбачається, що ці представлення дозволять підвищити відмовостійкість. Проте для того, щоб це передбачення могло бути втілено в реальність, необхідні відповідні проміжні методи.

Основним тут є метод створення кластерів на основі однакового значення, який передбачає, що всі вершини з різними кодами та однаковими номерами є «однією сутністю». Звісно, це не означає, що ця сутність має бути буквально однією вершиною – мова йде радше про певний імпліцитний кластер – підмережу, невидиму на рівні загальної мережі. Питання реалізації такого кластеру є досить комплексним і буде розглянуто пізніше. До тих пір приймемо за факт, що для вершин з однаковими номерами існує невидимий зв'язок. Таким чином, вершини з однаковими номерами можуть бути віртуально склеєні і представлені як одна «логічна вершина». Для надлишкової топології де Бруйна це матиме вигляд, представлений на рис. 3.6.

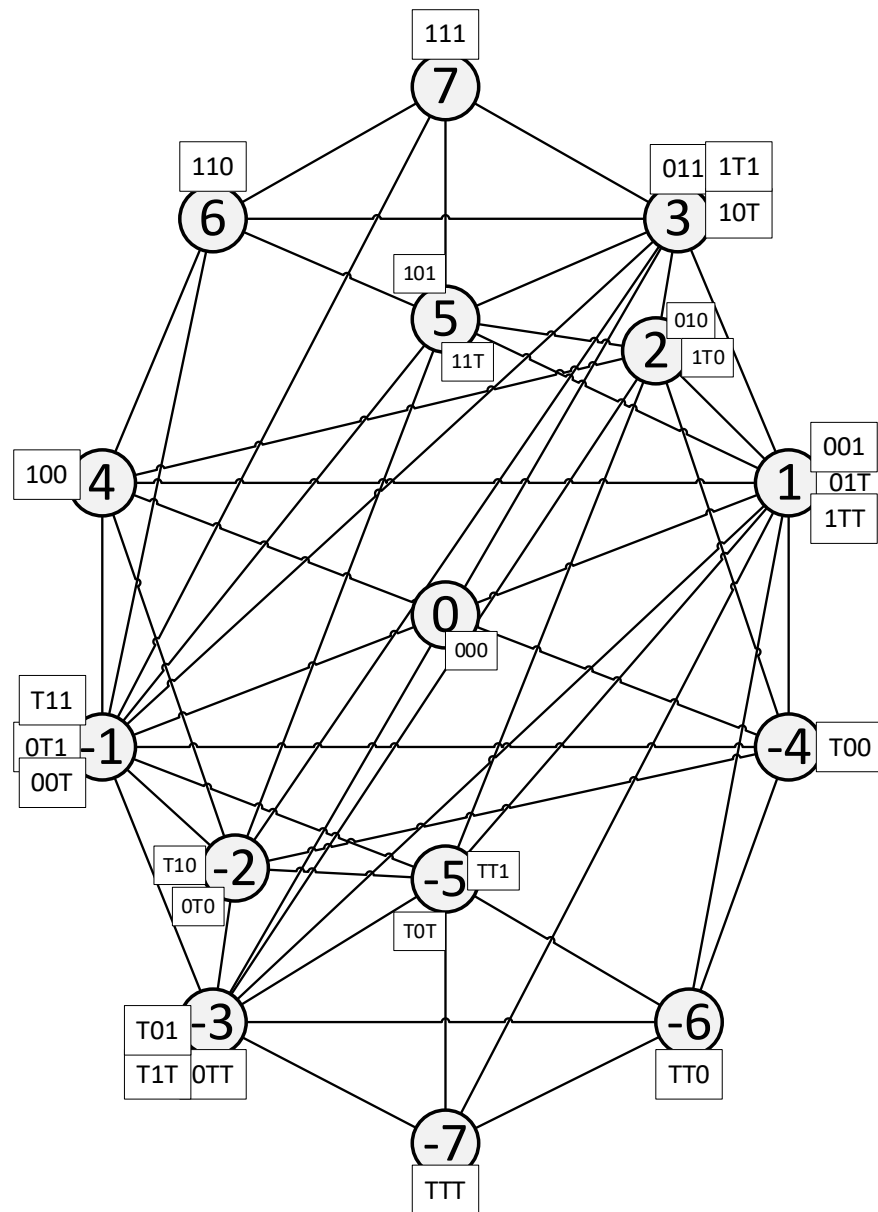


Рис. 3.6. Надлишкова топологія де Бруйна з імпліцитними кластерами. Форма «діамант» [4].

Для використання додаткових зв'язків пропонується метод маршрутизації, який полягає у тому, що на основі перетворень над кодом конкретної вершини можуть бути отримані всі її альтернативні представлення. Основою даного методу є таке поняття як шаблон – дворозрядна кодова підпоследовність, що може мати альтернативні представлення – такі шаблони далі називатимуться альтернаційними. В матеріалах конференції [8] було представлено 2 альтернаційні шаблони для надлишкової двійкової системи: 01-1Т (1) та 0Т-Т1 (-1). З ними пов'язано наступне твердження: будь-який код в надлишковій системі, що містить хоча б один альтернаційний шаблон

багатоканальної маршрутизації. По-третє, сам по собі перехід є досить непростим: хоча операція заміщення і відноситься до елементарних, але пошук всіх альтернативних представлень таким методом може бути занадто ємкою задачею.

Таким чином, існуючий метод потребує розвитку.

По-перше, шаблонне заміщення – це кодовий метод, а не деревний, тож немає сенсу розглядати поєднання цих методів без нагальної потреби. По-друге, обхід відмови – це операція, яка не потребує знаходження всіх альтпредставлень. Достатньо можливості шукати нові представлення при неможливості використати наявні.

Тож, опис розвинутого методу шаблонної маршрутизації варто почати із більш детального огляду класичної маршрутизації на зсувах. Її суть полягає у тому, що частини коду призначення поетапно вставляються в код джерела, таким чином, формуючи маршрут. Наприклад, для пересилки $10100 \rightarrow T1T10$ маршрут через зсув вправо виглядатиме так:

$$10100_0 \rightarrow \underline{0}1010_1 \rightarrow \underline{10}101_2 \rightarrow \underline{T1}010_3 \rightarrow \underline{1T}101_4 \rightarrow \underline{T1T}10_5$$

Підкресленням та курсивом тут виділені частини, що вставляються. Нижні індекси позначають номер вершини на маршруті. Таким чином, код будь-якої проміжної (1–4) вершини може бути чітко розділеним на 2 частини: від джерела та від призначення. Таким чином, наступні ситуації можуть бути виявлені.

Ситуація 1. Заміщення в частині джерела. Будь-який шаблон, замінений в частині джерела, не впливає на кінцеву точку маршруту. Це пов'язано із тим, що частина джерела (непідкреслена) поступово «виштовхується» з коду. Наприклад, нехай вершина 2 шляху є несправною. Тоді наступний альтернативний шлях може бути отримано (шаблон заміщення виділено сірим):

$$10100_0 \rightarrow \underline{0}1010_1 \gg \underline{011T}0_1^* \rightarrow \underline{1011T}_2 \rightarrow \underline{T1011}_3 \rightarrow \underline{1T101}_4 \rightarrow \underline{T1T10}_5$$

Як видно, шаблон повністю усувається зі шляху, тож виконання такого заміщення «від джерела» не потребує жодних додаткових дій.

Ситуація 2. Заміна в частині призначення. Будь-який шаблон, замінений в частині призначення, може бути «виправлений» зворотнім переходом в точці призначення. Аналогічно, нехай точкою відмови є вершина 4. Тоді:

$$10100_0 \rightarrow \underline{0}1010_1 \rightarrow \underline{10}101_2 \rightarrow \underline{T1010}_3 \gg \underline{0T010}_3^* \rightarrow \underline{10T01}_4^* \rightarrow \underline{T10T0}_5^* \gg \underline{T1T10}_5$$

Важливо розуміти, що зміна в частині призначення еквівалентна зміні точки призначення. Тож, якщо маршрутизація виконується від джерела, або ж якщо у вершині 1 стало відомо про відмову 3, є можливість скоротити число переходів і почати обхідний маневр ще у вершині 1. Цей «стрибок із випередженням» виглядатиме наступним чином:

$$10100_0 \rightarrow 01010_1 \rightarrow \underline{T}0101_{2*} \rightarrow \underline{0T}010_{3*} \rightarrow \underline{10T}01_{4*} \rightarrow \underline{T10T}0_{5*} >> \underline{T1T}10_5$$

Ситуація 3. Заміна між частинами. Будь-який шаблон, замінений в точці стику частин джерела та призначення, має бути «виправлений» зворотнім переходом за 1 крок до досягнення точки призначення. Наступний приклад ілюструє дану ситуацію (точкою відмови є вершина 2):

$$10100_0 \rightarrow \underline{01010}_1 >> \underline{1T}010_{1*} \rightarrow \underline{11T}01_{2*} \rightarrow \underline{T11T}0_{3*} \rightarrow \underline{1T11T}_{4*} >> \underline{1T10}1_4 \rightarrow \underline{T1T}10_5$$

Завдяки тому, що в описаних ситуаціях раніше пройдений маршрут зберігається, таке заміщення не потребує ні повторного пошуку маршрутів, ні ускладнення алгоритму: достатньо знати крок, на якому відбувся «стрибок», та позицію заміщення, щоб доставити повідомлення, використавши лише 1–2 додаткових хопи для обходу. Ситуація з багаторазовою «телепортацією» також вирішується досить просто: якщо мова йде про заміщення призначення чи стику, достатньо просто додати до заголовку інформацію про це і виконувати повернення в тому порядку, в якому були виконані заміни, зважаючи, що заміщення стику має бути оброблено за крок до точки призначення, а всі заміщення призначення можуть бути «розкриті» в самій точці призначення. Тут варто зазначити, що хоча гіпотетично замін в точці призначення може бути багато, проте, оскільки мова йде про одну і ту ж логічну вершину, зворотній перехід може бути значно простішим ніж це передбачається логікою маршрутизації.

3.2.4. Особливості методу синтезу на основі надлишкових кодів

Як було продемонстровано вище, метод топологічного синтезу на основі надлишкового бінарного коду є досить цікавим як з точки зору відмовостійкості, так і з точки зору ефективності. Загалом, наступні його переваги можуть бути виділені:

- Невисока ступінь, що дозволяє скоротити використання мережевого обладнання і зробити мережу безпосередньо-зв'язаною.

- Низький діаметр та висока швидкість масштабування порівняно із класичними топологіями для безпосередньо-зв'язаних мереж.
- Підтримка простих методів маршрутизації на основі кодових перетворень, що може давати кращу швидкодію і менші витрати пам'яті для алгоритму, ніж використання табличних методів.
- Підтримка маршрутизації на основі дерев, перевагою якої є швидка реакція на відмову / перевантаження, що дозволяє як підвищити відмовостійкість, так і скоротити час пересилки.
- Підтримка маршрутизації на основі шаблонів, що сумісна з елементарною маршрутизацією на основі кодів і передбачає максимально оперативне реагування як на відмову, так і на високий трафік, та дозволяє виконати обхід ціною всього лише 1–2 додаткових кроків.
- Створення імпліцитних кластерів в топології мережі, що дозволяє приховати реальну мережеву структуру і використати кластеризовані елементи в якості резерву, тим самим відновлюючи мережу при відмові, зберігаючи при цьому можливість окремої адресації таких елементів через код.

Як видно із означених переваг, метод є досить цікавим з точки зору поставленої задачі. З одного боку, він дозволяє досягти високої ефективності передачі даних, з іншої – надає високу відмовостійкість. Особливо цікавими тут є імпліцитні кластери, адже елементи «логічного вузла» не обов'язково мають виконувати лише одну задачу. Так, задачі можуть бути розподілені між елементами всередині логічного вузла, при цьому одні задачі можуть бути захищені гарячим резервом для уникнення помилок, інші – захищені програмно, щоб забезпечити максимальну продуктивність виконання, причому апаратно ролі елементів (основний чи резерв) можуть змінюватись.

Звісно, у методу є і недоліки. Ідея імпліцитних кластерів звучить гарно, проте не дослідженим є питання їх фактичної організації. Як саме вони мають бути організовані, щоб забезпечити високу відмовостійкість та ефективність? Розглянута в Розділі 2 теорія надлишкового коду пропонує реалізацію через багатовимірну форму

для кодів виду (tb, b) , проте такий код є досить специфічним. Таким чином, розглянута математична модель топологій на основі надлишкового коду потребує інтеграції в запропонований метод синтезу для його розвитку.

3.3. Розвиток методу синтезу на основі надлишкового коду

3.3.1. Створення імпліцитних кластерів на основі багатовимірної форми

На основі теорії надлишкового кодування можна запропонувати декілька шляхів розвитку існуючого методу. І ключовим із них є вирішення питання зв'язування вузлів із однаковим номером. Попередній метод не передбачає якогось універсального алгоритму виконання даної операції, проте гіпотеза 2.4 показує, що гіперкуб з основою t є оптимальною топологічною структурою для таких кластерів.

Втім, проблематичною передумовою даної гіпотези є особливий тип коду – так, дане рішення є справедливим для кодів виду (tb, b) . Таким чином, постає питання щодо формування кластерів для довільних кодів. І найпростішим способом вирішити дану проблему є штучне приведення коду до потрібного вигляду.

Як саме це можна зробити? Нехай дано деяку топологію G , отриману на основі системи числення з алфавітом A і основою b . Очевидно, що якщо дана система числення є надлишковою, то в ній існуватимуть вузли з однаковими номерами, що мають бути окремо згруповані в кластери Q_V . Тепер розглянемо інший граф G' , що отриманий на основі системи числення з основою b , проте з нормальним алфавітом A' , отриманим через розширення вихідного алфавіту A до розміру, що відповідає співвідношенню (tb, b) . Згідно до теореми про самоподібність, якщо $A \in A'$ то і $G \in G'$. При цьому, якщо розмір A' пропорційний b , то ідеальною формою для Q_V є гіперкуби з основою t .

Звісно, повноцінне підвищення порядку і перетворення бажаного графу G в розширений G' є складною і вкрай дорогою операцією, і надлишковість системи у цьому випадку на порядок перевищуватиме реальні потреби. Тож, для вирішення задачі пропонується віртуальне підвищення порядку з керованою надлишковістю.

В чому суть даного методу? Нехай дано довільний $Q_V \in G$. Очевидно, що рішення структурної проблеми для Q_V існує в G' , але не в G . Проте, якщо записати багатовимірну форму Q_V для G' , її елементи можна буде умовно розділити на 2 частини: ті, що одночасно належать і G і G' , і ті, що належать лише G' . З першими все очевидно: це основні вузли топології, що, з одного боку, мають зв'язок із загальною мережею, але в той же час поєднані і між собою. Щодо другого класу вузлів, то це резервні вузли: вони існують лише в рамках кластеру Q_V , але не мають власного зв'язку із загальним графом G . Технічно, для загальної мережі їх просто не існує, а їх код є недопустимим в рамках обраної системи кодування.

Тепер поговоримо про особливості цих резервних вузлів. По-перше, очевидно, їх кількість буде досить великою, оскільки вони доповнюють Q_V до розміру $(tb)^r$. По-друге, вони невидимі для загальної мережі і номінально не мають жодних зовнішніх зв'язків, а оскільки їх коди невидимі ззовні – їм не можна направлено призначити жодних задач. Таким чином, на дані вузли можуть бути покладені наступні функції:

- **Зв'язування.** Оскільки додаткові вузли доповнюють кластер до форми гіперкуба з надлишковою основою, це приводить до появи нових альтернативних шляхів всередині Q_V .
- **Чистий резерв.** Оскільки додаткові вузли існують лише всередині кластера, але не ззовні, вони можуть бути використані для заміщення відмови основних вузлів. При цьому код та зовнішні зв'язки мають бути передані резервному вузлу від основного.
- **Прискорення.** Оскільки додаткові вузли все ще є обчислювальними пристроями, вони можуть бути використані для обчислень. Це потребує додавання додаткового забезпечення, відповідального за планування, проте дозволить задіяти ці «неіснуючі» вузли для забезпечення безпечних, швидких та локалізованих обчислень, поклавши на основні вузли функції дистрибуції та агрегації трафіку.

Таким чином, можемо висунути кілька пропозицій щодо типів описаних додаткових вузлів:

- **Віртуальний міст.** Реалізується через присвоєння окремому вузлу декількох різних кодів. Є одним вузлом з точки зору обчислень, проте багатьма – з точки зору маршрутизації. Має реалізувати всі зв'язки, що топологічно належать його кодам.
- **Віртуальний вузол.** Реалізується у вигляді контейнеру. Відрізняється від мосту тим, що є окремою обчислювальною сутністю (може виконувати обчислення незалежно від інших віртуальних вузлів). Звісно, у такому випадку реальна продуктивність фізичного вузла розділяється між усіма логічними (віртуальними) вузлами, розгорнутими на ньому.
- **Ізольований фізичний вузол.** Є фізичним вузлом з усіма необхідними зв'язками всередині кластеру, проте без можливості комунікації із загальною системною мережею. Не може адресуватись ззовні, а отже, має отримувати завдання від інших компонентів всередині кластера – наприклад, від основних вузлів. Такого роду вузли можуть використовуватись для додаткового розпаралелювання обчислень на рівні кластеру, а також для забезпечення високої безпеки обчислень.
- **Резервний вузол.** Реалізує всі потрібні внутрішньокластерні зв'язки, а також має можливість доступатися до зовнішніх зв'язків одного чи кількох основних вузлів. Таким чином, при відмові одного із резервованих основних вузлів даний вузол може прийняти на себе його номер в зовнішній мережі і продовжити роботу від його імені, тим самим замаскувавши факт відмови.

Як видно із запропонованих типів, кожен наступний тип вузла розширює можливості попереднього. Так, в стані нормальної роботи основних вузлів резервний вузол цілком може виконувати роль ізольованого фізичного вузла, на якому, в свою чергу, можна розгорнути декілька екземплярів віртуальних вузлів, тим самим зменшивши апаратну складність кластера. В свою чергу віртуальний міст є повним аналогом віртуального вузла з тією різницею, що існує лише для підтримки маршрутизації в багатовимірній формі і не може брати на себе жодних обчислень. Це дозволяє уникнути беззмістовного розмноження обчислювальних сутностей виключно з метою забезпечення топологічної цілісності мережі.

3.3.2. Декомпозиція на основі самоподібності

Другим аспектом, що розширює класичний синтез на основі надлишкового коду, є декомпозиція загального графу на підграфи того ж типу і рангу, але для системи числення меншого порядку. Це має відразу декілька наслідків.

По-перше, в будь-якому надлишковому графі можна знайти підграфи меншого порядку. Таким чином, обчислювальні ресурси можуть бути за необхідності рівномірно розподілені між користувацькими завданнями, що в поєднанні із розкладом на дебруйнівські дерева та імпліцитні кластери-гіперкуби дає змогу надавати користувацьким задачам мережі, що мають стандартну (або близьку до стандартної) топологічну структуру.

По-друге, такого роду розклад може бути використаний для маршрутизації. Особливо яскраво це видно на топології де Бруйна, одним із методів маршрутизації для якої є розклад топології на дерева і пошук шляхів вже у цих деревах. Оскільки самоподібні декомпозиції можуть як перетинатись, так і не перетинатись в залежності від розміру алфавіту, це дозволяє (при перетині алфавітів не більш ніж одною цифрою) виділити в топології ряд незалежних дебруйнівських дерев.

Таким чином, необхідно більш докладно розглянути властивості такої декомпозиції. Для більшої наочності варто почати із прикладу. Розглянемо надлишкову топологію де Бруйна, отриману на основі надлишкового двійкового представлення. Як було показано на рис. 2.11 в Розділі 2, така мережа може бути розкладена на 3 класичні дебруйнівські мережі, а ті, в свою чергу, містять в собі по 2 дерева. Втім, розклад без залишку, показаний на рисунку, характерний лише мережі рангу 2. Починаючи із рангу 3 в мережі з'являтимуться вузли, що не відносяться до жодної підмережі.

Оскільки підмережі включатимуть в себе всі вершини, що містять 2 види цифр у своєму коді (а вершини з цифрами одного типу завжди є коренями дерев), не відноситимуться до підмереж лише ті вершини, чий код містить в собі всі 3 види цифр в тій чи іншій послідовності. Так, для топології рангу 3 можна назвати 6 таких кодів, причому вони можуть бути згруповані в 2 кільця, сформовані через циклічний зсув послідовностей 01Т та 0Т1 відповідно. Загалом, довжина кільця завжди буде рівною

рангу топології, що декомпонується. На рис. 3.8 представлено декомпозицію для топології рангу 3.

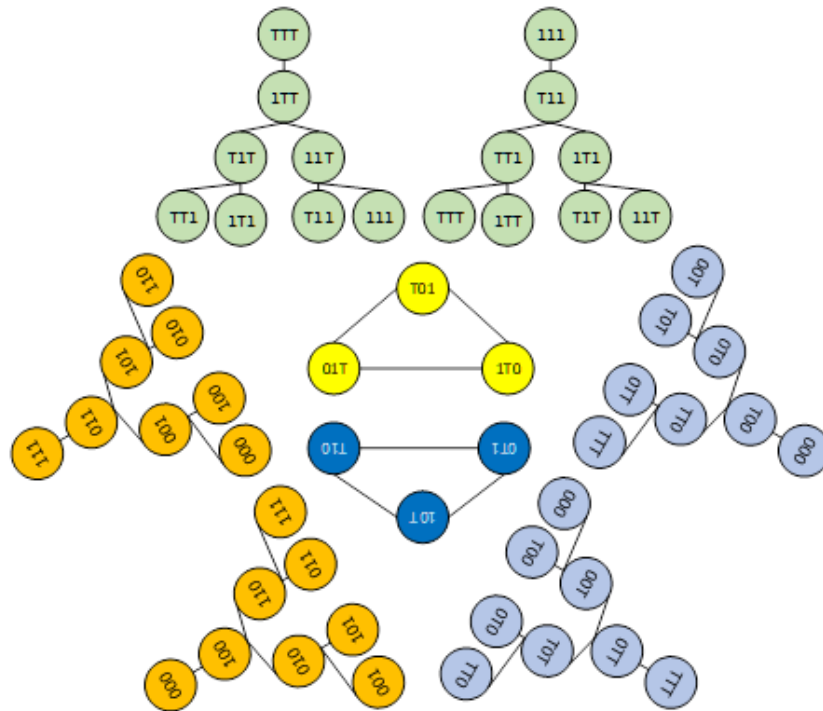


Рис. 3.8. Декомпозиція топології надлишкового де Бруїна рангу 3

Метод маршрутизації з використанням декомпозиції [9] передбачає наступне. Мережа поділяється на підмережі дебруїнівської та кільцевої структури, і кожній вершині у відповідність ставиться множина виходів до інших підмереж. Подальша логіка дуже проста: алгоритм маршрутизації в кожній підмережі знає лише її структуру і зв'язки з іншими підмережами. Коли потрібно передати повідомлення за межі підмережі – необхідно знайти вихід до неї, а якщо його нема – то виконати спробу передачі через проміжну підмережу, віддаючи перевагу підмережам іншого типу.

Сам алгоритм маршрутизації в рамках підмережі при цьому не регулюється: це може бути як будь-який топологічно-залежний алгоритм, так і більш складний, наприклад, адаптивна маршрутизація. Завдяки розбиттю на підмережі поле пошуку рішень та обсяг необхідної інформації суттєво зменшується, що дозволяє ефективно використати переваги адаптивного алгоритму, мінімізувавши його недоліки.

Висновок до розділу 3

В розділі представлено удосконалений метод синтезу відмовостійких топологій на основі надлишкового коду, що відрізняється від існуючого використанням надлишкової системи числення для кодування номерів вузлів і дозволяє отримати топології, що мають кращі (порівняно із класичними графами на основі двійкового коду) характеристики ступеня та діаметру, а також містять імпліцитні кластери, які складаються із вершин, коди яких є альтернативними представленнями одного числа.

Представлено спосіб формування імпліцитних кластерів на основі багатовимірної форми, що дозволяє в графі на основі будь-якого надлишкового коду сформувати імпліцитні кластери і тим самим забезпечити високу відмовостійкість топології.

За допомогою удосконаленого методу синтезовано топологію надлишкового гіперкуба, що базується на перетвореннях заміщення в надлишкових кодах. Даний граф є еквівалентним гіперкубу з надлишковою основою, проте дозволяє сформувати в графі імпліцитні кластери, а також забезпечити просту маршрутизацію на основі кодових перетворень.

За допомогою удосконаленого методу синтезовано топологію надлишкового графа де Бруйна (де Брейна, de Bruijn), що базується на зсувах та вставках в надлишкових кодах. Даний граф має незмінну з масштабуванням ступінь і діаметр, що еквівалентний діаметру аналогічного надлишкового гіперкуба. Особливістю графа є можливість його декомпозиції на дерева (т.зв. дерева де Бруйна).

Представлено алгоритм маршрутизації на основі кодових перетворень, що дозволяє використати перетворення синтезу для пошуку маршрутів у графі та обходу невеликого числа відмов.

Представлено алгоритм маршрутизації на основі дерев де Бруйна, який дозволяє виконати обхід кількості відмов, що на 1 менше ніж порядок алфавіту k СЧ, використаної для синтезу графа.

Представлено алгоритм маршрутизації на основі шаблонних переходів, що дозволяє використати імпліцитні кластери для обходу відмов, розглянуті особливості такого шаблонного переходу.

РОЗДІЛ 4.

МЕТОД МАСШТАБУВАННЯ ІЄРАРХІЧНИХ ТОПОЛОГІЙ

4.1 Опис методу масштабування ієрархічних топологій.

Метод синтезу, розглянутий в Розділі 3, має велику кількість переваг, втім його проблема також є досить вагомою. Звісно, існують методи для нівелювання такого впливу – наприклад, різноманітні універсальні засоби, які, у поєднанні із топологічно-орієнтованими методами та декомпозицією дозволять обійти проблему. Втім, є сенс розглянути і альтернативний підхід – що як даної проблеми можна уникнути, використовуючи інші методи синтезу? Чи може це дати краще рішення і чи можуть ці рішення бути поєднані для отримання кумулятивного ефекту?

Найбільш перспективним в даному контексті є метод топологічної інтеграції, який передбачає поєднання декількох графів за певним правилом. Методів інтеграції існує багато: заміщення вершин кластерами, декартовий добуток, додавання зв'язків, тощо. Оскільки вихідними даними в цьому методі є не коди та перетворення, а вже готові графи, це дає відразу декілька переваг. По-перше, знаючи переваги того чи іншого графу, завжди можна сконструювати мережу так, щоб ці переваги максимізувати. По-друге, використовуючи в якості основи популярні топології, такі як жирне дерево, гіперкуб чи dragonfly, можна отримати гарну сумісність розроблених рішень із уже існуючими.

По-третє, метод не накладає жодного обмеження на тип чи структуру графів, що інтегруються. Так, це може бути граф, отриманий іншим методом синтезу, включаючи надлишкові топології із Розділу 3. При цьому їх основна структура виявиться прихованою всередині загальної мережі, що дозволить ввести ще один рівень абстрагування. Завдяки цьому користувачеві не потрібно буде працювати із надлишковими кодами взагалі – для нього це буде звичайна мережа, що, безумовно, суттєво спростить програмування та підтримку системи.

Таким чином, переваги різних графів можуть бути максимізовані, а недоліки – нівельовані.

Формальний опис запропонованого методу передбачає наступні кроки:

1. Вибір графа G_c , що є неієрархічним графом (базовим кластером) і потребує масштабування.
2. Вибір графа G_T , що описує структуру ієрархії вищого / вищих порядків.
3. Формування ієрархії у відповідності до обраного типу ієрархії та способу масштабування.
4. Формування додаткових зв'язків (за необхідності) у відповідності до правил топологічної інтеграції

4.2. Синтез графів зі вкладеною ієрархічністю

Під ієрархічністю такого роду розумітимемо ситуацію, коли в загальній топології КС є як мінімум 2 рівні: найнижчий рівень, він же (базовий) кластер або група, і рівні вищого порядку, що складаються із ряду графів нижчого рівня, об'єднаних певним чином. Так, рівень 2 складається із базових кластерів, рівень 3 – із графів рівня 2 і т.д.

Типовим прикладом класичної ієрархічної топології можна вважати dragonfly.

4.2.1. Синтез на основі декартового добутку

Одним із найпростіших методів створення багаторівневого графу є використання декартового добутку. При цьому один граф «вставляється» в інший, породжуючи кластери. У випадку топологій на основі надлишкового коду доцільним рішенням є використовувати надлишковість на рівні кластеру, «приховавши» її всередині топології з класичним кодуванням вершин [7].

Такого роду структура має важливу перевагу: абстрагування маршрутизації. Так, в мережі існує 2 алгоритми маршрутизації, родин з яких відноситься до мережі, інший – до кластеру. На рівні загальної системної топології мережа є класичною і реалізує звичайні протоколи маршрутизації. В той же час на рівні кластеру мережа є надлишковою і, потенційно, реалізує імпліцитні кластери, породжуючи «кротові нори» в мережі – тобто, приховані зв'язки, що долають межі кластеру і дозволяють за необхідності досягнути до віддаленого сегменту системи, тим самим обходячи відмови на звичайному шляху. Алгоритм маршрутизації такої мережі має стійкість до відмов, що також дозволяє підвищити параметри відмовостійкості.

Прикладом такого роду топології є Hyper excess De Bruijn [8], що є поєднанням гіперкубічної на надлишкової де Бруйнівської мережі. Розглянутими є 2 види системи: з зовнішньою класичною топологією, що дозволяє абстрагувати надлишковість в кластерах, і з зовнішньою надлишковою топологією – для розширення самих кластерів, укрупнення обчислювальних груп та агрегації обчислювальних вузлів.

4.2.2. Синтез на основі рекурентних вкладень

Відомо, що проблема (Δ, D) опирається на 3 характеристики, а саме ступінь, діаметр та число вершин. Баланс цих параметрів визначає і питання вартості, і відмовостійкість, і пропускну здатність. Зменшення S і D при сталому N є нетривіальною задачею: оптимізація одного параметру відразу ж веде до погіршення іншого. Втім, якщо контролювати один із параметрів і зробити темпи зростання N дуже високими, є шанс на певному кроці r отримати топологію, яка за SD -характеристикою буде вкрай близькою до оптимальності [6].

Одними із найшвидших в масштабуванні є методи кодового синтезу, які дають швидкість росту k^r , де k – розмір алфавіту. При цьому такі топології можуть забезпечувати, проте не гарантують самоподібності, тобто, наявності в топології підграфів того ж типу, але меншого порядку або рангу. Як було продемонстровано раніше, у Розділі 2, ця властивість є досить цінною, оскільки дає і ефективну відмовостійку маршрутизацію, і можливість більш тонкого управління мережею.

Тож, як забезпечити і самоподібність, і високий ріст N ? Відповідь на це питання лежить в специфічному методі синтезу. Що як топологія рангу r буквально складатиметься із кластерів рангу $r-1$, ті, в свою чергу, міститимуть підграфи рангу $r-2$ і так далі?! Звісно, тут на заваді стають характеристики SD , оскільки таке масштабування призведе або до стрімкого росту ступеня, або – до суттєвого збільшення діаметра. Втім, і ця проблема має вирішення: псевдоповнозв'язні топології, такі як Dragonfly, мають невисокий діаметр і прийнятну ступінь. Це робить їх гарним вибором для формування взаємозв'язків між різними рангами.

Формальний опис даного способу синтезу виглядає наступним чином:

1. Дано G_c довільної форми з числом вузлів N_1 , номери вузлів якого представлені кодами довжини p деякої СЧ
2. Наступний рівень G_2 містить N_1 графів G_c , коди вершин якого можуть представлятись як $b.a$, де $a, b \in C$ і a є номером вузла в кластері G_c , а b є номером кластера в G_2 , $|a| = |b| = p$.
3. Зовнішній зв'язок формується за принципом: $\forall b.a, a \neq b$ та $a, b \in C: f(b.a) = a.b$.
4. Для графа рівня ієрархії r граф G_r містить N_{r-1} графів G_{r-1} , і кожен вузол G_r може бути закодований як $b.a$, $|a| = |b| = 2^{r-1}p$. При цьому перетворення f залишається незмінним для будь-якого r .

4.2.3. Простий рекурентний граф

Запропонований метод [1, 6] може бути описано наступним чином. Нехай є деякий вихідний кластер, вершини якого кодуються будь-яким способом. Найпростіший варіант – гіперкуб порядку 1 (лінія із 2 вузлів, 0 та 1), який ми розглянемо в якості прикладу. Це – топологія рангу 1. Топологія рангу 2 включає в себе рівно 2 кластери рангу 1, які у випадку гіперкуба кодуватимуться, відповідно, 00, 01, 10 та 11. Як сформувати зв'язки між ними? Очевидно, 00 зв'язаний з 01, а 10 – з 11, оскільки це відповідає зв'язкам рангу 1. Щодо зв'язків рангу 2, вони формуються наступним чином: розділимо код на 2 частини, a і b , які презентують, відповідно, номер кластера на номер вершини у кластері. Тоді, для кожної вершини $V = a.b$ існуватиме єдиний зовнішній зв'язок із вершиною $V^* = b.a$. Виключенням є лише вершина $a.a$, що містить петлю і, таким чином, не має зовнішнього зв'язку.

Цей же метод справедливий і для рангу 3, де повний код вершини кодуватиметься чотирьохбітовою послідовністю, яка розділятиметься на 2 двобітові. А кластерів, відповідно, буде 4 – як і вершин в кожному з них. Приклад топології третього рангу продемонстровано на рис. 4.1.

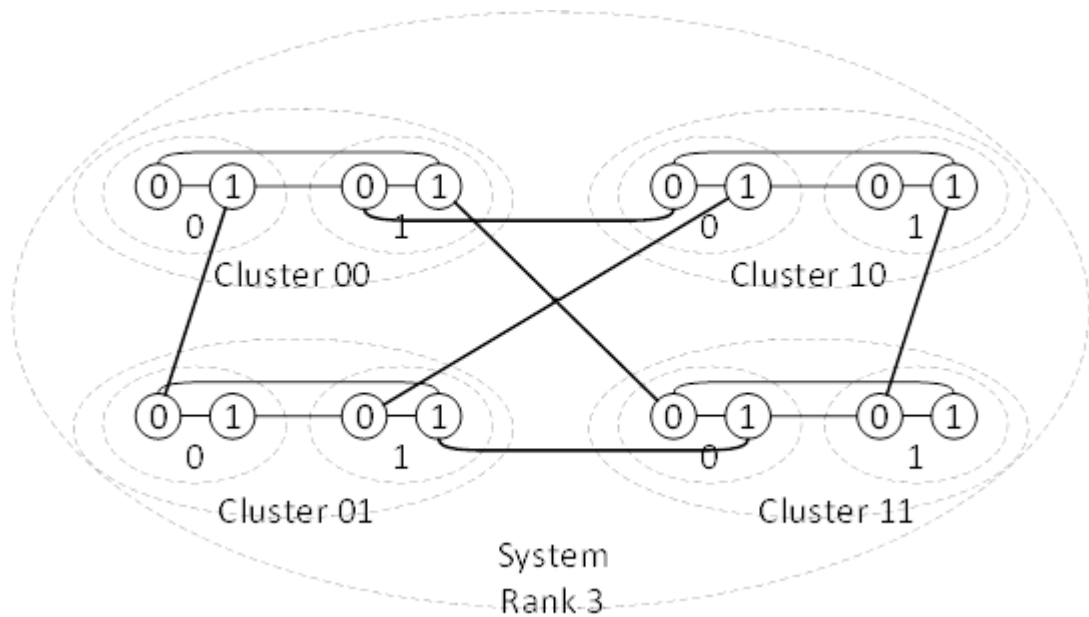


Рис. 4.1. Простий рекурентний граф рангу 3 [6]

Що це дає? З кожним кроком довжина коду подвоюється. Ступінь – зростає рівно на 1. Єдиною проблемою є діаметр. Зовнішні зв'язки кожного рангу r є повнозв'язними, тож із кожного кластера можна досягти будь-якого іншого, але діаметр кластерів не є одиничним: він відповідає топології рангу $r-1$, яка також передбачає 2 маршрутизації всередині кластерів і рівно 1 хоп до цільового кластеру.

Позначимо параметри вихідного графу як N_l , S_l та D_l . Тоді, для характеристик топології деякого рангу r справедливі наступні формули [1, 6]:

$$N_r = N_{r-1}^2 = N_1^{2^r} \quad (4.1)$$

$$S_r = S_{r-1} + 1 = S_1 + r - 1 \quad (4.2)$$

$$D_r = 2D_{r-1} + 1 = 2^{r-1}(D_1 + 1) - 1 \quad (4.3)$$

Таким чином, як і у випадку більшості класичних топологій деякі властивості рекурентного графу можуть бути розраховані теоретично, без необхідності синтезу та симуляції. Звісно, щодо формул вище постає питання: це гарні характеристики чи ні? Властивості рекурентного графу, як було показано вище, залежать від двох параметрів: графа, що лежить в основі, та коду, який використовується для репрезентації вершин. Тож, базуючись на тому, що бінарний гіперкуб та простий рекурентний граф мають спільний «базовий кластер» і спільне кодування, можемо

виконати розрахунки і порівняти отриманий результат (характеристика SD в табл. 4.1, повна версія – табл. Д.3).

Крім того, є сенс додати до порівняння dragonfly-подібну ситуацію, коли в основі лежить повнозв'язний кластер. Також, оскільки на даний момент не існує суперкомп'ютерів із сотнями мільйонів вузлів, є сенс позначити (табл. Д.3) зеленим ті топологічні ранги, що відповідають сучасним потребам людства, жовтим – ті, що гіпотетично можуть бути корисними, а червоним – ті, що є суто теоретичними і не мають практичного сенсу.

Табл. 4.1

Порівняння характеристики SD рекурентного графа та гіперкуба

Число бітів коду	2	4	8	16
Число вершин N	4	16	256	65 536
Гіперкуб	4	16	64	256
Рекурентний граф	6	21	60	155
Рекурентний граф на основі повнозв'язного кластеру з 4 вершин	3	12	35	90

Як видно із представлених даних, запропонований метод забезпечує досить непогані характеристики SD . При цьому маршрутизація в такій мережі також не є складною і базується на звичайній перестановці бітів коду. Немає складнощів і з відмовостійкістю пересилок: оскільки в основі зв'язків лежить топологія Dragonfly, всі методи, придатні для неї, можуть бути використані і в рекурентному графі.

4.2.4. Насичення топології

Хоча запропонований метод і є досить потужним, проте він все ще має ряд проблем, без вирішення яких практична реалізація такої системи не буде доцільною. І першою в цьому списку є топологічна проблема – недовикористання зовнішніх зв'язків [1]. В кожному кластері на кожному кроці масштабування з'являється рівно 1 вершина, чий зовнішній зв'язок замикається в петлю. Коли така вершина одна – це не страшно, проте для кожного рангу $r \in N_{r-1}$ кластерів і рівно стільки ж «висячих» вершин, в яких, в свою чергу, є по N_{r-2} таких вершини на $r-1$ рангу масштабування.

Загалом же, для рангу r число петель на всіх рангах можна отримати із наступної формули [1, 6]:

$$L_r = \sum_{i=2}^r N_i N_{i-1} = \sum_{i=2}^r N_{i-1}^3 \quad (4.4)$$

Вирішити цю проблему можна досить просто: оскільки на кожному рангу є рівно N_{r-1} висячих вершин, по 1 на кластер, необхідно просто додати «резервний» кластер на цьому рівні. Таким чином, для будь-якої вершини з кодом виду $a.a$ існуватиме один зовнішній зв'язок до вершини $X.a$, де X позначає «нenumерований» кластер.

Окрім очевидного виграшу з точки зору проблематики (Δ, D) , це також створює підвалини для кращої відмовостійкості, адже кластер X може розглядатись і як резерв. Відповідно, завдяки цьому з'являється можливість відновити структуру топології у випадку відмови. На рис. 4.2 продемонстровано насичену топологію рангу 3, де насичення було виконано двічі – на рангу 2 та рангу 3 відповідно.

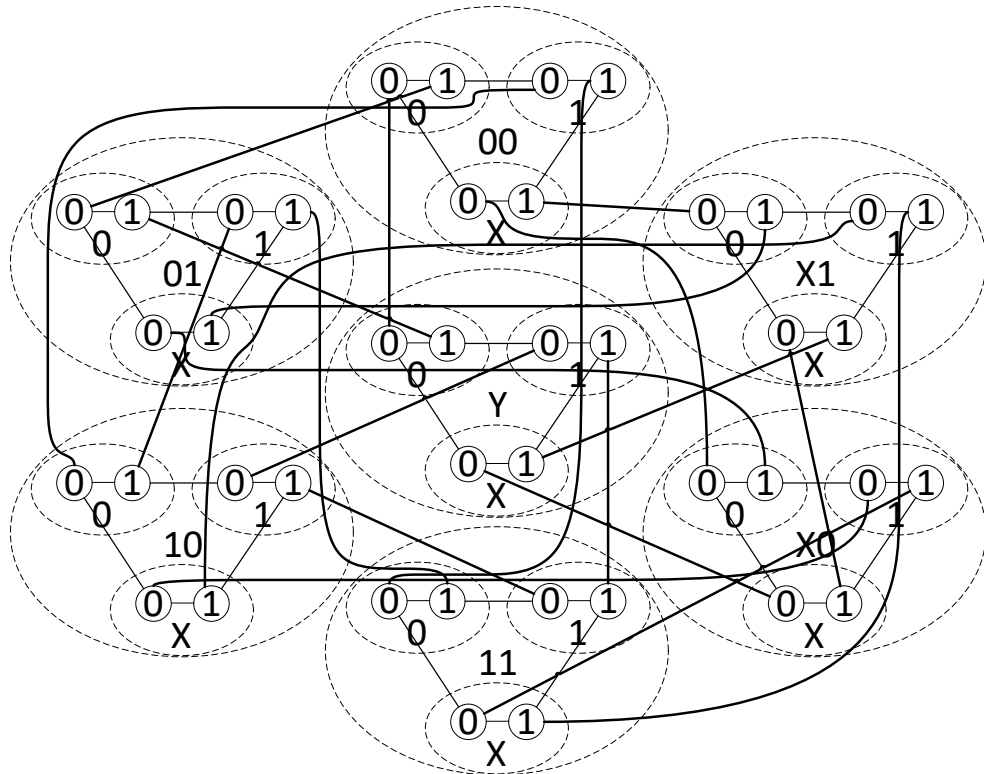


Рис. 4.2. Насичений рекурентний граф [1]

Втім, у цього методу є і певні проблеми. При насиченні на одному рівні все добре, проте при подальшому масштабуванні з'являються «напівкодовані» кластери, які ускладнюють маршрутизацію. Елементарним рішенням цієї проблеми є відхід від чистого кодового представлення на користь псевдо-кодового. Нехай ідентифікатор кожного рангу є додатнім цілим числом – псевдоцифрою, яка має власну основу. Перший (молодший) ідентифікатор закодований двійковим кодом і має три кластери. Другий – закодований трійковим кодом 012 і має 7 кластерів. Відповідно, третій ранг кодуватиметься семірковою системою, четвертий – СЧ з основою 43 і так далі.

Таким чином, маршрутизація в топології теж ускладнюється. Якщо раніше вона полягала в простій перестановці частин коду, то тепер алгоритм у псевдокоді виглядатиме наступним чином:

```

route (Src =  $s_{r-1}s_{r-2} \dots s_0$ , Dest =  $d_{r-1}d_{r-2} \dots d_0$ ,  $k = r$ ):
    if  $s_k = d_k$ :
        route (Src, Dest,  $k-1$ )
        return
     $V[r-1: k+1] \leftarrow d_i$ 
     $V[k] = s_k$ 
     $V[k-1: 0] \leftarrow \text{decompose } d_k = v_{k-1}v_{k-2} \dots v_0$ 
    # некоректне значення означає, що  $d_k \in$  позначенням некованого кластеру рангу  $k$ 
    if not is_correct ( $V[k-1: 0]$ ):
         $V[k-1: 0] \leftarrow \text{decompose } s_k$  # шукаємо маршрут до вершини виду  $a.a$ 
    route (Src, V,  $k-1$ ) # виконуємо внутрішню маршрутизацію від джерела
    # виконуємо стрибок по зовнішньому зв'язку
     $\text{jump } (V, k) \gg W = d_{r-1}d_{r-2} \dots d_{k+1}d_k w_{k-1}w_{k-2} \dots w_0$ 
    route (W, Dest,  $k-1$ ) # виконуємо внутрішню маршрутизацію до призначення

```

Складність тут полягає в операції *decompose*, алгоритм якої є наступним:

```

decompose (value, b, k):
    for  $i = k-1 \dots 0$ :
         $\text{code}[i] \leftarrow \text{value} \bmod b[i]$ 
         $\text{value} \leftarrow \text{value} \div b[i]$ 
    return code

```

Цей елементарний алгоритм має суттєвий недолік: багатократне виконання операцій ділення та отримання модуля. Подібні операції є досить тривалими, як наслідок, швидкість виконання такого алгоритму буде невисокою.

Таким чином, практичний сенс має лише насичення топології старшого рангу, адже у такому разі не буде зміни кодування, а стандартна маршрутизація через перестановки ускладниться лише однією додатковою умовою: якщо вершина призначення знаходиться в кластері X , то проміжною вершиною має бути $s_h \cdot s_h$, де s_h , відповідно, старша частина коду вершини джерела.

4.2.5. Редукція топології

Другою проблемою, що має бути вирішена для запропонованого методу, є головна його особливість – надшвидкий ріст [1]. Це є головною перевагою топології як графу, проте в той же час є її недоліком як топологічної організації. При проектуванні системи завжди є обмеження, пов'язані із фінансовим аспектом, тож топологія має бути не лише потужною, а і гнучкою, подібно до раніше згаданих Fat Tree та Dragonfly. Таким чином, виникає задача редукції топології, тобто, зниження числа вершин без втрати топологічних характеристик (а в ідеалі – з деяким їх покращенням). Три її рішення це пряма редукція, редукція усіченням та редукція розрідженням [1].

Пряма редукція полягає в простому факті: якщо змінити зовнішню топологію то і метод масштабування зміниться. Так, якщо бажаним числом вузлів системи є N , один із найпростіших способів цього досягти – це виконати наступні дії:

1. Взяти рекурентний граф із $M < N$ вузлами, такими, що $N \bmod M \rightarrow 0$
2. Обрати топологію із кількістю вузлів $M' = \frac{N}{M}$, таку, що задовольняє вимогам щодо топологічних характеристик та масштабування
3. Провести декартове множення рекурентного графа і обраної топології

Інші підходи відрізняються тим, що число вершин M вихідного рекурентного графа передбачається більшим, ніж бажане N . Причому різниця може бути дуже суттєвою: наприклад, розробляється система на приблизно 256 000 вузлів, а найближчим підходящим $M > N$ для рекурентного графу є 4 мільярди.

Усічення базується на тому, що розрядність кодування «обрізається» в старшій частині. Так, 32 розряди кодують 4 мільярди вершин, проте для 256 000 достатньо 18 бітів, з яких 16 є кодом всередині ієрархічного кластера, 2 – кодують номер кластера. При цьому зовнішні зв'язки не зникають, а «зациклюються» на існуючі кластери за таким принципом, що лише молодші розряди приймають участь у обміні. Тобто, для прикладу вище вершина з кодом $a_1a_0.b_{15}...b_2b_1b_0$ має одного зовнішнього сусіда – $b_1b_0.b_{15}...b_2a_1a_0$. Це робить зовнішні зв'язки щільнішими і, як наслідок, зменшує діаметр, адже для переходу між кластерами найвищого рівня достатньо відшукати найближчу вершину із потрібним номером в молодшій частині. Як правило, це потребує маршрутизації лише на одному чи двох молодших рівнях ієрархічного кластера, що еквівалентно 3 крокам для системи на основі повнозв'язного чотирьохвершинного кластера. При цьому діаметр ієрархічного кластера є рівним 15. Відповідно, таким чином повний діаметр системи знижується із 31 до 18, а SD стає рівним 126. В порівнянні, для еквівалентного гіперкуба цей показник дорівнює 324.

Розрідження також є досить простим: ним передбачається, що деякі кластери «вилучаються» із топології разом із їх зовнішніми зв'язками. Це не дозволяє покращити характеристики ціною зменшення числа вершин, проте має іншу перевагу: через специфіку маршрутизації не важливо, які саме номери були вилучені – алгоритм ніколи не стикнеться із ситуацією пересилки в неіснуючий кластер. Це означає, що немає необхідності якось змінювати маршрутизацію чи запам'ятовувати «заборонені» номери.

4.2.6. Надлишковий рекурентний синтез. DDB топологія

Звісно, коли мова йде про синтез на основі кодів, завжди є місце питанню: а чи можна змінити код і за рахунок цього отримати ще якісь вигоди? Очевидним кандидатом є, звісно, надлишкові коди, за допомогою яких можна отримати вигравш у відмовостійкості.

Як було продемонстровано вище, кодовою основою рекурентний синтезу є базовий кластер. Кодування його вершин «розповсюджується» в процесі синтезу і стає основою для маршрутизації. Таким чином, взявши в якості кластера надлишкову топологію, можна без жодних складнощів отримати надлишковий рекурентний граф.

дерезовидна мережа може як містити обмежене число рівнів (і масштабуватись вшир), так і бути необмежено масштабованою вглиб.

Типовим прикладом такого роду ієрархії є жирне дерево, що містить вузли 4 типів: кореневі, агрегатори, крайові комутатори та обчислювальні вузли. Класичне жирне дерево масштабується вшир через нарощування числа модулів (pods). Інший приклад – класичне n -арне дерево, що масштабується через нарощування глибини.

4.3.1 Синтез дерезовидних мереж з використанням топологічної інтеграції

Ієрархічність у вигляді вкладених рівнів є гарним способом організації крупномасштабної мережі, втім коли мова йде про мережі середнього чи малого розміру, цей метод не дозволяє ефективно реалізувати всі потрібні рівні. Таким чином, виникає потреба в ієрархічній структурі іншого типу, що з одного боку матиме чіткий розділ на рівні, а з іншого – міститиме мінімум рівнів вкладеності. Класичним прикладом такої структури є дерезовидна топологічна організація.

Втім, проблемою класичних дерев (включаючи жирне дерево) є відносно погані топологічні характеристики. З одного боку, існує необхідність в ущільненні корневих зв'язків, від чого страждає ступінь, а з іншого є проблеми із діаметром, оскільки для пересилки між різними гілками необхідно звертатись до кореня. Крім того, не можна забувати і про альтернативні маршрути: хоча жирне дерево штучним чином вводить додаткові зв'язки, проте їх кількість є обмеженою, а локальність – високою. Що це значить? Це означає, що основні і резервні маршрути розташовані в рамках однієї структурної частини топології, а значить, якщо в цій частині станеться збій – з високою імовірністю під удар потраплять вони всі.

Як виправити цю проблему? Не-дерезовидні топології, такі як dragonfly чи гіперкуб, такої проблеми не мають: відносно рівномірне розташування вузлів одне відносно одного урівнює і ризики, тож відмова в одному маршруті має невисокі шанси зачепити всі інші. Таким чином, постає питання: як, не втрачаючи дерезовидності, забезпечити подібні властивості?

Відповідь на це питання відома вже давно: це метод синтезу на основі топологічної інтеграції, який дозволяє поєднати декілька графів в один. Звісно, тут є

два питання: які графи поєднувати і як саме це робити? Таким чином, для практичного застосування цей метод потребує певного розвитку.

Пропонується наступне рішення. По-перше, інтеграція має виконуватись на кожному із рівнів дерева. Саме внутрішньорівневі зв'язки, а точніше, їх повна відсутність є ключовим недоліком дерева. Більш того, інтеграція має зачіпати різні гілки – таким чином, між двома вузлами в мережі існуватимуть альтернативні шляхи принципово різного характеру. Це нівелює проблему локальності, а заодно згладить гостру потребу жирного дерева в стрімкому нарощенні пропускної здатності зі зростанням рівня.

По-друге, необов'язково виконувати інтеграцію однотипно. Класичний метод суміщення графів передбачає, що поєднання на різних рівнях має бути однаковим, проте це не завжди може бути доцільним. По-третє, оскільки передбачається високопродуктивна мережа із, відповідно, великою кількістю вузлів, немає сенсу розглядати кожен вузол окремо. Більш продуктивним підходом є робота із кластерами, що попередньо інтегруються в дерево через декартовий добуток. Це дозволить одночасно і мінімізувати топологічні характеристики, і певним чином поєднати відразу декілька дерев в одне загальне дерево, що дозволить у свою чергу мінімізувати число рівнів.

Таким чином, запропонований метод дозволить синтезувати топології з кращими властивостями ніж у звичайних жирних дерев чи класичних інтегрованих топологій.

4.3.2. Опис методу. Топологія Tree-Dragonfly

Основою синтезу є три графи – G_T , G_C та G_L . Перший граф G_T представляє деревовидну структуру рангу r_T , у якій є корінь, проміжні вузли і листя. Адреса кожної вершини в такій топології може бути представлена двома параметрами (L, m) , де L визначає рівень дерева (від кореня 0 до листя r_T-1), на якому розташована вершина, а m – положення вершини на відповідному рівні. В залежності від зовнішніх зв'язків можна виділити наступні види деревовидних структур:

- Строге дерево, $\forall (L, m) \in G_T$ та $(L', m') \in G_T$ зв'язок може існувати тоді і тільки тоді, коли $|L - L'| = 1$. Тобто, зв'язані лише вершини сусідніх рівнів.
- Нестроге дерево, де може існувати деяка кількість додаткових зв'язків із несусідніми рівнями.
- Регулярне дерево, де зв'язки між сусідніми рівнями мають регулярний характер.
- Нерегулярне, де міжрівневі зв'язки змінюються за певним правилом.

Відповідно, такі топології як Leaf/Spine, жирне та звичайне k -арне дерево відносяться до строгих регулярних дерев. Дерева де Бруйна, розглянуті раніше – до строгих нерегулярних, оскільки від кореня в них йде на 1 зв'язок вниз менше, ніж від інших некінцевих вершин.

Граф G_C відповідає за кластер, що інтегрується в дерево. Не існує жодних обмежень на структуру кластеру, проте оптимальними є графи з мінімальним параметром D . Інтеграція відбувається через декартовий добуток, таким чином, мінімізуються втрати діаметру. При цьому нумерація вершин трансформується із пари параметрів у трійку (L, m, k) , де параметр m вже визначає номер кластера на рівні дерева, а k – номер конкретного вузла у кластері.

Граф G_L представляє міжрівневі зв'язки. При цьому правило формування міжрівневих зв'язків не обов'язково є однаковими для всіх вершин кластеру – воно може задаватись формульно, де вершині (L, m, k) ставиться у відповідність інша вершина (чи вершини) (L, m', k') .

Гарним прикладом, що ілюструє функціонування запропонованого методу, є топологія Tree-Dragonfly [5]. Дана структура базується на поєднанні трьох простих графів – бінарного дерева, повнозв'язного кластеру з 3 вершин і dragonfly, причому щодо останнього вводиться обмеження: для кожної вершини існує лише $h = l$ dragonfly зв'язок.

Синтез графа Tree-Dragonfly може бути описаний наступним чином. Нехай дано ранг r бажаного графа. Таким чином, вихідними інтегрованими графами є бінарне

дерево рангу r (з $N_T=2^{r+1}-1$ вершинами), повнозв'язний трикутник ($N_C=3$) і Dragonfly на 4 групи, де групами є повнозв'язні трикутники. Число вершин N результуючої топології є рівним $N_T * N_C$, що безпосередньо витікає із властивостей декартового добутку. Це дозволяє чітко визначити області значень параметрів $L \in [0, r-1]$; $m \in [0, 2^{L+1}-1]$; $k \in [0, 2]$. Для зручності потужність множин значень для параметрів m та k також можна позначити як M і K відповідно.

Відомо, що бінарне дерево є топологією зі статичним ступенем 3. При цьому кластер не масштабується, тож також має статичну ступінь 2. І, оскільки Dragonfly зв'язки обмежені умовою синтезу, то для кожної вершини існує рівно 1 такий зв'язок. Таким чином, для кожної вершини з наперед відомим номером (L, m, k) можливо визначити рівно 6 сусідів, таких, що:

$$V_1 = (L-1, \frac{m}{2}, k); \quad (4.5)$$

$$V_2 = (L+1, 2m, k); \quad (4.6)$$

$$V_3 = (L+1, 2m+1, k); \quad (4.7)$$

$$V_4 = (L, m, (k+1) \bmod K); \quad (4.8)$$

$$V_5 = (L, m, (k+2) \bmod K); \quad (4.9)$$

$$V_6 = (L, m', k'); \quad (4.10)$$

Перші 3 формули тут витікають з властивостей бінарного дерева і є добре відомими. Формули 4.8 та 4.9, відповідно, беруть початок у властивостях кластеру і тому також не представляють інтересу. Щодо формули 4.10, вона базується на перестановках номерів. Наступні формули описують зовнішні зв'язки в dragonfly із 4 кластерів:

$$m' = \begin{cases} k, & k < m \\ k+1, & k \geq m \end{cases} \quad (4.11)$$

$$k' = \begin{cases} m, & m < m' \\ m-1, & m \geq m' \end{cases} \quad (4.12)$$

Втім, формули 3.11 та 3.12 не підходять для опису будь-якого етапу масштабування – лише для етапу із 4 кластерів, що відповідає рангу 3 масштабування дерева.

Рис. 4.4 ілюструє топологію Tree-Dragonfly рангу 3 [5]. Чорні зв'язки є типовими для дерева, декартово помноженого на повнозв'язний кластер. Зелені – ті, що з'являються внаслідок інтеграції Dragonfly. Відповідно, кореневий рівень не приймає участі в інтеграції, а на першому рівні (оскільки існує лише 2 кластери) Dragonfly повнозв'язність вироджується у декартовий зв'язок. Цікавість представляє третя група, оскільки в ній і досягається псевдоповнозв'язність, причому задіяними є всі вершини. Це породжує питання: яким чином виконувати масштабування далі? Для того, щоб проілюструвати високорангові топології, зручніше використати кластерне представлення (для топології рангу 3 воно також зображено на рис. 4.4). В ньому вершинами є кластери, подвійні лінії представляють декартові зв'язки (тобто, кожна вершина кластеру a зв'язана з відповідною вершиною кластера b), а пунктирна лінія, яка поєднує 4 вершини – це dragonfly-повнозв'язність. Тобто, кожен кластер має зв'язки зі всіма іншими, при цьому якщо кластерів менше ніж вершин у кластері – зв'язки ущільнюються (між деякими парами кластерів виникає більше одного зв'язка).

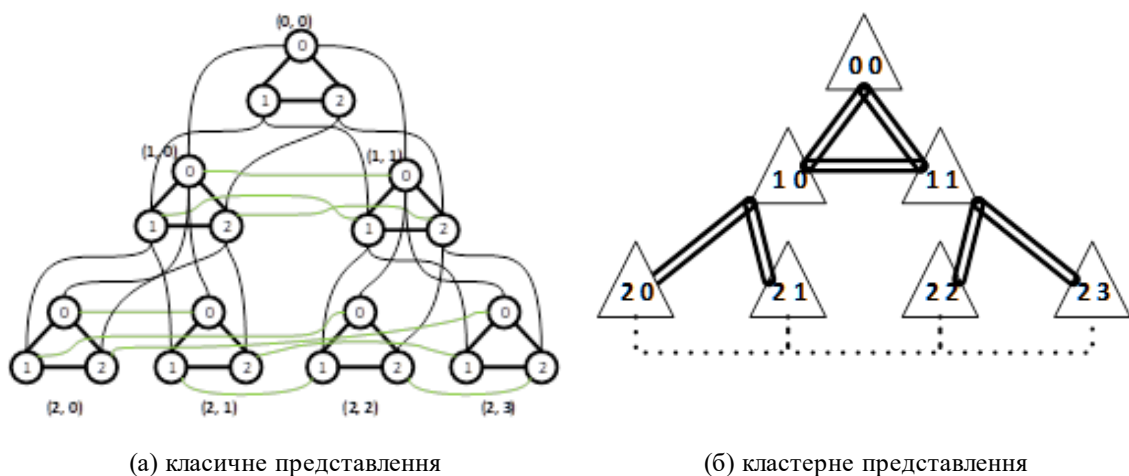


Рис. 4.4. Tree-Dragonfly, ранг 3 [5]

Таким чином, декілька підходів можуть бути запропоновані для подальшого масштабування. Ідея групування полягає у тому, що рівень L з M кластерами розбивається на групи по $K+1$ кластерів, які і поєднуються між собою. Це дозволяє зберегти статичну ступінь і при цьому скоротити діаметр. Ідеальним для бінарного дерева тут є випадок, коли $K = 2^t - 1$, де t – будь-яке натуральне число. В такому випадку розділення відбуватиметься на однакові за розміром групи, що дозволяє значно спростити роботу із ними.

На рис. 4.5 представлено 3 способи такого розділення. Ближній спосіб передбачає групування сусідніх кластерів, дальній – групування з певним кроком, що дозволяє об'єднувати віддалені точки. Комбінований метод, відповідно, реалізує чергування, що покликано максимізувати скорочення діаметру.

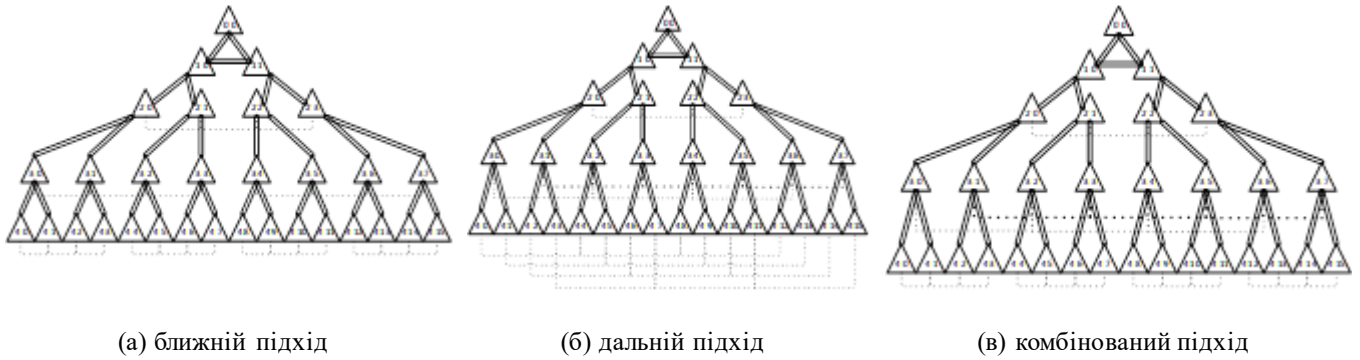


Рис. 4.5. Способи групування Tree-Dragonfly [5].

Відповідно, питання постає у тому, як математично описати виконані трансформації. Звісно, ключовий інтерес тут представляють лише ближній та дальній підходи, оскільки будь-яка версія комбінованого підходу так чи інакше базуватиметься на попередніх двох. На перший погляд, для формул 4.11 та 4.12 існує проблема розгалужень. Проте, якщо проаналізувати їх уважніше, розгалуження $k | k+1$ та $m | m-1$ існують виключно для уникнення ситуації, коли вузли виду (L, k, k) не матимуть сусідів. Тож для урахування впливу групування ці розгалуження не важливі.

Щоб спростити формули 4.11 і 4.12, введемо два додаткових параметри – g , що вказує номер групи, до якої належить поточна вершина, і G_L – кількість груп на рівні L . Нехай дано ідеальний випадок в бінарному дереві (як це проілюстровано на рис. 4.5). Тоді:

$$G_L = \frac{M}{K+1} = \frac{2^{L+1}}{K+1} = \frac{2^{L+1}}{2^t - 1 + 1} = 2^{L-t+1} \quad (4.13)$$

Відомо, що в ближньому методі групуються сусідні кластери, а в дальньому – з певним кроком, що залежить від кількості груп. Обидва ці випадки достатньо просто врахувати, використовуючи, відповідно, зміщення та остачу, тож:

$$g_{\text{ближ}} = \frac{m}{K+1} = \frac{m}{2^t} \quad (4.14)$$

$$g_{\text{дал}} = m \bmod G = m \bmod 2^{L-t+1} \quad (4.15)$$

Як можна бачити, хоча ці формули передбачають ділення та отримання модулю, проте, оскільки в основі лежить ступінь 2, вони можуть бути замінені, відповідно, зсувом вправо на t розрядів та вилученням $L-t+1$ молодших бітів числа з використанням маски. Звісно, зазначені формули є справедливими лише у випадку $L \geq t$, тобто, на конкретному рівні має існувати хоча б 2 групи.

Таким чином, формула 4.11 набуває наступного вигляду:

$$m'_{\text{ближ}} = \begin{cases} g + k, k < m - g \\ g + k + 1, k \geq m - g \end{cases} \quad (4.16)$$

$$m'_{\text{дал}} = \begin{cases} g + Gk, k < \frac{m - g}{G} \\ g + Gk + 1, k \geq \frac{m - g}{G} \end{cases} \quad (4.17)$$

Відповідно, щоб із m отримати k' , необхідно виконати зворотне перетворення. Тож формула 4.12 приймає наступний вид:

$$k'_{\text{ближ}} = \begin{cases} m - g, m < m' \\ m - g - 1, m \geq m' \end{cases} \quad (4.18)$$

$$k'_{\text{дал}} = \begin{cases} \frac{m - g}{G}, m < m' \\ \frac{m - g}{G} - 1, m \geq m' \end{cases} \quad (4.19)$$

Практичне значення цих формул полягає, звісно, у маршрутизації. Загалом, з такої точки зору ближній метод є досить простим, оскільки передбачає лише елементарні перетворення зсуву, додавання та віднімання. На противагу, дальнє групування передбачає виконання операцій множення та ділення, які є доволі тривалими.

4.3.3. Деревовидна мережа з неоднорідною ієрархією

Запропонований вище метод дозволяє покращити топологічні характеристики і зробити мережу більш ефективною. Проте його недоліком, як видно, є складність маршрутизації, що зростатиме із кожним наступним кроком масштабування вниз. Втім, є й інший вихід: що як обмежити число рівнів дерева деяким наперед заданим числом і виконувати масштабування не вглиб, а вшир? Оскільки метод, що пропонується, не передбачає потреби в об'єднанні надвеликої кількості вузлів, такий

хід дозволить спростити маршрутизацію, а крім того – розбити повноваження між різними рівнями дерева.

В якості основи пропонується взяти раніше розглянуту топологію Tree-Dragonfly, додавши до неї певні зміни. Так, по-перше, глибину дерева обмежимо 4 рівнями (не включаючи корінь). При такій глибині дерево міститиме як мінімум 31 кластер з як мінімум 3 вузлами в кожному. Враховуючи, що вузлами мережі можуть бути не самі обчислювальні елементи, а комутатори, до яких може підключатись як мінімум по 2 вузли – маємо вже 186 обчислювальних вузлів, чого цілком достатньо для невеликого суперкомп'ютера.

По-друге, горизонтальні зв'язки dragonfly розглядатимемо як псевдоповнозв'язну структуру в межах рівня. Таким чином, всі кластери рівня матимуть прямий зв'язок із іншими кластерами того ж рівня, що суттєво зменшить діаметр. Альтернативним рішенням може бути розбиття на декілька підмереж з частковими їх перетинами, що може бути доцільним в умовах, коли число кластерів останнього рівня є досить великим.

Рис. 4.6 ілюструє запропоновану структуру в її мінімальному вигляді. Трикутниками тут позначені кластери, а зеленими шинами – псевдоповнозв'язні зв'язки типу dragonfly.

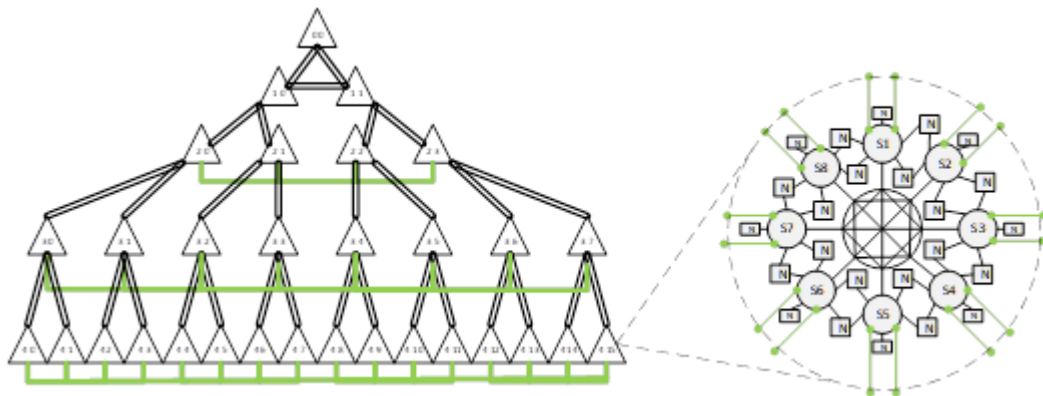


Рис. 4.6. Узагальнена деревовидна мережа з однорідною ієрархією

Окремо варто звернути увагу на кластери. В даному прикладі вони складаються із 8 комутаторів, поєднаних внутрішньою мережею. Також на рисунку продемонстровано 2 типи обчислювальних вузлів: власні, що приєднані до 1 комутатора, і спільні, що мають зв'язки з 2 сусідніми комутаторами. Така структура

дозволяє збільшити відмовостійкість ціною зменшення числа обчислювачів на комутатор.

З точки зору топологічної організації та маршрутизації така структура є набагато простішою за раніше запропоновану Tree-Dragonfly. Так, пересилка в такій мережі може виконуватись у 2 кроки: спочатку повідомлення пересилається на цільовий рівень дерева, а потім – через горизонтальні зв'язки до цільової вершини. Аналогічно, альтернативні маршрути в такій мережі також можуть бути легко знайдені.

Втім, з точки зору організації обчислень така структура може бути покращена. Для цього розділимо мережу на 4 рівні: рівень ядра (core), глибокий (deep), мілкий (fine) та крайовий (edge). Відповідно, корінь і перший рівень дерева відноситиметься до ядра, а інші 3 рівні дерева відповідатимуть наведеним іншим 3 рівням мережі. Зв'язки між рівнями формуватимуться наступним чином: ядро і глибокий рівень пов'язані leaf-spine топологією так, що кожен кластер кореня декартово зв'язаний із кожним глибоким кластером. Зв'язки між глибокою та мілкою частинами реалізовані за логікою k-арного n-дерева. Мілкий рівень та край зв'язані звичайним k-арним деревом. Рис. 4.7 наочно ілюструє таке неоднорідне дерево.

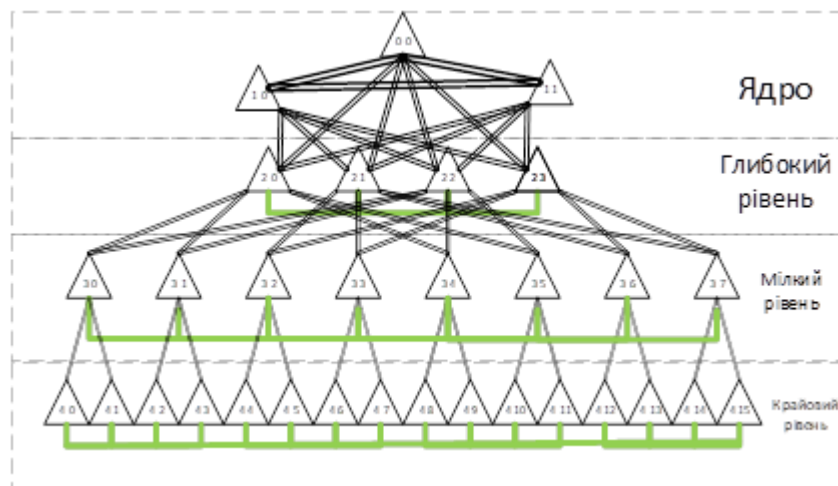


Рис. 4.7. Деревовидна мережа з неоднорідною ієрархією

Суть такого розділення полягає в наступному. По-перше, суперкомп'ютери, як правило, є гетерогенними системами. Одні їх вузли призначені для більш складних задач, інші – для більш простих. Таким чином, запропонована структура дозволяє зосередити найбільш потужні та малочисельні обчислювальні елементи в щільно

зв'язаному ядрі та глибокій частині, тим самим до мінімуму скорочуючи час пересилок між такими елементами. В той же час більш прості та численні вузли можуть бути рознесені по мілкій та крайовій частині мережі, що дозволяє використовувати їх для великої задачі лише тоді, коли для цього є потреба, а в інший час – завантажити простішими та менш вимогливими по часу завданнями.

По-друге, зі спуском по дереву роль горизонтальних зв'язків зростає. Так, ядро взагалі не містить dragonfly-повнозв'язності: її повністю заміняє декартовий зв'язок. На глибокому рівні горизонтальні зв'язки ущільнені, так що існує більше 1 прямого зв'язку між двома кластерами. Важливо зазначити, що в обидвох випадках таке ущільнення не потребує штучного втручання: воно виникає саме, через те, що число вузлів у кластері більше ніж число кластерів, які мають бути зв'язані. В той же час зв'язування всіх кластерів мілкого рівня чи, тим паче, краю, навіть в мінімальному варіанті «1 зв'язок до кожного іншого кластеру» може потребувати використання додаткових горизонтальних зв'язків. Таким чином, зміна щільності міжрівневих зв'язків, окрім очевидного ефекту підвищення відмовостійкості глибоких рівнів, має на меті ще і балансування числа зв'язків на кожен окремо взятий вузол.

Висновок до розділу 4

Існуючі популярні рішення в більшості своїй є ієрархічними графами, орієнтованими на комутовані мережі. Їх недоліком є завищений ступінь, що негативно позначається на вартості, зростання ступеня з масштабуванням, а також занижений топологічний трафік, що ускладнює ефективне використання альтернативних маршрутів для передачі даних чи забезпечення відмовостійкості.

Було виділено 2 види ієрархічності, такі як вкладена та деревовидна. В розділі було розглянуто вкладений підхід, на основі якого отримано топології hyper de Bruijn, а також описано рекурентний спосіб синтезу, за допомогою якого синтезовано рекурентний граф та топологію dragon de Bruijn. Також розглянуто деревовидний вид ієрархічності, описано синтез топології Dragonfly-Tree, розглянуто різні види групування. Для розглянутих топологій було запропоновано топологічно-орієнтовані алгоритми маршрутизації на основі їх властивостей, що має значення для практичної реалізації систем на основі даних графів.

Запропонований метод дозволяє використовувати для синтезу як класичні графи, так і графи, отримані за допомогою раніше розглянутого методу синтезу відмовостійких топологій на основі надлишкового коду. Такого роду гнучкість є потенційною перевагою запропонованого методу, оскільки дозволяє інтегрувати запропоновані засоби підвищення відмовостійкості та ефективності із сучасними популярними рішеннями, тим самим покращуючи їх характеристики і забезпечуючи сумісність із відомими мережевими протоколами та алгоритмами.

Не дослідженим залишається застосування алфавітної декомпозиції надлишкових графів в контексті синтезу ієрархічних топологій, а також масштабування надлишкових топологій з використанням деревовидного типу ієрархічності.

РОЗДІЛ 5

ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ НА ОСНОВІ РОЗРОБЛЕНИХ ЗАСОБІВ

5.1. Опис засобів оцінки ефективності та відмовостійкості топологій

Для аналізу запропонованих методів розроблено ряд інструментальних програмних засобів на основі раніше розглянутих способів:

- Засіб для моделювання графів, що дозволяє синтезувати графи за заданими правилами, а також вимірювати їх топологічні характеристики. Модель, що лежить в основі засобу, базується на класичній теорії графів. Мета розробки даного засобу полягає в експериментальній перевірці запропонованих методів для доведення виграшу в ефективності. Це можна зробити, використовуючи загальноприйняті топологічні характеристики, такі як ступінь, діаметр, тощо.
- Засіб для моделювання топології в умовах відмови. Концепція цієї моделі полягає в тому, щоб проаналізувати зміну топологічних характеристик графа в ситуації, коли вузли системи поступово відмовляють. Мета розробки засобу – доведення виграшу відмовостійкості у рішень на основі запропонованих методів порівняно із класичними.

Також в процесі аналізу є сенс розділити окремо аналіз топологій для безпосередньо зв'язаних мереж (БЗМ) та графів для комутованих мереж. Це необхідно з двох причин: по-перше, це дасть змогу скоротити число експериментів, а по-друге, це дозволить врахувати особливі вимоги різних мережевих класів та виконувати аналіз з урахуванням існуючих для них обмежень.

5.2. Засіб для моделювання топологій

Код розробленого програмного засобу приведено в Додатку А (параграф 1).

5.2.1. Опис моделі

Дана модель полягає в аналізі масштабованого графа за топологічними характеристиками. Вхідними даними моделі є:

- тип графа (гіперкуб, граф де Бруїна, тощо)
- параметри синтезу (наприклад, алфавіт СЧ)
- порядки масштабування, за якими виконуватиметься аналіз

Результатом є таблиця, де кожному проаналізованому кроку масштабування ставитиметься у відповідність набір топологічних характеристик графа. Загалом, проаналізовані характеристики можна поділити на наступні класи:

1. Класичні топологічні характеристики: визначають продуктивність, вартість та час пересилок в системі, побудованій на основі графа. Для більшості характеристик оптимальними є нижчі значення (крім N , де оптимальним є вище значення, та T , де оптимальним є значення 1)
 - a. Число вузлів N : в контексті системи визначає її номінальну продуктивність. Є сенс порівнювати лише графи з близькими показниками N .
 - b. Число ребер R : визначає фактичну вартість графа, тобто, кількість зв'язків, що мають бути побудовані між вузлами. В системному контексті з ростом R ростуть витрати на зв'язки і складність комутації.
 - c. Ступінь S : фактично, визначає кількість портів маршрутизаторів / комутаторів, що мають застосовуватись у системі.
 - d. Діаметр D : визначає мінімальну відстань, а отже і час пересилок між найвіддаленішими вузлами.
 - e. Середній діаметр \bar{D} (або aD): показує, наскільки вузли системи в середньому віддалені одне від одного. Варто очікувати, що чим більше середній діаметр – тим більше часу в середньому займатимуть пересилки, а отже – тим меншою буде ефективність.
 - f. Топологічний трафік T : демонструє, наскільки ефективно ребра графа забезпечують пересилки в умовах максимального паралельного трафіку.
 - g. Вартість C : визначає вартість графа з урахуванням того, що вузли, чия ступінь менша ніж ступінь графа, отримуватимуть нові зв'язки для максимального використання можливостей комутаторів.

2. Характеристики відмовостійкості: показують, наскільки складно ізолювати окремі вершини (спричинивши тим самим ситуацію, коли технічно працездатний вузол не може виконувати свої функції) чи розбити мережу на незв'язні частини (тим самим спричиняючи відмову системи). Оптимальними є вищі значення.

- a. Локальна зв'язність L : показує мінімум ребер, які мають бути розірвані для ізоляції однієї з вершин графа.
- b. Глобальна реберна зв'язність G або G_e : мінімальне число ребер, розрив яких дозволяє розбити граф на незв'язні підграфи.
- c. Глобальна вузлова зв'язність G_n : мінімальне число вузлів, відмова яких дозволяє розбити граф на незв'язні підграфи.

3. Додаткові характеристики:

- a. Локальна ефективність El : показує, наскільки швидко в середньому вузли здатні комунікувати між собою в рамках підграфів, що складаються з сусідів кожної конкретної вершини.
- b. Глобальна ефективність E або E_g : показує, наскільки ефективно комунікуватимуть між собою вершини всього графа (середнє значення ефективності для всіх можливих пар).

Оскільки інтерпретація даних характеристик є різною, слід збирати їх окремо.

Передбачається наступна серія досліджень з використанням даної моделі:

- 1. Відбір на основі класичних топологічних характеристик.
- 2. Відбір на основі відмовостійкості та ефективності.
- 3. Аналіз впливу додавання зв'язків на основі багатовимірної форми.
- 4. Порівняння результатів запропонованого методу із класичними рішеннями за цими 3 критеріями.

Після збору даних для кожного окремого набору вхідних даних передбачається виконання наступних етапів аналізу:

- 1. Групування, що полягає в розбитті вихідних даних на групи у відповідності до числа вузлів.
- 2. Усереднення даних для кожного графа кожної групи

3. Приведення до спільного виду, що полягає у трансформації характеристик таким чином, щоб мінімальні значення були оптимальними.
 - а. Для характеристик, де оптимальне найвище значення приведення полягає в тому, що для кожної групи визначається максимум X_{max} та мінімум X_{min} . Далі трансформація виконується за формулою: $X' = 1 + (X_{max} - X)/X_{min}$. Це дозволяє одночасно виконати і нормалізацію значень.
 - б. Для топологічного трафіка перетворення відбувається за формулою $T' = 2 - T$
4. Нормалізація, яка полягає в діленні всіх характеристик кожної групи на мінімальне значення в цій групі.
5. Отримання нормалізованої оцінки, що полягає в підсумовуванні нормалізованих значень для кожного набору вхідних даних.

5.2.2. Вибір графів БЗМ на основі класичних характеристик

Завданням даного етапу є мінімізація подальших експериментальних перевірок шляхом відкидання тих наборів параметрів, що дають результат, занадто далекий від оптимуму. Для перевірки розглядаються лише графи, отримані з використанням методу синтезу на основі надлишкового коду без урахування нових зв'язків. Знаючи, що характеристики чистих графів залежать лише від довжини алфавіту k і не залежать ні від його вмісту, ні від основи числення b , можна використати мінімальний набір експериментальних параметрів, що приведений в Таблиці 5.1.

Табл. 5.1.

Експериментальні параметри для відбору

Параметр	Можливі значення
Тип перетворення	Заміщення (гіперкуб); зсув (де Бруйн)
Порядок алфавіту k	2–10
Порядок топології r	2–12
Максимальне число вузлів N_{max}	4096

Перш ніж приступити до загального експерименту, розглянемо попередній експеримент з меншим діапазоном значень параметрів. В якості таких було обрано $N_{max} = 729$ та $k = 2, 3, 4, 5, 7$. Це дозволить отримати деякі попередні результати та протестувати розроблену модель.

Отримані дані було зібрано в 3 групи, умовно позначені англійськими літерами S (мала, 1–32 вузли), M (середня, 33–216 вузлів), L (велика, 217–729 вузлів). На рис. 5.1 представлена гістограма нормалізованих оцінок, згрупованих за типом перетворень по групам, де на легенді кольорами позначено різні k .

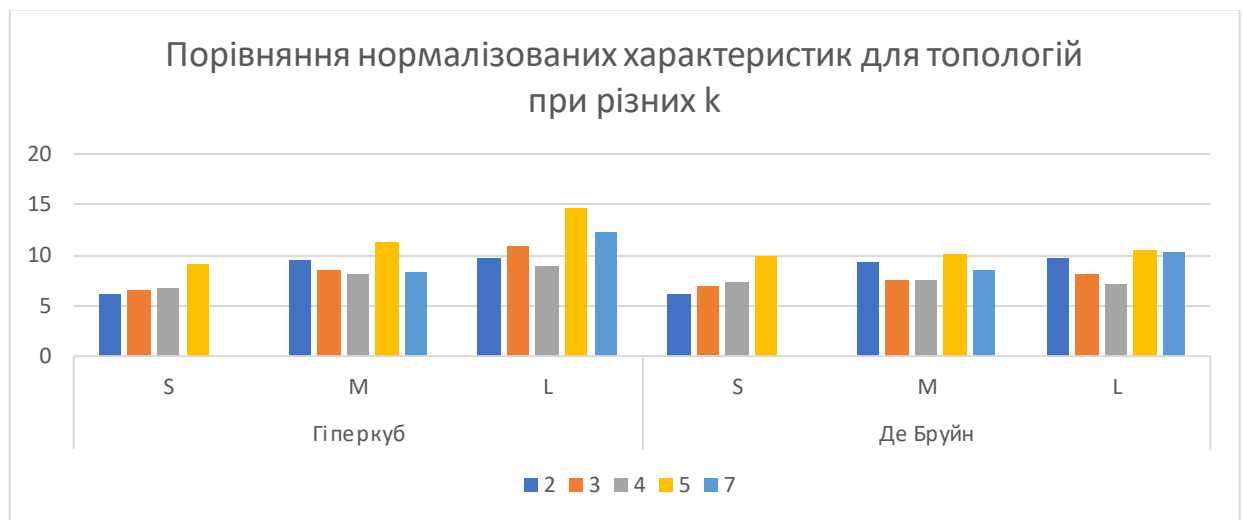


Рис. 5.1. Гістограма нормалізованих значень

Як можна бачити із рис 5.1, варіанти з меншими k загалом мають кращі характеристики, ніж інші. Найкращими можна вважати результати для трійкової та четвіркової СЧ, оскільки в такому разі забезпечується певний баланс між ступінню та діаметром. Напроти, мережі з великими k показують себе гірше, втім на такому наборі параметрів неможливо виявити якої б то не було загальної тенденції.

Таким чином, розширення діапазону експериментальних параметрів має дати більш загальне уявлення про поведінку мереж на різних етапах масштабування. В ході аналізу експериментальні дані було розбито на 4 групи S (0–63), M (64–255), L (256–1023), XL (1024–4096), проведено нормалізацію та отримано нормалізовану оцінку. Рис. 5.2 демонструє гістограми, побудовані на основі нормалізованих оцінок. Умовні позначки HC та DB позначають метод масштабування (гіперкубічний чи дебруїнівський), а цифра поряд – розмір алфавіту k . Менші значення нормалізованої

оцінки є кращими. Відсутні стовпці гістограми означають, що масштабування відповідних топологій не передбачає жодного кроку з числом вершин, що потрапляло б у відповідну область визначення групи. Ці топології мають бути пропущені при аналізі конкретної групи.

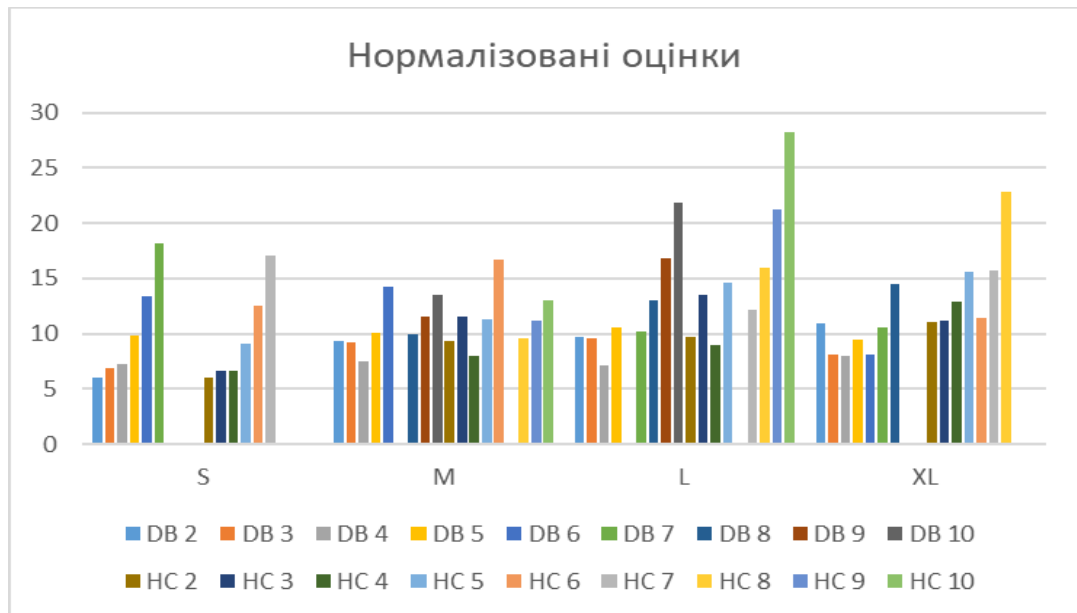


Рис. 5.2. Порівняння топологій за нормалізованою оцінкою

Також, в додаток до нормалізованих оцінок, є сенс звернути увагу на комплексний параметр SD , що дозволяє порівняти граfi за двома ключовими характеристиками, – ступенем та діаметром. Нижче значення SD показує, що система забезпечує краще співвідношення швидкості передачі даних та вартості ніж аналоги з вищим показником. Цей графік приведений на рис. 5.3.

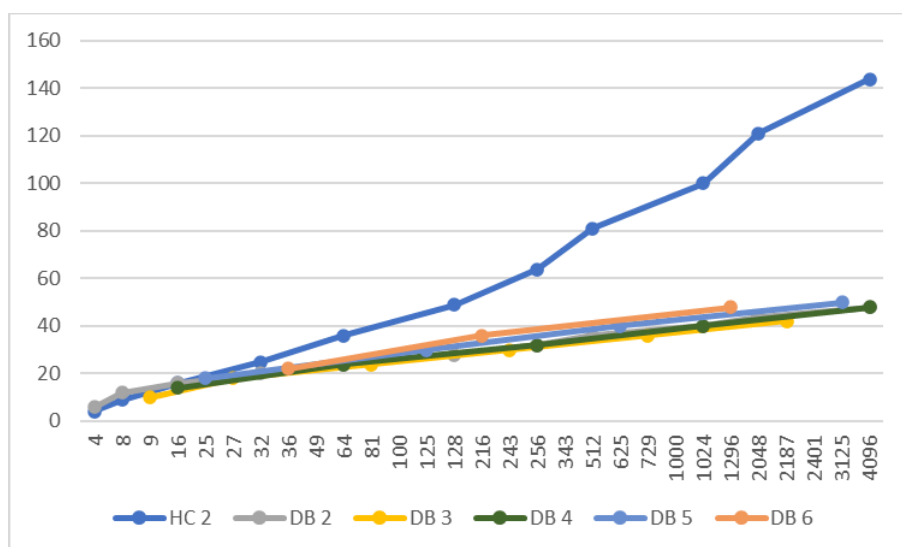


Рис. 5.3. Порівняння топологій за характеристикою SD

На основі цих оцінок можна відмітити наступні тенденції: загалом, мережі, побудовані на основі дебруйнівських перетворень, забезпечують кращі характеристики порівняно із мережами гіперкубічного типу. Ключовою причиною цього є ступінь, яка сильно зростає як і з ростом k , так і з масштабуванням. В той же час, найкращими з точки зору характеристик є дебруйнівські мережі з $k=3$ та 4. Якщо казати про групу XL, що є найцікавішою з точки зору високопродуктивних обчислень, варто також виділити дебруйнівську мережу із $k=6$, що, як і попередні варіанти, отримала оцінку 8, (для порівняння, нормалізована оцінка бінарного гіперкуба в цій групі – 11,01).

Відповідно, в подальших експериментах немає сенсу розглядати мережі з $k>7$, а також варіант $k=5$. Аналогічно, надлишковий гіперкуб хоч і є цікавим з точки зору щільності зв'язків, втім є занадто дорогим, особливо на високих N .

5.2.3. Аналіз відмовостійкості та топологічної ефективності

Вибравши із множини топологій ті, що є найкращими з точки зору характеристик, можна виконати оцінку їх відмовостійкості (зв'язності) та топологічної ефективності. Для аналізу пропонується розглянути дебруйнівські мережі на основі двійкового, трійкового, четвіркового та шестіркового кодів, а також бінарний гіперкуб в якості контрольного експерименту.

Рис. 5.4 представляє отримані результати (повна версія – рис. Д.6).

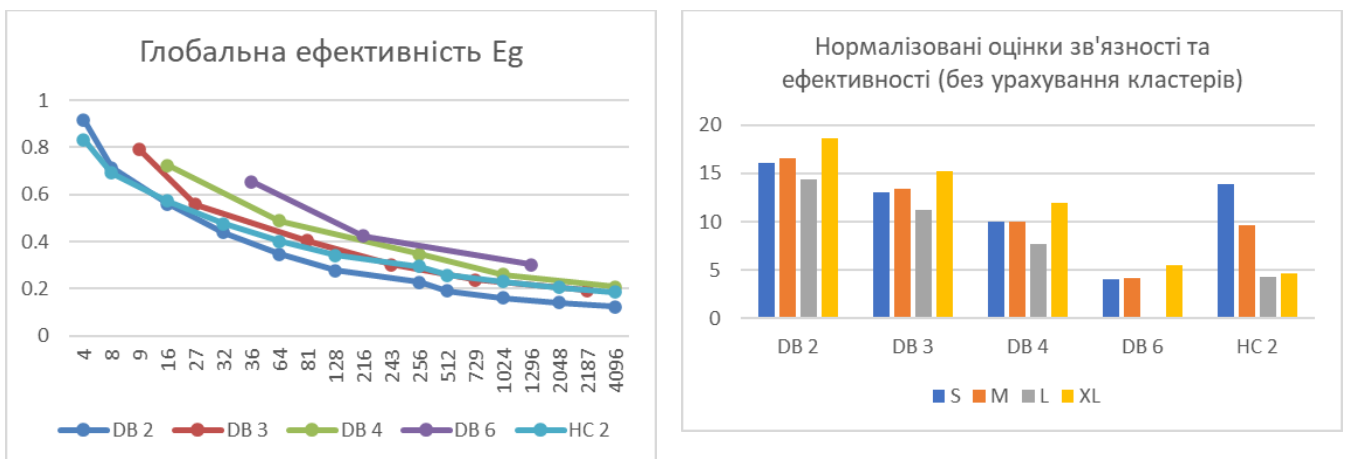


Рис. 5.4. Топологічна ефективність графів та нормалізовані оцінки

Представлені дані показують результати для топологій без імпліцитних кластерів. Загалом, зв'язність гіперкуба має тенденцію до постійного покращення, в той час як дебруйнівська мережа залишається відносно статичною.

Втім, отримані результати не є однозначними. По-перше, для всіх розглянутих графів локальна, глобальна реберна та глобальна вузлова зв'язності були рівними. Це означає, що єдиний спосіб розірвати таку мережу – ізолювати одну вершину, що загалом не є критичним. Таким чином, в подальшому аналізі є сенс обмежитись локальною зв'язністю як більш простим для вимірювання параметром. З іншого боку, локальна ефективність гіперкуба виявилась рівною 0, через що цей параметр довелося виключити із нормалізованої оцінки задля уникнення помилки.

В подальшому, щоб уникнути проблем із аналізом, пропонується обмежити аналіз зв'язності локальною зв'язністю, а аналіз топологічної ефективності – глобальною ефективністю.

5.2.4. Аналіз впливу імпліцитних кластерів

Наступним кроком є введення додаткових зв'язків у відповідності до багатовимірної форми. Табл. 5.2 ілюструє зміну характеристик після додавання імпліцитних кластерів у порівнянні з топологіями з тим же розміром алфавіту.

Табл. 5.2

Зміна характеристик після додавання кластерів

Крок	Хар.	(3, 2)	(4, 2)	(4, 3)	(6, 2)	(6, 3)	Крок	Хар.	(3, 2)	(4, 2)	(4, 3)
1	N	9	16	16	36	36	4	N	243	1024	1024
	ΔS	0	1	1	2	1		ΔS	4	4	4
	$\Delta \bar{D}$	0.00	-0.02	-0.01	-0.03	-0.01		$\Delta \bar{D}$	-0.32	-0.48	-0.19
	ΔE_g	0.000	0.008	0.004	0.013	0.007		ΔE_g	0.031	0.036	0.013
2	N	27	64	64	216	216	5	N	729	4096	4096
	ΔS	1	2	2	4	2		ΔS	5	5	5
	$\Delta \bar{D}$	-0.07	-0.15	-0.07	-0.23	-0.13		$\Delta \bar{D}$	-0.43	-0.62	-0.22
	ΔE_g	0.020	0.033	0.014	0.043	0.024		ΔE_g	0.028	0.031	0.010
3	N	81	256	256	1296	1296	6	N	2187	-	-
	ΔS	3	3	3	6	3		ΔS	6		
	$\Delta \bar{D}$	-0.19	-0.31	-0.14	-0.49	-0.28		$\Delta \bar{D}$	-0.52		
	ΔE_g	0.030	0.039	0.016	0.051	0.028		ΔE_g	0.023		

Варто зазначити, що такі характеристики як діаметр та локальна зв'язність при цьому не змінювались. Як можна бачити із таблиці, ступінь демонструє тенденцію до погіршення, в той час як середній діаметр спадає, а ефективність – росте. Причому більш вираженими ці «стрибки» є у топологій на кодах типу (tb, b) , а також у графів з меншою основою числення. Таким чином, є сенс розглядати коди $(4, 2)$, $(6, 2)$ та $(6, 3)$ в якості ключових, а також код $(3, 2)$, який є близьким за параметрами.

5.2.5. Порівняння результатів із класичними рішеннями

В якості класичних графів було обрано гіперкуб, 2D тор, 3D тор та 6D тор. Як показує аналіз в Розділі 1, це ті графи, що найчастіше використовуються для побудови БЗМ в суперкомп'ютерах. Серед графів, отриманих на основі методу синтезу з використанням надлишкового коду, розглядатимуться кластеризовані графи $(4, 2)$, $(6, 2)$ та $(6, 3)$. Результат такого порівняння проілюстровано на рис. 5.5.

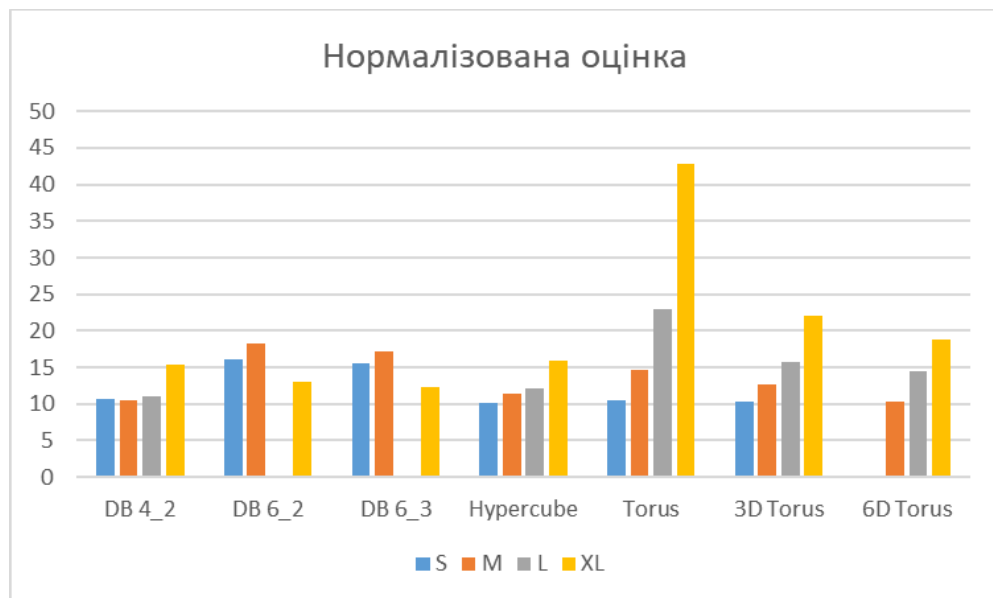


Рис. 5.5. Порівняння запропонованих графів із класичними рішеннями

В процесі аналізу було виявлено, що 2D тор забезпечує одні з найгірших характеристик на високих N . Отримані результати свідчать про наступне: незважаючи на негативний вплив створення кластерів, запропоновані топології забезпечують характеристики на рівні класичних рішень. В той же час їх показники ефективності є значно кращими.

5.2.6. Порівняння із популярними сучасними рішеннями

Аналіз вище порівнює результати із типовими топологічними організаціями, втім не враховує сучасних трендів в сфері високопродуктивних обчислень. Як було продемонстровано в Розділі 1, більшість сучасних популярних рішень опираються на 2 топології: це жирне дерево та dragonfly. Таким чином, неможливо робити висновки щодо ефективності методу без урахування цієї частини аналізу.

Для рівномірності масштабування для жирного дерева та dragonfly розглядатимемо число портів комутаторів в якості рангу, розділяючи зв'язки навпіл між зовнішніми та внутрішніми комутаціями.

Також, оскільки мова йде про комутовані мережі, є сенс додати до порівняння топології, отримані ієрархічним методом. Розглядатимемо топологію Hyper de Bruijn з групою у вигляді дебруйнівської мережі порядку 3 на основі кодів (3, 2) та (4, 2), а також топологію DDB з групою на основі коду (3, 2).

Варто додати, що в ієрархічних мережах розглядаються виключно мережі з 2 рівнями глибини. Причиною цього є забезпечення справедливості порівняння, а також уникнення ситуації, коли швидке масштабування унеможливить аналіз через нестачу обчислювальних ресурсів. Також при розгляді топологій не розглядатимемо обчислювальні вузли, оскільки вони є кінцевими для мережі і не впливають на її функціонування. Також, щоб зробити отримані результати більш наближеними до реальності, було проаналізовано тороїдальну топологію, що використовується в суперкомп'ютері Fugaku (1 місце за HPCG). Рис. 5.6 демонструє графіки порівняння характеристики SD для чистих топологій де Бруйна, а також для ієрархічних, таких як Hyper de Bruijn та Dragon de Bruijn. Нормалізація для цих графів не виконувалась, оскільки швидкість масштабування ієрархічних топологій є зависокою навіть із усіма накладеними обмеженнями, що не дає змоги розбити експериментальні результати на групи за кількістю вершин. Повна версія (таблиці та графіки) – у Додатку Г.

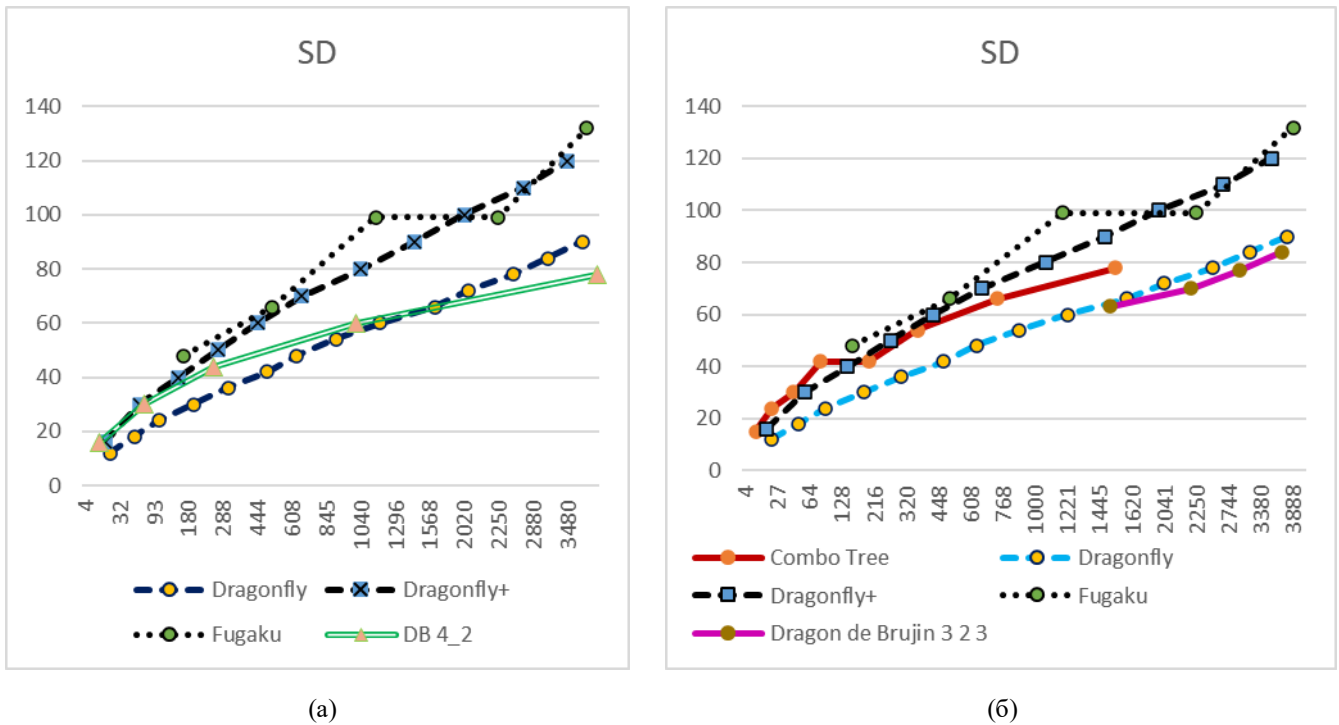


Рис. 5.6. Порівняння з популярними графами для (а) дебруйнівських мереж, (б) ієрархічних мереж

Як можна побачити із графіків, загалом запропоновані мережі показують кращі характеристики щодо ступеня ніж більшість класичних мереж, проте програють в діаметрі та середньому діаметрі. При цьому характеристика SD дебруйнівських мереж трохи кращою. Мережі деревовидного типу та Hyper de Bruijn переважають більшість топологій за параметром SD (але програють dragonfly), в той же час мережа Dragon de Bruijn забезпечує кращий результат ніж dragonfly. На рис. 5.7 продемонстровано шматок графіку SD для цих двох топологій.

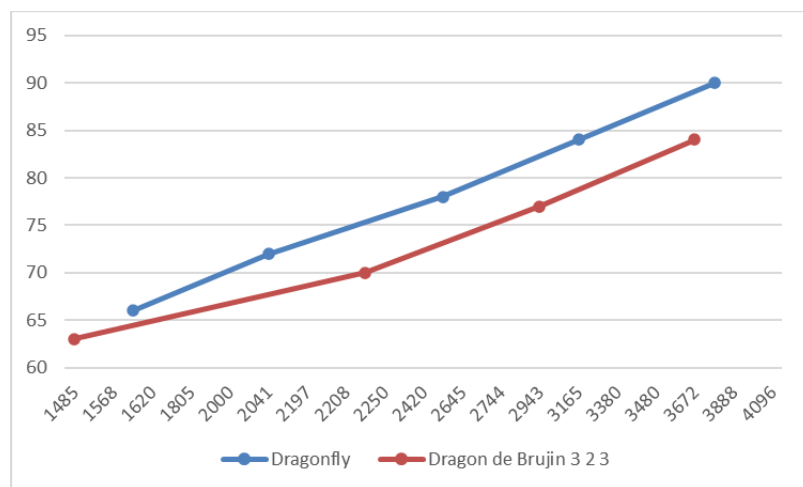


Рис. 5.7. Характеристика SD для мереж dragonfly та Dragon de Bruijn

Як видно із результатів, запропоновані рішення дійсно дозволяють отримати виграш в таких критеріях ефективності як характеристика SD.

5.2.7. Кількісна оцінка ефективності запропонованих рішень

Оскільки поставленою в дисертаційному дослідженні задачею є покращення ефективності, є сенс порівнювати лише найкращі отримані рішення із найкращими класичними, які використовуються в сучасних суперкомп'ютерах та датацентрах.

З точки зору характеристик найкращими запропонованими рішеннями є топологія де Бруйна на кодї типу (4,2) (може розглядатись як БЗМ чи комутована) та топологія Dragon de Bruijn (є комутованою). Серед класичних БЗМ варто виділити тороїдальну топологію, аналогічну до топології суперкомп'ютера Fugaku, а також класичний бінарний гіперкуб, що досі є популярним в деяких системах. Крім того, за визначенням немає сенсу порівнювати малі графи, оскільки мова йде про високопродуктивні КС, які на даний час мають щонайменше тисячу окремих вузлів. Таким чином, порівнюються мають значення для найвищих проаналізованих N .

Серед комутованих мереж розглядатимемо мережу де Бруйна (4, 2) та Dragon de Bruijn, і порівнюватимемо її з жирним деревом, dragonfly та dragonfly+.

Для порівняння відношення характеристик скористаємось формулою 5.1.

$$\delta X (topo) = \frac{X_{default topo} - X_{topo}}{X_{default topo}} \quad (5.1)$$

Таким чином, позитивні зміни (для характеристик, де менші значення кращі) відображатимуться позитивними значеннями, негативні – негативними. Для характеристик, де кращими є більші значення, змінимо значення в чисельнику місцями (таким чином, покращення відображатиметься позитивним числом відносно «класичного» значення). Для характеристики топологічного трафіка, де ідеальним є значення 1, варто виконати приведення $\forall T < 1: T' = 2 - T$. Для зручності читання також є сенс представити зміну характеристики як відсоток.

Результати порівняння методів за характеристиками ефективності представлені в таблиці 5.3.

Табл. 5.3.

Порівняння методів синтезу на основі характеристик отриманих графів

Характеристика	БЗМ		Запропоновані рішення		Комутовані мережі		
	Гіпер-куб	Fugaku	Де Бруйн (4, 2)	DDB	Жирне дерево	dragonfly	dragonfly+
N_{max}	4096	3888	4096	3672	3920	3856	3480
$SD(N_{max})$	196	132	78	84	224	90	120
δSD (де Бруйн)	60%	69.2%			65.1%	15.3%	35%
δSD (DDB)	57.1%	36.4%			62.5%	6.7%	30%
$E(N_{max})$	0.186	0.117	0.238	0.243	0.34	0.359	0.288
δE (де Бруйн)	28%	103.4%			-30%	-33.7%	-17.4%
δE (DDB)	30.6%	108%			-28.5%	-32.3%	-15.6%
T (реальне)	1.000	1.152	0.709	0.780	0.141	0.191	0.408
T (приведене)	1.000	1.152	1.291	1.220	1.859	1.809	1.592
δT (де Бруйн)	-29,1%	-12,1%			30,6%	28,6%	18,9%
δT (DDB)	-22%	-5,9%			34,4%	32,6%	23,3%
L	12	11	6	9	30	28	12
δL (де Бруйн)	-50%	-45,5%			-80%	-78,6%	-50%
δL (DDB)	-25%	-18,2%			-70%	-68,9%	-25%

Аналізуючи результати, варто зазначити, що *топологічна* ефективність тут не є рівною *фактичній* ефективності системи. Це лише параметр, що показує, наскільки короткими є маршрути між вузлами. Логічно, що щільнозв'язані мережі на кшталт dragonfly матимуть набагато вищу топологічну ефективність, ніж інші.

Втім, ціна такої «ефективності» – дуже висока надлишковість, що породжує іншу проблему: мережа технічно не здатна використати ті можливості, що в неї закладені. Таким чином, параметр топологічного трафіку тут дає змогу оцінити, наскільки реально мережа здатна використати ті зв'язки, що в ній є. Невелика

надлишковість (значення менше 1) може свідчити про наявність потенціалу до використання нових зв'язків у випадку виходу з ладу старих, проте занижений трафік – ознака збиткової надлишковості, що не йде на користь системі. Як можна бачити, з точки зору трафіку виграш, який комутовані топології мають від топологічної ефективності, нівелюється складністю (майже неможливістю) її практичного застосування.

Звісно, досягнене підвищення ефективності має і недоліки, один із яких – проблеми з діаметром та середнім діаметром. Хоча у порівнянні з типовими (гіперкуб, тор) графами запропоновані рішення демонструють виграш, втім, сучасні рішення (жирне дерево, dragonfly) забезпечуватимуть кращу середню швидкість передачі даних. Таким чином, виграш в *SD* та топологічному трафіку призводить до втрати іншого критерія – топологічної ефективності. Таким чином, остаточне рішення щодо найкращого рішення має прийматись з урахуванням конкретних вимог до мережі / системи, що проектується.

Втім, запропоновані рішення мають і недоліки, серед яких:

- Гірша локальна зв'язність (до -80% порівняно із жирним деревом), що робить окремі вузли топології більш вразливими.
- Порівняно з dragonfly та жирним деревом – вищий діаметр та нижча топологічна ефективність

5.3. Засіб моделювання відмов у топологіях

Код розробленого програмного засобу наведено в Додатку А (частина 2).

5.3.1. Спосіб моделювання відмов у топологіях

Запропоновано новий спосіб [4] моделювання відмов у топологіях з використанням принципу формування черги відмов (потоків відмов) на основі коефіцієнту посередництва, що передбачає наступні кроки:

1. Задається початковий граф G та число експериментів M .

2. Для всіх вершин топології розраховується імовірність відмови $p_n(f)$, де n – вершина графа, f – число вершин, що вже відмовили. Принцип розрахунку цієї імовірності називається потоком відмов.
3. На основі розрахованої імовірності обирається чергова вершина n , яка видаляється з графа разом зі своїми ребрами.
4. Якщо граф внаслідок відмови втратив зв'язність ($D = \infty$), даний експеримент завершується. Це означає, що топологія «вмирає», тобто, стає несправною.
5. Виконується вимірювання топологічних характеристик і повернення на крок 2.
6. Експеримент повторюється M разів, збираючи, таким чином, статистичні дані.

В кінці експерименту підбивається підсумок і формується статистика: для кожного значення $N' = N - f$ справних вузлів графа формується середня характеристика $X(N') = \frac{\sum_{i=1}^{M'} X_i(N')}{M'}$, де M' – число експериментальних моделей, які «вижили» при f відмовах вузлів, X – довільна топологічна характеристика.

Ключовою особливістю даного методу є можливість безпосереднього визначення відмовостійкості (яка далі по тексту також називається життєздатністю J). Розрахувати її можна за наступною формулою:

$$J(N') = \frac{M}{M'} \quad (5.2)$$

Практичний зміст характеристики J – це імовірність того, що при заданій кількості несправних вузлів f топологія зберігатиме зв'язність (тобто, що система на її основі буде здатною виконувати свої функції). Таким чином, можна розглядати її як *ключовий критерій відмовостійкості*. Іншими критеріями, згаданими раніше, є зміна усереднених характеристик з масштабуванням, що може описуватись наступною формулою:

$$\Delta X(N') = X(N') - X(0) \quad (5.3)$$

Таким чином, позитивні значення свідчитимуть про зростання характеристики відносно початкових значень, негативні – про спадання.

5.3.2. Потік відмов на основі посередництва

Для реальної системи майже неможливо врахувати всі можливі фактори, які впливають на частоту відмов вузлів. Проте можливою є наближена оцінка. Одним із факторів, що впливає на частоту відмов у мережі, є навантаженість елементів. Логічно, що більш навантажені вузли швидше зношуються, через них частіше розповсюджуються помилки і, відповідно, ці елементи є вразливішими за інші.

Найпростішим способом розрахунку навантаженості окремих вузлів на рівні топології є розрахунок числа найкоротших маршрутів, що проходять через кожен конкретний вузол – це так званий коефіцієнт посередництва (англ. *betweenness*). Таким чином, при моделюванні відмов доцільно робити моделювання не чисто випадковим, а керуватись даним параметром для вибору кожної наступної точки відмови. Це – модель потоку відмови на основі посередництва [2].

Різні графи мають свою логіку розподілу посередництва. Так, регулярні топології, на кшталт гіперкуба та dragonfly, мають рівномірний розподіл посередництва. Нерегулярні графи містять точки більшого та меншого навантаження. Наприклад, табл. 5.4 ілюструє мінімальні та максимальні коефіцієнти посередництва для топологій рангу 2 із заданими алфавітами.

Табл. 5.4

Характеристики посередництва для дворангових мереж де Бруйна [2]

Алфавіт	Мінімальне число найкоротших маршрутів через вершину	Максимальне число найкоротших маршрутів через вершину
01T	4 (корені)	10 (інші вузли)
012TZ	24 (корені)	50 (інші вузли)
0123TZE	60 (корені)	122 (інші вузли)

Відповідно, в таких графах є вершини з більшими та меншими шансами вийти із ладу. Подібна нерівномірність може бути як недоліком, так і перевагою: з одного боку проблемою є перевантаження, з іншого – простіше забезпечити резервування для

невеликого числа вразливих вершин або розробити логіку заміщення та обходу несправностей, як це реалізовано в мережі де Бруйна..

5.3.3. Моделювання відмов для топологій БЗМ

Було проведено 1000 симуляцій для наступних графів: (1) - граф де Бруйна рангу 3, код (4, 2), (2) - бінарний гіперкуб рангу 6, (3) - класичний тор рангу 8. Для наступних графів було проведено 100 симуляцій: (4) - граф де Бруйна рангу 3, код (6, 2), (5) - граф де Бруйна рангу 3, код (6, 3), (6) - 3D тор рангу 6, (7) - граф де Бруйна рангу 4, код (4, 2), (8) - бінарний гіперкуб рангу 8. Результати – в табл. 5.5.

Табл. 5.5.

Результати моделювання відмов для БЗМ (повна версія – табл. Д.4)

f, %	Характ.	Група 1 (N = 64)			Група 2 (N = 216)			Група 3 (N=256)	
		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
0%	M	1000	1000	1000	100	100	100	100	100
	N	64	64	64	216	216	216	256	256
	S	10	6	4	16	14	6	11	8
	D	3	6	8	3	3	9	4	8
	\bar{D}	2.170	3.048	4.063	2.342	2.439	4.521	2.867	4.016
	E	0.520	0.401	0.316	0.466	0.448	0.268	0.384	0.295
50 %	M'	944	628	0	100	99	9	89	68
	N	32	32	32	108	108	108	128	128
	ΔS	-2.582	-0.798	X	-3.750	-2.798	-0.111	-1.798	-0.662
	ΔD	+1.968	+2.170	X	+1.800	+2.030	+3.333	+2.652	+1.206
	$\Delta \bar{D}$	+0.331	+0.531	X	+0.300	+0.350	+1.138	+0.546	+0.396
	ΔE	-0.043	-0.035	X	-0.042	-0.044	-0.042	-0.048	-0.023
Останнє “живе” N		2 (3%)	15 (23%)	37 (58%)	35 (16%)	34 (16%)	90 (42%)	52 (20%)	85 (33%)

Порівнюючи результати, варто відмітити, що всі запропоновані топології продемонстрували високу життєздатність, витримуючи значно більше відмов та зазнаючи меншого погіршення характеристик ніж всі розглянуті класичні графи.

Недоліком рішень виявилась їх вартість. Її можна оцінити, виходячи із початкового ступеня: так, оскільки $C = N \cdot S$, при рівних N $C \sim S$, а отже, відношення характеристик можна розрахувати за формулою:

$$\delta C(N) = \frac{S_{\text{де Бруйн}} - S_{\text{гіперкуб}}}{S_{\text{гіперкуб}}} \quad (5.4)$$

Використовуючи формулу (5.4), можемо розрахувати погіршення вартості на забезпечення відмовостійкості. Для найбільш перспективних рішень, а саме (1) та (7), порівняно з еквівалентними гіперкубами (2) та (8) ці значення рівні -50% та -37,5% відповідно (середнє значення: **-43,75%**).

5.3.4. Моделювання відмов для графів комутованих мереж

Аналогічно, комутовані мережі також було проаналізовано по групам. На жаль, через різниці масштабування існують об'єктивні проблеми із підбором таких графів, щоб їх число вузлів N було однаковим. Таким чином, було виділено лише 2 групи:

- Група 1 ($N = 90$, $M = 1000$ експериментів):
 - DDB з групою ранга 2 на основі коду (3, 2)
 - Dragonfly з $h=1$, $c=9$
 - Dragonfly+ з $h=1$, $c=4$, $m=14$
- Група 2 ($N = 272$, $M = 100$ експериментів):
 - DDB з групою ранга 2 на основі коду (4, 2)
 - Dragonfly з $h=1$ та $c=16$ вузлів

Скорочені результати моделювання відмов даних топологій представлено в табл. 5.6. Повна версія - табл. Д.5. Варто зазначити, що жоден із проаналізованих графів не «дожив» до 50% відмов вузлів, тому для аналізу було обрано значення $f=40\%$.

Табл. 5.6

Моделювання відмов для комутованих мереж (повна версія – табл. Д.5)

f, %	Характ.	Група 1 (N=90)			Група 2 (N=272)	
		DDB	Dragonfly	Dragonfly +	DDB	Dragonfly
0 %	M	1000			100	
	S	6	9	15	9	16
	D	5	3	5	5	3
	\bar{D}	3.205	2.618	2.911	3.623	2.771
	E	0.361	0.431	0.393	0.307	0.391
40 %	M'	145	21	0	60	41
	ΔS	-0.586	-1.571	X	-1.333	-3.146
	ΔD	+12.097	+12.571	X	+9.467	+9.220
	$\Delta \bar{D}$	+3.585	+3.884	X	+2.565	+2.718
	ΔE	-0.124	-0.163	X	-0.096	-0.137
Останнє “живе” N		49	54	77	142	153
Частка (%) мережі		54%	60%	86%	52%	56%

В даному випадку вартість графа ϵ , навпаки, меншою. Порівнявши DDB та Dragonfly, маємо +33% в першій групі та +43,8% в другій (середнє **+38,4%**). Втім, дані графі є гіршими за діаметром: **-66%** в обох групах. Топологічна ефективність при цьому є гіршою на -16,2% та -21,5% (середнє **-18,9%**).

5.3.5. Порівняння на основі моделювання відмов

Згідно до даних таблиць 5.4 та 5.5, було виконано аналіз окремо для графів БЗМ та для графів комутованих мереж. Для категорії БЗМ було обрано наступні графи: гіперкуби на 64 та 256 вершин в якості найкращих стандартних рішень, граfi де Бруйна на основі коду (4,2) на 64 вершини як найбільш живуче рішення та на 256 вершин як середнє за розміром та характеристиками рішення. Для комутованих топологій розглянуто 4 граfi (всі, що є в табл. 5.6 крім dragonfly+, яка показала

високу вразливість до відмов). Табл. 5.7 показує поведінку БЗМ в умовах відмови, табл. 5.8 – поведінку комутованих мереж.

Табл. 5.7

Порівняння відмовостійкості БЗМ на основі посередництва

Мережі	Стандартні		Запропоновані				Середній ефект методу
	НС n64	НС n256	DB3 (4, 2)	Ефект	DB4 (4, 2)	Ефект	
25 %	99,7%	100%	100 %	+0,3% / 0%	100%	+0,3% / +0%	+0,15%
50%	62,8%	68%	94,4%	+31,6% / +26,4%	89%	+26,2% / +21%	+26,3%
75%	1,6%	0%	16,9%	+15,3% / +16,9%	2%	+0,4% / +2%	+8,65%

Як видно із таблиці, в середньому життєздатність зросла на +0,15% при 25% відмов, на +26,3% при 50% відмов, на +8,65% при 75% відмов.

Табл. 5.8

Порівняння методів синтезу на основі моделювання відмов (для комутованих мереж)

Відмов, %	Стандартні		Комутовані мережі				Середній ефект методу
	Dfly n90	Dfly n272	DDB n90	Ефект	DDB n272	Ефект	
20 %	99,2%	100%	98,8%	-0,4% / +1,2%	100%	+0,8% / 0%	+0,4%
30%	79,4%	94%	81,9%	2,5% / -12,1%	96%	+16,6% / +2%	+2,25%
40%	2,1%	41%	14,5%	+12,4% / -26,5%	60%	+57,9% / +19%	+15,7%

Таким чином, моделювання для комутованих мереж показало вигреш запропонованих рішень на +0,4% при 20% відмов, на +2,25% при 30% відмов і на +15,7% при 40% відмови системи.

Висновки до розділу 5

Було проведено дослідження запропонованих методів з використанням розробленого засобу моделювання топологічних характеристик. Отримано наступні результати:

- Графи, синтезовані на основі надлишкового коду, забезпечують кращі характеристики ніж типові топології. З точки зору характеристики SD найкращі результати показав граф де Бруйна на основі коду (4, 2), демонструючи $SD = 78$ при $N = 4096$ (для гіперкуба того ж розміру це 144, для топології суперкомп'ютера Fugaku – 132 при $N=3888$, для dragonfly з $c=16, h=15$ – 90 при $N = 3856$).
- Показники зв'язності та топологічної ефективності запропонованих рішень загалом є кращими ніж у класичних графів ($E = 0.238$ при $N=4096$ у де Бруйна (4, 2) проти 0.186 у гіперкуба), проте гіршими, ніж у сучасних популярних рішень (для жирного дерева $E = 0.34$ при $N=3920$).
- Графи, синтезовані на основі методу ієрархічного синтезу, показують в середньому кращі характеристики ніж типові мережі. Деревовидні граfi та Hyper de Bruijn загалом мають посередні характеристики, втім топологія Dragon de Bruijn за показником SD переважає всі інші граfi, в т.ч. незначно переважає dragonfly (84 при $N=3672$ проти 90 при $N = 3856$).

Також було проведено дослідження поведінки топологій в умовах наростаючого числа відмов з використанням розробленого програмного засобу моделювання відмов. Наступні результати було отримано:

- Всі дебруйнівські граfi ілюструють високу стійкість до відмов, забезпечуючи менші падіння характеристик та вищу відмовостійкість (всі класичні мережі виявились несправними при 77% відмов, в той час як топологія де Бруйна на основі коду (4, 2) на одному з експериментів досягла точки в 97% відмов).
- Топологія Dragon de Bruijn також ілюструє помірно гарні результати. Хоча її максимальна живучість і не є вражаючою, проте вона переважає як класичний dragonfly (вихід з ладу при 52% та 54% робочих вузлах проти

56%), так і dragonfly+ (всі 1000 експериментальних моделей вийшли з ладу при 86% робочих вузлів).

Запропонований метод має і недоліки, із яких:

- Гірша локальна зв'язність, що мало зростає з масштабуванням: 6 у де Бруїна на коді (4, 2), 9 – у dragon de Bruijn, порівняно з 11 у топології Fugaku, 28 у жирного дерева і 30 у dragonfly.
- Порівняно з графами БЗМ – гірший топологічний трафік: -29,1% порівняно з гіперкубом, -12,1% порівняно з топологією Fugaku для топології на основі коду (4, 2).
- Гірша топологічна ефективність (не плутати з ефективністю як такою!) у порівнянні класичними комутованими топологіями: -28,5% в порівнянні з жирним деревом -32,3% порівняно з dragonfly, -15,6% порівняно з dragonfly+.

Зазначені недоліки згладжуються тим, що:

- Локальна зв'язність є ненадійним показником відмовостійкості, що ілюструє дуже обмежений випадок (що доводиться моделюванням відмов)
- Топологічний трафік запропонованих рішень переважно лежить в межах від 0,6 до 1,0, що нормально для графів із невеликою надлишковістю.
- Висока топологічна ефективність класичних рішень нівелюється занижким топологічним трафіком, що робить ці теоретичні показники недосяжними і, таким чином, їх реальна ефективність буде значно меншою. В той же час запропоновані рішення цілком можуть використати наявну ефективність як для взаємодії вузлів, так і для нівелювання негативного впливу відмови.

ВИСНОВКИ

В дисертаційній роботі виконано теоретичне обґрунтування та одержано нове рішення задачі побудови топологій комп'ютерних систем, які забезпечують високу відмовостійкість та ефективність їх функціонування.

Основні наукові та практичні результати полягають у наступному:

1. Розроблено нову математичну модель топології на основі надлишкового коду, що відрізняється від існуючих моделей використанням алфавіту, основи числення та довжини коду для визначення кількості альтернативних представлень довільного числа в заданій системі числення, та дозволяє прогнозувати максимальну кількість вершин з однаковим номером у графі, кількість вершин з унікальними (не-надлишковими) номерами. За допомогою даної моделі **було доведено**, що форма розподілу кількості альтернативних представлень, а також похідні від неї характеристики надлишкового графа не залежать від вмісту алфавіту, а лише від його потужності, довжини коду та основи числення. Визначено, що код $(4, 2)$ має кращі характеристики ніж надлишкове бінарне представлення: при довжині коду 8 його **максимальне число представлень 128 (проти 34, виграш +94), а число унікальних представлень – 4 (проти 17, виграш -13)**.
2. Набув розвитку метод синтезу відмовостійких топологій на основі надлишкового коду, що відрізняється від існуючих використанням кодових перетворень, в тому числі послідовностей де Бруїна, в надлишкових системах числення та створенням нових зв'язків у таких топологіях за допомогою перетворень заміщення над кодами, які описують індекс альтернативного представлення в багатовимірній матриці надлишкових представлень, що дозволяє синтезувати відмовостійкі топології заданого порядку, в тому числі з імпліцитними кластерами. За допомогою даного методу **було синтезовано нові топології**, такі як графи де Бруїна на основі кодів $(4, 2)$, $(6, 2)$ та $(6, 3)$. Експериментальне дослідження показало, що топологія на основі коду $(4, 2)$ забезпечує **мультиплікативний критерій**

ступеня та діаметру (*SD*) на **60% кращий** ніж у гіперкуба та на **69,2% кращий** ніж у топології суперкомп'ютера Fugaku. Проте локальна зв'язність при цьому на **50% гірша** ніж у гіперкуба та на **45,5% гірша** - ніж у топології Fugaku.

3. Запропоновано новий метод масштабування ієрархічних топологій, що відрізняється від існуючих використанням декартового добутку, деревовидних структур та рекурентного вкладення кластерів, що дозволяє поєднати відмовостійкі топології, синтезовані на основі надлишкового коду, із класичними топологіями, такими як гіперкуб та dragonfly. За допомогою запропонованого методу було отримано нові топології, такі як hyper de Bruijn, dragon de Bruijn та дерева з близьким та комбінованим видом групування. Експериментальне дослідження показало, що топологія dragon de Bruijn на основі надлишкового дебруйнавського кластеру в 2-розрядному коді (4, 2) забезпечує **критерій *SD* на 62,5% краще** ніж у жирного дерева, **на 36,4% краще** ніж у топології Fugaku, **на 6,7% краще** ніж у dragonfly, **на 30% краще** ніж у dragonfly+. При цьому локальна зв'язність гірша на **70%** для жирного дерева, **на 18,2%** для графа Fugaku, **на 68,9%** ніж у dragonfly та **на 25%** ніж у dragonfly+.
4. Запропоновано новий спосіб формування імпліцитних кластерів в надлишкових топологіях, що відрізняється від існуючих використанням спеціальної багатовимірної матриці надлишкових представлень та кодування індексів в спеціальній системі числення та дозволяє формувати ребра між такими вершинами для топологій на основі кодів із певними співвідношеннями потужності алфавіту та основи числення. На основі даного способу було **розроблено інструментальний засіб для моделювання характеристик топологій на основі бібліотеки NetworkX**. Експериментальне дослідження з використанням розробленого засобу, показало, що графи на основі надлишкового коду, такі як граф надлишкового де Бруйна на коді (4, 2) забезпечують **кращу топологічну ефективність *E*** ніж гіперкуб (**+28%**) та граф Fugaku (**+103%**), проте програють по даному

параметру жирному дереву (-30%) та dragonfly (-33,7%). При цьому їх **топологічний графік T** є кращим ніж у жирного дерева (+30,6%) та dragonfly (+28,6%). Для топології dragon de Bruijn, відповідно, було проведено аналогічне порівняння з графом Fugaku ($\delta E = +30,6\%$, $\delta T = -12,1\%$), жирними деревом ($\delta E = -28,5\%$, $\delta T = +34,4\%$) та dragonfly ($\delta E = -32,3\%$, $\delta T = +32,6\%$).

5. Запропоновано новий спосіб моделювання відмов в топологіях, що відрізняється від існуючих використанням різних підходів до випадкового формування черги відмов, в тому числі з урахуванням коефіцієнту посередництва, та дозволяє при заданій кількості відмов вузлів аналізувати імовірність розриву зв'язності графа, підрахувати топологічні характеристики та їх зміну відносно початкового (безвідмовного) стану топології. На його основі було **розроблено інструментальний засіб для моделювання відмов в топологіях**. Експериментальне дослідження показало, що граф де Бруїна на коді (4, 2) забезпечує **життєздатність J** (імовірність збереження працездатності системи при заданій f кількості вузлів, що відмовили) в середньому **на 26,3% кращу** ніж гіперкуб при 50% відмов, проте його недоліком є **вартість, гірша на 43,75%**. Для топології dragon de Bruijn виконано порівняння з dragonfly, що має **вартість, кращу на 38,4%** і при 40% відмов дозволяє покращити **життєздатність на +15,7%**, проте його недоліками є гірший діаметр (-66%) та гірша топологічна ефективність (-18,9%).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2022). Extended DragonDeBruijn topology synthesis method. *International Journal of Computer Network and Information Security*, 9(6), 23.
2. Volokyta, A., Loutskii, H., Honcharenko, O., Cherevatenko, O., Rusinov, V., Kulakov, Y., & Tsybulia, S. (2023). Fault Tolerance Exploration and SDN Implementation for de Bruijn Topology based on betweenness Coefficient. *Computer Network and Information Security*, 5 (pp. 1-17).
3. О Гончаренко, О., & Череватенко, О. (2021). СПОСОБИ МУЛЬТИКАНАЛЬНОЇ МАРШРУТИЗАЦІЇ В МЕРЕЖАХ НАДЛИШКОВОГО ДЕ БРУЙНА. *Технічні науки та технології*, (2 (24)), 123-130.
4. Гончаренко, О., & Волокита, А. (2024). МЕТОД СИНТЕЗУ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ З ІМПЛІЦИТНИМИ КЛАСТЕРАМИ НА ОСНОВІ ПЕРЕТВОРЕНЬ ДЕ БРУЙНА В НАДЛИШКОВИХ СИСТЕМАХ ЧИСЛЕННЯ. *Проблеми інформатизації та управління*, (4 (80)), 20-27.
5. Rusinov, V., Honcharenko, O., Volokyta, A., Loutskii, H., Pustovit, O., & Kyrianov, A. (2023, March). Methods of Topological Organization Synthesis Based on Tree and Dragonfly Combinations. In *International Conference on Computer Science, Engineering and Education Applications* (pp. 472-485). Cham: Springer Nature Switzerland.
6. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2021). Topology synthesis method based on excess de Bruijn and dragonfly. In *Advances in Computer Science for Engineering and Education IV* (pp. 315-325). Springer International Publishing.
7. Loutskii, H., Volokyta, A., Rehida, P., Honcharenko, O., & Thinh, V. D. (2021). Method for synthesis scalable fault-tolerant multi-level topological organizations based on excess code. In *Advances in Computer Science for Engineering and Education III 3* (pp. 350-362). Springer International Publishing.
8. Loutskii, H., Volokyta, A., Rehida, P., Honcharenko, O., Ivanishchev, B., & Kaplunov, A. (2019, December). Increasing the fault tolerance of distributed systems for the

Hyper de Bruijn topology with excess code. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)* (pp. 1-6). IEEE.

9. Honcharenko, O., Volokyta, A., & Loutskii, H. (2024, August). Method of fault tolerant routing in distributed systems based on non-binary de Bruijn topology. In *The International Conference on Security, Fault Tolerance, Intelligence*.

10. November 2023 | TOP500.Home - | TOP500.
URL: <https://www.top500.org/lists/top500/2023/11/> (date of access: 10.12.2024).

11. HPCG - November 2023 | TOP500.Home - | TOP500.
URL: <https://www.top500.org/lists/hpcg/2023/11/> (date of access: 10.12.2024).

12. Torell, W., & Avelar, V. (2004). Mean time between failure: Explanation and standards. white paper, 78, 6-7.

13. Duer, S., Woźniak, M., Paś, J., Zajkowski, K., Bernatowicz, D., Ostrowski, A., & Budniak, Z. (2023). Reliability Testing of Wind Farm Devices Based on the Mean Time between Failures (MTBF). *Energies*, 16(4), 1659.

14. Egwutuoha, I. P., Levy, D., Selic, B., & Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65, 1302-1326.

15. Amdahl, G. M. (1967, April). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485).

16. Dongarra, J., Herault, T., & Robert, Y. (2015). Fault tolerance techniques for high-performance computing (pp. 3-85). Springer International Publishing.

17. Heroux, M. A., Dongarra, J., & Luszczek, P. (2013). HPCG benchmark technical specification (No. SAND2013-8752). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

18. Cook, J. (2011). Supercomputers and supercomputing. In *Visual Analytics and Interactive Technologies: Data, Text and Web Mining Applications* (pp. 282-294). IGI Global.

19. Dongarra, J., Sterling, T., Simon, H., & Strohmaier, E. (2005). High-performance computing: clusters, constellations, MPPs, and future directions. *Computing in Science & Engineering*, 7(2), 51-59.
20. Resch, M. M., & Gabriel, E. (2011). Supercomputers in grids. In *Cloud, Grid and High Performance Computing: Emerging Applications* (pp. 1-9). IGI Global.
21. Kozielski, S., & Mrozek, D. (2020). Development of high performance computing systems. In *Computer Networks: 27th International Conference, CN 2020, Gdańsk, Poland, June 23–24, 2020, Proceedings 27* (pp. 52-63). Springer International Publishing
22. Yin, F., & Shi, F. (2022). A comparative survey of big data computing and HPC: From a parallel programming model to a cluster architecture. *International Journal of Parallel Programming*, 50(1), 27-64.
23. Pattnaik, A., Tang, X., Kayiran, O., Jog, A., Mishra, A., Kandemir, M. T., ... & Das, C. R. (2019, June). Opportunistic computing in gpu architectures. In *Proceedings of the 46th international symposium on computer architecture* (pp. 210-223).
24. Gomes, E., & Dantas, M. A. R. (2014, August). An advance reservation mechanism to enhance throughput in an opportunistic high performance computing environment. In *2014 IEEE 13th International Symposium on Network Computing and Applications* (pp. 221-228). IEEE.
25. Garcés, N., Sotelo, G. A., Villamizar, M., & Castro, H. (2012, November). Running mpi applications over opportunistic cloud infrastructures. In *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (pp. 309-314). IEEE.
26. Charlot, M., De Fabritis, G., de Lomana, A. G., Gomez-Garrido, A., Groen, D., Guylas, L., ... & Villa-Freixa, J. (2007). The QosCosGrid project: Quasi-opportunistic supercomputing for complex systems simulations. Description of a general framework from different types of applications. In *Ibergrid 2007 conference, Centro de Supercomputacion de Galicia (GESGA)*.
27. Kravtsov, V., Carmeli, D., Dubitzky, W., Orda, A., Schuster, A., Silberstein, M., & Yoshpa, B. (2008). Quasi-opportunistic supercomputing in grid environments. In

Algorithms and Architectures for Parallel Processing: 8th International Conference, ICA3PP 2008, Cyprus, June 9-11, 2008 Proceedings 8 (pp. 233-244). Springer Berlin Heidelberg.

28. Liu, C. Y., Shie, M. R., Lee, Y. F., Lin, Y. C., & Lai, K. C. (2014, May). Vertical/horizontal resource scaling mechanism for federated clouds. In 2014 International Conference on Information Science & Applications (ICISA) (pp. 1-4). IEEE.

29. Millnert, V., & Eker, J. (2020, December). HoloScale: horizontal and vertical scaling of cloud resources. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)* (pp. 196-205). IEEE.

30. *20 Years of Excellence in Computational Science* (Annual report 2011-2012). (2012). Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/wp-content/uploads/2010/03/AR2012_Final.pdf

31. Barker, K. J., Davis, K., & Kerbyson, D. J. (2009, May). Performance modeling in action: Performance prediction of a cray xt4 system during upgrade. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (pp. 1-8). IEEE.

32. Joubert, W., & Su, S. Q. (2012, June). An analysis of computational workloads for the ORNL Jaguar system. In *Proceedings of the 26th ACM international conference on Supercomputing* (pp. 247-256).

33. Bland, B. (2012, November). Titan-early experience with the titan system at oak ridge national laboratory. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (pp. 2189-2211). IEEE.

34. Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., ... & Shimizu, T. (2020, November). Co-design for a64fx manycore processor and" fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-15). IEEE.

35. Matsuoka, S. (2021, June). Fugaku and A64FX: the first exascale supercomputer and its innovative arm CPU. In *2021 Symposium on VLSI Circuits* (pp. 1-3). IEEE.

36. Popov, G., Mastorakis, N., & Mladenov, V. (2010). Calculation of the acceleration of parallel programs as a function of the number of threads. *Proceedings of the ICCOMP, 10*, 411-414.

37. Zhu, Y., Taylor, M., Baden, S. B., & Cheng, C. K. (2008, November). Advancing supercomputer performance through interconnection topology synthesis. In *2008 IEEE/ACM International Conference on Computer-Aided Design* (pp. 555-558). IEEE.
38. Bokhari, S. H. (2012). *Assignment problems in parallel and distributed computing* (Vol. 32). Springer Science & Business Media.
39. Bokhari, S. H. (1988). Partitioning problems in parallel, pipeline, and distributed computing. *IEEE transactions on Computers*, 37(1), 48-57.
40. Fox, G. C., Williams, R. D., & Messina, G. C. (2014). *Parallel computing works!*. Elsevier.
41. Barborak, M., Dahbura, A., & Malek, M. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys (CSur)*, 25(2), 171-220.
42. Haider, S., & Nazir, B. (2016). Fault tolerance in computational grids: perspectives, challenges, and issues. *SpringerPlus*, 5, 1-20.
43. Kuhl, J. G., & Reddy, S. M. (1986). Fault-tolerance considerations in large, multiple-processor systems. *Computer*, 19(03), 56-67.
44. Domke, J., Matsuoka, S., Radanov, I., Tsushima, Y., Yuki, T., Nomura, A., ... & Dubé, N. (2019, August). The first supercomputer with hyperx topology: A viable alternative to fat-trees?. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)* (pp. 1-4). IEEE.
45. Mollah, M. A., Faizian, P., Rahman, M. S., Yuan, X., Pakin, S., & Lang, M. (2018, May). A comparative study of topology design approaches for HPC interconnects. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (pp. 392-401). IEEE.
46. Andújar, F. J., Coll, S., Alonso, M., Martínez, J. M., López, P., Sánchez, J. L., & Alfaro, F. J. (2023). Energy efficient HPC network topologies with on/off links. *Future Generation Computer Systems*, 139, 126-138.
47. Wang, F., Ramamritham, K., & Stankovic, J. A. (1995). Determining redundancy levels for fault tolerant real-time systems. *IEEE Transactions on Computers*, 44(2), 292-301.

48. Xu, C., Holzemer, M., Kaul, M., Soto, J., & Markl, V. (2017). On fault tolerance for distributed iterative dataflow processing. *IEEE Transactions on Knowledge and Data Engineering*, 29(8), 1709-1722.
49. Vedavalli, P., & Ch, D. (2022). Data Recovery Approach for Fault-Tolerant IoT Node. *International Journal of Advanced Computer Science and Applications*, 13(1).
50. Treaster, M. (2005). A survey of fault-tolerance and fault-recovery techniques in parallel systems. *arXiv preprint cs/0501002*.
51. Castro, P. F., D'Argenio, P. R., Demasi, R., & Putruele, L. (2019, April). Measuring masking fault-tolerance. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 375-392). Cham: Springer International Publishing.
52. Putruele, L., Demasi, R., Castro, P. F., & D'Argenio, P. R. (2022, March). MaskD: a tool for measuring masking fault-tolerance. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 396-403). Cham: Springer International Publishing.
53. Riva, O., Nzouonta, J., & Borcea, C. (2008, July). Context-aware fault tolerance in migratory services. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services* (pp. 1-12).
54. Pickartz, S., Gad, R., Lankes, S., Nagel, L., Süß, T., Brinkmann, A., & Krempel, S. (2014). Migration techniques in HPC environments. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20* (pp. 486-497). Springer International Publishing.
55. Bosilca, G., Delmas, R., Dongarra, J., & Langou, J. (2009). Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4), 410-416.
56. Liu, N., Haider, A., Sun, X. H., & Jin, D. (2015, June). Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (pp. 199-210).

57. Liu, N., Haider, A., Jin, D., & Sun, X. H. (2017). Modeling and simulation of extreme-scale fat-tree networks for HPC systems and data centers. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 27(2), 1-23.
58. Kim, J., Dally, W. J., Scott, S., & Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3), 77-88.
59. McGlohon, N., Carothers, C. D., Hemmert, K. S., Levenhagen, M., Brown, K. A., Chunduri, S., & Ross, R. B. (2021, November). Exploration of congestion control techniques on dragonfly-class hpc networks through simulation. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (pp. 40-50). IEEE.
60. Ajima, Y., Sumimoto, S., & Shimizu, T. (2009). Tofu: A 6D mesh/torus interconnect for exascale computers. *Computer*, 42(11), 36-40.
61. Bhatele, A., Jain, N., Isaacs, K. E., Buch, R., Gamblin, T., Langer, S. H., & Kale, L. V. (2014, December). Optimizing the performance of parallel applications on a 5D torus via task mapping. In *2014 21st International Conference on High Performance Computing (HiPC)* (pp. 1-10). IEEE.
62. Li, K., Qiu, Y., Jiang, C., Malawski, M., & Nabrzyski, J. (2020, December). Improving system utilization on wireless HPC systems with torus interconnects. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (pp. 60-69). IEEE.
63. Domke, J., Matsuoka, S., Ivanov, I. R., Tsushima, Y., Yuki, T., Nomura, A., ... & Dubé, N. (2019, November). Hyperx topology: First at-scale implementation and comparison to the fat-tree. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-23).
64. Domke, J., Matsuoka, S., Radanov, I., Tsushima, Y., Yuki, T., Nomura, A., ... & Dubé, N. (2019, August). The first supercomputer with hyperx topology: A viable alternative to fat-trees?. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)* (pp. 1-4). IEEE.

65. Adamowski, J., law Bednarek, S., & Iomiej Szafran, B. (2005). Quantum Computing with Quantum Dots. *Schedae Informaticae*, 14, 95-111.
66. Lanzagorta, M., & Uhlmann, J. (2008, April). Is quantum parallelism real?. In *quantum information and computation VI* (Vol. 6976, pp. 172-178). SPIE.
67. Korolija, N., Popović, J., Cvetanović, M., & Bojović, M. (2017). Dataflow-based parallelization of control-flow algorithms. In *Advances in computers* (Vol. 104, pp. 73-124). Elsevier.
68. Mazumdar, S., Scionti, A., Zuckerman, S., & Portero, A. (2023). NoC-based hardware software co-design framework for dataflow thread management. *The Journal of Supercomputing*, 79(16), 17983-18020.
69. Zhao, J., Korpan, B., Gonzalez, A., & Asanovic, K. (2020, May). Sonicboom: The 3rd generation Berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V* (Vol. 5, pp. 1-7).
70. Cardoso, D. M., Tonetto, R., Brandalero, M., Nazar, G., Beck, A. C., & Azambuja, J. R. (2019). Exploring the limitations of dataflow shift techniques in out-of-order superscalar processors. *Microelectronics Reliability*, 100, 113406.
71. Vestias, M., & Neto, H. (2014, September). Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1-6). IEEE.
72. Liu, B., Zydek, D., Selvaraj, H., & Gewali, L. (2012, December). Accelerating high performance computing applications: Using cpus, gpus, hybrid cpu/gpu, and fpgas. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies* (pp. 337-342). IEEE.
73. Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., ... & Kepner, J. (2018). Scalable system scheduling for HPC and big data. *Journal of Parallel and Distributed Computing*, 111, 76-92.
74. Fan, Y., Lan, Z., Rich, P., Allcock, W. E., Papka, M. E., Austin, B., & Paul, D. (2019, June). Scheduling beyond CPUs for HPC. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (pp. 97-108).

75. Ashraf, M. U., Eassa, F. A., Osterweil, L. J., Albeshri, A. A., Algarni, A., & Ilyas, I. (2021). AAP4All: An Adaptive Auto Parallelization of Serial Code for HPC Systems. *Intelligent Automation & Soft Computing*, 30(2).
76. Basthikodi, M., Faizabadi, A. R., & Ahmed, W. (2019). HPC Based Algorithmic Species Extraction Tool for Automatic Parallelization of Program Code. *International Journal of Recent Technology and Engineering*, 8, 1004-1009.
77. Ramirez-Gargallo, G., Garcia-Gasulla, M., & Mantovani, F. (2019, May). TensorFlow on state-of-the-art HPC clusters: a machine learning use case. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (pp. 526-533). IEEE.
78. Moiz, A. A., Pal, P., Probst, D., Pei, Y., Zhang, Y., Som, S., & Kodavasal, J. (2018). A machine learning-genetic algorithm (ML-GA) approach for rapid optimization using high-performance computing. *SAE International journal of commercial vehicles*, 11(2018-01-0190), 291-306.
79. Munawar, A., Wahib, M., Munetomo, M., & Akama, K. (2008, September). A survey: Genetic algorithms and the fast evolving world of parallel computing. In *2008 10th IEEE International Conference on High Performance Computing and Communications* (pp. 897-902). IEEE.
80. Koren, I., & Krishna, C. M. (2020). *Fault-tolerant systems*. Morgan Kaufmann.
81. Nelson, V. P. (1990). Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7), 19-25.
82. Radojkovic, P., Marazakis, M., Carpenter, P., Jeyapaul, R., Gizopoulos, D., Schulz, M., ... & Unsal, O. (2020). *Towards resilient EU HPC systems: A blueprint* (Doctoral dissertation, European HPC resilience initiative).
83. Jia, J., Liu, Y., Zhang, G., Gao, Y., & Qian, D. (2023). Software approaches for resilience of high performance computing systems: a survey. *Frontiers of Computer Science*, 17(4), 174105.
84. Gray, J. (1990). A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on reliability*, 39(4), 409-418.

85. Charng-Da, L. (2005). Scalable diskless checkpointing for large parallel systems [Ph. D. thesis].
86. Oliner, A., & Stearley, J. (2007, June). What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)* (pp. 575-584). IEEE.
87. Di Martino, C., Kalbarczyk, Z., Iyer, R. K., Baccanico, F., Fullop, J., & Kramer, W. (2014, June). Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (pp. 610-621). IEEE.
88. Oppenheimer, D., & Patterson, D. A. (2002). Architecture and dependability of large-scale internet services. *IEEE Internet Computing*, 6(5), 41-49.
89. Arifeen, T., Hassan, A. S., & Lee, J. A. (2020). Approximate triple modular redundancy: A survey. *IEEE Access*, 8, 139851-139867.
90. Lockner, P. A., & Hancock, P. D. (1990). Redundancy in fault-tolerant systems. *Mechanical Engineering-CIME*, 112(5), 76-84.
91. Shobana, M., & Senthil, M. S. (2015, March). Reconfigurable data processing using duplex fault tolerance system. In *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)* (pp. 1-5). IEEE.
92. Gray, J., & Siewiorek, D. P. (1991). High-availability computer systems. *Computer*, 24(9), 39-48.
93. Malipatil, S., Gour, A., & Maheshwari, V. (2020). Design & implementation of reconfigurable adaptive fault tolerant system for ALU. *International Journal of Electrical Engineering and Technology*, 11(9), 01-07.
94. Rushby, J. (1991). Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: Second International Symposium Nijmegen, The Netherlands, January 8-10, 1992 Proceedings 2* (pp. 237-257). Springer Berlin Heidelberg.
95. Mitra, S., Saxena, N. R., & McCluskey, E. J. (2000, April). Fault escapes in duplex systems. In *Proceedings 18th IEEE VLSI Test Symposium* (pp. 453-458). IEEE.

96. Kubalík, P., Kvasnicka, J., & Kubátová, H. (2007, April). Fault injection and simulation for fault tolerant reconfigurable duplex system. In *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems* (pp. 1-4). IEEE.
97. Yin, M. L., Blough, D. M., & Bic, L. (1992). Recovery modeling in performability analysis for multi-computer systems.
98. Wang, Z., Xu, X., & Du, Y. (2023). Assessing Transportation Network Redundancy by Integrating Route-Diversity and Spare-Capacity Dimensions. *Available at SSRN 4368082*.
99. Kellner, A., Kolinowitz, H. J., & Urban, G. (2001, March). A novel approach to fault tolerant computing [in space systems]. In *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)* (Vol. 3, pp. 3-1127). IEEE.
100. Lyons, R. E., & Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2), 200-209.
101. Hudson, S., Shyama Sundar, R. S., & Koppu, S. (2018). Fault control using triple modular redundancy (TMR). In *Progress in Computing, Analytics and Networking: Proceedings of ICCAN 2017* (pp. 471-480). Springer Singapore.
102. Driscoll, K., Hall, B., Sivencrona, H., & Zumsteg, P. (2003, September). Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security* (pp. 235-248). Berlin, Heidelberg: Springer Berlin Heidelberg.
103. Veeravalli, V. S. (2009, March). Fault tolerance for arithmetic and logic unit. In *IEEE Southeastcon 2009* (pp. 329-334). IEEE.
104. Babić, I., Miljković, A., Čabarkapa, M., Nikolić, V., Đorđević, A., Randelović, M., & Randelović, D. (2021). Triple modular redundancy optimization for threshold determination in intrusion detection systems. *Symmetry*, 13(4), 557.
105. Sousa, D. (1978). Sift-out modular redundancy. *IEEE Transactions on Computers*, 100(7), 624-627.
106. Shawkat, S., Alkady, G. I., Amer, H. H., Daoud, R. M., & Adly, I. (2020, December). Extensible fault secure Sift-out technique for FPGA-based applications. In *2020*

15th International Conference on Computer Engineering and Systems (ICCES) (pp. 1-6). IEEE.

107. Avizienis, A. (1995). The methodology of n-version programming. *Software fault tolerance*, 3, 23-46.

108. Avizienis, A. (1977). On the implementation of N-version programming for software fault tolerance during execution. In *Proc. Compsac* (pp. 149-155).

109. Gujarati, A., Gopalakrishnan, S., & Pattabiraman, K. (2020, October). New wine in an old bottle: N-version programming for machine learning components. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 283-286). IEEE.

110. Barbirotta, M., Cheikh, A., Mastrandrea, A., Menichelli, F., & Olivieri, M. (2022). Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors. *IEEE Access*, 10, 126074-126088.

111. Barbirotta, M., Cheikh, A., Mastrandrea, A., Menichelli, F., Ottavi, M., & Olivieri, M. (2022). Evaluation of dynamic triple modular redundancy in an interleaved-multi-threading risc-v core. *Journal of Low Power Electronics and Applications*, 13(1), 2.

112. Garcia-Astudillo, L. A., Entrena, L., Lindoso, A., Martín, H., Martín-Holgado, P., & Garcia-Valderas, M. (2022). Analyzing reduced precision triple modular redundancy under proton irradiation. *IEEE Transactions on Nuclear Science*, 69(3), 470-477.

113. Almukhaizim, S., & Makris, Y. (2003, November). Fault tolerant design of combinational and sequential logic based on a parity check code. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems* (pp. 563-570). IEEE.

114. Mejia, A., Flich, J., Duato, J., Reinemo, S. A., & Skeie, T. (2006, April). Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium* (pp. 10-pp). IEEE.

115. Bossard, A., & Kaneko, K. (2020). Cluster-fault tolerant routing in a torus. *Sensors*, 20(11), 3286.

116. Shu, C., Wang, Y., Fan, J., & Zhang, H. (2021, September). Fault-tolerant routing of generalized hypercubes under 3-component connectivity. In *2021 IEEE Intl Conf*

on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom) (pp. 1320-1327). IEEE.

117. Gomez, M. E., Nordbotten, N. A., Flich, J., Lopez, P., Robles, A., Duato, J., ... & Lysne, O. (2006). A routing methodology for achieving fault tolerance in direct networks. *IEEE transactions on Computers*, 55(4), 400-415.

118. Li, X. Y., Lin, W., Liu, X., Lin, C. K., Pai, K. J., & Chang, J. M. (2021). Completely independent spanning trees on BCCC data center networks with an application to fault-tolerant routing. *IEEE Transactions on Parallel and Distributed Systems*, 33(8), 1939-1952.

119. Aspnes, J., Diamadi, Z., & Shah, G. (2002, July). Fault-tolerant routing in peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (pp. 223-232).

120. Alwan, H., & Agarwal, A. (2009, June). A survey on fault tolerant routing techniques in wireless sensor networks. In *2009 third international conference on sensor technologies and applications* (pp. 366-371). IEEE.

121. Alam, M. S., & Melhem, R. G. (1991). An efficient modular spare allocation scheme and its application to fault tolerant binary hypercubes. *IEEE Transactions on Parallel & Distributed Systems*, 2(01), 117-126.

122. Yang, C. S., Zu, L. P., & Wu, Y. N. (1994). A reconfigurable modular fault-tolerant hypercube architecture. *IEEE Transactions on Parallel and Distributed Systems*, 5(10), 1018-1032.

123. Bruck, J., Cypher, R., & Ho, C. (1993). Fault-tolerant meshes and hypercubes with minimal numbers of spares. *IEEE Transactions on Computers*, 42(9), 1089-1104.

124. Bhanu, P. V., Govindan, R., Kattamuri, P., Soumya, J., & Cenkeramaddi, L. R. (2021). Flexible spare core placement in torus topology based nocs and its validation on an fpga. *IEEE Access*, 9, 45935-45954.

125. Hori, A., Yoshinaga, K., Herault, T., Bouteiller, A., Bosilca, G., & Ishikawa, Y. (2020). Overhead of using spare nodes. *The International Journal of High Performance Computing Applications*, 34(2), 208-226.

126. Hayes, T., Rustagi, N., Saia, J., & Trehan, A. (2008, August). The forgiving tree: a self-healing distributed data structure. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing* (pp. 203-212).
127. Hayes, T. P., Saia, J., & Trehan, A. (2009, August). The forgiving graph: a distributed data structure for low stretch under adversarial attack. In *Proceedings of the 28th ACM symposium on Principles of distributed computing* (pp. 121-130).
128. Sarma, A. D., & Trehan, A. (2012, March). Edge-preserving self-healing: keeping network backbones densely connected. In *2012 Proceedings IEEE INFOCOM Workshops* (pp. 226-231). IEEE.
129. Dutt, S., & Mahapatra, N. R. (1997). Node-covering, error-correcting codes and multiprocessors with very high average fault tolerance. *IEEE Transactions on Computers*, 46(9), 997-1015.
130. Renella, M. T. (1996). *Field replaceable unit (FRU) life: A statistical technique for analyzing the quality of field products*. University of Northern Colorado.
131. Jackson, D. S., Pant, H., & Tortorella, M. (2002). Improved reliability-prediction and field-reliability-data analysis for field-replaceable units. *Ieee Transactions on reliability*, 51(1), 8-16.
132. Zhu, M., Xu, Z., Guo, M., & Li, B. (2017, October). Spare Parts Demand Analysis Method Based on Field Replaceable Unit. In *IOP Conference Series: Materials Science and Engineering* (Vol. 250, No. 1, p. 012071). IOP Publishing.
133. Issarny, V., Tartanoglu, F., Romanovsky, A., & Levy, N. (2003, October). Coordinated forward error recovery for composite web services. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* (pp. 167-176). IEEE.
134. Fasi, M., Langou, J., Robert, Y., & Ucar, B. (2016). A backward/forward recovery approach for the preconditioned conjugate gradient method. *Journal of computational science*, 17, 522-534.
135. Campbell, R. H., & Randell, B. (1986). Error recovery in asynchronous systems. *IEEE transactions on software engineering*, (8), 811-826.

136. Elnozahy, E. N., Alvisi, L., Wang, Y. M., & Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3), 375-408.
137. Koo, R., & Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1), 23-31.
138. Wang, Z., & Aiken, A. (2024). Efficient Fault Tolerance for Pipelined Query Engines via Write-ahead Lineage. *arXiv preprint arXiv:2403.08062*.
139. Stamatakis, G., Pappas, N., Fragkiadakis, A., Petroulakis, N., & Traganitis, A. (2024). Semantics-aware active fault detection in status updating systems. *IEEE Open Journal of the Communications Society*.
140. Díaz, Á. F., Fredlund, L. Å., Benac-Earle, C., & Mariño, J. (2023). A formal semantics for agent distribution and fault tolerance in Jason. *Journal of Logical and Algebraic Methods in Programming*, 133, 100874.
141. Sari, A., & Akkaya, M. (2015). Fault tolerance mechanisms in distributed systems. *International Journal of Communications, Network and System Sciences*, 8(12), 471-482.
142. Egwutuoha, I. P., Chen, S., Levy, D., Selic, B., & Calvo, R. (2014). Cost-oriented proactive fault tolerance approach to high performance computing (HPC) in the cloud. *International Journal of Parallel, Emergent and Distributed Systems*, 29(4), 363-378.
143. Linping, W., Hongbing, L., Jianfeng, Z., & Dan, M. (2011, October). A runtime fault detection method for HPC cluster. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies* (pp. 68-72). IEEE.
144. Ghiasvand, S., & Ciorba, F. M. (2019, June). Anomaly detection in high performance computers: A vicinity perspective. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)* (pp. 112-120). IEEE.
145. Pelaez, A., Quiroz, A., Browne, J. C., Chuah, E., & Parashar, M. (2014, December). Online failure prediction for hpc resources using decentralized clustering. In *2014 21st International Conference on High Performance Computing (HiPC)* (pp. 1-9). IEEE.

146. Gainaru, A., Cappello, F., Snir, M., & Kramer, W. (2012, November). Fault prediction under the microscope: A closer look into HPC systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (pp. 1-11). IEEE.
147. Gainaru, A., Cappello, F., & Kramer, W. (2012, May). Taming of the shrew: Modeling the normal and faulty behaviour of large-scale hpc systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (pp. 1168-1179). IEEE.
148. Borghesi, A., Molan, M., Milano, M., & Bartolini, A. (2021). Anomaly detection and anticipation in high performance computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(4), 739-750.
149. Dani, M. C., Doreau, H., & Alt, S. (2017). K-means application for anomaly detection and log classification in hpc. In *Advances in Artificial Intelligence: From Theory to Practice: 30th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2017, Arras, France, June 27-30, 2017, Proceedings, Part II 30* (pp. 201-210). Springer International Publishing.
150. Fulp, E. W., Fink, G. A., & Haack, J. N. (2008). Predicting Computer System Failures Using Support Vector Machines. *WASL*, 8, 5-5.
151. Zhu, B., Wang, G., Liu, X., Hu, D., Lin, S., & Ma, J. (2013, May). Proactive drive failure prediction for large scale storage systems. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)* (pp. 1-5). IEEE.
152. Ganguly, S., Consul, A., Khan, A., Bussone, B., Richards, J., & Miguel, A. (2016, March). A practical approach to hard disk failure prediction in cloud platforms: Big data model for failure management in datacenters. In *2016 IEEE second international conference on big data computing service and applications (bigdataservice)* (pp. 105-116). IEEE.
153. Das, A., Mueller, F., Siegel, C., & Vishnu, A. (2018, June). Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th international symposium on high-performance parallel and distributed computing* (pp. 40-51).

154. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., & Snir, M. (2009). Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4), 374-388.
155. Cappello, F., Al, G., Gropp, W., Kale, S., Kramer, B., & Snir, M. (2014). Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations: an International Journal*, 1(1), 5-28.
156. Milojević, D. S., Douglass, F., Paindaveine, Y., Wheeler, R., & Zhou, S. (2000). Process migration. *ACM Computing Surveys (CSUR)*, 32(3), 241-299.
157. Guan, S. U., & Li, S. (2001, December). Task decomposition based on output parallelism. In *10th IEEE International Conference on Fuzzy Systems*. (Cat. No. 01CH37297) (Vol. 1, pp. 260-263). IEEE.
158. Midolo, A., & Tramontana, E. (2022, July). An API for Analysing and Classifying Data Dependence in View of Parallelism. In *Proceedings of the 10th International Conference on Computer and Communications Management* (pp. 61-67).
159. Rul, S., Vandierendonck, H., & De Bosschere, K. (2007). Function level parallelism driven by data dependencies. *ACM SIGARCH Computer Architecture News*, 35(1), 55-62.
160. Appelbaum, S. H., Marchionni, A., & Fernandez, A. (2008). The multi-tasking paradox: Perceptions, problems and strategies. *Management Decision*, 46(9), 1313-1325.
161. Zhuang, X., Eichenberger, A. E., Luo, Y., O'Brien, K., & O'Brien, K. (2009, September). Exploiting parallelism with dependence-aware scheduling. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques* (pp. 193-202). IEEE.
162. Hromkovič, J. (2013). *Communication complexity and parallel computing*. Springer Science & Business Media.
163. Chen, G., Li, G., Pei, S., & Wu, B. (2009, December). High performance computing via a GPU. In *2009 First International Conference on Information Science and Engineering* (pp. 238-241). IEEE.

164. Nguyen, T., MacLean, C., Siracusa, M., Doerfler, D., Wright, N. J., & Williams, S. (2022). FPGA-based HPC accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience*, 34(20), e6570.
165. Lant, J., Navaridas, J., Luján, M., & Goodacre, J. (2019). Toward FPGA-based HPC: Advancing interconnect technologies. *IEEE Micro*, 40(1), 25-34.
166. Bennett, C. H., Brassard, G., & Ekert, A. K. (1992). Quantum cryptography. *Scientific American*, 267(4), 50-57.
167. Orús, R., Mugel, S., & Lizaso, E. (2019). Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4, 100028.
168. McArdle, S., Endo, S., Aspuru-Guzik, A., Benjamin, S. C., & Yuan, X. (2020). Quantum computational chemistry. *Reviews of Modern Physics*, 92(1), 015003.
169. Bova, F., Goldfarb, A., & Melko, R. G. (2021). Commercial applications of quantum computing. *EPJ quantum technology*, 8(1), 2.
170. Rawat, B., Mehra, N., Bist, A. S., Yusup, M., & Sanjaya, Y. P. A. (2022). Quantum computing and ai: Impacts & possibilities. *ADI Journal on Recent Innovation*, 3(2), 202-207.
171. Bayerstadler, A., Becquin, G., Binder, J., Botter, T., Ehm, H., Ehmer, T., ... & Winter, F. (2021). Industry quantum computing applications. *EPJ Quantum Technology*, 8(1), 25.
172. Spector, L., Barnum, H., Bernstein, H. J., & Swamy, N. (1999). Quantum computing applications of genetic programming.
173. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., ... & Martinis, J. M. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779), 505-510.
174. Ball, P. (2020). Physicists in China challenge Google's quantum advantage'. *Nature*, 588(7838), 380.
175. Hossain, K. A. (2023). The potential and challenges of quantum technology in modern era. *Scientific Research Journal*, 11, 41-49.
176. Ezratty, O. (2023). Perspective on superconducting qubit quantum computing. *The European Physical Journal A*, 59(5), 94.

177. Choi, C. Q. (2023). Ibm's quantum leap: The company will take quantum tech past the 1,000-qubit mark in 2023. *IEEE Spectrum*, 60(1), 46-47.
178. Hou, S. Y., Feng, G., Wu, Z., Zou, H., Shi, W., Zeng, J., ... & Zeng, B. (2021). SpinQ Gemini: a desktop quantum computing platform for education and research. *EPJ Quantum Technology*, 8(1), 1-23.
179. Boerkamp, M. (2023). Intel releases 12-qubit silicon quantum chip. *Physics World*, 36(8), 11ii.
180. Ding, Y., Gonzalez-Conde, J., Lamata, L., Martín-Guerrero, J. D., Lizaso, E., Mugel, S., ... & Sanz, M. (2023). Toward prediction of financial crashes with a d-wave quantum annealer. *Entropy*, 25(2), 323.
181. Montanez-Barrera, J. A., van den Heuvel, P., Willsch, D., & Michielsen, K. (2023, September). Improving Performance in Combinatorial Optimization Problems with Inequality Constraints: An Evaluation of the Unbalanced Penalization Method on D-Wave Advantage. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)* (Vol. 1, pp. 535-542). IEEE.
182. Kelly, J. (2018, 5 березня). A preview of Bristlecone, Google's new quantum processor. *Google Research Blog*. <https://research.google/blog/a-preview-of-bristlecone-googles-new-quantum-processor/>
183. Wang, C., Li, X., Xu, H., Li, Z., Wang, J., Yang, Z., ... & Yu, H. (2022). Towards practical quantum computers: Transmon qubit with a lifetime approaching 0.5 milliseconds. *npj Quantum Information*, 8(1), 3.
184. Somoroff, A., Ficheux, Q., Mencia, R. A., Xiong, H., Kuzmin, R., & Manucharyan, V. E. (2023). Millisecond coherence in a superconducting qubit. *Physical Review Letters*, 130(26), 267001.
185. Wang, P., Luan, C. Y., Qiao, M., Um, M., Zhang, J., Wang, Y., ... & Kim, K. (2021). Single ion qubit with estimated coherence time exceeding one hour. *Nature communications*, 12(1), 233.
186. McGeoch, C. C. (2022). *Adiabatic quantum computation and quantum annealing: Theory and practice*. Springer Nature.

187. Honcharenko, O., & Loutsikii, H. (2022). Methods of effectiveness of scalable systems: review. *Information, Computing and Intelligent systems*, (3), 63–76.
188. Wang, B., Hu, F., Yao, H., & Wang, C. (2020). Prime factorization algorithm based on parameter optimization of Ising model. *Scientific reports*, 10(1), 7106.
189. Wang, C., Hu, Q., Yao, H., Wang, S., & Pei, Z. (2023). Deciphering a Million-Plus RSA Integer with Ultralow Local Field Coefficient h and Coupling Coefficient J of the Ising Model by D-Wave 2000Q. *Tsinghua Science and Technology*, 29(3), 874-882.
190. Ji, X., Wang, B., Hu, F., Wang, C., & Zhang, H. (2021). New advanced computing architecture for cryptography design and analysis by D-Wave quantum annealer. *Tsinghua Science and Technology*, 27(4), 751-759.
191. Sousa, T. B. (2012, November). Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering* (Vol. 130).
192. Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., O’Boyle, M. F., & Leather, H. (2021, July). Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning* (pp. 2244-2253). PMLR.
193. Gasmi, K., & Hasnaoui, S. (2024). Dataflow-based automatic parallelization of MATLAB/Simulink models for fitting modern multicore architectures. *Cluster Computing*, 1-12.
194. Kumar, A., Wang, Z., Ni, S., & Li, C. (2020). Amber: a debuggable dataflow system based on the actor model. *Proceedings of the VLDB Endowment*, 13(5), 740-753.
195. Zhao, J., Korpan, B., Gonzalez, A., & Asanovic, K. (2020, May). Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V* (Vol. 5, pp. 1-7).
196. Gobieski, G., Nagi, A., Serafin, N., Isgenc, M. M., Beckmann, N., & Lucia, B. (2019, October). Manic: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 670-684).

197. Chang, K. W., & Chang, T. S. (2019, May). VSCNN: Convolution neural network accelerator with vector sparsity. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)* (pp. 1-5). IEEE.
198. Bhagyanath, A., & Schneider, K. (2023, June). Program balancing in compilation for buffered hybrid dataflow processors. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)* (pp. 57-66). IEEE.
199. Diaz, J. M. M. (2021). *Sequential codelet model: a supercodelet program execution model and architecture*. University of Delaware.
200. Gévay, G. E., Soto, J., & Markl, V. (2021). Handling iterations in distributed dataflow systems. *ACM Computing Surveys (CSUR)*, 54(9), 1-38.
201. Anil, R., Capan, G., Drost-Fromm, I., Dunning, T., Friedman, E., Grant, T., ... & Yilmazel, Ö. (2020). Apache mahout: Machine learning on distributed dataflow systems. *Journal of Machine Learning Research*, 21(127), 1-6.
202. Arvind, & Brobst, S. (1993). The evolution of dataflow architectures: from static dataflow to P-RISC. *International Journal of High Speed Computing*, 5(02), 125-153.
203. Gigerl, B., Primas, R., & Mangard, S. (2021). Secure and efficient software masking on superscalar pipelined processors. In *Advances in Cryptology—ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part II 27* (pp. 3-32). Springer International Publishing.
204. Cowley, M., & Sawalha, L. (2021). RISC-V Dataflow Extension. In *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV'21)* (p. 7).
205. Soundararajan, N., Gupta, S., Natarajan, R., Stark, J., Pal, R., Sala, F., ... & Subramoney, S. (2019, October). Towards the adoption of local branch predictors in modern out-of-order superscalar processors. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 519-530).
206. Becker, T., Burovskiy, P., Nestorov, A. M., Palikareva, H., Reggiani, E., & Gaydadjiev, G. (2017, March). From exaflop to exaflow. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (pp. 404-409). IEEE.

207. Baskin, C., Liss, N., Zheltonozhskii, E., Bronstein, A. M., & Mendelson, A. (2018, May). Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 162-169). IEEE.
208. Sau, C., Fanni, T., Rubattu, C., Raffo, L., & Palumbo, F. (2021). The Multi-Dataflow Composer tool: An open-source tool suite for optimized coarse-grain reconfigurable hardware accelerators and platform design. *Microprocessors and Microsystems*, 80, 103326.
209. Guo, Y., & Luo, G. (2020, November). Pillars: An Integrated CGRA Design Framework. In *Third Workshop on Open-Source EDA Technology (WOSET)*.
210. Charitopoulos, G., Pnevmatikatos, D. N., & Gaydadjiev, G. (2021). MC-DeF: Creating customized CGRAs for dataflow applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(3), 1-25.
211. Heo, J. H., Fayyazi, A., Esmaili, A., & Pedram, M. (2022, August). Sparse Periodic Systolic Dataflow for Lowering Latency and Power Dissipation of Convolutional Neural Network Accelerators. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (pp. 1-6).
212. Mastinu, M. (2013). *Design flow to support dynamic partial reconfiguration on Maxeler architectures* (Doctoral dissertation, University of Illinois at Chicago).
213. Holzinger, A., Langs, G., Denk, H., Zatloukal, K., & Müller, H. (2019). Causability and explainability of artificial intelligence in medicine. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4), e1312.
214. Carleo, G., Cirac, I., Cranmer, K., Daudet, L., Schuld, M., Tishby, N., ... & Zdeborová, L. (2019). Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4), 045002.
215. Ikonomakis, M., Kotsiantis, S., & Tampakas, V. (2005). Text classification using machine learning techniques. *WSEAS transactions on computers*, 4(8), 966-974.
216. Bertomeu, J., Cheynel, E., Floyd, E., & Pan, W. (2021). Using machine learning to detect misstatements. *Review of Accounting Studies*, 26, 468-519.

217. Kitano, H., & Hendler, J. A. (1994). *Massively parallel artificial intelligence* (Vol. 11). Menlo Park, CA: Aaa Press.
218. Bäck, T., & Schwefel, H. P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1), 1-23.
219. Slowik, A., & Kwasnicka, H. (2020). Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32, 12363-12379.
220. Hruschka, E. R., Campello, R. J., & Freitas, A. A. (2009). A survey of evolutionary algorithms for clustering. *IEEE Transactions on systems, man, and cybernetics, Part C (applications and reviews)*, 39(2), 133-155.
221. Elbaz, K., Shen, S. L., Zhou, A., Yin, Z. Y., & Lyu, H. M. (2021). Prediction of disc cutter life during shield tunneling with AI via the incorporation of a genetic algorithm into a GMDH-type neural network. *Engineering*, 7(2), 238-251.
222. Bertoni, A., & Dorigo, M. (1993). Implicit parallelism in genetic algorithms. *Artificial Intelligence*, 61(2), 307-314.
223. Del Grosso, A., & Righetti, G. (1988). Finite element techniques and artificial intelligence on parallel machines. *Computers & Structures*, 30(4), 999-1007.
224. Tournavitis, G., Wang, Z., Franke, B., & O'Boyle, M. F. (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan notices*, 44(6), 177-187.
225. Li, J., Ma, X., Singh, K., Schulz, M., de Supinski, B. R., & McKee, S. A. (2009, April). Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE international symposium on performance analysis of systems and software* (pp. 89-100). IEEE.
226. Kerr, R., & Szelke, E. (Eds.). (2016). *Artificial intelligence in reactive scheduling*. Springer.
227. Anghelescu, P. (2021). Parallel Optimization of Program Instructions Using Genetic Algorithms. *Computers, Materials & Continua*, 67(3).

228. Mahjoub, S., Golsorkhtabaramiri, M., Amiri, S. S. S., Hosseinzadeh, M., & Mosavi, A. (2022). A New Combination Method for Improving Parallelism in Two and Three Level Perfect Nested Loops. *IEEE Access*, 10, 74542-74554.
229. Liang, P., Tang, Y., Zhang, X., Bai, Y., Su, T., Lai, Z., ... & Li, D. (2023). A Survey on Auto-Parallelism of Large-Scale Deep Learning Training. *IEEE Transactions on Parallel and Distributed Systems*.
230. Ansari, A., & Bakar, A. A. (2014, December). A comparative study of three artificial intelligence techniques: Genetic algorithm, neural network, and fuzzy logic, on scheduling problem. In *2014 4th international conference on artificial intelligence with applications in engineering and technology* (pp. 31-36). IEEE.
231. Kari, S. R., Ocampo, C. A. R., Jiang, L., Meng, J., Peserico, N., Sorger, V. J., ... & Youngblood, N. (2023). Optical and electrical memories for analog optical computing. *IEEE Journal of Selected Topics in Quantum Electronics*, 29(2: Optical Computing), 1-12.
232. Eckmiller, R., & vd Malsburg, C. (Eds.). (2012). *Neural computers*. Springer Science & Business Media.
233. Martens, S., Landuyt, A., Espeel, P., Devreese, B., Dawyndt, P., & Du Prez, F. (2018). Multifunctional sequence-defined macromolecules for chemical data storage. *Nature Communications*, 9(1), 4451.
234. Ceze, L., Nivala, J., & Strauss, K. (2019). Molecular digital data storage using DNA. *Nature Reviews Genetics*, 20(8), 456-466.
235. Ghafar, A. M. A., & Said, M. F. M. (2023). Parallel Processing-A Case Study on Automatic Parallelization. *Journal of Advanced Computing Research Vol*, 5(1), 1-5.
236. Midkiff, S. (2022). Automatic parallelization: an overview of fundamental compiler techniques.
237. Esfahanian, & Hakimi. (1985). Fault-tolerant routing in debruijn communication networks. *IEEE Transactions on Computers*, 100(9), 777-788.
238. Atchley, S., Zimmer, C., Lange, J. R., Bernholdt, D. E., Vergara, V. G. M., Beck, T., ... & Yeung, P. K. (2023, November). Frontier: Exploring Exascale The System

Architecture of the First Exascale Supercomputer. In *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-16). IEEE.

239. *Aurora* | *Argonne Leadership Computing Facility*. (n.d.). <https://www.alcf.anl.gov/aurora>

240. *Eagle System Configuration*. (n.d.). High-Performance Computing | NREL. <https://www.nrel.gov/hpc/eagle-system-configuration.html>

241. Ajima, Y. (2020). High-dimensional interconnect technology for the K computer and the supercomputer Fugaku. *Fujitsu Technical Review*.

242. *Documentation - Network and interconnect*. (n.d.). <https://docs.lumi-supercomputer.eu/hardware/network/>

243. *About | Leonardo pre-exascale supercomputer*. (2024, February 21). Leonardo Pre-exascale Supercomputer. <https://leonardo-supercomputer.cineca.eu/about/#:~:text=Leonardo%20features%20a%20Dragonfly%2B%20topology,HPC%20application%20performance%20and%20scalability>.

244. Stunkel, C. B., Graham, R. L., Shainer, G., Kagan, M., Sharkawi, S. S., Rosenberg, B., & Chochia, G. A. (2020). The high-speed networks of the Summit and Sierra supercomputers. *IBM Journal of Research and Development*, 64(3/4), 3-1.

245. *MareNostrum* 5. (n.d.). BSC-CNS. <https://www.bsc.es/ca/marenostrum/marenostrum-5>

246. Morgan, T. P. (2022, October 26). *The NVSwitch fabric that is the hub of the DGX H100 SuperPOD*. The Next Platform. <https://www.nextplatform.com/2022/03/23/nvidia-will-be-a-prime-contractor-for-big-ai-supercomputers/>

247. Wang, T., Su, Z., Xia, Y., & Hamdi, M. (2014). Rethinking the data center networking: Architecture, network protocols, and resource sharing. *IEEE access*, 2, 1481-1496.

248. Jain, N., Bhatele, A., Howell, L. H., Böhme, D., Karlin, I., León, E. A., ... & Leininger, M. L. (2017, November). Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-13).

249. Ohring, S. R., Ibel, M., Das, S. K., & Kumar, M. J. (1995, April). On generalized fat trees. In *Proceedings of 9th international parallel processing symposium* (pp. 37-44). IEEE.
250. Zahavi, E. (2010). D-Mod-K routing providing non-blocking traffic for shift permutations on real life fat trees. *CCIT Report*, 776, 840.
251. Alizadeh, M., & Edsall, T. (2013, August). On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st annual symposium on high-performance interconnects* (pp. 71-74). IEEE.
252. Sabir, E., Mamut, A., & Vumar, E. (2019). The extra connectivity of the enhanced hypercubes. *Theoretical Computer Science*, 799, 22-31.
253. Shpiner, A., Haramaty, Z., Eliad, S., Zdornov, V., Gafni, B., & Zahavi, E. (2017, February). Dragonfly+: Low cost topology for scaling datacenters. In *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)* (pp. 1-8). IEEE.
254. Гончаренко, О. О. (2021). *Метод синтезу топологій для підвищення відмовостійкості розподілених систем* (Master's thesis, КНУ ім. Ігоря Сікорського).

Додаток А

Частина програмного коду

Розроблене програмне забезпечення знаходиться за посиланням <https://github.com/O-Honcharenko/PhD.git>

Лістинг А.1 – модуль *NumericSystems.py*

```
import itertools as iter
import math
import random as rand

denotation = {
    '0':0, '1':1, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8, '9':9, 'A':10, 'B':11, 'C':12, 'D':13, 'E':14, 'F':15,
    'O':-0,
    'T':-1, # T like 1 with upscore
    'Z':-2, # Z like warped 2
    'W':-3, # W like Roman III
    'Q':-4, # Q like Quadro
    'P':-5, # P like Penta
    'H':-6, # H like Hexa
    'S':-7, # S like Septa
    'K':-8, # K like oKto (right is Octa, but letters O, C, T and A is still allocated)
    'N':-9, # N like Nona
    'X':-10, # X like Roman X
    'L':-11, # L like eLeven
    'Y':-12, # Y like previosly letter of Z (-2)
    'M':-13, # M like inverted W (-3)
    'U':-14, # U like 2nd letter of qUadrodeca
    'P':-15 # P like Pentadeca
}

# get value of 1 simple letter
def getValue(letter):
    if type(letter)==int:
        return letter
    else:
        return denotation[letter]

def get_value(letter):
    return getValue(letter)

# get value of code in concrete numeric system
# alphabetic denotations of numbers are predefined by 'denotations' dict
def valueOf(code, base):
    value = 0
    w = 1
    high_pos = len(code)-1
    for i in range(high_pos, -1, -1): # go from the last symbols
        digit = code[i]
        digit_val = getValue(code[i])
        value += w * digit_val # get value of symbol and multiply by weight
        w *= base # change weight to higher (w = base*pos)
    return value

def value_of(code, base):
    return valueOf(code, base)

# denote a value (int) as digit (str) or returns direct string representation if it is impossible
def denote(value):
    for digit in denotation:
        if denotation[digit] == value: return digit
    return str(value)

# returbs alphabet with values from list
def denote_all(values):
    return map(denote, values)

def value_all(alphabet):
    return map(getValue, alphabet)

# returns lower digit in the alphabet (list of digits)
def lower_value(alphabet):
```

```

    return min(value_all(alphabet))

def higher_value(alphabet):
    return max(value_all(alphabet))

##### Системи числення та коди в них #####

def all_codes(alphabet, rank):
    return iter.product(alphabet, repeat=rank)

def create_alphabet(first, last):
    return create_alpha_in_range(range(first, last+1))

def create_alpha_in_range(_range):
    return list(denote_all([i for i in _range]))

# 1 аргумент - остання цифра last
# 2 аргументи - набір цифр алфавіту (з first по last)
def alphabet(*args):
    if not args: return []
    if len(args) == 1:
        if type(args) == range: return create_alpha_in_range(args[0])
        return create_alphabet(0, args[0])
    return create_alphabet(args[0], args[1])

def as_alphabet(*args):
    return list(denote_all(args))

BINARY = ['0', '1']
BIN = BINARY
TERNARY = ['0', '1', '2']
TERN = TERNARY
RED_BINARY = ['T', '0', '1']
RB = RED_BINARY
QUADRARY = ['0', '1', '2', '3']
QUAD = QUADRARY
RED_TERNARY = ['T', '0', '1', '2']
RT = RED_TERNARY
PENTARY = alphabet(0, 4)
PENT = PENTARY
HEXARY = alphabet(0, 5)
HEXA = HEXARY

# Генерує випадковий код в заданій СЧ
# За замовчуванням система бінарна, а ранг = 0 (випадковий ранг від 1 до 255)
def get_random_code(alphabet = BINARY, rank = 0):
    a = alphabet.copy()
    if rank == 0:
        rank = rand.randint(1, 255)
    code = rand.choices(a, k = rank)
    return code

# генерує код, де всі цифри однакові
def monocode(digit, rank):
    return tuple([digit for i in range(rank)])

##### Робота з мітками #####

# може розпізнавати одно- і багаторівневі мітки
# може інтерпретувати цілочисельні та неітеровані значення
def as_label(code, sep = "", denote_ints = False):
    if code == None: return 'None' # якщо код порожній - повертаємо рядком None
    if type(code) == str: return code # якщо код вже є рядком - нічого не конвертуємо
    if hasattr(code, '__iter__'): return sep.join(map(label_d, code)) # якщо код є ітерованим значенням (стандартний варіант) - конвертуємо
    # конверсія обробляє (* of int) незалежно від параметра denote_ints
    if type(code) == int: # з цілими числами діємо залежно від того, що у нас сказано в denote_ints
        return denote(code) if denote_ints else str(code)
    return str(code) # якщо це невідомо що - перетворюємо стандартною функцією приведення

# перетворює цифру в мітку
def label_d(d):
    if type(d) == str: return d
    if type(d) == int: return denote(d)
    if type(d) == tuple: return f'({as_label(d)})'
    if type(d) == list: return f'[{as_label(d)}]'
    return str(d)

# аліас для as_label

```

```

def label(code, sep = "):
    return as_label(code, sep=sep)

# перетворення однорівневої мітки в послідовність
def parse(label):
    code = []
    for s in label:
        code.append(s)
    return tuple(code)

##### Операції над СЧ #####

# повертає порядок (кількість унікальних символів) в послідовності
# послідовністю може бути як код так і алфавіт СЧ
# функціонує через приведення в set
def order(sequence):
    return len(set(sequence))

# перевіряє, чи належить послідовність іншій послідовності
# послідовністю може бути як код так і алфавіт СЧ
# супермножиною має бути алфавіт СЧ, але може бути і код (залізяці різниці нема)
def belong(sequence, potential_superset):
    return set(potential_superset).issuperset(sequence)

# повертає ПЕРШИЙ підходящий алфавіт
def get_alphabet(sequence, alphabets):
    for a in alphabets:
        if belong(sequence, a): return a
    else:
        return None

##### Алфавітна декомпозиція #####

"""
Алфавітна декомпозиція
codeset - набір кодів
alphabet - алфавіт, в якому сформовано набір кодів
desired_order - порядок декомпозиції (якщо значення не вказане чи некоректне - береться порядок на 1 менше ніж порядок алфавіту)
"""
def decomposition(codeset, alphabet, desired_order = 0):
    k = order(alphabet) # отримуємо порядок вихідного алфавіту

    if desired_order <= 0 or desired_order >= k: # перевіряємо коректність бажаного порядку
        desired_order = k-1

    subalphabets = iter.combinations(alphabet, desired_order) # генеруємо підалфавіти
    result = {tuple(a):[] for a in subalphabets} # результатом є словник, де кожному підалфавіту ставиться множина кодів з codeset, які до нього
    належать
    result[None] = [] # коди, що не належать до жодного субалфавіту, записуються в словник із ключем "None"
    for code in codeset:
        ord = order(code) # отримуємо порядок коду
        if ord > desired_order:
            result[None].append(code) # якщо він більше бажаного - то тут і думати нічого, це None
        else:
            for a in result:
                if a == None: continue
                if belong(code, a):
                    result[a].append(code) # перевіряємо належність до кожного із підалфавітів. Якщо належить - додаємо під відповідним ключем

def dec_to_string(dec, kdelim='\n', vdelim = ' '):
    strings = []
    for a in dec:
        strings.append(f'{label(a)}:{vdelim.join(map(label, dec[a]))}')
    return kdelim.join(strings)

```

Лістинг A.2 – Модуль NumericGraphs.py

```

import networkx as nx
import math

from numeric import NumericSystems as ns

# create unlinked excess graph
# alphabet: alphabet for creation (array of symbols). Example: [0, 1, T]
def create_nodes(alphabet, rank):
    G = nx.Graph(
        name='none',

```

```

    alphabet = alphabet,
    rank = rank) # create an empty graph with attributes 'name', 'alphabet' and 'rank'
codes = ns.all_codes(alphabet, rank) # generate all 'rank'-digital codes for defined alphabet (base is not important here)
G.add_nodes_from(codes) # create unconnected graph with nodes, encoded by rank-digital codes
return G

```

```

#####
##### СТАНДАРТНІ ТОПОЛОГІЇ #####
#####

```

```

def create_links_de_Brujin(G, exclusion=[]):
    rank = G.graph['rank']
    for node in list(G.nodes):
        for digit in G.graph['alphabet']:
            good = not (digit in exclusion) # все добре, якщо цифри digit нема в заборонених
            if not good:
                continue # якщо цифра digit заборонена - пропускаємо ітерацію
            for ex in exclusion:
                good = good and not (ex in node) # шукаємо кожну заборонену цифру в коді вершини. Добре - якщо її там нема

            if good: # якщо все добре - виконуємо зсуви
                shift_left = list(node[1:rank])
                shift_left.append(digit)
                shift_right = [digit]
                shift_right.extend(list(node[0:rank-1]))
                left = tuple(shift_left)
                right = tuple(shift_right)
                if not node==left:
                    G.add_edge(node, left)
                if not node==right:
                    G.add_edge(node, right)
    return G

```

```

def create_links_hypercube(G, exclusion=[]):
    rank = G.graph['rank']
    for node in list(G.nodes):
        for digit in G.graph['alphabet']:
            good = not (digit in exclusion) # все добре, якщо цифри digit нема в заборонених
            if not good:
                continue # якщо цифра digit заборонена - пропускаємо ітерацію
            for ex in exclusion:
                good = good and not (ex in node) # шукаємо кожну заборонену цифру в коді вершини. Добре - якщо її там нема

            if good: # якщо все добре - виконуємо зсуви
                for i in range(len(node)):
                    if node[i] != digit:
                        nei = list(node)
                        nei[i] = digit
                        G.add_edge(node, tuple(nei))
    return G

```

```

#####
#####
##### НАДЛИШКОВА КЛАСТЕРИЗАЦІЯ #####
#####
#####

```

```

def allowed_values(rank, alphabet, base, exclusions = []):
    alpha = list(alphabet.copy())
    for e in exclusions: alpha.remove(e)
    codes = ns.all_codes(alpha, rank)
    values = [ns.value_of(c, base) for c in codes]
    return set(values)

```

```

"""
Групує вершини по кластерам.
Числового номеру ставиться у відповідність список кодів, які цьому номеру відповідають
"""

```

```

def group(graph, base, exclusions = []):
    G = nx.Graph.copy(graph)
    G.graph['basis'] = base
    rank = G.graph['rank']
    alphabet = G.graph['alphabet']

```



```

allowed = allowed_values(rank, alphabet, base, exclusions = exclusions)

clusters = {}
nodes_to_remove = []

for node in list(G.nodes):
    value = ns.valueOf(node, base)
    if not value in allowed:
        nodes_to_remove.append(node)
        continue
    G.nodes[node]['value'] = value
    if value in clusters:
        clusters[value].append(node)
    else:
        clusters[value] = [node, ]

G.remove_nodes_from(nodes_to_remove)
return {'graph': G, 'clusters': clusters}

"""Операції по роботі з ІСЧ.
index_system alphabet - дає алфавіт індексної системи. Варто зазначити, що ІСЧ завжди має тип (t, b), причому алфавіт має вигляд 0...t-1
to_index - конвертує заданий код в ІСЧ за формулою (a-lower mod t). Пам'ятаємо, що молодша цифра (rank-1 елемент кортежу) не враховується у
формулі. В нашому випадку, оскільки індексація масивів завжди йде з нуля від старшої частини до молодшої (а не як в дисертації, де 0 - молодший
розряд), встановлення відповідності між i+1 елементом вихідного коду та i-м індексного реалізується банальним скороченням масиву для індексного
коду (тобто, по дисертації зв'язуються i-len(node) та i-len(index), а len(index) сам по собі якраз менше на 1 за len(node))."""
# отримати коефіцієнт t (відношення порядку алфавіту до основи)
def get_t(alphabet, base):
    return math.ceil(len(alphabet)/base)

# згенерувати алфавіт індексної системи (генерується як int)
def index_system_alphabet(alphabet, base, t):
    return [i for i in range(0, t)]

# конвертувати в індексну систему
# елементами індексного коду є цифри (тип int)
def to_index(node, lower, t, rank):
    index = [(ns.getValue(node[i]) - lower) % t for i in range(0, rank-1)]
    return tuple(index)

# шукаємо зв'язки для заданого індексу
# пошук - по перетворенню Exchange (гіперкуб)
def neighbors_in_cluster_hc(index, rank, ind_alphabet):
    neighbors = []
    ind_list = list(index)
    for i in range(0, rank-1):
        for a in ind_alphabet:
            if index[i] != a:
                nei = ind_list.copy()
                nei[i] = a
                neighbors.append(tuple(nei))
    return neighbors

def clusterize(grouped):
    G = grouped['graph'] # граф
    clusters = grouped['clusters'] # словник, де значенням у відповідність поставлено список кодів

    # підготовка до перетворення в ІСЧ
    alphabet = G.graph['alphabet'] # вихідний алфавіт
    rank = G.graph['rank'] # ранг топології
    lower = ns.lower_value(alphabet) # мінімальне значення вихідного алфавіту
    base = G.graph['basis'] # основа числення
    t = get_t(alphabet, base) # відношення k/b (назвемо його надлишковістю. Звичайна СЧ має t=1, надлишкова - t>1, анти-надлишкова - t<1)
    G.graph['ideal redundancy'] = t # тут нас цікавить найближче ціле t зверху

    index_alpha = index_system_alphabet(alphabet, base, t) # алфавіт ІСЧ
    G.graph['index alphabet'] = index_alpha # оскільки алфавіт єдиний для всіх вершин - зберігаємо його в графі
    G.graph['indices'] = {}

    for value in clusters:
        cluster_nodes = clusters[value] # список вершин кластеру
        indices = {} # робимо словник, де індексу ставимо у відповідність код
        # це треба для зворотнього перетворення, щоб не використовувати складну формулу із дисертації
        # Формат: (індекс [tuple of int] : код вершини)

    for node in cluster_nodes:

```

```

index = to_index(node, lower, t, rank) # генеруємо для всіх вершин кластеру індекси
indices[index] = node                 # заносимо індекси в словник
G.nodes[node]['index'] = index        # додаємо кожній вершині графа властивість "індекс"

G.graph['indices'][value] = indices    # в характеристики графа додаємо словник індексів для потрібного значення.
# так ми зможемо отримати код вершини по її значенню та індексу
G.graph['incluster edges'] = {}

for index in indices:
    this = indices[index]             # отримуємо код даної вершини (дійсний)
    neighbors = neighbors_in_cluster_hc(index, rank, index_alpha) # отримуємо всіх індексних сусідів по певному правилу (не всі з них дійсні)

    for nei in neighbors:              # перебираємо сусідів
        if nei in indices:             # якщо сусід присутній в indices - це дійсна вершина багатовимірної форми (яка має порожні елементи)
            target = indices[nei]       # перетворюємо індекс в код
            G.add_edge(this, target, cluster = value) # формуємо ребро
            edge = (this, target)
            if value in G.graph['incluster edges']:
                G.graph['incluster edges'][value].append(edge)
            else:
                G.graph['incluster edges'][value] = [edge, ]
return G # граф повертаємо

def link_excess_clusters(G, base = 2, exclusions = []):
    grouped = group(G, base, exclusions = exclusions)
    return clusterize(grouped)

def node_label(node, G = None):
    s = ns.label(node)
    if not G == None:
        try:
            index = ns.label(G.nodes[node]['index'])
            s += ('+index+')
        except Exception as e:
            return s
    return s

def edge_label(edge, G = None):
    return f"({node_label(edge[0], G)}, {node_label(edge[1], G)})"

def edge_in_nbunch(edge, nbunch):
    return (edge[0] in nbunch) and (edge[1] in nbunch)

def as_edge(label1, label2):
    return (ns.parse(label1), ns.parse(label2))

```

Лістинг А.3 – модуль NumericGenerator.py

```

import networkx as nx
import pandas as pd

from numeric import NumericSystems as ns
from numeric import NumericGraphs as ngraphs

# ФУНКЦІЇ-ЛІНКЕРИ (із модуля NumericGraphs)

HYPERCUBE = ngraphs.create_links_hypercube
DE_BRUIJN = ngraphs.create_links_de_Brujin

# ФУНКЦІЇ-КЛАСТЕРИЗАТОРИ
DIS_CLUSTERIZER = ngraphs.link_excess_clusters # кластеризація за багатовимірною формою (дисертація)

##### ФУНКЦІЇ-ГЕНЕРАТОРИ ТОПОЛОГІЙ #####
# це зручніше ніж щоразу виписувати всі етапи генерації

# генератор звичайної (не кластеризованої) топології будь-якого типу
# за замовчуванням - генерує гіперкуб на надлишковій бінарній системі числення
def simple_creator(rank, f_linker = HYPERCUBE, alphabet = ns.RED_BINARY, exclusions = []):
    G = ngraphs.create_nodes(alphabet, rank)
    return f_linker(G, exclusion = exclusions)

# генератор кластеризованої топології будь-якого типу
# за замовчуванням - генерує гіперкуб на надлишковій бінарній системі числення
def clustered_creator(rank, f_linker = HYPERCUBE, f_clusterizer = DIS_CLUSTERIZER, alphabet = ns.RED_BINARY, base = 2, exclusions = []):
    G = simple_creator(rank, f_linker = f_linker, alphabet = alphabet, exclusions = exclusions)
    return f_clusterizer(G, base = base, exclusions = exclusions)

```

СТАНДАРТНІ ГЕНЕРАТОРИ ТОПОЛОГІЙ

ФУНКЦІЇ-ГЕНЕРАТОРИ ДЛЯ СПРОЩЕННЯ РОБОТИ

```

_HYPERCUBE = lambda rank, alphabet: simple_creator(rank, f_linker = HYPERCUBE, alphabet = alphabet)
_DE_BRUJIN = lambda rank, alphabet: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = alphabet)
_HYPERCUBE_E = lambda rank, alphabet, exclusions: simple_creator(rank, f_linker = HYPERCUBE, alphabet = alphabet, exclusions = exclusions)
_DE_BRUJIN_E = lambda rank, alphabet, exclusions: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = alphabet, exclusions = exclusions)

_HYPERCUBE_C = lambda rank, alphabet, base: clustered_creator(rank, f_linker = HYPERCUBE, alphabet = alphabet, base = base)
_DE_BRUJIN_C = lambda rank, alphabet, base: clustered_creator(rank, f_linker = DE_BRUJIN, alphabet = alphabet, base = base)
_HYPERCUBE_EC = lambda rank, alphabet, base, exclusions: clustered_creator(rank, f_linker=HYPERCUBE, alphabet=alphabet, base=base,
exclusions=exclusions)
_DE_BRUJIN_EC = lambda rank, alphabet, base, exclusions: clustered_creator(rank, f_linker=DE_BRUJIN, alphabet=alphabet, base=base,
exclusions=exclusions)

```

#hypercubes, no exclusions, no clusterization

```

GEN_HYPERCUBE_BIN = lambda rank: simple_creator(rank, f_linker = HYPERCUBE, alphabet = ns.BINARY)
GEN_HYPERCUBE_RBIN = lambda rank: simple_creator(rank, f_linker = HYPERCUBE, alphabet = ns.RED_BINARY)
GEN_HYPERCUBE_QUAD = lambda rank: simple_creator(rank, f_linker = HYPERCUBE, alphabet = ns.QUADRARY)

```

#de brujins, no exclusions, no clusterization

```

GEN_DE_BRUJIN_BIN = lambda rank: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.BINARY)
GEN_DE_BRUJIN_RBIN = lambda rank: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.RED_BINARY)
GEN_DE_BRUJIN_QUAD = lambda rank: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.QUADRARY)

```

#hypercubes with base 2, clustered

```

GEN_HYPERCUBE_RBIN_C = lambda rank: clustered_creator(rank, f_linker = HYPERCUBE, alphabet = ns.RED_BINARY, base = 2)
GEN_HYPERCUBE_QUAD_C = lambda rank: clustered_creator(rank, f_linker = HYPERCUBE, alphabet = ns.QUADRARY, base = 2)

```

#de brujins with base 2, clustered

```

GEN_DE_BRUJIN_RBIN_C = lambda rank: clustered_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.RED_BINARY, base = 2)
GEN_DE_BRUJIN_QUAD_C = lambda rank: clustered_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.QUADRARY, base = 2)

```

#de brujins with exclusions

```

GEN_DE_BRUJIN_ERBIN = lambda rank: simple_creator(rank, f_linker = DE_BRUJIN, alphabet = ns.RED_TERNARY, exclusion=['2'])
GEN_DE_BRUJIN_ERBIN_C = lambda rank: clustered_creator(rank, f_linker=DE_BRUJIN, alphabet = ns.RED_TERNARY, base = 2, exclusion=['2'])

```

Лістинг А.4 – модуль Topologies.py

```

import networkx as nx
import matplotlib.pyplot as plt
from pathlib import Path

```

Немасштабовані топології

```

def petersen():
    return nx.petersen_graph()

```

```

def tutte():
    return nx.tutte_graph()

```

Стантартні топології

насправді це просто переадресація викликів NetworkX
нашо це треба? Щоб не шукати по документації, як та чи інша стандартна топологія генерується

```

def circle(rank):
    return nx.circulant_graph(rank, [1])

```

```

def line(rank):
    return nx.path_graph(rank)

```

```

def star(rank):
    return nx.star_graph(rank)

```

```

def hypercube(rank):
    return nx.hypercube_graph(rank)

```

```

def grid(m, n):
    return nx.grid_2d_graph(m, n)

```

```

def grid(rank):
    return nx.grid_2d_graph(rank, rank)

```

```

def mesh(m, n):
    return nx.grid_2d_graph(m, n, periodic = True)

```

```

def mesh(rank):
    return nx.grid_2d_graph(rank, rank, periodic = True)

def nd_torus(rank, d = 3):
    return nx.grid_graph([rank for i in range(d)], periodic = True)

def wheel(rank):
    return nx.wheel_graph(rank)

def full(rank):
    return nx.complete_graph(rank)

def tree(rank, base = 2, integer_num = False):
    G = nx.Graph()
    for layer in range(rank):
        count = base**layer
        nodes = [(layer, i) for i in range(count)]
        G.add_nodes_from(nodes)
        if layer != 0:
            for node in nodes:
                i_parrent = node[1]//base
                G.add_edge(node, (layer-1, i_parrent))
    if integer_num: nx.convert_node_labels_to_integers(G, ordering = 'sorted')
    return G

##### Додаткові функції графових перетворень #####

"""
Функція для видалення самоциклів (ребра виду (n, n))
Виправляє граф або "на місці" або попередньо робить копію.
"""
def unloop(graph, copy = False):
    G = graph.copy() if copy else graph
    loops = nx.selfloop_edges(G)
    G.remove_edges_from(loops)
    return G

"""
Реплікація графа
Повертає граф H, що містить n незв'язаних копій графа G
Може використовувати як аргумент або параметр n (число копій) або seq (номери копій)
Намагається зберегти всі параметри ребер
"""
def replicate(G, n = 1, seq = None):
    if n<=1 and seq == None: return G

    H = nx.Graph()
    edges_data = {e: G.get_edge_data(*e) for e in G.edges}

    if seq == None:
        seq = [i for i in range(n)]
    for n in seq:
        map = {m: (n, m) for m in G.nodes}
        g = nx.relabel_nodes(G, map, copy=True)
        H.update(nodes = g.nodes, edges = g.edges)
        for e in edges_data:
            H.get_edge_data((n, e[0]), (n, e[1])).update(edges_data[e])
    return H

##### Просунуті топології #####

"""
Генератор топології dragonfly
c - число вершин в кластері
h - число зовнішньокластерних зв'язків
"""
def dragonfly(c, h):
    if c<2 or h<1: raise Exception(f"topo.dragonfly: bad parameters: c = {c}, h = {h}")
    #print(f'Dragonfly ({c}, {h})')
    cluster = full(c)
    g = c*h+1
    #print(f'tg = {g}')
    H = replicate(cluster, n = g)
    #print(f'tnodes degrees {H.degree()}')
    # йтимемо "вгору"

```

```

for a in range(g):      # номер групи
    for b in range(c):  # номер вузла в групі
        node = (a, b)
        #print(f'\tfor node ({a}, {b})...')
        for t in range(h): # номер зв'язку
            bt = c-b-1
            at = (a+bt*h+t+1)%g
            target = (at, bt)
            #print(f'\t\tlink to ({at}, {bt})')
            H.add_edge(node, target, hierarchy = 2)
return H

"""
Генератор топології dragonfly з довільним кластером
h - число зовнішньокластерних зв'язків
cluster - передвизначений кластер
"""

def dragonfly_custom(h, cluster = full(4)):
    if h<1: raise Exception(f'topo.dragonfly: bad parameters: c = {c}, h = {h}')
    c = len(cluster)
    #print(f'Dragonfly ({c}, {h})')
    g = c*h+1
    print(f'\tg = {g}')
    nlist = list(cluster.nodes.keys())
    #print(f'\tnodes = {nlist}')
    H = replicate(cluster, n = g)
    #print(f'\tH.nodes = {H.nodes.keys()}')
    # йтимемо "вгрупу"
    for a in range(g):      # номер групи
        for b in range(c):  # номер вузла в групі
            node = (a, nlist[b])
            #print(f'\tfor node ({a}, {b})...')
            for t in range(h): # номер зв'язку
                bt = c-b-1
                at = (a+bt*h+t+1)%g
                target = (at, nlist[bt])
                #print(f'\t\tlink to {target}')
                H.add_edge(node, target, hierarchy = 2)
    return H

def dragonfly_k(k):
    k2 = k//2
    return dragonfly(k2+1, k2)

"""
Генератор топології dragonfly+
m - число крайових вершин в кластері
c - число вершин-агрегаторів в кластері
h - число зовнішньокластерних зв'язків
"""

def dragonfly_plus(m, c, h):
    if c<2 or h<1: raise Exception(f'topo.dragonfly: bad parameters: c = {c}, h = {h}')
    #print(f'Dragonfly ({c}, {h})')
    cluster = nx.Graph() #
    cluster.add_nodes_from(range(c)) # вузли-агрегатори не зв'язані між собою
    g = c*h+1
    #print(f'\tg = {g}')
    H = replicate(cluster, n = g)
    #print(f'\tnodes degrees {H.degree()}')
    # йтимемо "вгрупу"
    for a in range(g):      # номер групи
        edge_nodes = [('edge', a, i) for i in range(m)]
        H.add_nodes_from(edge_nodes)
        for b in range(c):  # номер вузла в групі
            node = (a, b)
            #print(f'\tfor node ({a}, {b})...')
            for t in range(h): # номер зв'язку
                bt = c-b-1
                at = (a+bt*h+t+1)%g
                target = (at, bt)
                #print(f'\t\tlink to ({at}, {bt})')
                H.add_edge(node, target, hierarchy = 2)
        for enode in edge_nodes:
            H.add_edge(node, enode)
    return H

# генерація по числу входів комутаторів
# k - бажане число входів

```

```

def dragonfly_plus_k(k):
    return dragonfly_plus(k//2, k//2, k//2)

def leaf_spin(n, m):
    if m<1 or n<1: raise Exception(f"topo.leaf_spin: bad parameters: n = {n}, m = {m}")
    G = nx.Graph()
    G.add_nodes_from([i for i in range(m+n)])
    for i in range(m):
        for j in range(m, m+n):
            G.add_edge(i, j) # всі вузли частини m зв'язуються з усіма вузлами частини n
    return G

'''
Топологія класичного FatTree визначається числом входів комутаторів (це і є ранг) складається з 4 рівнів:
1. Core
2. Agregation
3. Edge
4. Computing
Рівень ядра складається з ( $N_1 = (k/2)^2$ ) комутаторів. При k=4 це 4 вузли, при k=8 - 16 вузлів
Кожному комутатору ядра відповідають модулі (pods)
Кожен под містить k/2 комутаторів агрегації та стільки ж крайових.
Кожен комутатор агрегації в групі підключений до ( $k/2$ )=(4/2)=2 основних комутаторів і ( $k/2$ )=(4/2)=2 крайових комутаторів
Кожен периферійний комутатор у модулі підключений до ( $k/2$ )=(4/2)=2 серверів і ( $k/2$ )=(4/2)=2 комутаторів агрегації
'''

def fat_tree(rank, no_servers = True):
    G = nx.Graph()
    k2 = rank//2 # передобчислили (k/2)
    npods = k2**2 # число коренів = (k/2)^2
    G.add_nodes_from(range(npods)) # корені нумеруватимемо як числа
    for pod in range(rank): # йдемо по групах (pod-ам)
        aggregation_nodes = [('aggr', pod, i) for i in range(k2)] # агрегаторам ставимо у відповідність трійку значень (перше - мітка). Кількість в поді: k/2
        G.add_nodes_from(aggregation_nodes) #
        edge_nodes = [('edge', pod, i) for i in range(k2)] # крайовим вузлам - також. Їх кількість в поді: k/2
        G.add_nodes_from(edge_nodes) #
        for aggr in aggregation_nodes: # йдемо по агрегаторам
            id = aggr[2] # отримуємо власний номер
            for core in range(id*k2, (id+1)*k2):
                G.add_edge(aggr, core) # кожен вузол-агрегатор зв'язаний з k/2 коренями. Причому зі зсувом id*k2
            edges_down = [(aggr, enode) for enode in edge_nodes] # кожен агрегатор зв'язаний з k/2 крайовими вузлами в поді. HINT: в поді всього k/2
            крайових
            G.add_edges_from(edges_down)
            if no_servers: continue # параметр no_servers дозволяє не генерувати кінцеві вузли (тільки комутатори)
            for node in edge_nodes: # йдемо по крайовим вузлам
                e_num = node[2] #
                servers = [(pod, e_num, i) for i in range(k2)] # серверам ставитимемо у відповідність 3 значення (pod, e_num, i). Їх кількість теж k/2
                G.add_nodes_from(servers) # додаємо сервери до графа
                links = [(serv, node) for serv in servers] # кожен сервер зв'язаний зі своїм крайовим роутером
                G.add_edges_from(links) #
    return G

##### Топології-константи #####
# заготовлені константи - щоб не обтяжувати пам'ять розмноженням графів.
# добре підходить, коли треба використати граф в режимі read only
# ахтунг: не використовувати, якщо функція / клас змінюватиме графи
PETERSEN = petersen()
TUTTE = tutte()
LINE_2 = line(2)
LINE_3 = line(3)
CIRCLE_3 = circle(3)
FULL_4 = full(4)

##### Стантартні методи синтезу #####
def add_cycle(G):
    for n in G.nodes:
        G.add_edge(n, (n+1)%N)
    return G

# синтез за латинським квадратом
# сам метод банальний: обрання стовпця a - це те саме що хордове кільце з кроком a.
# зв'язок завжди двосторонній (і в плюс і в мінус), оскільки якщо x вказує на x+a, то x-a в той же час вказуватиме на x
def latin_square(N, columns):
    #return nx.circulant_graph(N, columns)
    G = nx.Graph()
    G.add_nodes_from([i for i in range(N)])

```

```

for n in G.nodes:
    for c in columns:
        y = (n+c) % N
        G.add_edge(n, y)
return G if nx.is_connected(G) else add_cycle(G)

# синтез методом лінійних рівнянь
# вид лінійного рівняння: ax+b
# a_s та b_s - масиви коефіцієнтів a та b відповідно
# якщо в якомусь з масивів значення кінчаються, то інший вважається рівним
def linear_equations(N, a_s = [], b_s = []):
    la = len(a_s)
    lb = len(b_s)
    count = max(la, lb)
    G = nx.Graph()
    G.add_nodes_from([i for i in range(N)])
    for i in range(count):
        a = a_s[i] if i<la else 0
        b = b_s[i] if i<lb else 0
        for x in G.nodes:
            y = (a*x + b) % N
            G.add_edge(x, y)
    return G if nx.is_connected(G) else add_cycle(G)

def square_equations(N, a = [], b = [], c = []):
    la = len(a)
    lb = len(b)
    lc = len(c)
    count = max(la, lb, lc)
    G = nx.Graph()
    G.add_nodes_from([i for i in range(N)])
    for i in range(count):
        _a = a[i] if i<la else 0
        _b = b[i] if i<lb else 0
        _c = c[i] if i<lc else 0
        for x in G.nodes:
            y = (_a*x*x + _b*x + _c) % N
            G.add_edge(x, y)
    return G if nx.is_connected(G) else add_cycle(G)

"""
Таблицею може бути будь-яка ітерована сутність
Наприклад - список списків
Порядок - порядок поліному (рівняння)
0 - вироджений випадок
1 - лінійні
2 - квадратні
...
якщо розмір таблиці менший за порядок - вважається, що молодші коефіцієнти рівні 0
Якщо навпаки - молодша частина таблиці ігнорується
"""

def equations(N, table, order = 2):
    G = nx.Graph()
    G.add_nodes_from([i for i in range(N)])
    for x in range(N):
        for s in table:
            # s - це серія (рядок таблиці)
            y = 0
            # акумулятор
            i = order
            # порядок
            for a in s:
                # витягаємо по чергово коефіцієнти
                if i<0: break
                # якщо порядок став від'ємним - кінець
                y += a*(x**i) % N
                # щоб не переповнити розрядну сітку - беремо модуль
                i -= 1
            y = y % N
            # беремо модуль від результату поліному
            G.add_edge(x, y)
            # додаємо ребро
    return G if nx.is_connected(G) else add_cycle(G)

# ===== Supercomputers topology =====

"""
Топологія суперкомп'ютера Фугаку.
Представляє собою групи з 12 нод, розташованих по 3 осям A, B, C (в кількості 2, 3, 3)
Ці групи, в свою чергу, об'єднуються 3D тором.
"""

def fugaku(rank):
    group = nx.grid_graph([2, 3, 3], periodic = True)
    torus3D = nx.grid_graph([rank, rank, rank], periodic = True)
    G = nx.cartesian_product(group, torus3D)

```

```

return unloop(G)

def frontier(comp_groups, manage_groups = 1, io_groups = 5, nodes_per_cg = 32, nodes_per_mg = 16, nodes_per_iog = 16):
    # створили групи 3 типів
    computing_groups = {f'C({i})': unloop(full(nodes_per_cg)) for i in range(comp_groups)}
    management_groups = {f'M({i})': unloop(full(nodes_per_mg)) for i in range(manage_groups)}
    inout_groups = {f'IO({i})': unloop(full(nodes_per_iog)) for i in range(io_groups)}
    # об'єднали групи різних типів воедино
    G_C = nx.union_all(computing_groups.values(), rename = computing_groups.keys())
    G_M = nx.union_all(management_groups.values(), rename = management_groups.keys())
    G_IO = nx.union_all(inout_groups.values(), rename = inout_groups.keys())
    # генеруємо внутрішні зв'язки груп вводу-виводу
    for i in range(io_groups):
        for j in range(io_groups):
            if i != j: G_IO.add_edge(f'IO({i})_0', f'IO({j})_0')
    # зводимо граф воедино
    G = nx.union_all([G_C, G_M, G_IO])
    # генеруємо всі зовнішні зв'язки
    for i in range(manage_groups):
        for j in range(comp_groups): G.add_edge(f'M({i})_0', f'C({j})_0') # зв'язуємо керування та обчислення
        for j in range(io_groups):
            G.add_edge(f'M({i})_0', f'IO({j})_0') # зв'язуємо керування та IO
            for k in range(comp_groups): G.add_edge(f'IO({j})_0', f'C({k})_0') # зв'язуємо IO та обчислення (1 внутр.цикл - для економії часу)
    return G

```

Лістинг А.5 – модуль *Characteristics.py*

```

import networkx as nx
import itertools as iter
import math
import pandas as pd
import numpy as np

from numeric import NumericSystems as ns
from numeric import NumericGraphs as ngraphs
from numeric import NumericGenerator as gen
from numeric import NumericPaint as paint

##### Вузлові характеристики #####

def degrees(G, values_only = True):
    degs = nx.degree(G)
    return [s[1] for s in degs] if values_only else degs

def betweenes(G, precission = None, normalized = False):
    return nx.betweenness_centrality(G, k=precission, normalized = normalized)

##### Топологічні характеристики #####
# у всіх функцій обрахунку (навіть якщо вони рахуються від вже наявних характеристик) має бути версія виду f(G, characteristics = None)
# це треба для автообрахунку в циклі

# UPD 30.11.2024
# get використовується для отримання передобчислених даних.
# Оскільки ця ситуація є занадто рідкісною - було вирішено винести її із функцій. Все одно це не використовується ніде крім таблиці

def get(name, characteristics, func):
    if characteristics != None:
        X = characteristics.get(name, -1)
        if X != -1: return X
    return func()

def nodes_count(G, characteristics = None):
    return G.number_of_nodes()

def edges_count(G, characteristics = None):
    return G.number_of_edges()

def degree(G, characteristics = None):
    degs = nx.degree(G)
    s_vals = [s[1] for s in degs]
    return max(s_vals)

def diameter(G, characteristics = None):
    return nx.diameter(G)

```



```

def avg_diameter(G, characteristics = None):
    return nx.average_shortest_path_length(G)

# use formulae  $T = N \cdot aD/R$ 
def traffic(G, characteristics = None, regularized = False):
    aD = get('aD', characteristics, nx.average_shortest_path_length(G))
    if regularized:
        S = get('S', characteristics, degree(G))
        return 2*aD/S
    else:
        N = G.number_of_nodes()
        R = G.number_of_edges()
        return N*aD/R

def cost(G, characteristics = None):
    N = G.number_of_nodes()
    S = get('S', characteristics, lambda: degree(G))
    return N*S

def local_connectivity(G, characteristics = None):
    degs = nx.degree(G)
    s_vals = [s[1] for s in degs]
    return min(s_vals)

# коефіцієнт "багатого клубу" (число ребер до потенційного числа ребер всіх вершин зі степенем більше k)
# щоб привести його до числа, я його усереднюю: беру суму і ділю на число елементів
def mrich(G, characteristics = None):
    r = nx.rich_club_coefficient(G)
    return sum(r.values())/len(r)

"Середнє посередництво"
def mbetw(G, characteristics = None):
    betws = betweenes(G, precission = None, normalized = True)
    return sum(betws)/len(betws)

# не будемо включати вартість у список, бо вартість - це наближена оцінка R, а ми його і так можемо оцінити
CHAR = {
    "N": nodes_count,          # число вузлів
    "R": edges_count,          # число ребер (воно ж фактична вартість)
    "S": degree,               # ступінь
    "D": diameter,             # діаметр
    "SD": lambda G, characteristics = None: degree(G, characteristics = characteristics)*diameter(G, characteristics = characteristics), #SD-характеристика
    "aD": avg_diameter,        # середній діаметр
    "T": traffic,               # топологічний трафік
    "C": cost,                  # вартість (по стандартній формулі)
    "L": local_connectivity,    # локальна зв'язність
    # "aL": lambda G, characteristics = None: nx.average_node_connectivity(G), # середня зв'язність - краще не рахувати, бо гальмує дуже сильно
    "Gn": lambda G, characteristics = None: nx.node_connectivity(G),    # глобальна вузлова зв'язність
    "Ge": lambda G, characteristics = None: nx.edge_connectivity(G),    # глобальна реберна зв'язність
    "El": lambda G, characteristics = None: nx.local_efficiency(G),      # лок. ефективність (те ж, що і глоб. еф-ть, але на підграфі сусідів кожної вершини)
    "Eg": lambda G, characteristics = None: nx.global_efficiency(G),      # глобальна ефективність (метрика, протилежна довжині найкоротших шляхів)
    # "Cc": lambda G, characteristics = None: nx.clustering(G),            # коефіцієнт кластеризації
    # "mRich": mrich,               # середній коефіцієнт багатого клубу
    # "estrada": lambda G, characteristics = None: nx.estrada_index(G)      # індекс Естрада (3-вимірна компактність)
}

SCHAR = {name: CHAR[name] for name in ["N", "R", "S", "D", "SD", "aD", "T", "C"]}
FTCHAR = {name: CHAR[name] for name in ["L", "Gn", "Ge"]}
ACHAR = {name: CHAR[name] for name in ["El", "Eg"]}

def subchar(*names):
    return {name: CHAR[name] for name in names}

def characteristics(G, functions = SCHAR):
    results = {}
    for name in functions:
        function = functions[name]
        X = function(G, characteristics = results)
        results[name] = X
    return results

##### Таблиці характеристик #####
# функція генерації таблиці характеристик для будь-якої топології
# відразу генерує pandas.DataFrame
# f_generator - функція, що генерує граф

```

```

# єдина вимога до функції-генератора - мати аргумент "rank", що міститься в першому полі. Це має визначати структуру графа.
# Інші аргументи можна передати через спеціальні аргументи args і kwargs. Вони однакові для графа в процесі генерації.
# functions - перелік характеристик, які вимірюються
# ranks - діапазон рангів
# Nmax - необов'язковий аргумент, що припиняє масштабування по досягненню графом певного розміру
# add_graph - булевий аргумент, що визначає, чи додається згенерований граф до таблиці. Це має сенс якщо є плани в подальшому якось працювати з цими графами.
def table(f_generator, args = None, kwargs = None, functions = SCHAR, ranks = range(2, 8), Nmax = -1, add_graph = False):
    frame = pd.DataFrame(columns = list(functions.keys()))

    # створюємо генератор через лямбди, щоб не рахувати розгалуження в циклі
    if args != None:
        if kwargs != None:
            generator = lambda rank: f_generator(rank, *args, **kwargs)
        else:
            generator = lambda rank: f_generator(rank, *args)
    else:
        if kwargs != None:
            generator = lambda rank: f_generator(rank, **kwargs)
        else:
            generator = f_generator

    for rank in ranks:
        G = generator(rank)
        N = len(G)
        if (Nmax != -1) and (N > Nmax): break
        try:
            chars = characteristics(G, functions = functions)
        except Exception as e:
            print(f'Exception occurs on rank {rank}: {e}')
            edges_colors = paint.color_incluster_edges(G)
            labels = {n: f'{ns.label(n)}({ns.label(G.nodes[n].get("index", "!")))}' for n in G.nodes}
            paint.paint(G, edges_colors = edges_colors, labels = labels)
            return table
        if add_graph: chars['graph'] = G
        frame.loc[rank] = chars
    return frame

"""
Функція, що генерує таблицю для графів.
graphs - ітерований об'єкт: список чи словник
якщо словник - іменами графів є ключі словника
якщо список - іменами графів є індекси або відповідні індекси списку names
functions - характеристики
names - імена
"""

def table_for(graphs, functions = SCHAR, names = None, Nmax = -1, add_graph = False):
    frame = pd.DataFrame()

    if type(graphs) == list:
        g = {}
        if names != None:
            for i in range(len(graphs)):
                index = names[i] if i < len(names) else i
                g[index] = graphs[i]
        else:
            g = {i: graphs[i] for i in range(len(graphs))}
        graphs = g

    for rank in graphs:
        G = graphs[rank]
        N = len(G)
        if (Nmax != -1) and (N > Nmax): break
        chars = characteristics(G, functions = functions)
        if add_graph: chars['graph'] = G
        frame.loc[rank] = chars
    return frame

# Просунута функція генерації
# не потребує лямбд та креаторів
# потрібно лише передати функції всі необхідні параметри (лінкер, алфавіт, виключення та (опціонально) основу) і вона сама створює потрібний креатор
def numeric_table(f_linker, functions = SCHAR, alphabet = ns.BINARY, exclusions = [], base = 0, ranks = range(2, 8), Nmax = -1, add_graph = False):
    if base <= 0:
        f_g = lambda rank: gen.simple_creator(rank, f_linker = f_linker, alphabet = alphabet, exclusions = exclusions)
    else:
        f_g = lambda rank: gen.clustered_creator(rank, f_linker = f_linker, alphabet = alphabet, base = base, exclusions = exclusions)
    return table(f_g, functions = functions, ranks = ranks, Nmax = Nmax, add_graph = add_graph)

```

Збір даних для багатьох топологій одночасно

```
class Experimental_Data:
    def __init__(self):
        self.data = {'name':[], 'is_numeric':[], 'f':[], 'args':[], 'kwargs':[], 'alphabet':[], 'exclusions':[], 'base':[], 'ranks':[]}
        self.index = 0

    def append(self, name, is_numeric, f, args=None, kwargs=None, alphabet=None, exclusions=None, base=0, ranks=range(2, 8)):
        self.data['name'].append(name)
        self.data['is_numeric'].append(is_numeric)
        self.data['f'].append(f)
        self.data['args'].append(args)
        self.data['kwargs'].append(kwargs)
        self.data['alphabet'].append(alphabet)
        self.data['exclusions'].append(exclusions)
        self.data['base'].append(base)
        self.data['ranks'].append(ranks)
        self.index += 1

    def add_generator(self, name, f_generator, ranks = range(2, 8)):
        self.append(name, False, f_generator, ranks=ranks)

    def add_pure(self, name, f_generator, args = None, kwargs = None, ranks = range(2, 8)):
        self.append(name, False, f_generator, args=args, kwargs=kwargs, ranks=ranks)

    def add_numeric(self, name, f_linker, alphabet = ns.BINARY, exclusions = [], base = 0, ranks = range(2, 8)):
        self.append(name, True, f_linker, alphabet=alphabet, exclusions=exclusions, base=base, ranks=ranks)

    def get(self, index, col):
        return self.data[col][index]

    def __getitem__(self, key):
        return {k: self.data[k][key] for k in self.data}

    def __len__(self):
        return self.index

    def as_frame(self):
        return pd.DataFrame(self.data)

def create_tables(experimental_data, chars = SCHAR, nmax = 4096, add_graph = False, logging = True):
    result = {}
    for i in range(len(experimental_data)):
        name = experimental_data.get(i, 'name')
        f = experimental_data.get(i, 'f')
        ranks = experimental_data.get(i, 'ranks')
        is_numeric = experimental_data.get(i, 'is_numeric')

        #print(f'{i}: {name} for {str(ranks)}')

        if is_numeric:
            alphabet = experimental_data.get(i, 'alphabet')
            exclusions = experimental_data.get(i, 'exclusions')
            base = experimental_data.get(i, 'base')
            #print(f'\t{i}: a = {alphabet}, ex = {exclusions}, base = {base}')
            try:
                tab = numeric_table(f,
                                    functions = chars,
                                    alphabet = alphabet,
                                    exclusions = exclusions,
                                    base = base,
                                    ranks = ranks,
                                    Nmax = nmax,
                                    add_graph = add_graph)
            except Exception as e:
                print(f'\nTABLE FOR {name}: ({ns.label(alphabet)} with base {base}, ex {exclusions}): exception {e}')
                tab = None

            if logging:
                print(f'\nTABLE FOR {name}: ({ns.label(alphabet)} with base {base}, ex {ns.label(exclusions)})\n')
                print(tab)
        else:
            args = experimental_data.get(i, 'args')
            kwargs = experimental_data.get(i, 'kwargs')
            tab = table(f,
```

```

        args = args,
        kwargs = kwargs,
        functions = chars,
        ranks = ranks,
        Nmax = nmax,
        add_graph = add_graph)
    if logging:
        print(f'\nTABLE FOR {name}:\n')
        print(tab)
    result[name] = tab
    result['characteristics'] = chars.keys()
return result

##### Обробка характеристик #####

def define_labels(count):
    if count == 2: return ['Low', 'High']
    if count == 3: return ['S', 'M', 'L']
    if count == 4: return ['S', 'M', 'L', 'XL']
    return [f'G{i}' for i in range(count)]

def define_groups(Nmax, count, exponential = False, base = 2):
    splits = [] # точки розбиття (к-кість буде count-1)
    if exponential:
        power = math.floor(math.log(Nmax, base)) # беремо логарифм з округленням вниз. Це - найближча до Nmax ступінь base
        if count > power:
            count = power # силою приводимо число груп до степеня, якщо розбити не вдається
            step = 1
        else:
            step = power // count # крок зміни
        i = 0
        for i in range(count-1):
            power -= step
            point = base ** power # точками є base ** power.
            # Наприклад, при N=4096, count = 4, b=2 буде: power = 12, step = 3. G4 = [2^9, 2^12], G3 = [2^6, 2^9], ...
            splits.append(point)
        splits.reverse()
    else:
        step = Nmax // count
        splits = [i*step for i in range(1, count)]

    splits.append(Nmax)
    labels = define_labels(count)
    res = {}
    p1 = 0
    for i in range(count):
        p2 = splits[i]
        res[labels[i]] = (p1, p2)
        p1 = p2
    return res

def fetch_group(frame, column, start, end):
    rows = []
    for i in frame.index:
        row = frame.loc[i]
        N = row.loc[column]
        if N >= start and N <= end:
            rows.append(row)
    return pd.DataFrame(rows)

##### Обробка характеристик #####

# розділяє фрейм по типам характеристик, припускаючи, що характеристики в колонках таблиці
def split_types(frame):
    std = frame.loc[:, [col for col in frame.columns if col in SCHAR.keys()]]
    ft = frame.loc[:, [col for col in frame.columns if col in FTCHAR.keys()]]
    add = frame.loc[:, [col for col in frame.columns if col in ACHAR.keys()]]
    return [std, ft, add]

# зменшує фрейм, замінюючи набір значень середнім значенням
def reduce(frame):
    return frame.mean(axis = 0) # беремо середнє значення по рядкам в кожній колонці. Має повернутись pd.Series, індексоване колонками

DEFGROUPS = {'S': (0, 64), 'M': (64, 256), 'L': (256, 1024), 'XL': (1024, 65536)}

# ріже фрейм на частини у відповідності до заданих груп, збираючи результат у словник, де мітці відповідає серія усереднених значень
# Якщо значень немає - повертає None
# повертає словник, де міткам груп ставляться у відповідність усереднені характеристики

```

```

def split_reduce(frame, groups = DEFGROUPS):
    reduced = {}
    for label in groups:
        rng = groups[label]
        subframe = fetch_group(frame, 'N', rng[0], rng[1]-1)
        if not subframe.empty:
            reduced[label] = reduce(subframe) # dataframe -> series
    return reduced

# ріже словник результатів на групи.
# Результат - словник фреймів по групам, де рядок - топологія, стовпець - характеристика, на перетині - середнє значення характеристики топології в діапазоні.
#
def split_reduce_all(result, groups = DEFGROUPS):
    # Mtable = pd.MultiIndex()
    chars = result['characteristics']
    meta = {g: pd.DataFrame(index = [k for k in result.keys() if k != 'characteristics'], columns = chars) for g in groups}
    for name in result:
        if name == 'characteristics': continue
        reduced = split_reduce(result[name], groups = groups) # отримуємо скорочений фрейм для даної топології
        for group in reduced:
            series = reduced.get(group, None)
            for c in chars:
                meta[group].loc[name, c] = series.loc[c] # додаємо щось у фрейм лише якщо дані є. Якщо даних нема - топологія не додається в групу, от і все

# Видалення порожніх значень
rm = []
for group in meta:
    meta[group].dropna(axis = 0, how = 'any', inplace = True)
    if meta[group].empty:
        rm.append(group)
for k in rm:
    del meta[k]
return meta

MORE_BETTER = ['N', 'L', 'aL', 'Gn', 'Ge', 'El', 'Eg']
ONE_BETTER = ['T']

# уніфікуємо дані так, щоб найменші значення були оптимальними
# Потім - нормалізуємо всі дані так, щоб вони попадали в область [1, max/min]
def unify_normalize(frame):
    nframe = pd.DataFrame(index = frame.index, columns = frame.columns)
    frame = frame.copy()
    for char in ONE_BETTER:
        if not char in frame.columns: continue
        for i in frame.index:
            v = frame.loc[i, char]
            if v < 1:
                frame.loc[i, char] = 2 - v

    for char in frame.columns:
        slice = frame.loc[:, char]
        cmin = slice.min()
        if cmin == 0: cmin = 0.0001
        if char in MORE_BETTER:
            cmax = slice.max()
            for i in frame.index:
                v = frame.loc[i, char]
                nframe.loc[i, char] = 1+(cmax-v)/cmin
        else:
            for i in frame.index:
                v = frame.loc[i, char]
                res = v/cmin
                if res<1: res +=1
                nframe.loc[i, char] = res
    return nframe

"""
Зібрані та відфільтровані дані після split_reduce_all
"""
def unify_normalize_all(reduced):
    normalized = {}
    for group in reduced:
        normalized[group] = unify_normalize(reduced[group])
    return normalized

"""
Отримання нормалізованих оцінок в групах (стовпці) по топологіям (рядки)

```

```

'''
def normalized_mark(normalized, dropchars = ['N', 'SD']):
    marks = {}
    for group in normalized:
        frame = normalized[group]
        marks[group] = {}
        for name in frame.index:
            row = frame.loc[name]
            s = 0
            for char in row.index:
                if char in dropchars: continue
                s += row.loc[char]
            marks[group][name] = s
    return pd.DataFrame(marks)

```

Лістинг А.6 – модуль FaultTolerance.py

```

import networkx as nx
import itertools as iter
import math
import pandas as pd
import random as rand

from topologies import Characteristics as chars

# поле властивостей вузла/ребра, що позначає відмову. False - вузол справний, True - відмова
# це поле є сенс тиражувати і на інші класи
DEF_FAULTFIELD = 'faulted'

# ремаркує граф, додаючи вузлам/ребрам поля відмов
def remark(graph, copy=False, faultfield = DEF_FAULTFIELD):
    G = graph.copy() if copy else graph
    for n in G.nodes:
        G.nodes[n][faultfield] = False
    for e in G.edges:
        G.get_edge_data(*e)[faultfield] = False
    return G

##### Потоки відмов #####
# УВАЖНО! ПОТІК ВІДМОВ (для економії пам'яті) МОДИФІКУЄ ГРАФИ!
# Для безпечності треба перед експериментом робити G.copy()
# Також синтаксис функції має включати 1 обов'язковий параметр і будь-яку кількість необов'язкових (в ідеалі kwargs)
# потік відмов завжди повертає граф з вилученими вершинами чи ребрами

# Поле faultfield (str) змінює логіку роботи: тепер замість того, щоб видаляти вузол з графа, система записує True в поле відповідного вузла
# це означає, що вузол став несправним

# pure random. Nodes only
def fault_rand(G, seed = None, k = 1, faultfield = None):
    if seed != None: rand.seed(seed)
    if k < 1 or k >= len(G.nodes): k=1
    faults = rand.sample(list(G.nodes), k = k) # завжди повертає ітерований об'єкт, навіть при k=1
    if faultfield == None:
        G.remove_nodes_from(faults)
    else:
        for fault in faults:
            G.nodes[fault][faultfield] = True
    return {'G': G, 'faults': faults, 'par': []}

# by maximal betweenes. Nodes only
def fault_maxbetwn(G, seed = None, k = 1, faultfield = None):
    if k < 1 or k >= len(G.nodes): k=1
    betw = chars.betweenes(G) # це - словник
    faults = []
    fbetw = []
    while len(faults) < k:
        n = None
        b = -1
        for node in G.nodes:
            betweenes = betw[node]
            if betweenes > b:
                n = node
                b = betweenes
        if n == None:
            break

```

```

else:
    faults.append(n)
    fbetw.append(betw[n])
if faultfield == None:
    G.remove_nodes_from(faults)
else:
    for fault in faults:
        G.nodes[fault][faultfield] = True
return {'G': G, 'faults': faults, 'par': fbetw}

# by maximal betweenes. Nodes only
"""Ідея тут така: ми нормалізуємо значення посередництва і потім генеруємо випадкове число від 0 до 1.
Чим більше посередництво - тим більший сегмент рулетки співвідноситься з відповідною ногою. Отже, зростає імовірність вибірки саме цієї
ноги."""
def fault_maxbetwn_rand(G, betw = None, seed = None, k = 1, faultfield = None):
    if seed != None: rand.seed(seed)
    if k < 1 or k >= len(G.nodes): k=1
    if betw == None: betw = chars.betweenes(G, normalized = True) # це - словник
    faults = []
    par = []
    seq = list(betw.keys()) # послідовність, з якої відбувається вибірка
    weights = list(betw.values()) # ваги вибірки
    while len(faults)<k:
        f = rand.choices(seq, weights, k=1)
        fault = f[0]
        faults.append(fault)
        i = seq.index(fault)
        par.append(weights[i])
        del seq[i]
        del weights[i]
    if faultfield == None:
        G.remove_nodes_from(faults)
    else:
        for fault in faults: G.nodes[fault][faultfield] = True
    return {'G': G, 'faults': faults, 'par': par}

##### Експерименти #####
"""Почергові відмови
Дано: граф G, потік відмов.
Вихід: таблиця / графік (pd.DataFrame)
по осі абсцис (рядки) - множина точок відмови (видалених вершин) або число відмов
по осі ординат (стовпці) - характеристики:
- Стандартні топологічні характеристики (ступінь, діаметр, сер. діаметр, ...)
- Зв'язність (середня локальна, реберна, вузлова)
- Середня живучість (обчислюємо живучості і беремо середнє. При роз'єднанні графа отримаємо -INF)
Постановка експерименту:
Дано граф. В ньому за певним правилом відмовляють вершини. Питання: як зміниться та чи інша характеристика графа після N відмов?

Аргументи:
graph - граф
f_fault - потік відмов
seed - зерно випадкового генератора
functions - параметри, що вимірюються
mode - режим зміни числа вершин (-d - віднімання d вершин, /d - ділення на d, *d - зміна в d разів відносно початкового значення)
d - крок зміни
batchfault - чи допускається одночасна відмова ряду вершин, чи потік має все одно генерувати відмови почергово (навіть якщо проміжні стани не
вимірюються)
f_label - функція помітки номерів вершин
cumfaults - чи писати у виводі весь перелік вершин, що відмовили з самого початку і до кроку j, чи лише ті, що відмовили конкретно на кроці j
save_graphs - чи зберігати копії графа G в словнику-результаті?
save_param - чи зберігати параметр, за яким відбувався відбір (str)?
"""
# MODES = ["-d", "/d", "*d"]
# різниця між "/d" та "*d" в тому, що /d ділить граф відносно поточного розміру, а *d - відносно початкового
def experiment_1(graph,
    f_fault,
    seed = None,
    functions = chars.CHAR,
    mode = '-d',
    d=1,
    batchfault = True,
    f_label = None,
    cumfaults = False,
    save_graphs = False,
    save_param = False
):
    G = graph.copy()
    N = len(graph)

```

```

results = {
    'faults': [],
    'fcount': []
}
if save_param:
    results['par'] = []
for c in functions:
    results[c] = []          # ініціалізуємо масив результатів
if save_graphs:
    results['G'] = []

if (mode == '/d') and (k <= 1): d = 1.01    # ініціалізуємо крок відмови
if (mode == '-d'):
    d = math.floor(d)
    if d < 1: d = 1
if (mode == '*d') and (d <= 0.0) and (d >= 1.0): d = 0.1
if seed != None: rand.seed(seed)

faults = []
fcount = 0
step = 0

k = d if mode == '-d' else math.floor(G.number_of_nodes()*d) if (mode == '*d') else 1 # для режимів -d та *d розраховуємо k статично
if k < 1: k=1                                # k за визначенням не може бути менше 1 чи більше N
if k >= G.number_of_nodes(): k = G.number_of_nodes() - 1          # не можна видалити більше вершин ніж є
par = ""

while nx.is_connected(G) and len(G)>=2:
    # запис інформації
    res = chars.characteristics(G, functions = functions)
    results['fcount'].append(fcount)          # число відмов - це ціле число, яке показує, скільки вершин відмовило до цього часу
    if f_label == None:                      # f_label - функція розмітки (представлення нод у текст). Формат f_label(node) return str
        results['faults'].append(' '.join(str(faults.copy())))) # якщо її немає - додаємо копію поточного стану списку faults як рядок через join
    else:
        results['faults'].append(' '.join(map(f_label, faults))) # якщо є функція розмітки - робимо мапінг і додаємо результат як рядок через join
    for c in res:
        results[c].append(res[c])           # вносимо в словник всі розраховані характеристики

    if save_graphs:
        results['G'].append(G.copy())
    if save_param:
        results['par'].append(par)
    # модифікація графа
    if (mode == '/d'):                      # перерахунок потрібен лише в режимі /d
        P = G.number_of_nodes()           # P - поточне число працездатних вузлів
        k = P - (P/d)                     # від графа ми віднімаємо вузли так, щоб їх число стало P/d
        if k < 1: k = 1                   # число відмов на крок не може бути менше 1
        if k > P: break                   # якщо число відмов на крок перевищило число вузлів - виконання алгоритму не має сенсу

    if batchfault:
        fault = f_fault(G, k = k)
        stepfaults = fault['faults']
        G = fault['G']
        if save_param:
            param = fault['par']
            s = [str(p) for p in param]
            par = ' '.join(s)
        else:
            fault = [f_fault(G, k=1) for i in range(k)]
            stepfaults = [fault[i]['faults'] for i in range(k)]
            if save_param:
                s = [str(fault[i]['par']) for i in range(k)]
                par = ' '.join(s)
            G = fault[k-1]['G']
    if cumfaults:
        faults.append(*stepfaults)
    else:
        faults = stepfaults
    step += 1
    fcount += k

return {'results': results, 'G': G} if save_graphs else results

##### Посередництво #####

def lv(graph, speed, seed = None, use_betw = False):
    G = graph.copy()

```



```

result = {'N': [], 'fcount': []}
N = len(G)
if seed != None: rand.seed(seed)

fcount = 0
stop = 2 + speed
while nx.is_connected(G) and len(G) >= stop:
    result['N'].append(len(G))
    result['fcount'].append(fcount)

    if use_betw: faults = fault_maxbetwn_rand(G, k = speed)
    else: faults = fault_rand(G, k = speed)

    fcount += speed
return result

def liveability(graph, speed = 1, f_label = None, normalized = True, seed = None, use_betw = False, save_betw = False, save_graphs = False):
    G = graph.copy()
    result = {'N': [], 'fcount': []}
    if save_betw:
        for node in G.nodes:
            key = node if f_label == None else f_label(node)
            result[key] = []
    if save_graphs: result['G'] = []
    N = len(G)
    if seed != None: rand.seed(seed)

    fcount = 0
    step = 0

    stop = 2 + speed
    while nx.is_connected(G) and len(G) >= stop:
        result['N'].append(len(G))
        result['fcount'].append(fcount)

        if use_betw:
            betweenes = nx.betweenness(G, normalized = normalized)
            if save_betw:
                for node in graph.nodes:
                    key = node if f_label == None else f_label(node)
                    result[key] = betweenes.get(node, None)
            # sum = sum([betweenes[key] for key in betweenes])
            # betw = {key: betweenes[key]/_sum for key in betweenes}
            faults = fault_maxbetwn_rand(G, betw = betweenes, k = speed)
        else: faults = fault_rand(G, k = speed)

        G = faults['G']
        if save_graphs: result['G'].append(G.copy())

        step += 1
        fcount += speed
    return result

```

Додаток Б

Список публікацій здобувача.

1. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., **Honcharenko, O.**, & Korenko, D. (2022). Extended DragonDeBruijn topology synthesis method. *International Journal of Computer Network and Information Security*, 9(6), 23. DOI: [10.5815/ijcnis.2022.06.03](https://doi.org/10.5815/ijcnis.2022.06.03)
2. Volokyta, A., Loutskii, H., **Honcharenko, O.**, Cherevatenko, O., Rusinov, V., Kulakov, Y., & Tsybulia, S. (2023). Fault Tolerance Exploration and SDN Implementation for de Bruijn Topology based on betweenness Coefficient. *Computer Network and Information Security*, 5 (pp. 1-17). DOI: [10.5815/ijcnis.2024.01.08](https://doi.org/10.5815/ijcnis.2024.01.08)
3. **Гончаренко, О.**, & Череватенко, О. (2021). СПОСОБИМУЛЬТИКАНАЛЬНОЇ МАРШРУТИЗАЦІЇ В МЕРЕЖАХ НАДЛИШКОВОГО ДЕ БРУЙНА. *Технічні науки та технології*, (2 (24)), 123-130. DOI: [10.25140/2411-5363-2021-2\(24\)-123-130](https://doi.org/10.25140/2411-5363-2021-2(24)-123-130)
4. **Гончаренко, О.**, & Волокита, А. (2024). МЕТОД СИНТЕЗУ ВІДМОВОСТІЙКИХ ТОПОЛОГІЙ З ІМПЛІЦИТНИМИ КЛАСТЕРАМИ НА ОСНОВІ ПЕРЕТВОРЕНЬ ДЕ БРУЙНА В НАДЛИШКОВИХ СИСТЕМАХ ЧИСЛЕННЯ. *Проблеми інформатизації та управління*, (4 (80)), 20-27. DOI: <https://doi.org/10.18372/2073-4751.80.19784>
5. Rusinov, V., **Honcharenko, O.**, Volokyta, A., Loutskii, H., Pustovit, O., & Kyrianov, A. (2023, March). Methods of Topological Organization Synthesis Based on Tree and Dragonfly Combinations. In *International Conference on Computer Science, Engineering and Education Applications* (pp. 472-485). Cham: Springer Nature Switzerland. DOI: [10.1007/978-3-031-36118-0_43](https://doi.org/10.1007/978-3-031-36118-0_43)
6. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., **Honcharenko, O.**, & Korenko, D. (2021). Topology synthesis method based on excess de bruijn and dragonfly. In *Advances in Computer Science for Engineering and Education*

- IV* (pp. 315-325). Springer International Publishing. DOI: [10.1007/978-3-030-80472-5_27](https://doi.org/10.1007/978-3-030-80472-5_27)
7. Loutskii, H., Volokyta, A., Rehida, P., **Honcharenko, O.**, & Thinh, V. D. (2021). Method for synthesis scalable fault-tolerant multi-level topological organizations based on excess code. In *Advances in Computer Science for Engineering and Education III 3* (pp. 350-362). Springer International Publishing. DOI: [10.1007/978-3-030-55506-1_32](https://doi.org/10.1007/978-3-030-55506-1_32)
 8. Loutskii, H., Volokyta, A., Rehida, P., **Honcharenko, O.**, Ivanishchev, B., & Kaplunov, A. (2019, December). Increasing the fault tolerance of distributed systems for the Hyper de Bruijn topology with excess code. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)* (pp. 1-6). IEEE. DOI: [10.1109/ATIT49449.2019.9030487](https://doi.org/10.1109/ATIT49449.2019.9030487)
 9. **Honcharenko, O.**, Volokyta, A., & Loutskii, H. (2024, August). Method of fault tolerant routing in distributed systems based on non-binary de brujin topology. In *The International Conference on Security, Fault Tolerance, Intelligence*. Link: <https://icsfti-proc.kpi.ua/article/view/307020>

Додаток В

Вплив алфавіту на розподіл альтернативних представлень

В.1. Опис експерименту

Гіпотеза експерименту полягає у тому, що для будь-якого не-аномального алфавіту розподіл альтернативних представлень залежить лише від розміру алфавіту k і основи числення b . Склад алфавіту впливає лише на зміщення і не міняє самої форми розподілу.

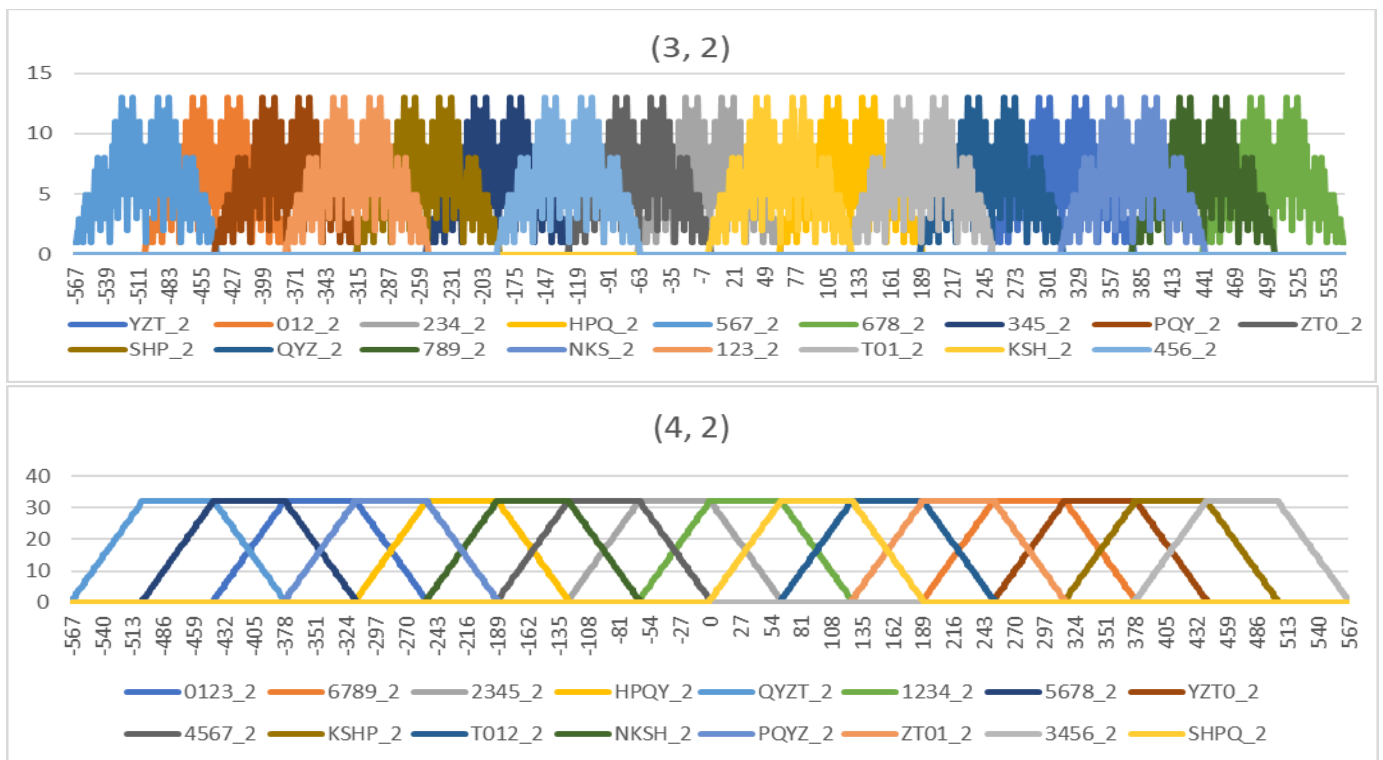
Для перевірки даної гіпотези пропонується сформувати розподіл представлень для систем числення. Параметрами експерименту є довжина алфавіту k , основа b і довжина (ранг) коду m . Генерація виконується за наступними правилами:

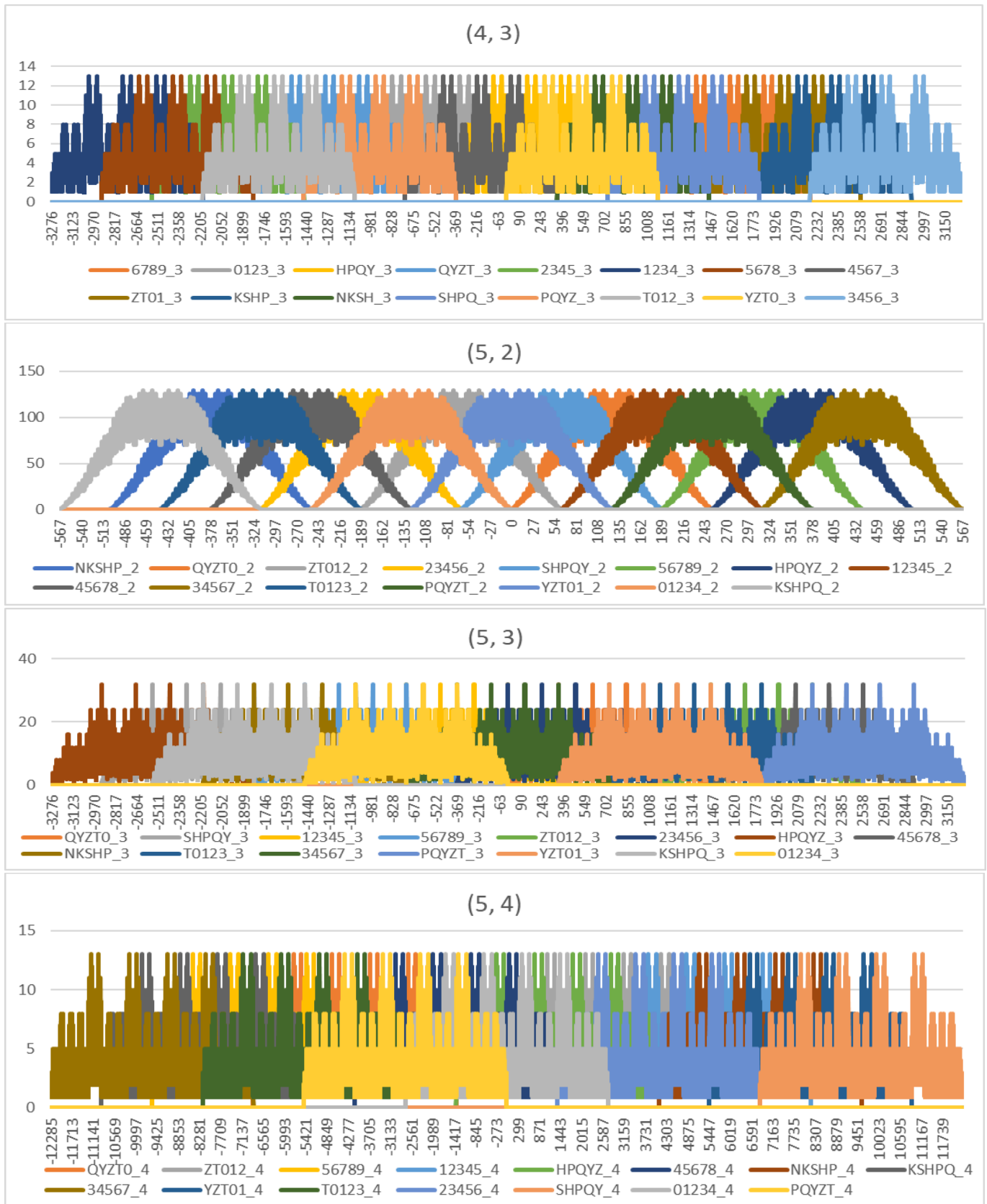
- Алфавіти можуть містити цифри від -9 до 9. При цьому в рамках експерименту використовуються наступні позначки негативних цифр: T (-1), Z (-2), Y (-3), Q (-4), P (-5), H (-6), S (-7), K (-8), N (-9).
- Алфавіт довжиною k містить всі цифри від деякого t до $t+k-1$ без пропусків чи повторень (виконується умова нормальності алфавіту).
- В рамках одного експерименту генеруються всі можливі k -значні алфавіти, що задовольняють умовам вище.
- Система числення і довжина кодів є спільною для всіх алфавітів.
- Результатом експерименту є таблиця (.csv), де стовпці позначають різні коди, а рядки – число альтернативних представлень для певного числа. Значення 0 у комірці означає, що певна m -розрядна система не може представити зазначене число.

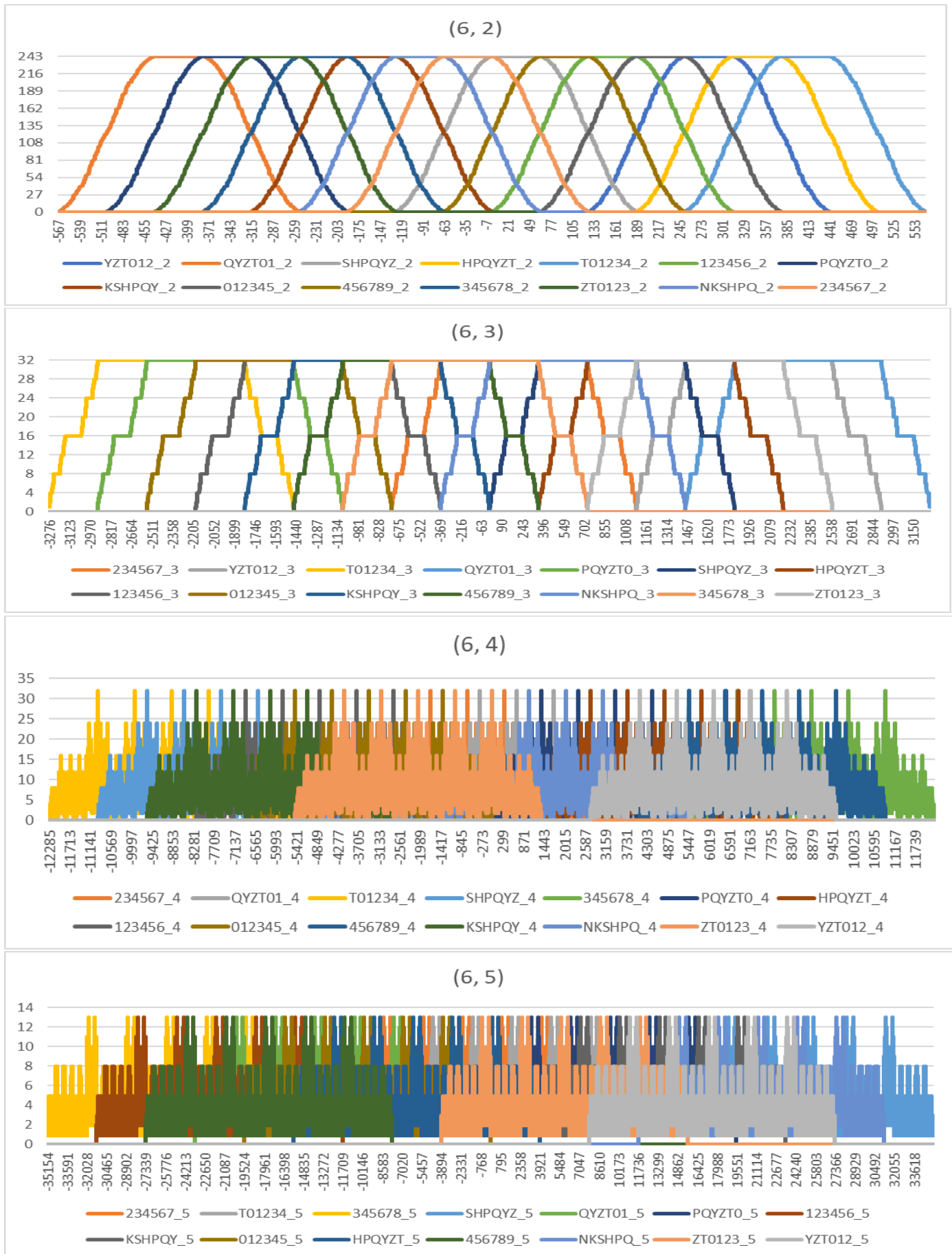
Для зручного відображення даних доцільно використати графічне представлення розподілу, де осі абсцис відповідатимуть числам, осі ординат – кількість альтернативних представлень відповідного числа, а різні графіки представлятимуть різні можливі алфавіти для відповідної системи.

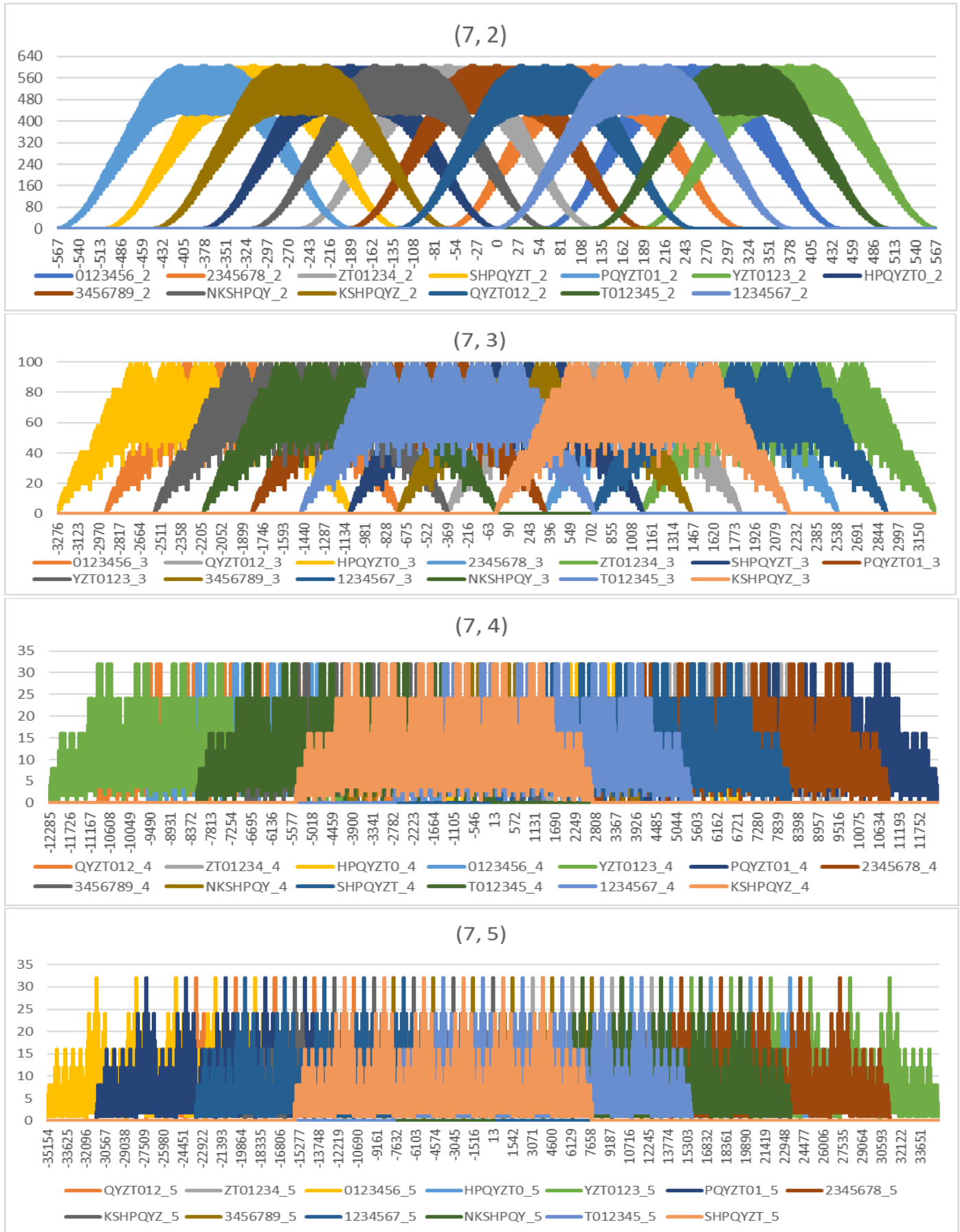
В.2. Експериментальні дані

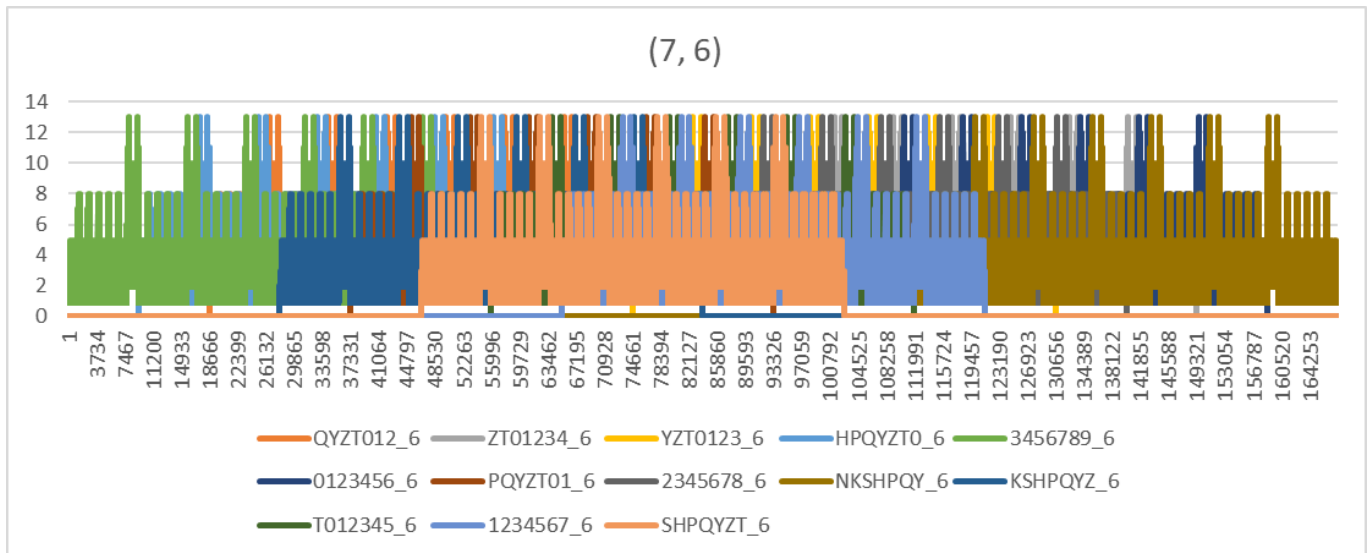
Для кодів із невеликою довжиною r і невеликою розмірністю алфавіту k можливо виконати повний перебір. В якості вихідних даних було обрано k від 3 до 7 і $r = 6$. Такі параметри обрано через те, що будь-які небінарні коди в цифрових системах представляються у вигляді групи бітів – таким чином, дуже великі алфавіти є недоцільними, оскільки потребуватимуть великої кількості цифр і суттєво збільшать число елементарних операцій при роботі з кодами. Щодо довжини, значення було обрано з огляду на той факт, що, з одного боку, в контексті високопродуктивних обчислень немає сенсу досліджувати невеликі r , а з іншого, програмна генерація розподілу потребує повного перебору. При $r = 6$ результат передбачає N від 3^6 до 7^6 , що дозволяє досягти компромісу між часом симуляції і практичною цінністю результатів.











Додаток Г

Аналіз характеристик топологічних організацій

Г.1. Попередні дані

Запропонований метод працює з двома типами зв'язків – з гіперкубічними (перетворення заміщення) та де Бруйнівськими (перетворення зсуву). Оскільки для топології ключовою характеристикою є довжина алфавіту k , на даному етапі немає сенсу враховувати інші параметри, такі як основа числення чи структура алфавіту. Ці параметри мають значення лише при формуванні кластерів.

Результати експерименту представлені у таблиці Г.1.

Табл. Г.1

Дані для «чистих» мереж (де Бруйн та гіперкуб)

Парам.			Тип зв'язків													
			Гіперкубічні							Де Бруйнівські						
k	R	N	L	S	D	\bar{D}	T	C	SD	L	S	D	\bar{D}	T	C	SD
2	2	4	2	2	2	1.3333	1.3333	8	4	2	3	2	1.1667	0.7778	12	6
	3	8	3	3	3	1.7143	1.1429	24	9	2	4	3	1.6429	0.8214	32	12
	4	16	4	4	4	2.1333	1.0667	64	16	2	4	4	2.1417	1.0708	64	16
	5	32	5	5	5	2.5806	1.0323	160	25	2	4	5	2.754	1.377	128	20
	6	64	6	6	6	3.0476	1.0159	384	36	2	4	6	3.4534	1.7267	256	24
	7	128	7	7	7	3.5276	1.0079	896	49	2	4	7	4.2148	2.1074	512	28
	8	256	8	8	8	4.0157	1.0039	2048	64	2	4	8	5.028	2.514	1024	32
	9	512	9	9	9	4.5088	1.002	4608	81	2	4	9	5.8844	2.9422	2048	36
3	2	9	4	4	2	1.5	0.75	36	8	4	5	2	1.4167	0.5667	45	10
	3	27	6	6	3	2.0769	0.6923	162	18	4	6	3	2.0769	0.6923	162	18
	4	81	8	8	4	2.7	0.675	648	32	4	6	4	2.8333	0.9444	486	24
	5	243	10	10	5	3.3471	0.6694	2430	50	4	6	5	3.6738	1.2246	1458	30
	6	729	12	12	6	4.0055	0.6676	8748	72	4	6	6	4.5722	1.5241	4374	36
4	2	16	6	6	2	1.6	0.5333	96	12	6	7	2	1.55	0.4429	112	14
	3	64	9	9	3	2.2857	0.5079	576	27	6	8	3	2.3214	0.5804	512	24
	4	256	12	12	4	3.0118	0.502	3072	48	6	8	4	3.1781	0.7945	2048	32

Продовження табл. Г.1

Парам.			Тип зв'язків														
			Гіперкубічні							Де Бруйнівські							
k	R	k	R	k	R	k	R	k	R	k	R	k	R	k	R	k	
5	2	25	8	8	2	1.6667	0.4167	200	16	8	9	2	1.6333	0.363	225	18	
	3	125	12	12	3	2.4194	0.4032	1500	36	8	10	3	2.471	0.4942	1250	30	
	4	625	16	16	4	3.2051	0.4006	10000	64	8	10	4	3.3779	0.6756	6250	40	
7	2	49	12	12	2	1.75	0.2917	588	24	12	13	2	1.7321	0.2665	637	26	
	3	343	18	18	3	2.5789	0.2865	6174	54	12	14	3	2.6391	0.377	4802	42	

Маючи ці дані, можна виконати нормалізацію і оцінити, які з розглянутих мережевих топологій мають кращі значення основних топологічних характеристик. Для аналізу було виконано поділ на 3 приблизно однакові за розміром групи у відповідності до N : таким чином, можна відслідкувати поведінку кожної топології з масштабуванням. Враховуючи, що діаметр у гіперкуба та де Бруйна є рівним, його було виділено окремим стовпцем. Також, не розглядалась сукупна характеристику SD , оскільки інакше вплив характеристик S і D на оцінку буде невиправдано підвищеним, а отже, буде складно оцінити реальний стан. Нижче представлені груповані усереднені характеристики (табл. Г.2) та норми по групам (табл. Г.3).

Табл. Г.2

Груповані та усереднені експериментальні дані (без кластерів)

k	Група	D	Гіперкубічні мережі				Дебруйнівські мережі			
			S	\bar{D}	T	C	S	\bar{D}	T	C
2	S (1-32)	3.5	3.5	1.94039939	1.1437788	64	3.75	1.92630568	1.21216078	59
	M (33-216)	6.5	6.5	3.28758905	1.01187352	640	4	3.834093	1.9170465	384
	L (217-729)	8.5	8.5	4.26224627	1.00293926	3328	4	5.45623309	2.72811655	1536
3	S (1-32)	2.5	5	1.78846154	1.27884615	99	5.5	1.74679487	1.37051282	103.5
	M (33-216)	4	8	2.7	1.325	648	6	2.83333333	1.05555556	486
	L (217-729)	5.5	11	3.67630097	1.33149805	5589	6	4.12302568	1.37434189	2916
4	S (1-32)	2	6	1.6	1.46666667	96	7	1.55	1.55714286	112
	M (33-216)	3	9	2.28571429	1.49206349	576	8	2.32142857	1.41964286	512
	L (217-729)	4	12	3.01176471	1.49803922	3072	8	3.178125	1.20546875	2048
5	S (1-32)	2	8	1.66666667	1.58333333	200	9	1.63333333	1.63703704	225
	M (33-216)	3	12	2.41935484	1.59677419	1500	10	2.47096774	1.50580645	1250
	L (217-729)	4	16	3.20512821	1.59935897	10000	10	3.37794872	1.32441026	6250
7	M (33-216)	2	12	1.75	1.70833333	588	13	1.73214286	1.73351648	637
	L (217-729)	3	18	2.57894737	1.71345029	6174	14	2.63909774	1.62298604	4802

Табл. Г.3

Норми по групам

Група	D	S	\bar{D}	T	C
S (1-32)	2	3.5	1.55	1.1437788	59
M (33-216)	2	4	1.73214286	1.01187352	384
L (217-729)	3	4	2.57894737	1.00293926	1536

Маючи норми, можна виконати нормування згрупованих усереднених значень і отримати суми, за якими можна комплексно порівняти якість топологічних організацій одна відносно одної (Табл. Г.4).

Табл. Г.4

Нормалізовані усереднені значення

K	Група	D	Гіперкубічні мережі					Дебруйнівські мережі				
			S	\bar{D}	T	C	СУМА	S	\bar{D}	T	C	СУМА
2	S (1-32)	1.750	1.000	1.252	1.000	1.085	6.087	1.071	1.243	1.060	1.000	6.124
	M (33-216)	3.250	1.625	1.898	1.000	1.667	9.440	1.000	2.213	1.895	1.000	9.358
	L (217-729)	2.833	2.125	1.653	1.000	2.167	9.778	1.000	2.116	2.720	1.000	9.669
3	S (1-32)	1.250	1.429	1.154	1.118	1.678	6.628	1.571	1.127	1.198	1.754	6.901
	M (33-216)	2.000	2.000	1.559	1.309	1.688	8.556	1.500	1.636	1.043	1.266	7.445
	L (217-729)	1.833	2.750	1.426	1.328	3.639	10.975	1.500	1.599	1.370	1.898	8.201
4	S (1-32)	1.000	1.714	1.032	1.282	1.627	6.656	2.000	1.000	1.361	1.898	7.260
	M (33-216)	1.500	2.250	1.320	1.475	1.500	8.044	2.000	1.340	1.403	1.333	7.577
	L (217-729)	1.333	3.000	1.168	1.494	2.000	8.995	2.000	1.232	1.202	1.333	7.101
5	S (1-32)	1.000	2.286	1.075	1.384	3.390	9.135	2.571	1.054	1.431	3.814	9.870
	M (33-216)	1.500	3.000	1.397	1.578	3.906	11.381	2.500	1.427	1.488	3.255	10.170
	L (217-729)	1.333	4.000	1.243	1.595	6.510	14.681	2.500	1.310	1.321	4.069	10.533
7	M (33-216)	1.000	3.000	1.010	1.688	1.531	8.230	3.250	1.000	1.713	1.659	8.622
	L (217-729)	1.000	4.500	1.000	1.708	4.020	12.228	3.500	1.023	1.618	3.126	10.268

Г.2. Аналіз без формування кластерів (до N = 4096)

Аналогічний аналіз було виконано для всіх k від 2 до 10 з обмеженням на число вершин $N_{\max} = 4096$ на основі розробленого ПЗ для моделювання топологічних характеристик. Результати представлено в таблицях Г.5 – Г.22.

Табл. Г.5

Експериментальні результати для гіперкуба ($k=2$)

	N	R	S	D	SD	aD	T	C
2	4	4	2	2	4	1.333	1.333	8
3	8	12	3	3	9	1.714	1.143	24
4	16	32	4	4	16	2.133	1.067	64
5	32	80	5	5	25	2.581	1.032	160
6	64	192	6	6	36	3.048	1.016	384
7	128	448	7	7	49	3.528	1.008	896
8	256	1024	8	8	64	4.016	1.004	2048
9	512	2304	9	9	81	4.509	1.002	4608
10	1024	5120	10	10	100	5.005	1.001	10240
11	2048	11264	11	11	121	5.503	1.000	22528
12	4096	24576	12	12	144	6.001	1.000	49152

Табл. Г.6

Експериментальні результати для графа де Бруїна ($k=2$)

	N	R	S	D	SD	aD	T	C
2	4	5	3	2	6	1.167	0.933	12
3	8	13	4	3	12	1.643	1.011	32
4	16	29	4	4	16	2.142	1.182	64
5	32	61	4	5	20	2.754	1.445	128
6	64	125	4	6	24	3.453	1.768	256
7	128	253	4	7	28	4.215	2.132	512
8	256	509	4	8	32	5.028	2.529	1024
9	512	1021	4	9	36	5.884	2.951	2048
10	1024	2045	4	10	40	6.774	3.392	4096
11	2048	4093	4	11	44	7.689	3.847	8192
12	4096	8189	4	12	48	8.623	4.313	16384

Табл. Г.7

Експериментальні результати для гіперкуба ($k=3$)

	N	R	S	D	SD	aD	T	C
2	9	18	4	2	8	1.500	0.750	36
3	27	81	6	3	18	2.077	0.692	162
4	81	324	8	4	32	2.700	0.675	648
5	243	1215	10	5	50	3.347	0.669	2430
6	729	4374	12	6	72	4.005	0.668	8748
7	2187	15309	14	7	98	4.669	0.667	30618

Табл. Г.8

Експериментальні результати для графа де Бруїна (k=3)

	N	R	S	D	SD	aD	T	C
2	9	21	5	2	10	1.417	0.607	45
3	27	75	6	3	18	2.077	0.748	162
4	81	237	6	4	24	2.833	0.968	486
5	243	723	6	5	30	3.674	1.235	1458
6	729	2181	6	6	36	4.572	1.528	4374
7	2187	6555	6	7	42	5.506	1.837	13122

Табл. Г.9

Експериментальні результати для гіперкуба (k=4)

	N	R	S	D	SD	aD	T	C
2	16	48	6	2	12	1.600	0.533	96
3	64	288	9	3	27	2.286	0.508	576
4	256	1536	12	4	48	3.012	0.502	3072
5	1024	7680	15	5	75	3.754	0.500	15360
6	4096	36864	18	6	108	4.501	0.500	73728

Табл. Г.10

Експериментальні результати для графа де Бруїна (k=4)

	N	R	S	D	SD	aD	T	C
2	16	54	7	2	14	1.550	0.459	112
3	64	246	8	3	24	2.321	0.604	512
4	256	1014	8	4	32	3.178	0.802	2048
5	1024	4086	8	5	40	4.099	1.027	8192
6	4096	16374	8	6	48	5.054	1.264	32768

Табл. Г.11

Експериментальні результати для гіперкуба (k=5)

	N	R	S	D	SD	aD	T	C
2	25	100	8	2	16	1.667	0.417	200
3	125	750	12	3	36	2.419	0.403	1500
4	625	5000	16	4	64	3.205	0.401	10000
5	3125	31250	20	5	100	4.001	0.400	62500

Табл. Г.12

Експериментальні результати для графа де Бруїна (k=5)

	N	R	S	D	SD	aD	T	C
2	25	110	9	2	18	1.633	0.371	225
3	125	610	10	3	30	2.471	0.506	1250
4	625	3110	10	4	40	3.378	0.679	6250
5	3125	15610	10	5	50	4.334	0.868	31250

Табл. Г.13

Експериментальні результати для гіперкуба (k=6)

	N	R	S	D	SD	aD	T	C
2	36	180	10	2	20	1.714	0.343	360
3	216	1620	15	3	45	2.512	0.335	3240
4	1296	12960	20	4	80	3.336	0.334	25920

Табл. Г.14

Експериментальні результати для графа де Бруйна (k=6)

	N	R	S	D	SD	aD	T	C
2	36	195	11	2	22	1.690	0.312	396
3	216	1275	12	3	36	2.570	0.435	2592
4	1296	7755	12	4	48	3.505	0.586	15552

Табл. Г.15

Експериментальні результати для гіперкуба (k=7)

	N	R	S	D	SD	aD	T	C
2	49	294	12	2	24	1.750	0.292	588
3	343	3087	18	3	54	2.579	0.287	6174
4	2401	28812	24	4	96	3.430	0.286	57624

Табл. Г.16

Експериментальні результати для графа де Бруйна (k=7)

	N	R	S	D	SD	aD	T	C
2	49	315	13	2	26	1.732	0.269	637
3	343	2373	14	3	42	2.639	0.381	4802
4	2401	16779	14	4	56	3.592	0.514	33614

Табл. Г.17

Експериментальні результати для гіперкуба (k=8)

	N	R	S	D	SD	aD	T	C
2	64	448	14	2	28	1.778	0.254	896
3	512	5376	21	3	63	2.630	0.250	10752
4	4096	57344	28	4	112	3.501	0.250	114688

Табл. Г.18

Експериментальні результати для графа де Бруйна (k=8)

	N	R	S	D	SD	aD	T	C
2	64	476	15	2	30	1.764	0.237	960
3	512	4060	16	3	48	2.690	0.339	8192
4	4096	32732	16	4	64	3.654	0.457	65536

Табл. Г.19

Експериментальні результати для гіперкуба (k=9)

	N	R	S	D	SD	aD	T	C
2	81	648	16	2	32	1.800	0.225	1296
3	729	8748	24	3	72	2.670	0.223	17496

Табл. Г.20

Експериментальні результати для графа де Бруйна (k=9)

	N	R	S	D	SD	aD	T	C
2	81	684	17	2	34	1.789	0.212	1377
3	729	6516	18	3	54	2.729	0.305	13122

Табл. Г.21

Експериментальні результати для гіперкуба (k=10)

	N	R	S	D	SD	aD	T	C
2	100	900	18	2	36	1.818	0.202	1800
3	1000	13500	27	3	81	2.703	0.200	27000

Табл. Г.22

Експериментальні результати для графа де Бруйна (k=10)

	N	R	S	D	SD	aD	T	C
2	100	945	19	2	38	1.809	0.191	1900
3	1000	9945	20	3	60	2.759	0.277	20000

На основі отриманих даних програмним засобом було виконано групування та нормалізацію характеристик. Результати для різних груп представлено в табл. Г.23 – Г.26.

Табл. Г.23

Нормалізовані значення для групи S (0-63)

	S	D	SD	aD	T	C
HC 2	1.000	1.750	1.125	1.252	1.001	1.085
HC 3	1.429	1.250	1.083	1.154	1.119	1.678
HC 4	1.714	1.000	1.000	1.032	1.284	1.627
HC 5	2.286	1.000	1.333	1.075	1.386	3.390
HC 6	2.857	1.000	1.667	1.106	1.450	6.102
HC 7	3.429	1.000	2.000	1.129	1.495	9.966
DB 2	1.071	1.750	1.125	1.243	1.000	1.000
DB 3	1.571	1.250	1.167	1.127	1.157	1.754
DB 4	2.000	1.000	1.167	1.000	1.348	1.898
DB 5	2.571	1.000	1.500	1.054	1.425	3.814
DB 6	3.143	1.000	1.833	1.091	1.477	6.712
DB 7	3.714	1.000	2.167	1.118	1.514	10.797

Табл. Г.24

Нормалізовані значення для групи М (64-255)

	S	D	SD	aD	T	C
HC 2	1.625	3.250	1.771	1.864	1.000	1.667
HC 3	2.250	2.250	1.708	1.714	1.312	4.008
HC 4	2.250	1.500	1.125	1.296	1.475	1.500
HC 5	3.000	1.500	1.500	1.372	1.578	3.906
HC 6	3.750	1.500	1.875	1.424	1.646	8.438
HC 8	3.500	1.000	1.167	1.008	1.726	2.333
HC 9	4.000	1.000	1.333	1.020	1.754	3.375
HC 10	4.500	1.000	1.500	1.031	1.777	4.688
DB 2	1.000	3.250	1.083	2.174	1.927	1.000
DB 3	1.500	2.250	1.125	1.845	1.089	2.531
DB 4	2.000	1.500	1.000	1.316	1.380	1.333
DB 5	2.500	1.500	1.250	1.401	1.476	3.255
DB 6	3.000	1.500	1.500	1.457	1.546	6.750
DB 8	3.750	1.000	1.250	1.000	1.742	2.500
DB 9	4.250	1.000	1.417	1.014	1.767	3.586
DB 10	4.750	1.000	1.583	1.026	1.787	4.948

Табл. Г.25

Нормалізовані значення для групи L (256-1023)

	S	D	SD	aD	T	C
HC 2	2.125	2.833	2.266	1.653	1.000	2.167
HC 3	3.000	2.000	2.250	1.553	1.329	5.695
HC 4	3.000	1.333	1.500	1.168	1.494	2.000
HC 5	4.000	1.333	2.000	1.243	1.595	6.510
HC 7	4.500	1.000	1.688	1.000	1.708	4.020
HC 8	5.250	1.000	1.969	1.020	1.744	7.000
HC 9	6.000	1.000	2.250	1.035	1.772	11.391
HC 10	6.750	1.000	2.531	1.048	1.795	17.578
DB 2	1.000	2.833	1.063	2.116	2.732	1.000
DB 3	1.500	2.000	1.125	1.773	1.524	2.848
DB 4	2.000	1.333	1.000	1.232	1.194	1.333
DB 5	2.500	1.333	1.250	1.310	1.317	4.069
DB 7	3.500	1.000	1.313	1.023	1.614	3.126
DB 8	4.000	1.000	1.500	1.043	1.656	5.333
DB 9	4.500	1.000	1.688	1.058	1.690	8.543
DB 10	5.000	1.000	1.875	1.070	1.717	13.021

Табл. Г.26

Нормалізовані значення для групи XL (1024-4096)

	S	D	SD	aD	T	C
HC 2	2.750	2.750	2.897	1.650	1.000	2.857
HC 3	3.500	1.750	2.333	1.400	1.332	3.204
HC 4	4.125	1.375	2.179	1.237	1.499	4.661
HC 5	5.000	1.250	2.381	1.199	1.599	6.539
HC 6	5.000	1.000	1.905	1.000	1.665	2.712
HC 7	6.000	1.000	2.286	1.028	1.713	6.029
HC 8	7.000	1.000	2.667	1.049	1.749	12.000
DB 2	1.000	2.750	1.048	2.307	3.848	1.000
DB 3	1.500	1.750	1.000	1.651	1.836	1.373
DB 4	2.000	1.375	1.048	1.372	1.145	2.143
DB 5	2.500	1.250	1.190	1.299	1.132	3.270
DB 6	3.000	1.000	1.143	1.051	1.413	1.627
DB 7	3.500	1.000	1.333	1.077	1.485	3.517
DB 8	4.000	1.000	1.524	1.095	1.542	6.857

Маючи нормалізовані характеристики (ідеальний результат – 1, вищі значення – гірші), можна сформувані нормалізовані оцінки для топологій по групам (табл. Г.27). Мітки HC позначають гіперкубічні графи, DB – дебруйнівські. Цифра – порядок системи числення (2 – двійкова, 3 – трійкова, тощо). В нормалізованих оцінках не враховувалась характеристика SD.

Табл. Г.27

Нормалізовані оцінки

	DB 2	DB 3	DB 4	DB 5	DB 6	DB 7	DB 8	DB 9	DB 10
S	6.06	6.88	7.35	10.12	13.93	19.01			
M	9.35	9.22	7.50	10.10	14.25		10.01	11.65	13.56
L	9.68	9.65	7.09	10.53		10.24	13.01	16.77	21.79
XL	10.91	8.11	8.03	9.45	8.09	10.58	14.49		
	HC 2	HC 3	HC 4	HC 5	HC 6	HC 7	HC 8	HC 9	HC 10
S	6.19	6.78	6.81	9.45	13.08	17.94			
M	9.43	11.60	8.04	11.42	16.89		9.60	11.20	13.07
L	9.79	13.60	9.00	14.71		12.24	16.04	21.24	28.24
XL	11.01	11.19	12.90	15.59	11.38	15.77	22.81		

Г.3. Аналіз топологічної ефективності та зв'язності

Табл. Г.28 ілюструє топологічні характеристики локальної, глобальної реберної та глобальної вузлової зв'язності, а також локальної та глобальної ефективності двійкового гіперкуба та двійкового де Бруйна.

Табл. Г.28

Зв'язність та ефективність бінарних гіперкуба та де Бруйна

	Гіперкуб						Де Бруйн					
	N	L	Gn	Ge	El	Eg	N	L	Gn	Ge	El	Eg
2	4	2	2	2	0	0.833	4	2	2	2	0.917	0.917
3	8	3	3	3	0	0.690	8	2	2	2	0.542	0.714
4	16	4	4	4	0	0.572	16	2	2	2	0.229	0.560
5	32	5	5	5	0	0.477	32	2	2	2	0.115	0.438
6	64	6	6	6	0	0.401	64	2	2	2	0.057	0.346
7	128	7	7	7	0	0.342	128	2	2	2	0.029	0.277
8	256	8	8	8	0	0.295	256	2	2	2	0.014	0.227
9	512	9	9	9	0	0.258	512	2	2	2	0.007	0.190
10	1024	10	10	10	0	0.229	1024	2	2	2	0.004	0.162
11	2048	11	11	11	0	0.205	2048	2	2	2	0.002	0.140
12	4096	12	12	12	0	0.186	4096	2	2	2	0.001	0.123

Також було проаналізовано дебруйнівські графи на основі кодів числення з довжиною алфавіту 3, 4 та 6. Табл. 29 ілюструє результати для них.

Табл. Г.29

Зв'язність та ефективність дебруйнівських надлишкових графів

	Трійковий						Четвірковий					
	N	L	Gn	Ge	El	Eg	N	L	Gn	Ge	El	Eg
2	9	4	4	4	0.778	0.792	16	6	6	6	0.706	0.725
3	27	4	4	4	0.258	0.558	64	6	6	6	0.162	0.487
4	81	4	4	4	0.075	0.403	256	6	6	6	0.036	0.346
5	243	4	4	4	0.025	0.303	1024	6	6	6	0.009	0.260
6	729	4	4	4	0.008	0.237	4096	6	6	6	0.002	0.207
7	2187	4	4	4	0.003	0.192						
	Шестірковий											
	N	L	Gn	Ge	El	Eg						
2	36	10	10	10	0.622	0.655						
3	216	10	10	10	0.087	0.423						
4	1296	10	10	10	0.013	0.301						

В табл. Г.30 – нормалізовані результати по всіх групах, в табл. Г.31 – нормалізовані оцінки по характеристикам зв'язності та ефективності (крім локальної ефективності, оскільки її нормалізована оцінка виявилась всюди рівна нескінченності через ділення на 0, і без розділення на глобальну реберну та глобальну вузлову зв'язності, оскільки вони рівні і їх практичний сенс є дуже схожим)

Табл. Г.30

Нормалізовані результати по групах										
	S (0-63)					M (64-255)				
	L	Gn	Ge	El	Eg	L	Gn	Ge	El	Eg
DB 2	5.00	5.00	5.00	inf	1.11	5.00	5.00	5.00	inf	1.56
DB 3	4.00	4.00	4.00	inf	1.08	4.00	4.00	4.00	inf	1.43
DB 4	3.00	3.00	3.00	inf	1.00	3.00	3.00	3.00	inf	1.00
DB 6	1.00	1.00	1.00	inf	1.11	1.00	1.00	1.00	inf	1.20
HC 2	4.25	4.25	4.25	inf	1.13	2.75	2.75	2.75	inf	1.37
	L (256-1023)					XL (1024-4096)				
	L	Gn	Ge	El	Eg	L	Gn	Ge	El	Eg
DB 2	4.25	4.25	4.25	inf	1.66	5.50	5.50	5.50	inf	2.13
DB 3	3.25	3.25	3.25	inf	1.52	4.50	4.50	4.50	inf	1.77
DB 4	2.25	2.25	2.25	inf	1.00	3.50	3.50	3.50	inf	1.48
DB 6						1.50	1.50	1.50	inf	1.00
HC 2	1.00	1.00	1.00	inf	1.33	1.00	1.00	1.00	inf	1.67

Табл. Г.31

Нормалізовані оцінки за характеристиками зв'язності та ефективності

	S	M	L	XL
DB 2	16.105	16.564	14.408	18.627
DB 3	13.078	13.430	11.272	15.267
DB 4	10.000	10.000	7.750	11.976
DB 6	4.109	4.205		5.500
HC 2	13.877	9.621	4.329	4.665

Г.4. Аналіз впливу імпліцитних кластерів

А табл. Г.32 – Г.34 ілюструють характеристики дебруйнівських графів з кластерами. Конкретні типи кодів вказано в форматі (k , b).

Табл. Г.32

Характеристики трійкового графа де Бруйна на основі коду (3, 2)

	N	R	S	D	aD	T	L	El	Eg
2	9	21	5	2	1.417	0.607	4	0.778	0.792
3	27	83	7	3	2.003	0.652	4	0.301	0.578
4	81	285	9	4	2.641	0.751	4	0.088	0.434
5	243	931	10	5	3.358	0.877	4	0.030	0.334
6	729	2981	11	6	4.145	1.014	4	0.010	0.264
7	2187	9459	12	7	4.983	1.152	4	0.003	0.216

Табл. Г.33

Характеристики четвіркових графів де Бруйна

	Однакові			Код (4, 2)						Код (4, 3)					
	N	D	L	R	S	aD	T	El	Eg	R	S	aD	T	El	Eg
2	16	2	6	56	8	1.533	0.438	0.706	0.733	55	8	1.542	0.448	0.709	0.729
3	64	3	6	290	10	2.170	0.479	0.197	0.520	262	10	2.250	0.550	0.169	0.502
4	256	4	6	1330	11	2.867	0.552	0.046	0.384	1096	11	3.037	0.709	0.037	0.362
5	1024	5	6	5878	12	3.623	0.631	0.013	0.297	4444	12	3.906	0.900	0.009	0.274
6	4096	6	6	25602	13	4.432	0.709	0.003	0.238	17860	13	4.830	1.108	0.002	0.217

Табл. Г.34

Характеристики шестіркових графів де Бруйна

	Однакові			Код (6, 2)						Код (6, 3)					
	N	D	L	R	S	aD	T	El	Eg	R	S	aD	T	El	Eg
2	36	2	10	211	13	1.665	0.284	0.625	0.667	204	12	1.676	0.296	0.621	0.662
3	216	3	10	1611	16	2.342	0.314	0.124	0.466	1455	14	2.439	0.362	0.086	0.448
4	1296	4	10	11103	18	3.012	0.352	0.035	0.352	9525	15	3.229	0.439	0.013	0.329

Г.5. Характеристики класичних та популярних графів

Табл. Г.35-Г. представляють характеристики класичних топологій комп'ютерних систем. В якості контрольних було обрано такі топології як гіперкуб (характеристики для нього приведені у попередніх таблицях – див. топологію НС 2), класичні 2, 3 та 6-вимірні тори.

Табл. Г.35

Характеристики 2-вимірного тора

	N	R	S	D	aD	T	L	Eg
4	16	32	4	4	2.133333	1.066667	4	0.572222
8	64	128	4	8	4.063492	2.031746	4	0.316345
12	144	288	4	12	6.041958	3.020979	4	0.217511
16	256	512	4	16	8.031373	4.015686	4	0.165633
20	400	800	4	20	10.02506	5.012531	4	0.133714
24	576	1152	4	24	12.02087	6.010435	4	0.112102
28	784	1568	4	28	14.01788	7.00894	4	0.096502
32	1024	2048	4	32	16.01564	8.00782	4	0.084712
36	1296	2592	4	36	18.0139	9.00695	4	0.075489
40	1600	3200	4	40	20.01251	10.00625	4	0.068076
44	1936	3872	4	44	22.01137	11.00568	4	0.061989
48	2304	4608	4	48	24.01042	12.00521	4	0.056901
52	2704	5408	4	52	26.00962	13.00481	4	0.052585
56	3136	6272	4	56	28.00893	14.00447	4	0.048877
60	3600	7200	4	60	30.00834	15.00417	4	0.045658
64	4096	8192	4	64	32.00781	16.00391	4	0.042836

Табл. Г.36

Характеристики 3-вимірного тора

	N	R	S	D	aD	T	L	Eg
2	8	12	3	3	1.714286	1.142857	3	0.690476
3	27	81	6	3	2.076923	0.692308	6	0.564103
4	64	192	6	6	3.047619	1.015873	6	0.401323
5	125	375	6	6	3.629032	1.209677	6	0.329032
6	216	648	6	9	4.52093	1.506977	6	0.267593
7	343	1029	6	9	5.157895	1.719298	6	0.231421
8	512	1536	6	12	6.011742	2.003914	6	0.199757
9	729	2187	6	12	6.675824	2.225275	6	0.178407
10	1000	3000	6	15	7.507508	2.502503	6	0.159199
11	1331	3993	6	15	8.18797	2.729323	6	0.145166
12	1728	5184	6	18	9.005211	3.001737	6	0.132297
13	2197	6591	6	18	9.696721	3.23224	6	0.122383
14	2744	8232	6	21	10.50383	3.501276	6	0.113166
15	3375	10125	6	21	11.20332	3.73444	6	0.105793
16	4096	12288	6	24	12.00293	4.000977	6	0.098869

Табл. Г.37

Характеристики 6-вимірного тора

	N	R	S	D	aD	T	L	Eg
2	64	192	6	6	3.047619	1.015873	6	0.401323
3	729	4374	12	6	4.005495	0.667582	12	0.280769
4	4096	24576	12	12	6.001465	1.000244	12	0.186012

Також було проаналізовано додаткові графи, що використовуються в сучасних комп'ютерних системах. Рис. Г.1 демонструє структуру топології суперкомп'ютера Фуґаку, що складається з тороїдальних груп із 12 вершин (осі А, В, С, що мають статичний розмір 2, 3 і 3 відповідно) і поєднані між собою масштабованим 3-вимірним тором. Реальний суперкомп'ютер містить 158 976 вузлів.

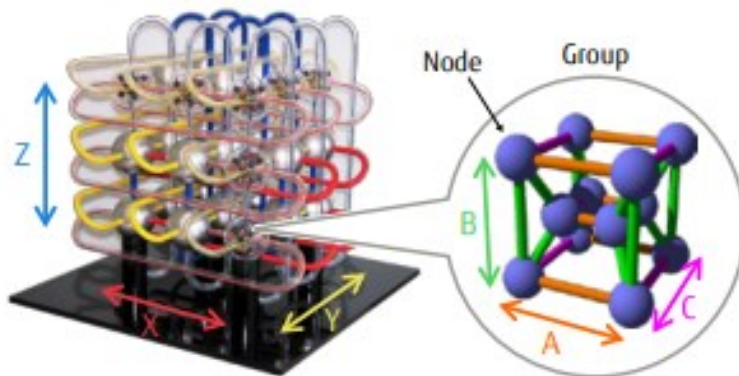


Рис. Г.1 – структура топології суперкомп'ютера Fugaku [239]

Результати щодо топологічних характеристик представлені в табл. Г.38. Масштабування відбувалось за правилами 3-вимірного тору по всім 3 вимірам, інтеграція груп виконана через декартовий добуток.

Табл. Г.38

Характеристики топології суперкомп'ютера Fugaku

	N	R	S	D	aD	T	L	Eg	SD
2	144	576	8	6	3.356643	0.839161	8	0.35373	48
3	486	2673	11	6	3.841237	0.698407	11	0.296357	66
4	1152	6336	11	9	4.837533	0.879551	11	0.235045	99
5	2250	12375	11	9	5.435749	0.988318	11	0.205574	99
6	3888	21384	11	12	6.334963	1.151811	11	0.177133	132

Також проаналізовано такі топології як жирне дерево, dragonfly та dragonfly+. Жирне дерево, використане для аналізу, було сформовано у відповідності до числа портів комутаторів (ступінь S) і складалось із 3 типів комутаторів: кореневих, агрегуючих та крайових вузлів (рис. Г.2). Обчислювальні вузли не аналізувались, оскільки їх параметри не критичні для мережі і лише зіпсують статистику.

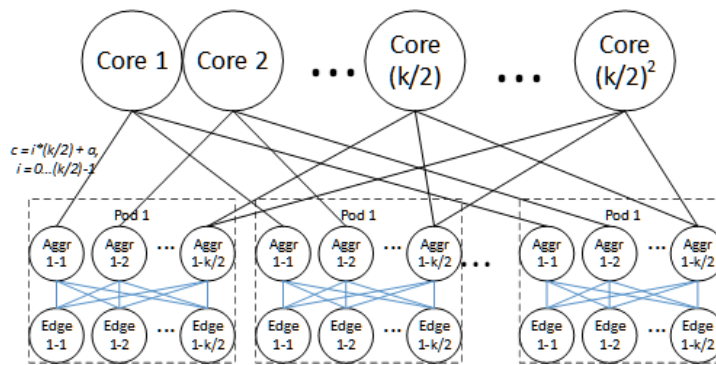


Рис. Г.2. Структура жирного дерева

Результати отриманого жирного дерева – в табл. Г.39.

Табл. Г.39

Характеристики жирного дерева (ранг – ступінь S комутаторів)

	N	R	S	D	aD	T	L	Eg	SD
4	20	32	4	4	2.589474	1.618421	2	0.475439	16
6	45	108	6	4	2.781818	1.159091	3	0.427273	24
8	80	256	8	4	2.881013	0.900316	4	0.403376	32
10	125	500	10	4	2.941935	0.735484	5	0.388978	40
12	180	864	12	4	2.98324	0.621508	6	0.37933	48
14	245	1372	14	4	3.013115	0.538056	7	0.372404	56
16	320	2048	16	4	3.035737	0.474334	8	0.367189	64
18	405	2916	18	4	3.053465	0.424092	9	0.363119	72
20	500	4000	20	4	3.067735	0.383467	10	0.359853	80
22	605	5324	22	4	3.07947	0.34994	11	0.357174	88
24	720	6912	24	4	3.089291	0.321801	12	0.354937	96
26	845	8788	26	4	3.09763	0.297849	13	0.353041	104
28	980	10976	28	4	3.104801	0.277214	14	0.351413	112

30	1125	13500	30	4	3.111032	0.259253	15	0.35	120
32	1280	16384	32	4	3.116497	0.243476	16	0.348762	128
34	1445	19652	34	4	3.12133	0.22951	17	0.347669	136
36	1620	23328	36	4	3.125633	0.217058	18	0.346695	144
38	1805	27436	38	4	3.12949	0.205888	19	0.345824	152
40	2000	32000	40	4	3.132966	0.19581	20	0.345039	160
42	2205	37044	42	4	3.136116	0.186674	21	0.344328	168
44	2420	42592	44	4	3.138983	0.178351	22	0.343682	176
46	2645	48668	46	4	3.141604	0.170739	23	0.343091	184
48	2880	55296	48	4	3.144008	0.16375	24	0.342549	192
50	3125	62500	50	4	3.146223	0.157311	25	0.342051	200
52	3380	70304	52	4	3.148269	0.151359	26	0.34159	208
54	3645	78732	54	4	3.150165	0.145841	27	0.341164	216
56	3920	87808	56	4	3.151927	0.140711	28	0.340767	224

Для мережі dragonfly було обрано схожий принцип масштабування на основі виділеного на мережу ступеня, причому зв'язки було розділено навпіл між внутрішньогруповими та міжгруповими ($a+1 = h = S/2$). Характеристики – в табл. Г.40.

Табл. Г.40.

Характеристики dragonfly ($a = S/2+1$, $g = a*h+1$, $h = S/2$)

	N	R	S	D	aD	T	L	Eg	SD
4	21	42	4	3	2.166667	1.083333	4	0.538889	12
6	52	156	6	3	2.372549	0.79085	6	0.477124	18
8	105	420	8	3	2.503846	0.625962	8	0.441667	24
10	186	930	10	3	2.592793	0.518559	10	0.419219	30
12	301	1806	12	3	2.65619	0.442698	12	0.403968	36
14	456	3192	14	3	2.703297	0.386185	14	0.39304	42
16	657	5256	16	3	2.739499	0.342437	16	0.38488	48
18	910	8190	18	3	2.768097	0.307566	18	0.378585	54
20	1221	12210	20	3	2.791207	0.279121	20	0.373597	60
22	1596	17556	22	3	2.81024	0.255476	22	0.369558	66
24	2041	24492	24	3	2.826169	0.235514	24	0.366227	72
26	2562	33306	26	3	2.839683	0.218437	26	0.363437	78
28	3165	44310	28	3	2.851285	0.203663	28	0.361069	84
30	3856	57840	30	3	2.861349	0.190757	30	0.359036	90

Мережа dragonfly+ синтезована аналогічним чином, причому параметрами внутрішньокластерної структури було обрано $l=s=S/2$ для дотримання бісекційної пропускної здатності (як це пропонувалось розробниками топології). Деталі структури – на рис. Г.3. Характеристики – в табл. Г.41.

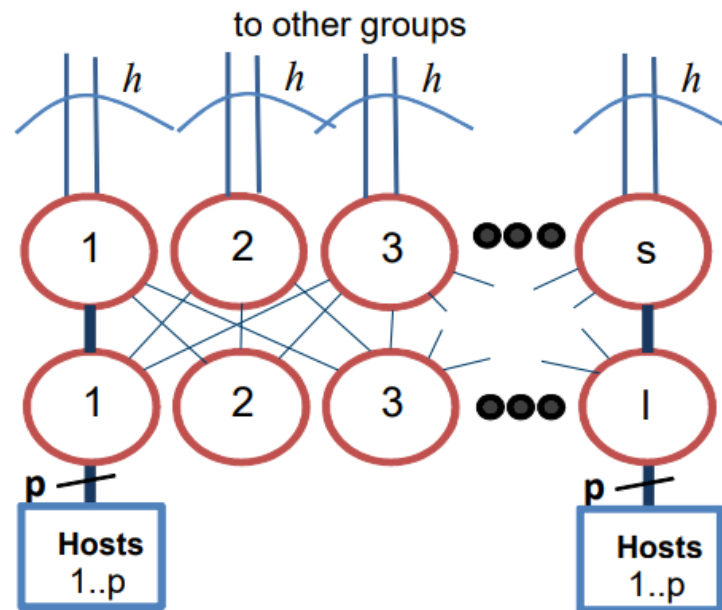


Рис. Г.3. Структура групи мережі dragonfly+ [56, 251]

Табл. Г.41

Характеристики dragonfly+ ($l = S/2$, $s = S/2$, $g = s \cdot h + 1$, $h = S/2$)

	N	R	S	D	aD	T	L	Eg	SD
4	20	30	4	4	2.5	1.666667	2	0.480263	16
6	60	135	6	5	2.875706	1.278092	3	0.400565	30
8	136	408	8	5	3.107407	1.035802	4	0.361358	40
10	260	975	10	5	3.264865	0.870631	5	0.338263	50
12	444	1998	12	5	3.374342	0.749854	6	0.323432	60
14	700	3675	14	5	3.456979	0.658472	7	0.313066	70
16	1040	6240	16	5	3.520693	0.586782	8	0.305474	80
18	1476	9963	18	5	3.571224	0.52907	9	0.29969	90
20	2020	15150	20	5	3.612234	0.481631	10	0.295145	100
22	2684	22143	22	5	3.646156	0.441958	11	0.291485	110
24	3480	31320	24	5	3.674667	0.408296	12	0.288477	120

Г.6. Структура запропонованих ієрархічних мереж

Для аналізу було обрано топологію Hyper de Bruijn (HDB) з кластерами рангу 3 на кодах (3, 2) та (4, 2), Dragon de Bruijn (DDB) з кластером на основі коду (3, 2) та дерева Dragonfly-Tree з горизонтальними зв'язками близького (Near) та комбінованого (Combo) типу. Результати – в табл. Г.42 – Г.45.

Табл. Г.42

Топологія DDB з кластером 3 порядку на коді (3, 2)

	N	R	S	D	aD	T	L	Eg	SD
2	1485	6050	9	7	4.374613	1.073769	6	0.243406	63
3	2214	10127	10	7	4.359107	0.953003	7	0.242981	70
4	2943	14933	11	7	4.351311	0.857558	8	0.242766	77
5	3672	20468	12	7	4.346611	0.779791	9	0.242636	84

Табл. Г.43

Топологія HDB з кластером 3 порядку на кодi (3, 2)

	N	R	S	D	aD	T	L	Eg	SD
2	1485	6050	9	7	4.374613	1.073769	6	0.243406	63
3	2214	10127	10	7	4.359107	0.953003	7	0.242981	70
4	2943	14933	11	7	4.351311	0.857558	8	0.242766	77
5	3672	20468	12	7	4.346611	0.779791	9	0.242636	84

Табл. Г.44

Топологія HDB з кластером 3 порядку на кодi (4, 2)

	N	R	S	D	aD	T	L	Eg	SD
2	256	1416	12	5	3.148039	0.569137	8	0.362692	60
3	512	3088	13	6	3.642857	0.603997	9	0.311379	78
4	1024	6688	14	7	4.139785	0.633843	10	0.271553	98
5	2048	14400	15	8	4.638007	0.659628	11	0.240104	120
6	4096	30848	16	9	5.136996	0.682091	12	0.214845	144

Табл. Г.45

Топологія Dragonfly-Tree з різними типами міжкластерних зв'язків [3]

	Спільні			Near Tree					Combo Tree				
	N	S	L	D	C	aD	T	SD	D	C	aD	T	SD
2	9	5	5	3	135	1.750	0.700	15	3	135	1.750	0.700	15
3	21	6	6	4	504	2.186	0.729	24	4	504	2.186	0.729	24
4	45	6	6	5	1350	2.919	0.964	30	5	1350	2.884	0.961	30
5	93	6	6	7	3906	3.941	1.314	42	7	3906	3.680	1.227	42
6	189	6	6	9	10206	5.291	1.764	54	7	7938	4.309	1.436	42
7	381	6	6	11	25146	6.869	2.290	66	9	20574	5.448	1.816	54
8	765	6	6	13	59670	8.607	2.869	78	11	50490	6.815	2.272	66
9	1533	6	6	15	137970	10.450	3.483	90	13	119574	8.447	2.816	78

Г.7. Порівняння запропонованих та популярних топологій

Оскільки різні мережі мають різні швидкості масштабування і різні експериментальні точки за N, є сенс робити порівняння в графічному виді, лінійно апроксимувачи порожні проміжки. Порівняння варто проводити з тими графами, які мають практичне застосування – жирне дерево, dragonfly, топологія Fugaku. Також, оскільки всі характеристики представлені вище в таблицях, немає сенсу робити повне порівняння в графіках, оскільки це зіпсує їх наочність. Порівнювати для кожної характеристики є сенс лише 2-3 найкращі класичні рішення і 1-2 запропоновані. Це порівняння для мереж на основі надлишкових кодів продемонстровано на рис. Г.4. Для мереж на основі ієрархічного синтезу таке порівняння представлено на рис. Г.5.

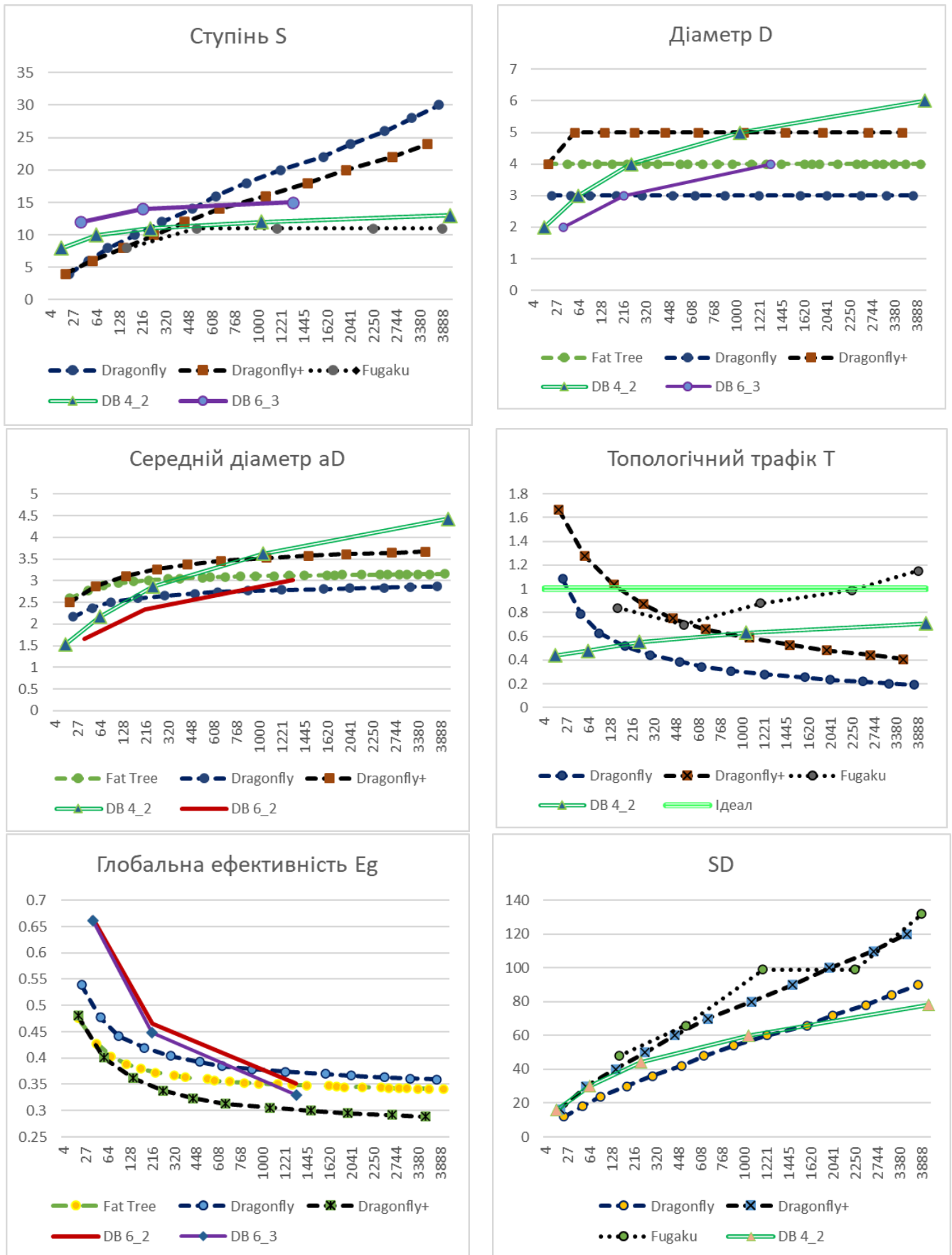
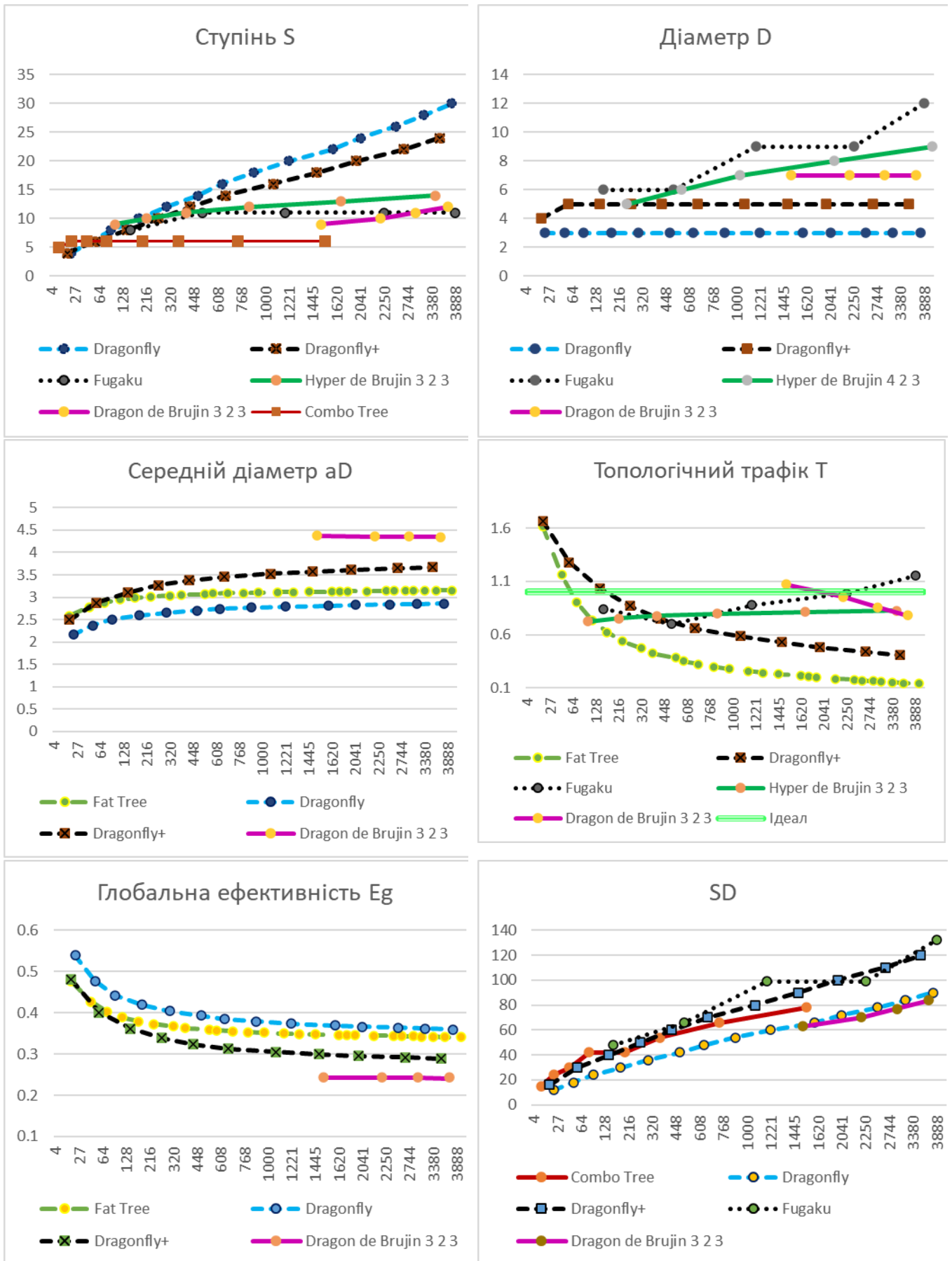


Рис. Г.4. Порівняння популярних графів та графів на основі надлишкових кодів



Г.5. Порівняння запропонованих ієрархічних та популярних графів

Додаток Г

Аналіз поведінки топологій в умовах відмови

Г.1. Результати для безпосередньо зв'язаних мереж

Для аналізу було обрано ряд графів, їх було розділено на групи за числом вершин. Було виділено наступні групи:

Група 1 ($N = 64$) - 1000 симуляцій:

- DB3 (4, 3): граф де Бруйна рангу 3, код (4, 3)
- HC n64: бінарний гіперкуб рангу 6
- T2D: 2-вимірний тор рангу 8

Група 2 ($N = 216$) – 100 симуляцій:

- DB3 (6, 2): граф де бруйна рангу 3, код (6, 2)
- DB3 (6, 3): граф де бруйна рангу 3, код (6, 3)
- T3D n216: 3D тор рангу 6

Група 3 ($N = 256$) – 100 симуляцій:

- HC n256: бінарний гіперкуб рангу 8
- DB4 (4, 2): граф де бруйна рангу 4, код (4, 2)

Оскільки масив отриманих даних є занадто великим для табличного представлення, є сенс представити результати в графічному вигляді.

Результати симуляції для першої групи топологій представлені на рис. Г.1. Отримані в ході кожного із 1000 експериментів характеристики було усереднено для відповідних значень N . Параметр `alive` тут і далі показує число моделей графів, що залишались в робочому (зв'язному, «живому») стані на кожній із ітерацій експерименту.

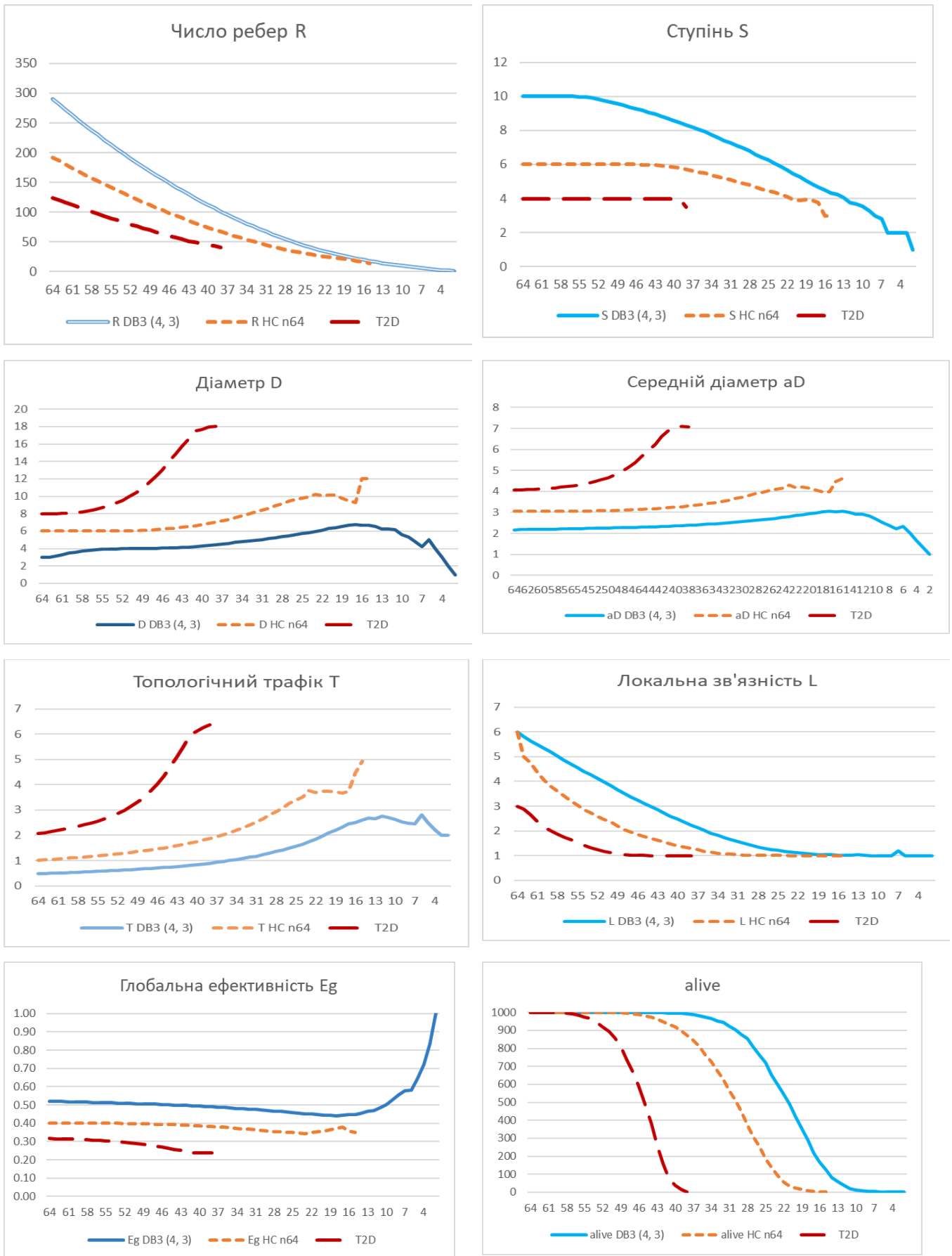
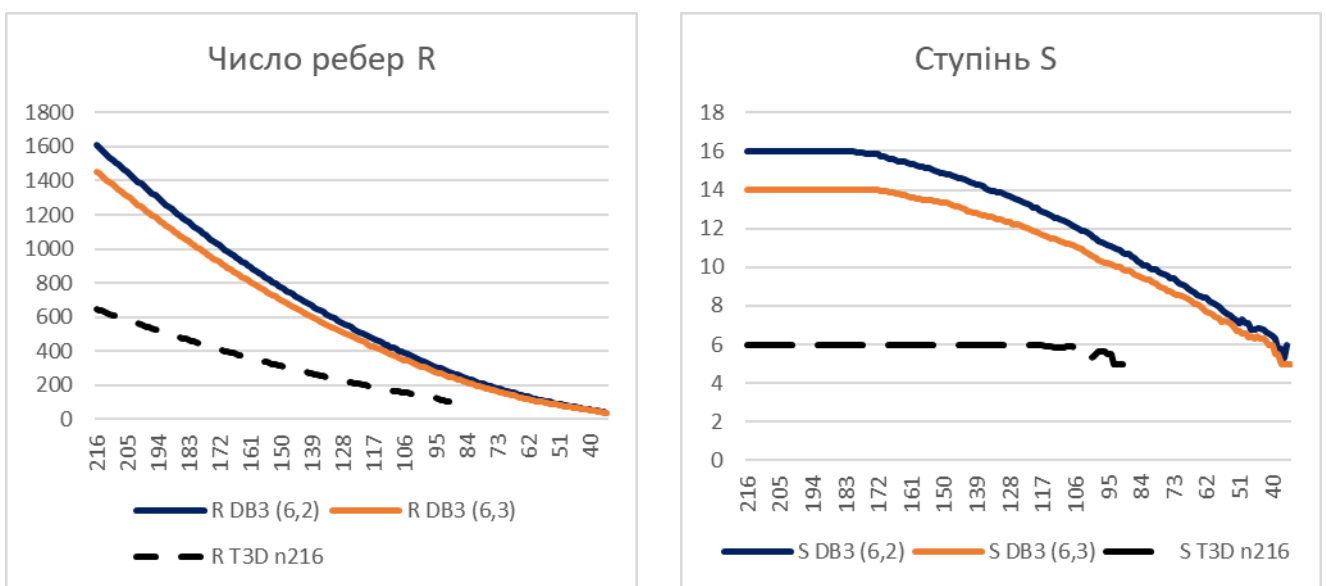


Рис. Г.1. Результати моделювання відмов для N=64

Зображені дані ілюструють наступні тенденції: число ребер дебруйнівської мережі скорочується набагато повільніше ніж у інших, те саме можна сказати і про ступінь. В той же час діаметр та середній діаметр ілюструють дивну тенденцію до спаду ближче до кінця кожного експерименту, що свідчить про групування залишків мережі в одну суцільну щільнозв'язану групу. Топологічний трафік також зберігається на низькому рівні, близькому до 1, в той час як ідеальний з т.з. трафіку гіперкуб через відмови втрачає зв'язність, що призводить нездатності мережі забезпечити достатню пропускну здатність. Аналогічно, високою залишається локальна зв'язність, що свідчить про те, що вразливі елементи потрапляли під відмову значно рідше, ніж в інших типах мереж. Врешті-решт, ефективність при цьому проілюструвала тенденцію до зростання, що також свідчить про перегрупування мережі у щільнозв'язану структуру.

Не менш важливим аспектом є і той факт, що запропонована мережа проявила значно більшу життєздатність. Загалом, значно більше число її екземплярів із початкової 1000 залишались в працездатному стані, ніж у альтернативних систем на основі класичних топологій, і більш того, деякі з них зберігали зв'язність, фактично, до перетворення графа в одну-єдину працездатну вершину. В той же час класичні мережі були повністю знищені, втративши $\frac{3}{4}$ своїх вершин.

Результати для середньої групи (N=216) показані на рис. Г.2.



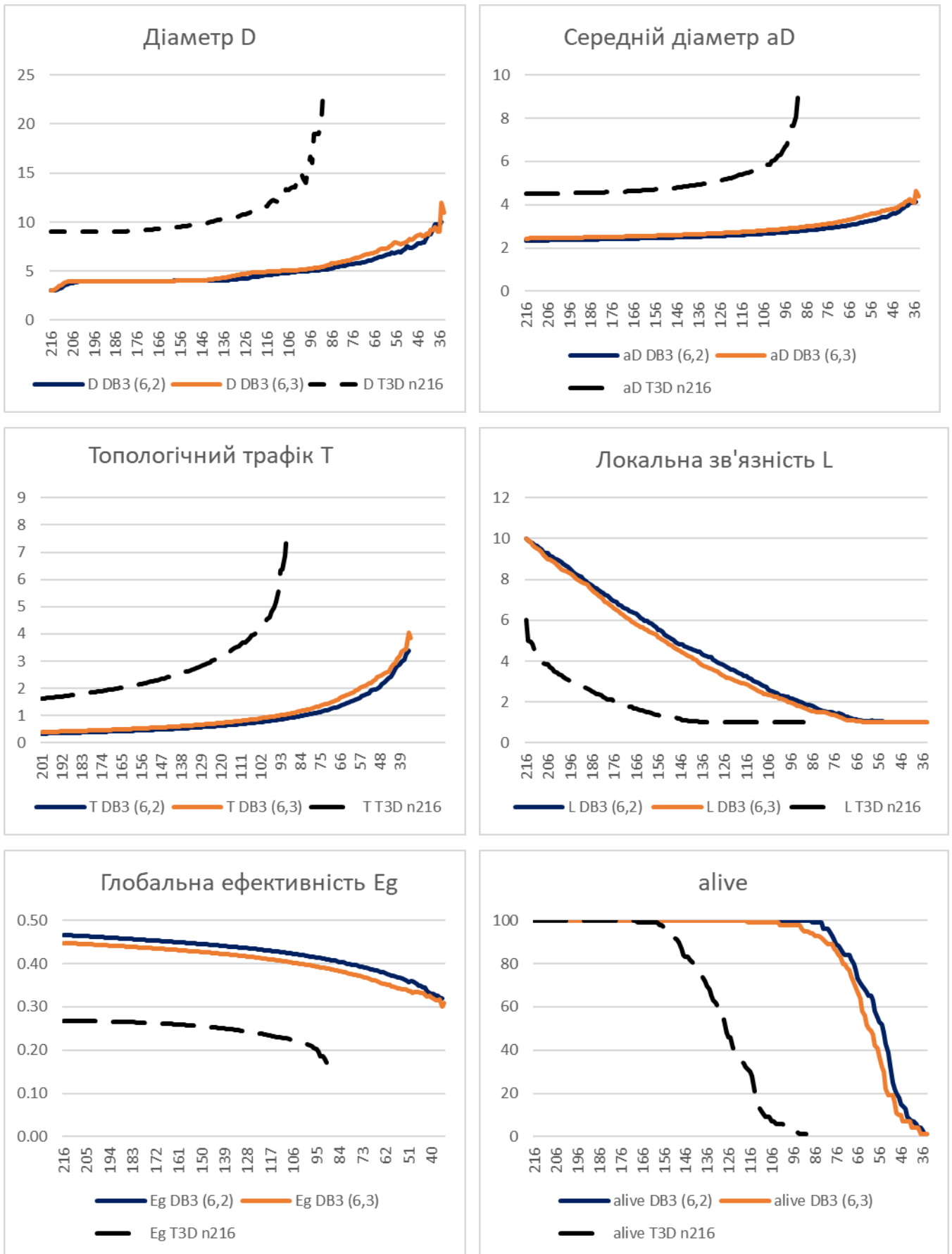
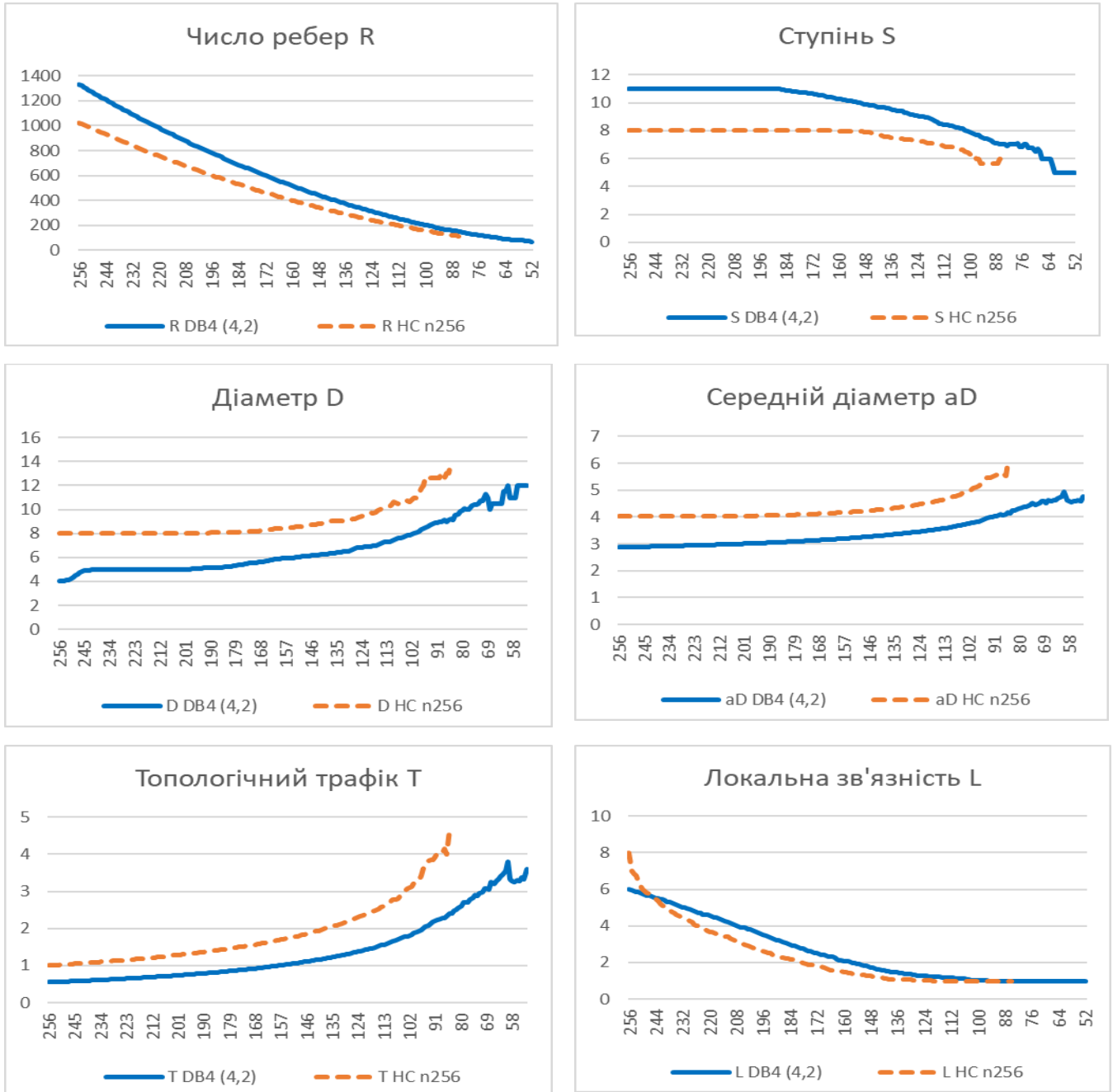


Рис. Г.2. Результати моделювання відмов для N=216

Отримані результати загалом відповідають тим, що були отримані в попередній групі, за виключенням різкого спаду діаметру і пов'язаних з цим флуктуацій. Загалом, варто відмітити, що топологічні характеристики дебруйнівських мереж погіршуються слабше, ніж у 3-вимірному тора, а момент остаточної відмови системи настає значно пізніше (останніми точками є $N=34$ і 35 проти $N=90$ у тора).

Рис. Г.3 ілюструє результати для останньої групи, $N=256$.



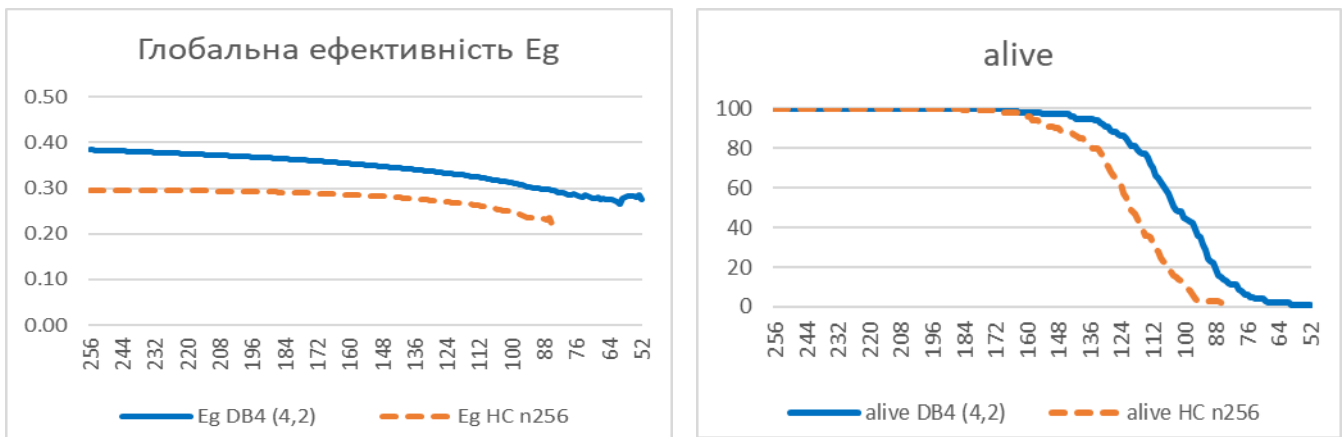


Рис. Г.3. Результати моделювання відмов для $N=256$

В даній групі можна також відмітити невеликі флуктуації, пов'язані із діаметром надлишкової мережі, що також свідчить про початок само-кластеризації в деякий момент часу. Також важливо зазначити зміни в локальній зв'язності: в той час, як при безвідмовній роботі надлишкова мережа програє гіперкубу по цьому параметру, з відмовою все нових і нових вузлів локальна зв'язність надлишкової мережі починає переважати відповідний параметр гіперкуба. Щодо ефективності та життєздатності, аналогічно, запропонована надлишкова мережа переважає по даному параметру, розпадаючись при $N=51$ (в гіперкубі розрив відбувся при $N=81$).

Г.2. Результати для комутованих мереж

Комутовані мережі було розбито на наступні групи:

- Група 1 ($N = 90$) - 1000 експериментів (рис. Г.4):
 - DDB: DDB з групою ранга 2 на основі коду (3, 2)
 - Dfly: Dragonfly з $h=1$, $c=9$
 - Dfly+: Dragonfly+ з $h=1$, $c=4$, $m=14$
- Група 2 ($N = 272$) - 100 експериментів (рис. Г.5):
 - DDB: DDB з групою ранга 2 на основі коду (4, 2)
 - Dfly: Dragonfly з $h=1$ та $c=16$ вузлів

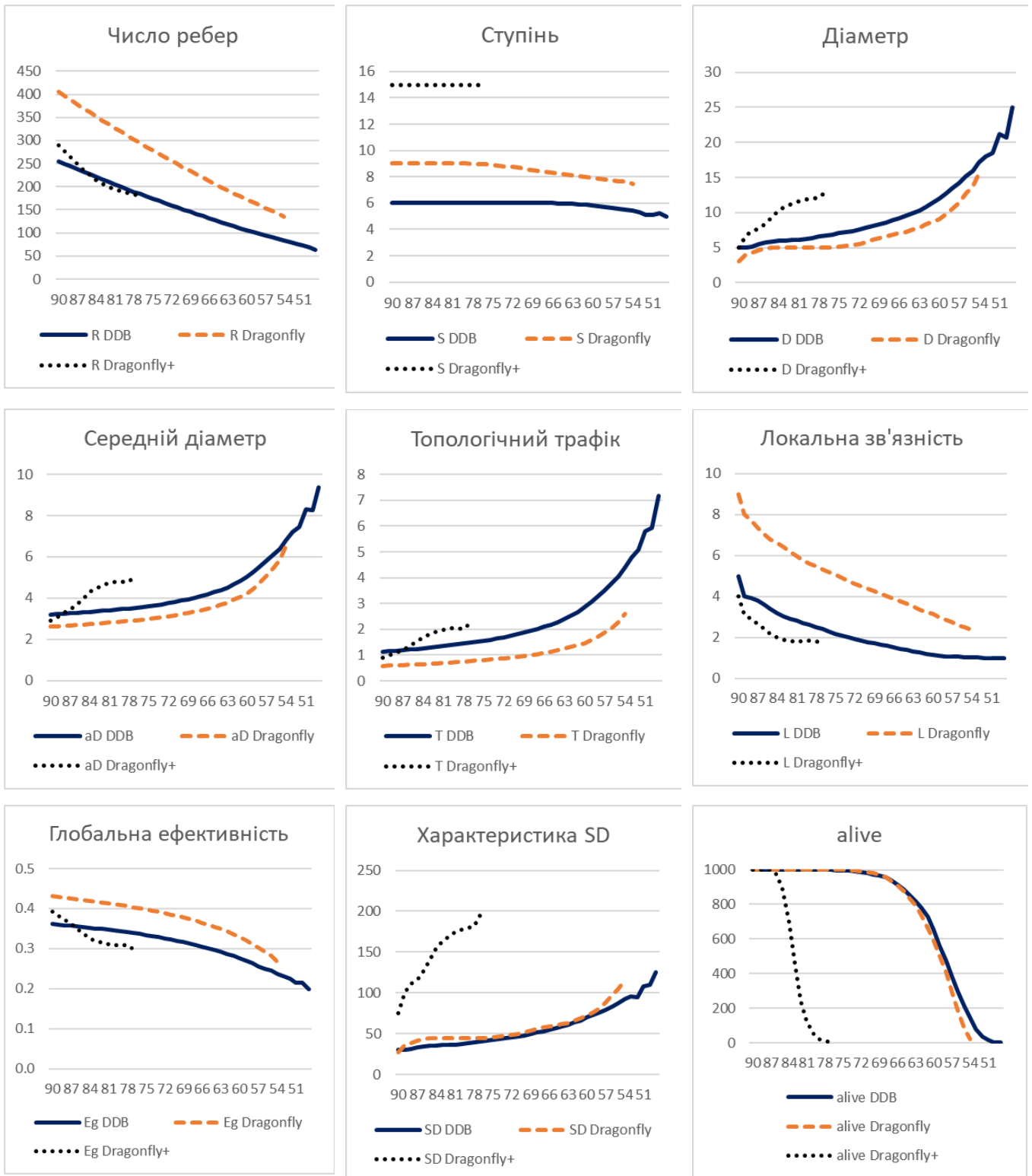


Рис. Г.4. Результати для групи 1 (N = 90) комутуваних топологій

Як видно, результати моделювання є більш стриманими. Отримані рішення мають вищу життєздатність (alive), проте їх характеристики подекуди є гіршими.

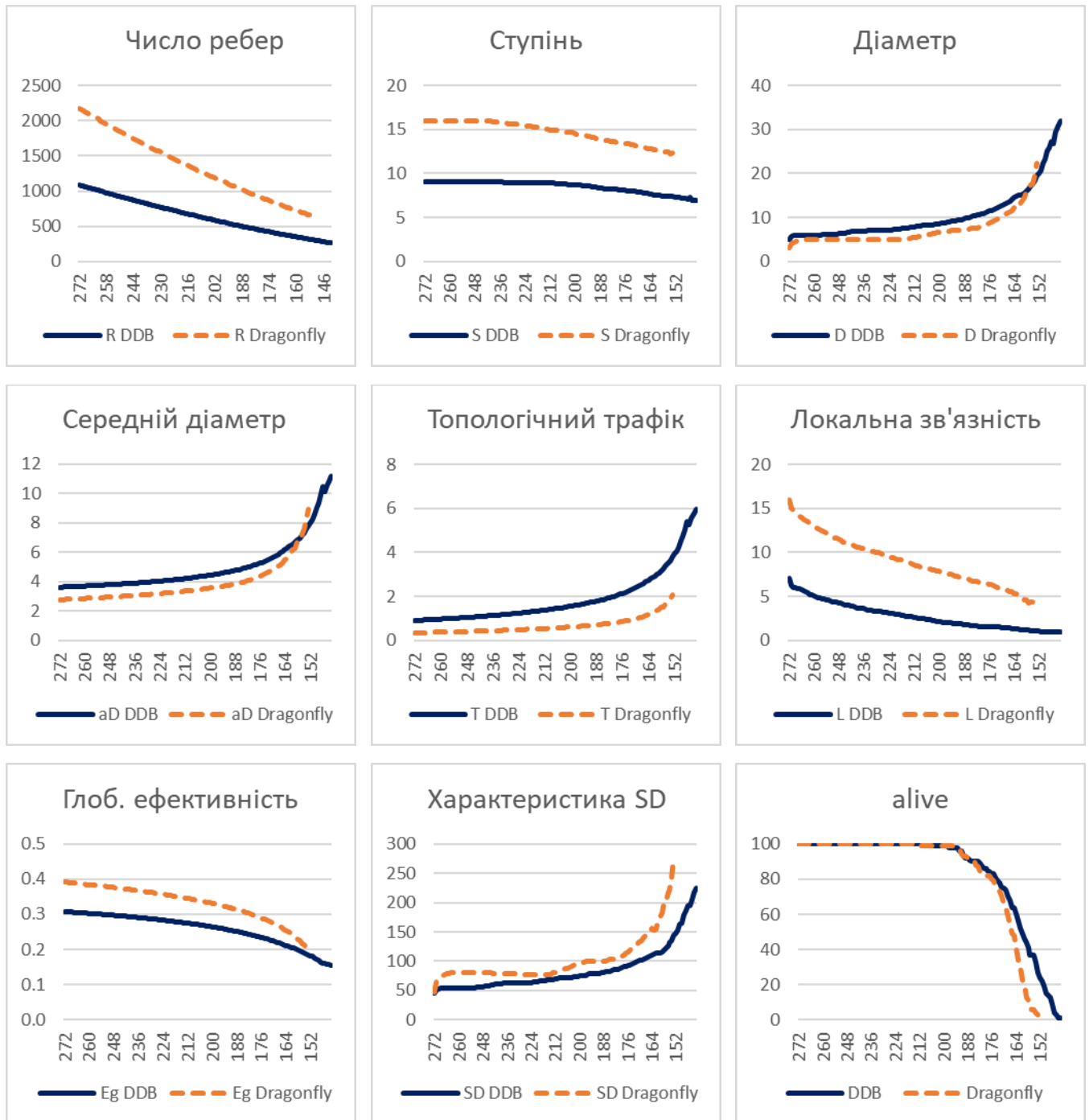


Рис. Г.5. Результати для групи 2 ($N = 272$) комутуваних топологій

Додаток Д

Розгорнуті пояснення до окремих частин основного тексту, додаткові малюнки та таблиці.

Д.1. Аналіз причин відмов у суперкомп'ютерах

Окрім наведених в Розділі 1, існують й інші способи розділити помилки за певними ознаками. Часто дослідниками вводяться власні класифікації, пов'язані з характеристиками систем, що аналізуються, та наявними даними. Наприклад, в аналізі Грея для тандемних систем [84] виділялись фатальні та нефатальні помилки; одиночні помилки та ланцюжки помилок; відмови, що були виявлені системою раннього виявлення, невиявлені помилки та ті, що були виявлені користувачем. Крім того, помилки поділялись на програмні (software), апаратні (hardware), обслуговування (maintenance), керування (operations), оточення (environment) та процесу (process). В результаті даного дослідження було встановлено, що основними причинами відмов / відключень системи є оточення (в основному, вимикання живлення), збої програмного забезпечення та проблеми керування. На апаратні несправності припало всього лише 4% від всіх збоїв.

Дещо схоже виглядає класифікація відмов, зроблена Лу для систем Національного центру суперкомп'ютерних додатків (National Center for Supercomputing Applications, NCSA) [85]. В роботі було виділено програмні збої, апаратні збої, відключення внаслідок планового обслуговування, збій мережі та проблеми з електроенергією або циркуляцією повітря. Загалом встановлено, що збій програмного забезпечення є причиною відмови в 59–83% випадків, втім в середньому на усунення програмного збою потрібно значно менше часу (0,6–1,5 годин), ніж на усунення апаратної несправності (6,3–100,7 годин).

В аналізі Олінера та Стерлі для 5 систем [86] основними типами помилок, що виділялись, були апаратні, програмні та невизначені помилки. Хоча сирі дані демонстрували, що основною причиною відмови є, як правило, апаратне забезпечення, втім подальший аналіз показав, що причиною 64% відмов були проблеми з програмним забезпеченням.

Не менш цікаві дані представлено в дослідженні системи Blue Waters, зробленому Ді Мартіно та ін. [87]. Встановлено, що апаратне забезпечення було причиною 42% всіх відмов, в той час як частка збоїв програмного забезпечення становила 20%. Втім, аналізуючи простої, викликані різними типами збоїв, дослідники встановили, що 53% часу відновлення було витрачено на програмні помилки, в той час як апаратні зайняли лише 23% часу.

Втім тенденція до превалювання програмних помилок над апаратними характерна не для всіх суперкомп'ютерів. Так, в системах Jaguar та Titan основну частку помилок складали саме несправності апаратури [33, 83].

Ще однією можливою причиною відмови є людський фактор. Наприклад, в дослідженні Оппенгеймера та Паттерсона [88] показано, що на втручання людини-оператора припало 14–30% від всіх помилок системи.

Узагальнюючи приведені дані, можна виділити 3 основних аспекти проблеми:

- апаратний аспект, який полягає в ненадійності апаратури та її схильності до деградації і потребує ефективних механізмів резервування вузлів та маскування помилок;
- програмний аспект, що базується на помилках при написанні, розгортанні та оновленні програмного забезпечення і потребує резервування на рівні обчислень та сервісів, а також різноманітних засобів відновлення / відкату стану;
- людський фактор, що ґрунтується на схильності людини помилятись і потребує максимального відсторонення користувача від керування внутрішнім станом системи – тобто, максимальної автоматизації питань відмовостійкості.

Д.2. Інші аспекти N-кратного резервування

Резервування програмного рівня

Однією із концепцій резервування програм є використання багатоядерності для усунення помилок. Його ідея полягає у використанні процесорних пар, де один процес (на одному процесорі) є основним, а інший – дублюючим. При відмові основного

процесу дублюючий бере його роль на себе [14]. Проблемою даного методу є те, що він передбачає дублювання обчислень [80], – як наслідок, це призводить до того, що при наявності N фізичних ядер система здатна підтримувати паралелізм лише для $N/2$ задач.

Ще один метод – модульне резервування, але не для апаратних компонентів, а для процесів / алгоритмів / програмного забезпечення (ПЗ). При збої в одному з процесів відмову можна виявити і замаскувати за допомогою голосування. Недоліком є те, що якщо ПЗ містить помилку, то скільки б реплік не було запущено, результат буде однаковий і при цьому некоректний.

Таким чином, коли постає питання відмовостійкості програмного рівня, важливим є не просто дублювання самого ПЗ, але розробка альтернативних програмних рішень. Наприклад, метод N -версійного програмування передбачає розробку N різних програмних рішень за однаковою специфікацією, причому різними незалежними групами людей, з використанням різних алгоритмічних рішень та різних засобів програмування [107–109]. Дані версії паралельно запускаються, причому в кінці та на певних проміжних етапах виконується перевірка коректності виконання з використанням голосування. Окрім цього, також застосовуваними є методи блокового відновлення та N -самоперевірки, які доповнюють зазначений метод приймальними тестами: спільними для всіх версій в першому підході або ж власними – в другому. Головною проблемою даної групи методів є складність перевірки програмного забезпечення. Через надвелику кількість станів в сучасних програмах дуже складно виявити конкретну причину несправності.

Також використовується метод для знешкодження тимчасових помилок – це часова надлишковість. Ідея даного підходу така: якщо трапилась помилка – необхідно повторити обчислення. Очевидною проблемою даного методу є зниження продуктивності, тому однією із модифікацій часової надлишковості є перевиконання інструкцій, що передбачає повторення обчислень лише для конкретних несправних команд. Іншим варіантом є часопросторова надлишковість, що дозволяє виявляти як тимчасові помилки, так і постійні. Загалом, основними недоліками всіх методів цього

типу є суттєве зменшення продуктивності, особливо у випадках, коли перевиконанню підлягають великі блоки коду.

Розвиток методів N-кратного резервування

Узагальнюючи огляд N-кратного модульного резервування із Розділу 1, варто зазначити, що проблемою всіх подібних методів є їх дороговизна, що передбачає надлишковість $N+\epsilon$ елементів на кожен окремий блок [92]. Таким чином, дослідниками розглядаються варіанти здешевлення даного методу з використанням специфічних методів чи за допомогою поєднання різних підходів.

Так, в роботі [89] запропоновано метод, що базується на наближених обчисленнях і дозволяє зменшити накладні витрати TMR. Продемонстровано ряд технік наближеного потрійного модульного резервування для різних рівнів та виконано їх докладний аналіз. Головною проблемою даного методу є складність проектування, а також більша вразливість до помилок.

Дослідження представлені в роботі [110] пропонують інший підхід – використання часо-просторової надлишковості в процесорах з черговою багатопотоковістю. Цей підхід носить назву буферизованого TMR. Такий підхід дозволяє зменшити апаратні витрати при збереженні відмовостійкості, проте його недоліком є зростання часових витрат на перевиконання інструкцій. Розвитком даного методу є використання динамічного TMR замість часової надлишковості [111]. Виявлено, що такий підхід дозволяє збільшити відмовостійкість у порівнянні із буферизованим TMR, але апаратні витрати при цьому також збільшуються.

В роботі [112] було досліджено метод TMR зниженої точності і проведено експеримент в умовах протонного опромінення схеми. Даний метод гарно показав себе для виявлення критичних несправностей, проте його недоліком є неточність та ігнорування невеликих помилок, що обмежує його сферу застосування.

Інформаційна надлишковість

Окремим видом надлишковості є інформаційна надлишковість. Її суть полягає в додаванні до інформації певного обсягу додаткових даних, які допомагають виявити і, – іноді, – виправити помилку. Сфера застосування інформаційної надлишковості є

досить обширною: це і виявлення помилки при обчисленнях, і зберігання даних, і обмін інформацією на різних рівнях.

Одним із найпростіших методів є кодування Хеммінга, також відоме як біти парності. Його суть полягає в застосуванні оператора «виключного АБО» до всіх бітів кодованого слова – таким чином, при непарній кількості одиниць в кодованому слові біт парності буде рівний 1, при парному – 0. Даний метод є використовуваним на апаратному рівні [113], оскільки не потребує ні великих апаратних витрат, ні довгої часової затримки. Ще однією сферою застосування є схеми пам'яті, що реалізують SEC/DED (Single Error Correction / Double Error Detection). В сучасних схемах використовуваними є коди Хеммінга (39, 7) та (72, 8) [80]. Недоліком даного кодування є той факт, що лише одиночні помилки можуть бути виправлені, що потребує досить великої кількості додаткових бітів.

Схожим є код Бергера, який для генерації кодового слова передбачає підрахунок числа одиниць в коді, його представлення в двійковому коді та побітову інверсію. Оскільки N -значне кодове слово може містити не більш ніж N одиниць, кодове слово, відповідно, має довжину, рівну $\log_2(N+1)$. Недолік даного коду – неможливість виправлення помилок.

Метод контрольних сум передбачає використання суми за модулем 2 між словами кодової інформації і застосовується для передачі даних. В залежності від того, який варіант коду використовується, це дозволяє виявляти різну кількість помилок. Даний метод широко застосовується при передачі даних, проте він також не дозволяє виправлення і, як наслідок, потребує повторної передачі даних.

Кодування M із N передбачає представлення інформації у вигляді N -бітного слова, що містить рівно M одиниць. Таким чином, цей код дозволяє виявляти всі 1-бітові помилки, а також множинні однонаправлені помилки. Його недоліком є нероздільність, оскільки кодове слово як таке відсутнє, замість цього сама інформація пере-представляється в іншому вигляді.

Циклічні коди є одним із широко використовуваних методів і називаються так, оскільки циклічний зсув над кодовим словом також дає в результаті кодове слово. Ідея даного методу полягає у множенні за модулем 2 певного слова на константу, що і

формує кодове слово. Відповідно, розшифровка відбувається через ділення, а отримання ненульового залишку свідчить про помилку. Даний метод широко застосовується в різних аспектах зберігання та передачі даних. Його єдиним суттєвим недоліком є більша складність порівняно із попередніми, і як наслідок – більші апаратні витрати / часові затримки, необхідні для роботи із інформацією.

Д.3. Візантійська помилка та методи її усунення

Важливою небезпечною властивістю помилок є їх розповсюдження, що в кінцевому випадку веде до відмови системи як такої. Це пов'язано зі специфікою поведінки несправних елементів у випадках, коли несправність не веде до повної її зупинки. Подібно до справних вузлів, вони виконують певні обчислення, приймають та посилають повідомлення, проте інформація, яка повідомляється ними іншим вузлам, є некоректною, причому кожному конкретному отримувачу передаються різні значення. Такі помилки в літературі носять назву візантійських.

Важливою специфікою візантійський помилок є складність їх усунення стандартними методами. Так, апаратна візантійська помилка не може бути усуненою класичною схемою N-модульної надлишковості через здатність візантійського сигналу долати вентиля «виключне АБО» [102]

Існує декілька методів вирішення такої проблеми. Класичний алгоритм передбачає надлишковість $N \geq 3m+1$, де N – загальне число елементів, m – число помилок, що мають бути виявлені [102]. Відповідно, проблемою тут є надвелике число елементів, що потребується для виявлення кожної окремої відмови. В той же час, алгоритм з аутентифікацією дозволяє зменшити ці накладні витрати [41], забезпечуючи тривіальність рішення при кількості процесорів $N \leq m+2$ [80].

Д.4. Огляд та аналіз проблематики квантових обчислень та відповідного апаратного забезпечення

Сфера потенційного застосування квантових обчислень є досить широкою: задачі криптографії [166], фінансова аналітика та банківська справа [167], фармацевтика та хімія [168], виробництво [169], задачі штучного інтелекту [170],

задачі оптимізації та моделювання [171], тощо. Розглянутими є методи автоматичної розробки квантових алгоритмів, наприклад, генетичне квантове програмування [172].

Втім, ключовим важливим обмеженням предметної області є не алгоритми, а фізична елементна база. Загалом, можна виділити 3 основні категорії сучасних квантових обчислювачів:

1. *Універсальні квантові комп'ютери*. До даного класу можна віднести такі системи як 54-кубітний Sycamore компанії Google [173], китайський фотонний квантовий комп'ютер Jiuzhang [174] чи 433-кубітний Osprey від IBM [175–177]. Їх визначальними особливостями є (а) універсальність, що дозволяє виконувати на таких системах будь-який квантовий алгоритм і (б) число кубітів, достатнє для досягнення квантової переваги.
2. *Малі квантові пристрої*. Прикладами таких є 2-кубітний Gemini від SpinQ Technology [178] та 12-кубітний процесор Tunnel Falls від компанії Intel [179]. Такого роду системи мають дуже мале число кубітів, зате можуть працювати в звичайних температурних умовах, на відміну від класичних систем, що потребують контрольованого середовища.
3. *Спеціалізовані квантові обчислювачі*. Гарним представником даного класу процесорів є розробки компанії D-Wave, такі як 2000Q [180] та 5000-кубітний Advantage [181]. Важливою особливістю даної категорії обчислювачів є наявність надвеликого числа кубітів (порівняно із універсальними квантовими системами) і обмеження на тип задач, що можуть бути вирішені такого роду пристроями.

Така категоризація дає змогу виділити наступні обмеження. Очевидно, малі пристрої мають недостатньо кубітів для досягнення квантової переваги. Звісно, в цій сфері існує тенденція до стрімкого розвитку, проте в кращому випадку це приведе до зрівняння параметрів майбутніх квантових процесорів з сучасними великими квантовими системами. Це є прогресом для галузі, проте аж ніяк не проривом.

Щодо сучасних великих систем, то мінімальними вимогами для досягнення квантової переваги є наявність 49 кубітів, глибина схеми 40 і двокубітна помилка не вище 50% [182]. Вперше це було досягнуто системою Sycamore в 2019 році, яка за 200

секунд виконала задачу, на яку у суперкомп'ютера Summit мало б піти 10 000 років [173]. Втім, варто зазначити, що все ще існує ряд серйозних проблем, що суттєво обмежують використання квантових обчислень в реальних задачах, а саме:

1. Обмеженість часу когерентності. Це час, протягом якого квантова система знаходиться в стані суперпозиції і може використовуватись для обчислень. В залежності від реалізації кубіта верхня межа цього параметру може як обмежуватись мілісекундами [183, 184] так і сягати декількох годин [185].
2. Обмеження точності. Квантова система є дуже вразливою до зовнішніх впливів, що в результаті виливається у різноманітні помилки, які зводять нанівець весь процес обчислень. Рішенням даної проблеми є контрольованість зовнішнього середовища та алгоритми квантової корекції помилок. Втім ці методи породжують похідне обмеження: з ростом числа послідовних квантових операцій зростає і шанс помилки.
3. Топологічні обмеження. Основою квантового обчислення є заплутаність, що зв'язує кілька незалежних кубітів в єдину квантову систему. Звісно, ця заплутаність можлива лише для сусідніх кубітів, а зв'язок між не-сусідніми потребує додаткових квантових операцій, що веде ускладнення алгоритму.

Кожне із описаних обмежень саме по собі не є критичним, втім їх поєднання створює загальну проблему: реальний квантовий алгоритм потребує заплутування великого числа кубітів, що виливається в зростання числа операцій і накопичення помилки. Протидія цій помилці потребує додаткових операцій і кубітів, внаслідок чого обчислення упирається в обмеження часу когерентності. Окрім цього, існують і менші проблеми, такі як складність ініціалізації системи, помилки читання стану, фізичні обмеження на глибину схеми, тощо. Як наслідок, множина алгоритмів, що можуть бути вирішені квантовим комп'ютером, суттєво звужується, що зводить універсальний квантовий комп'ютер до рівня спеціалізованого обчислювача, що може гарно вирішувати лише невеликі задачі або обмежені класи задач.

Іншим підходом є використання спеціалізованих елементів, на кшталт процесорів компанії D-Wave. Основою їх роботи є принцип квантової релаксації [186], що дозволяє знаходити глобальний мінімум функції в процесі квантової еволюції.

Такий пристрій не дозволяє виконання загальних квантових алгоритмів, втім в рамках доступного класу задач дозволяє отримувати вкрай високі показники продуктивності. Ще однією перевагою у порівнянні із універсальними системами є значно нижча ціна «квантових відпалювачів» і значно вище число кубітів, а також можливість поєднання квантових елементів із класичним обладнанням. Прикладом є архітектура 5000-кубітного D-Wave Advantage [187].

Аналізуючи перспективи такого роду обчислювачів в контексті їх інтеграції у високопродуктивні системи, варто зазначити недолік у вигляді все ще високої ціни (порівняно з GPU та FPGA) та обмеженість доступного класу задач. Проте варто відмітити, що з розвитком технології може відбутись здешевшення «відпалювачів». Крім цього є тенденція до розширення класу доступних задач. Прикладом є розробка алгоритму квантової факторизації на основі моделі Ізінга [188], що була продемонстрована для числа 1 245 407 [189]. Це дозволило розширити сферу використання «квантових відпалювачів» на задачі квантової криптографії [190].

Д.5. Підвищення ефективності за допомогою методів еволюційних обчислень та штучного інтелекту

Підвищення ефективності на основі методу вирішення задачі

Для кожної окремої задачі існують свої методи та алгоритми, вибір яких є правом розробників конкретного ПЗ. Втім, існують методи, що є майже універсальними і дають змогу вирішувати дуже широкий спектр задач. Це дозволяє розглядати їх як певну альтернативу класичним алгоритмам. Більш того, ці методи можуть бути застосовані і для проектування високопродуктивної системи або вирішення певних її внутрішніх проблем.

В нинішній час все більшої і більшої популярності набуває галузь штучного інтелекту (ШІ), яка включає в себе великий спектр підгалузей, таких як машинне навчання, нейромережі, комп'ютерний зір та ін. Загальною властивістю ШІ є той факт, що прикладна задача вирішується не через виконання детермінованого алгоритму, а через навчання моделі на певних даних. Це дозволяє ШІ вирішувати задачі, для яких

не існує детермінованого рішення або ж воно є занадто складним для вирішення класичними методами.

Існує багато різних сфер та задач, що сьогодні вирішуються з допомогою методології штучного інтелекту [213–216], і цей перелік лише продовжує зростати. Втім, в контексті проблеми ефективності цікавішим є те, що алгоритми ШІ мають досить високий ступінь внутрішнього паралелізму [217]. Це дозволяє обійти обмеження закону Амдала і отримати високе прискорення. Проте є і негативні аспекти. По-перше, використання методів ШІ для задач, що мають алгоритм точного вирішення, не завжди є доцільним. По-друге, навчання ШІ пов'язано із численними складнощами, тож адаптація (під)задачі, навчання моделі та її імплементація в процес вирішення становить окрему науково-практичну проблему, вирішення якої потребується.

Також варто окремо згадати про еволюційні обчислення, що використовуються для таких задач як оптимізація [218], автоматичне програмування (включаючи квантове) [172, 219], кластеризація [220], тощо. Багато в чому цей метод перетинається з попереднім і навіть використовується в задачах штучного інтелекту [221], втім його основні засади дещо відрізняються. Якщо методологія ШІ пов'язана, переважно, з нейробіологією, і використовує для оптимізації такі методи як градієнтний спуск та зворотне розповсюдження помилки, еволюційні алгоритми ґрунтуються на теорії еволюції і передбачають оптимізацію за рахунок «виживання» кращих рішень в процесі природнього відбору. На відміну від машинного навчання, генетичний алгоритм не потребує великого обсягу даних для навчання, проте має вимоги до фітнес-функції: ця функція має бути доступною до підрахунку. Втім, як і у попереднього методу, у еволюційних алгоритмів ступінь внутрішнього паралелізму є дуже високим [222].

Варто зазначити, що як і у попередньому випадку не кожен алгоритм може бути ефективно реалізований через еволюційні обчислення. Крім того, існує проблема, пов'язана зі збіжністю еволюції, що породжує проблему пошуку ефективніших реалізацій.

Таким чином, методи штучного інтелекту та генетичних алгоритмів (ГА) виглядають багатообіцяючими, проте їх застосування до кожної конкретної задачі є питанням дискусійним, а їх використання для обходу закону Амдала – рішенням, що близьке до утопії.

Підвищення ефективності розпаралелювання та планування

Застосування методів ШІ та еволюційних обчислень до прикладних задач може бути складним і в кінцевому випадку не гарантує високої ефективності для кожної окремої задачі. Втім в контексті високопродуктивних обчислень є й інший спосіб застосування цих методів. Такі проблеми паралелізму як автоматична декомпозиція на підзадачі та планування не мають універсального оптимального рішення, що робить їх потенційною сферою застосування обох вищезгаданих методів.

Такого роду підходи широко розглядаються в сучасній науковій літературі. Наприклад, робота [223] присвячена вирішенню питань прискорення та ефективності паралельних систем і пропонує метод на основі штучного інтелекту (ШІ) та технік кінцевого елементу. Робота [224] описує реалізацію автоматичного розпаралелювання з використанням машинного навчання. Розпаралелювання в процесі виконання з плануванням на основі машинного навчання розглядається в публікації [225]. Ряд публікацій, присвячених задачам реактивного, опортуністичного та динамічного планування з використанням ШІ представлено в книзі [226].

З іншого боку, розглядається використання генетичних алгоритмів в задачі оптимізації коду. Наприклад, в статті [227] пропонується метод розпаралелювання та оптимізації паралельного коду на основі ГА. Публікація [228] присвячена оптимізації паралельних циклів на основі алгоритму стрибка жаби, що поєднує в собі методи ГА та нечіткі методи. Робота [229] розглядає автоматичне розпаралелювання машинного навчання на основі генетичного алгоритму.

В роботі [230] порівнюються 3 методи, що використовуються дослідниками в задачі планування, а саме генетичні алгоритми, нейромережі та нечітка логіка. Встановлено, що хоча методи ШІ можуть бути досить ефективними, втім жоден із розглянутих методів сам по собі не задовольняє всього спектру вимог, що створює потребу в нових методах.

Д.6. Характеристики відмовостійкості та ефективності

Локальна та глобальна зв'язність. Ці два показники є важливими в контексті відмовостійкості, оскільки визначають мінімальне число вузлів та зв'язків, які мають вийти з ладу для того, щоб топологія була розділена на кілька частин.

Мультиплікативний критерій SD. В літературі його також називають проблемою (Δ, D) [237]. Дана проблема описує взаємозв'язок між трьома величинами: числом вершин, ступенем та діаметром. Стверджується, що коли задано 2 з цих показників, існує теоретичне обмеження для третього показника. В контексті високопродуктивних систем даний параметр є важливим, оскільки нарощування продуктивності передбачає збільшення числа вузлів, в той же час ступінь обмежена наявними технологіями, а діаметр – потребує оптимізації.

Стабільність діаметру DS [80]. Даний параметр визначає, яким чином діаметр змінюється при відмові вузлів. Виділяють детерміністичний та імовірнісний підхід до його визначення. В першому використовують таке поняття як витривалість (persistence) – мінімальне число вузлів, що мають вийти з ладу для того, щоб топологія змінила діаметр. В другому розглядається вектор $(p_{d+1}, p_{d+2}, \dots)$, що описує імовірність, з якою топологія змінить діаметр на $d+i$ через вплив несправностей. Також виділяють p_∞ – імовірність того, що граф буде розірвано на частини.

Д.7. Рисунки та таблиці, винесені із основного тексту

Табл. Д.1.

Ефективність суперкомп'ютерів за даними рейтингу TOP 500 (червень 2024 року)

[10]

Рейтинг		Система	Число ядер	Продуктивність, TFlop/s			Ефективність, %		
TOP 500	HPCG			Пікова (Rpeak)	LINPACK (Rmax)	HPCG (R)	LINPACK	HPCG	HPCG відносно Rmax
1	2	Frontier	8699904	1679818.75	1194000	14054	71.08	0.84	1.18
2	-	Aurora	4742808	1059325.75	585340	-	55.26	-	-
3	-	Eagle	1123200	846835.2	561200	-	66.27	-	-
4	1	Supercomputer Fugaku	7630848	537212	442010	16004.5	82.28	2.98	3.62
5	3	LUMI	2752704	531505.15	379700	4586.95	71.44	0.86	1.21
6	4	Leonardo	1824768	304465.92	238700	3113.94	78.40	1.02	1.3

7	5	Summit	2414592	200794.88	148600	2925.75	74.01	1.46	1.97
8	-	MareNostrum 5 ACC	680960	265574.4	138200	-	52.04	-	-

Табл. Д.2 демонструє шаблони для наступних типів кодувань: не-надлишкового коду типу (3, 3) і надлишкових кодів типів (3, 2), (4, 2), (4, 3), (5, 2), (5, 3). Жирним курсивом виділені ті коди, що мають властивість $\alpha 1$ -стабільності.

Табл. Д.2.

Шаблони надлишкових кодів

Значення	01T ₃	01T ₂	<i>0123₂</i>	<i>012T₂</i>	012T ₃	<i>01234₂</i>	<i>0123T₂</i>	0123T ₃	<i>012TZ₂</i>
12						<i>44</i>		33	
11						<i>43</i>		32	
10						<i>34, 42</i>		31	
9			<i>33</i>			<i>33, 41</i>		23, 30	
8			<i>32</i>		22	<i>24, 32, 40</i>		22, 3T	
7			<i>31, 23</i>		21	<i>23, 31</i>		21	
6			<i>30, 22</i>	<i>22</i>	20	<i>14, 22, 30</i>		13, 20	
5			<i>21, 13</i>	<i>21</i>	12, 2T	<i>13, 21</i>		12, 2T	
4	11		<i>20, 12</i>	<i>20, 12</i>	11	<i>04, 12, 20</i>		11	
3	10	11	<i>11, 03</i>	<i>11, 2T</i>	10	<i>03, 11</i>	<i>03</i>	03, 10	
2	1T	10	<i>10, 02</i>	<i>10, 02</i>	02, 1T	<i>02, 10</i>	<i>02</i>	02, 1T	<i>02</i>
1	01	01, 1T	<i>01</i>	<i>01, 1T</i>	01	<i>01</i>	<i>01, T3</i>	01	<i>01</i>
0	00	00	<i>00</i>	<i>00, T2</i>	00	<i>00</i>	<i>00, T2</i>	T3, 00	<i>00, T2</i>
-1	0T	0T, T1		<i>0T, T1</i>	T2, 0T		<i>0T, T1</i>	T2, 0T	<i>T1, 0T</i>
-2	T1	T0		<i>T0</i>	T1		<i>T0</i>	T1	<i>T0, Z2, 0Z</i>
-3	T0	TT		<i>TT</i>	T0		<i>TT</i>	T0	<i>TT, Z1</i>
-4	TT				TT			TT	<i>Z0, TZ</i>
-5									<i>ZT</i>
-6									<i>ZZ</i>

Табл. Д.3

Порівняння топологічних характеристик рекурентного графа та гіперкуба

Число бітів коду	2	4	8	16	32	64	128	256
Число вершин N	4	16	256	65 536	4 294 967 296	2 ⁶⁴	2 ¹²⁸	2 ²⁵⁶
Гіперкуб	r = 2 S = 2 D = 2 SD = 4	r = 4 S = 4 D = 4 SD = 16	r = 8 S = 8 D = 8 SD = 64	r = 16 S = 16 D = 16 SD = 256	r = 32 S = 32 D = 32 SD = 1024	r = 64 S = 64 D = 64 SD = 4096	r = 128 S = 128 D = 128 SD = 16384	r = 256 S = 256 D = 256 SD = 65536
Рекурентний граф	r = 2 S = 2 D = 3 SD = 6	r = 3 S = 3 D = 7 SD = 21	r = 4 S = 4 D = 15 SD = 60	r = 5 S = 5 D = 31 SD = 155	r = 6 S = 6 D = 63 SD = 378	r = 7 S = 7 D = 127 SD = 889	r = 8 S = 8 D = 255 SD = 2040	r = 9 S = 9 D = 511 SD = 4599
Рекурентний граф на основі повнозв'язного кластеру з 4 вершин	r = 1 S = 3 D = 1 SD = 3	r = 2 S = 4 D = 3 SD = 12	r = 3 S = 5 D = 7 SD = 35	r = 4 S = 6 D = 15 SD = 90	r = 5 S = 7 D = 31 SD = 217	r = 6 S = 8 D = 63 SD = 504	r = 7 S = 9 D = 127 SD = 1143	r = 8 S = 10 D = 255 SD = 2550

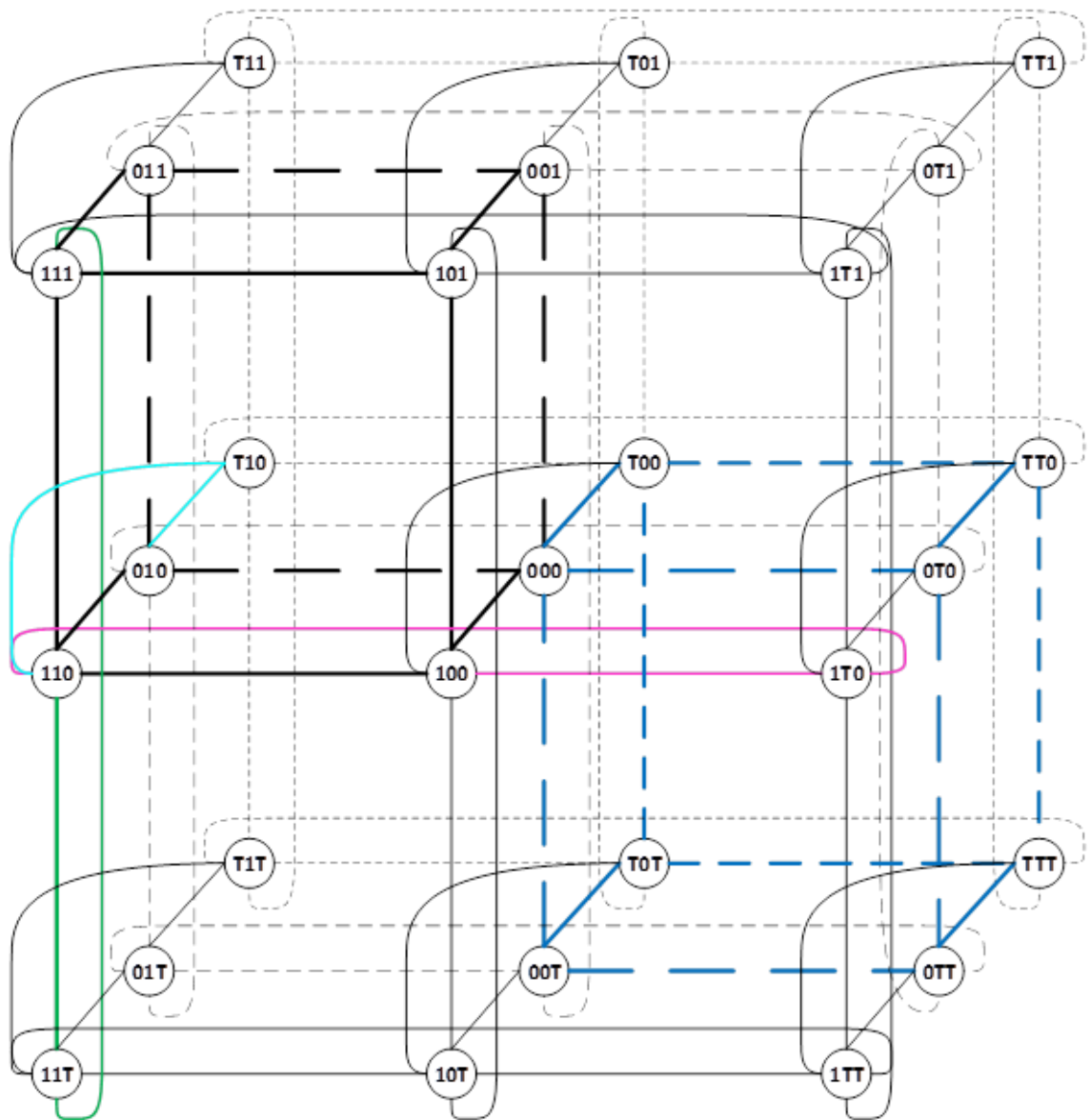


Рис. Д.1. Надлишковий гіперкуб. Збільшена версія.

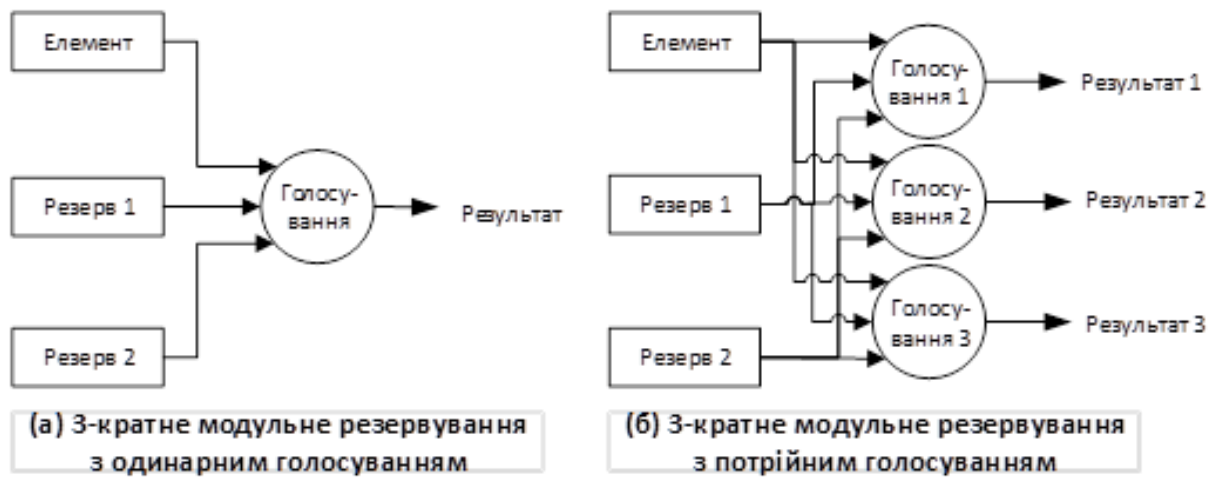


Рис. Д.2. TMR з (а) одиночною і (б) потрійною схемою голосування

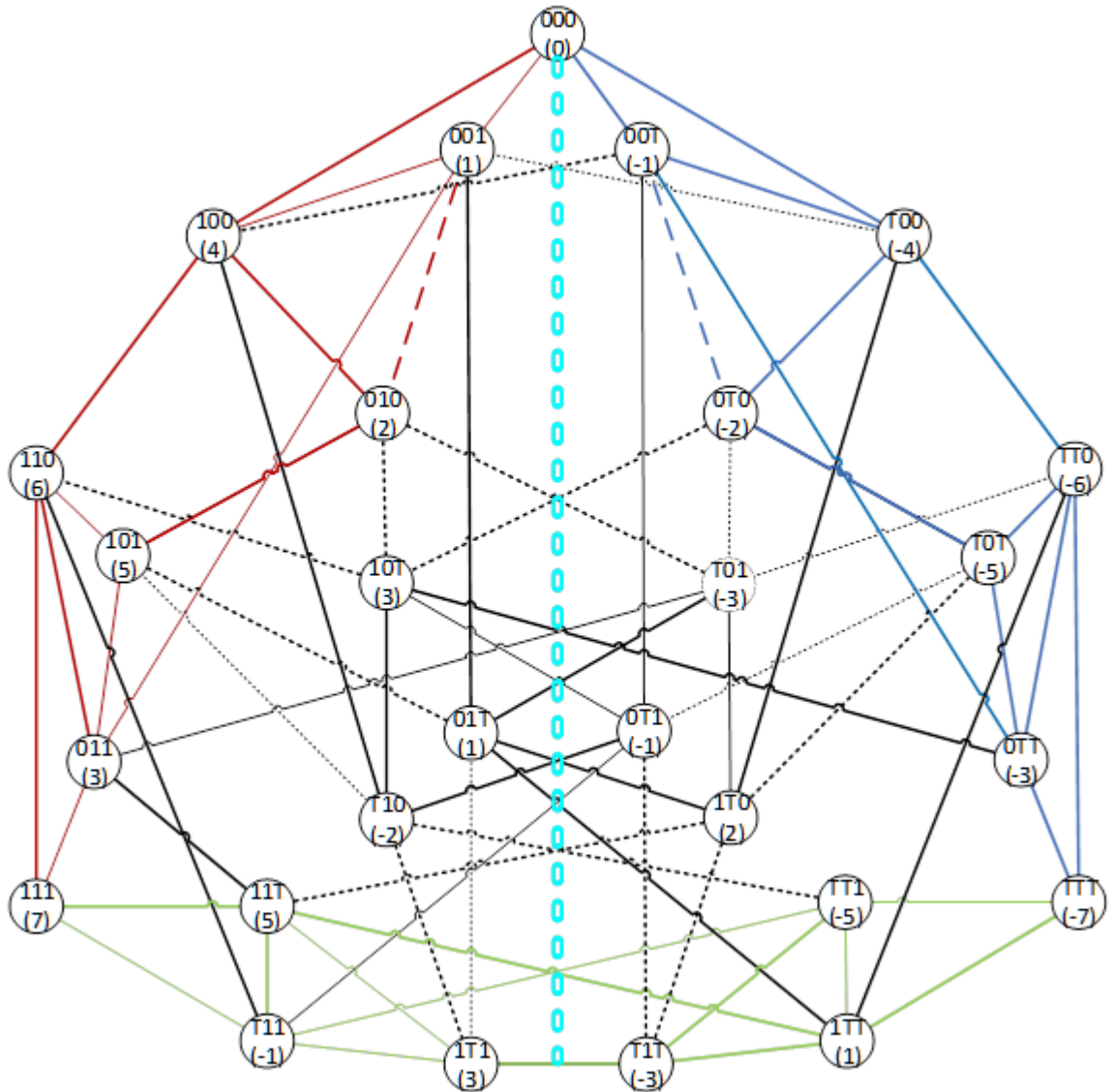


Рис. Д.3. Надлишковий де Бруйн: вигляд «сапфір». Збільшена версія.

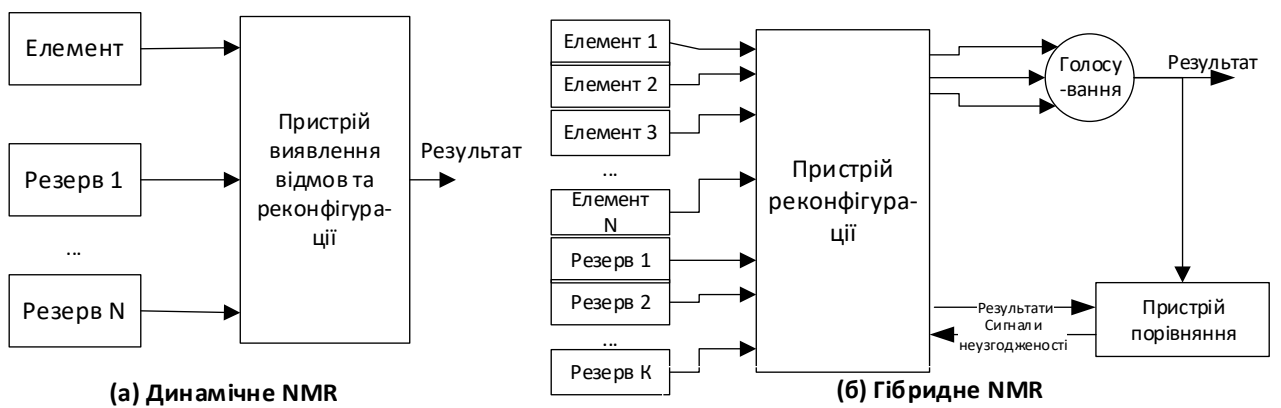


Рис. Д.4. N-кратне модульне резервування: (а) динамічний підхід та (б) гібридний підхід

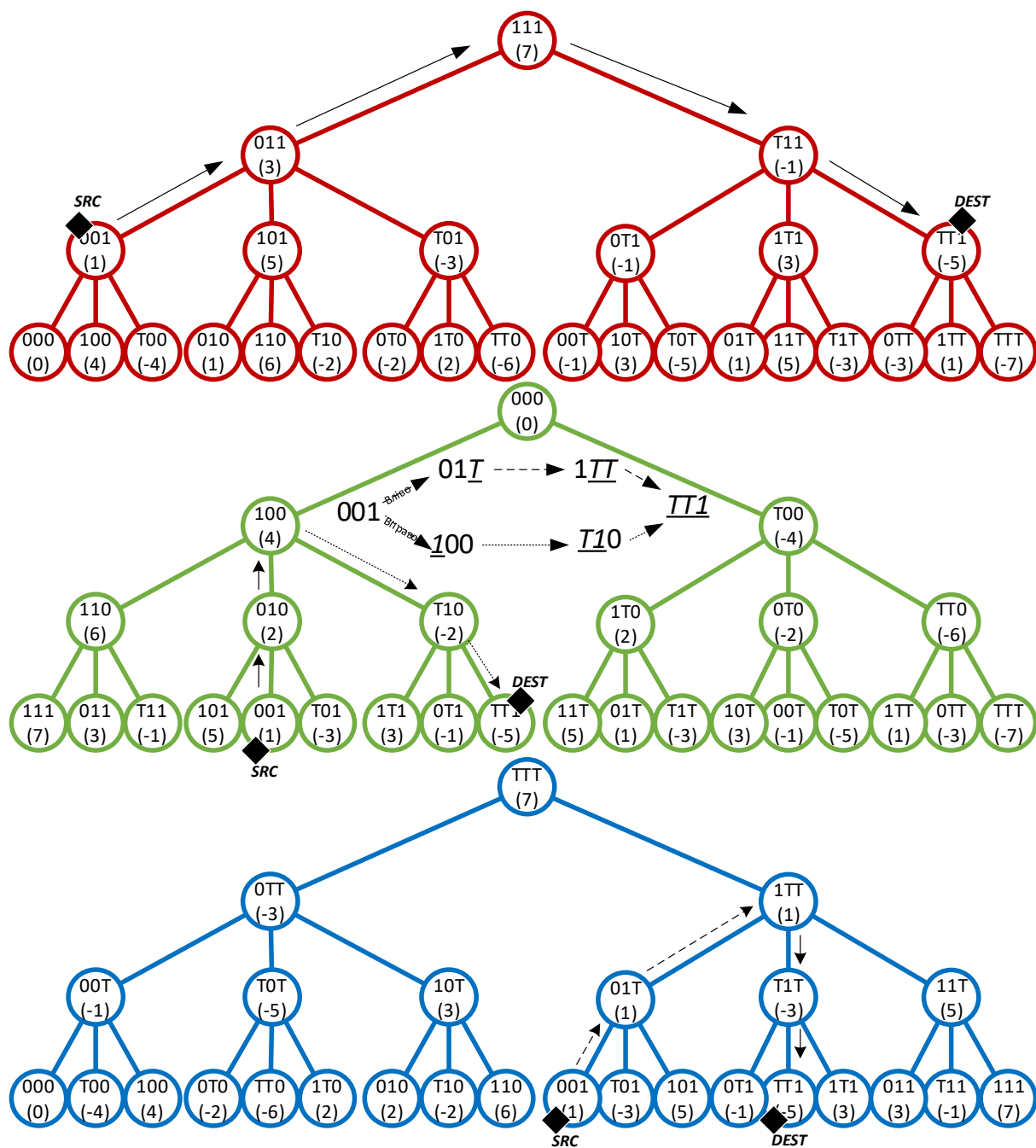


Рис. Д.5. Древа маршрутизації надлишкового де Бруїна. Збільшена версія.

Табл. Д.4

Результати моделювання відмов топологічних організацій для БЗМ

Відмов від загального N	Характ.	Група 1			Група 2			Група 3	
		DB3 (4, 2)	HC n64	T2D n64	DB3 (6, 2)	DB3 (6, 3)	T3D n216	DB4 (4, 2)	HC n256
f = 0%	Екземплярів	1000	1000	1000	100	100	100	100	100
	N	64	64	64	216	216	216	256	256
	S	10	6	4	16	14	6	11	8
	D	3	6	8	3	3	9	4	8
	\bar{D}	2.170	3.048	4.063	2.342	2.439	4.521	2.867	4.016
	Eg	0.520	0.401	0.316	0.466	0.448	0.268	0.384	0.295
f = 25 %	Екземплярів	1000	997	810	100	100	99	100	100
	N	48	48	48	162	162	162	192	192
	ΔS	-0.52	0	0	-0.610	-0.310	0	-0.010	0
	ΔD	+1.02	+0.128	+2.883	+1.000	+1.000	+0.374	+1.130	+0.040
	$\Delta \bar{D}$	+0.098	+0.058	+0.718	+0.103	+0.107	+0.137	+0.166	+0.036
	ΔE_g	-0.015	-0.006	-0.031	-0.016	-0.016	-0.008	-0.017	-0.003
f = 50 %	Екземплярів	944	628	0	100	99	9	89	68
	N	32	32	32	108	108	108	128	128
	ΔS	-2.582	-0.798	X	-3.750	-2.798	-0.111	-1.798	-0.662
	ΔD	+1.968	+2.170	X	+1.800	+2.030	+3.333	+2.652	+1.206
	$\Delta \bar{D}$	+0.331	+0.531	X	+0.300	+0.350	+1.138	+0.546	+0.396
	ΔE_g	-0.043	-0.035	X	-0.042	-0.044	-0.042	-0.048	-0.023
f = 75%	Екземплярів	169	4	0	48	30	0	2	0
	N	16	16	16	54	54	54	64	64
	ΔS	-5.485	-3.000	X	-8.500	-6.933	X	-5.000	X
	ΔD	+3.686	+6.000	X	+3.896	+4.767	X	+6.500	X
	$\Delta \bar{D}$	+0.863	+1.419	X	+0.960	+1.172	X	+1.771	X
	ΔE_g	-0.074	-0.045	X	-0.100	-0.107	X	-0.108	X
Останнє “живе” N		2 (3%)	15 (23%)	37 (58%)	35 (16%)	34 (16%)	90 (42%)	52 (20%)	85 (33%)

Табл. Д.5.

Результати моделювання відмов топологічних організацій для комутованих мереж

f(N), %	Характ.	Група 1 (п90)			Група 2 (п272)		f(N), %	Характ.	Група 1 (п90)			Група 2 (п272)	
		DDB	Dfly	Dfly+	DDB	Dfly			DDB	Dfly	Dfly+	DDB	Dfly
0 %	Екземплярів	1000			100		30 %	Екземплярів	819	794	0	96	94
	S	6	9	15	9	16		ΔS	-0.060	-0.907	X	-0.531	-1.957
	D	5	3	5	5	3		ΔD	+5.294	+4.850	X	+4.365	+4.043
	\bar{D}	3.205	2.618	2.911	3.623	2.771		$\Delta\bar{D}$	+1.316	+1.170	X	+1.046	+0.991
	Eg	0.361	0.431	0.393	0.307	0.391		ΔEg	-0.069	-0.082	X	-0.053	-0.072
10%	Екземплярів	1000	1000	129	100	100	40 %	Екземплярів	145	21	0	60	41
	ΔS	0	0	0	0	-0.020		ΔS	-0.586	-1.571	X	-1.333	-3.146
	ΔD	+1.139	+2.000	+6.620	+1.490	+2.000		ΔD	+12.097	+12.571	X	+9.467	+9.220
	$\Delta\bar{D}$	+0.204	+0.193	+1.833	+0.192	+0.201		$\Delta\bar{D}$	+3.585	+3.884	X	+2.565	+2.718
	ΔEg	-0.014	-0.017	-0.083	-0.012	-0.017		ΔEg	-0.124	-0.163	X	-0.096	-0.137
20%	Екземплярів	988	992	0	100	100	Останнє “живе” N		49	54	77	142	153
							Частка (%) мережі		54%	60%	86%	52%	56%
	ΔS	0	-0.262	X	-0.050	-0.800							
	ΔD	+2.572	+2.560	X	+2.460	+2.080							
	$\Delta\bar{D}$	+0.554	+0.506	X	+0.494	+0.500							
	ΔEg	-0.035	-0.042	X	-0.028	-0.040							

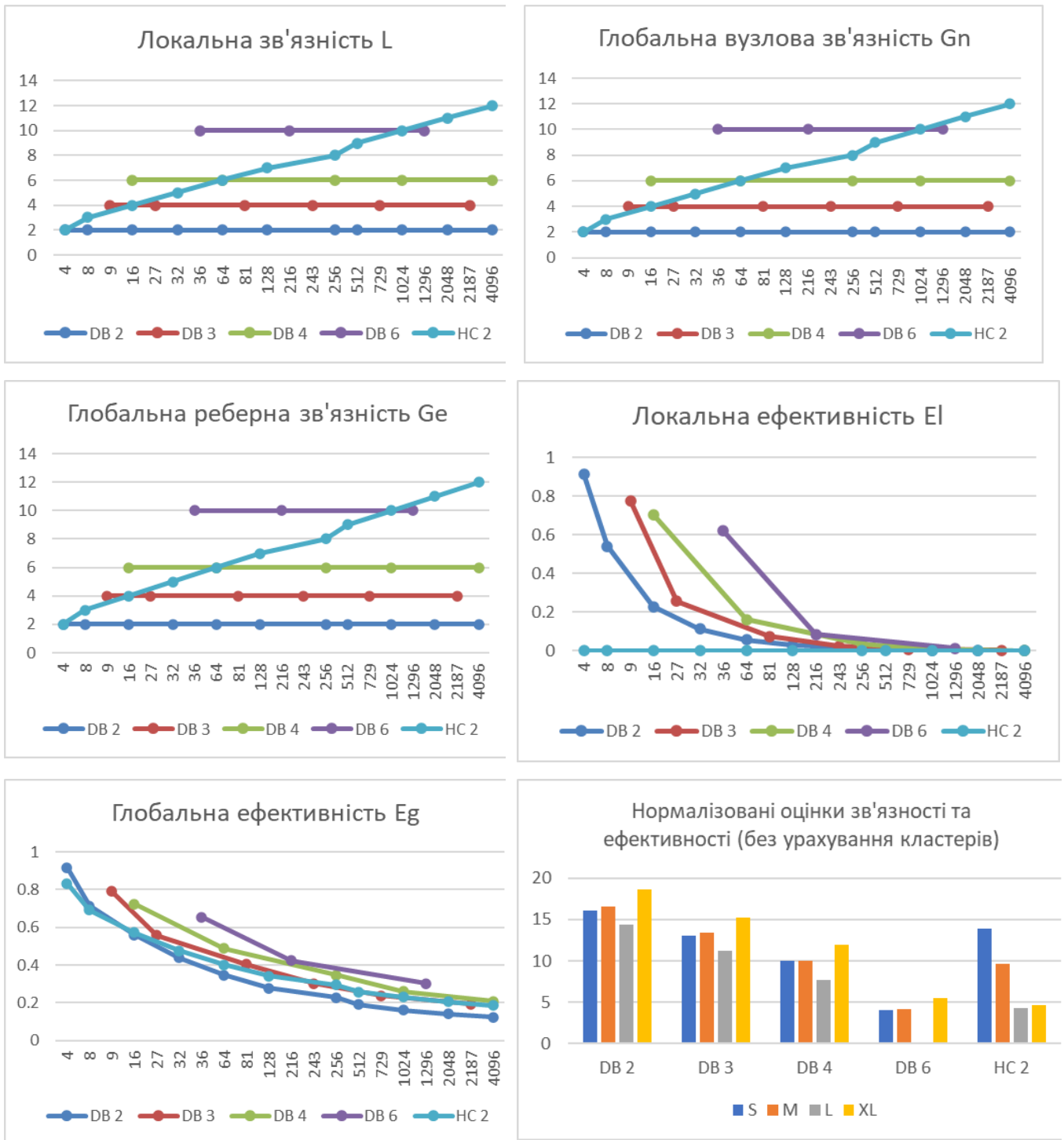


Рис. Д.6. Параметри зв'язності та топологічної ефективності. Повна версія рисунку.

Додаток Е**Акт впровадження результатів дисертаційного дослідження**