Український державний університет залізничного транспорту
Міністерство освіти і науки України

Харківський національний університет радіоелектроніки
Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

**КУРЦЕВ МАКСИМ СЕРГІЙОВИЧ**

УДК 621.391(043.3)

**ДИСЕРТАЦІЯ**

# МЕТОД ПЛАНУВАННЯ ВИКОНАННЯ ЗАВДАНЬ З УПРАВЛІННЯ ТЕЛЕКОМУНІКАЦІЙНИМИ МЕРЕЖАМИ НА ОСНОВІ ВИРІШЕННЯ ЗАДАЧ НЕЛІНІЙНОГО БУЛЕВОГО ПРОГРАМУВАННЯ

Спеціальність: 05.12.02 – телекомунікаційні системи та мережі
05 – технічні науки

Подається на здобуття наукового ступеня кандидата технічних наук

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

М.С. Курцев

Науковий керівник: доктор технічних наук, професор
Лістровий Сергій Володимирович

Ідентичність всіх примірників дисертації засвідчую
Вчений секретар спеціалізованої вченої ради /М.О. Євдокименко/

Харків – 2018

# АНОТАЦІЯ

*Курцев М.С.* Метод планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування. - Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня кандидата технічних наук (доктора філософії) за спеціальністю 05.12.02 «Телекомунікаційні системи та мережі». – Харківський національний університет радіоелектроніки МОН України, Харків, 2018.

Робота присвячена розробці методу планування виконання завдань з управління телекомунікаційними мережами. Метою дисертаційного дослідження є підвищення оперативності планування розподілу завдань з управління в кластерах телекомунікаційних систем за рахунок розробки методу планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування.

**Зміст дисертації.** У вступі обґрунтовано актуальність проблеми, що вирішується, сформульовано мету та задачі дослідження, наукову новизну та практичну цінність результатів роботи, а також наведено її загальну характеристику.

У **першому розділі** роботи на основі аналізу основних напрямків удосконалення кластерів телекомунікаційних сиситем (ТКС) і проблем, що виникають при вирішенні завдань планування виконання завдань з управління телекомунікаційними мережами в кластерах Grid-систем установлено, що важливим напрямком розвитку сучасних систем управління пакетною обробкою в кластерах ТКС є розробка відповідних методів моделей планування, які дозволяють підвищити ефективність та якість обслуговування завдань з управління телекомунікаційними мережами, що характеризуються: коефіцієнтом важливості, коефіцієнтом збереження важливості і коефіцієнтом прискорення виконання завдань, а також розширення функціональних можливостей розподілених ТКС на основі

використання обчислювальних кластерів із застосуванням Grid-технологій. Показано, що підвищення ефективності та якості обслуговування завдань можливо за рахунок розробки методу планування виконання завдань з управління телекомунікаційними мережами, який базується на зведенні задачі планування виконання завдань до вирішення задачі нелінійного булевого програмування і розробки ефективного методу вирішення даного завдання на основі рангового підходу.

У **другому розділі** запропоновано метод оперативного планування розподілу завдань з управління телекомунікаційними мережами на основі методу групової вибірки з індивідуальною сегментації.

На основі прикладів, показаних у роботі, можна зробити висновок, що використання методу групової вибірки з індивідуальною сегментацією дозволяє скоротити число етапів вирішення черги і зменшити частоту відмов в обслуговуванні завдань на вході системи при піковому навантаженні. Для забезпечення ефективності запропонованого методу необхідно використовувати в якості математичного апарату методи з малою часовою складністю для вирішення завдань лінійного та нелінійного програмування з булевими змінними.

Тому в якості методу планування пропонується використовувати рангові алгоритми вирішення задач нелінійного булевого програмування.

В роботі показана ефективність рангового підходу до вирішення довільних завдань булевого програмування на основі запропонованих процедур, що дозволяють вирішувати, як завдання лінійного, так і нелінійного програмування, з довільними нелінійностями, як в функціоналі, так і в обмеженнях, алгоритмами поліноміальної складності з невеликою похибкою.

Результати експериментального дослідження часової складності і похибки розроблених алгоритмів наочно підтверджують їх теоретичну оцінку. Збільшення числа обмежень в задачах нелінійного булевого

програмування призводить до зниження похибки їх вирішення, при цьому сама ступінь нелінійності несуттєво впливає на величину похибки.

Використання запропонованих процедур оптимізації другого типу дозволяють істотно підвищити оперативність вирішення завдань булевого програмування і точність їх вирішення, що суттєво для процесів планування виконання завдань в розподілених обчислювальних системах.

Запропоновані процедури ефективно можуть бути распаралелені, оскільки формування шляхів на ярусах у множинах $m_{sj}^{r}$ можна здійснювати одночасно. Таким чином, якщо мати $n$ - процесорних елементів для формування шляхів, то часова складність алгоритмів на основі розглянутих процедур не перевищить відповідно $O\left(pn^2\right)$ і $O\left(pn\right)$, якщо їх реалізовувати з використанням CUDA-технологій, то це дозволить застосовувати ці процедури для управління в масштабі реального часу, в розподілених системах обчислення.

У **третьому розділі** проведене моделювання роботи кластера телекомунікаційної мережі з плануванням виконання пакетів завдань на основі вирішення задачі нелінійного булевого програмування ранговим методом.

Представлена імітаційна модель дозволяє користувачеві в ручному режимі змінювати параметри роботи Grid-системи, включаючи методи планування, які і є предметом дослідження даної моделі, і отримувати результат у докладної формі з графіками. Також наведено математичний опис досліджуваних характеристик.

Як видно з результатів моделювання, використання в якості алгоритму планування запропонованого в роботі методу розв'язання задачі нелінійного булевого програмування дозволяє істотно збільшити такі показники, як коефіцієнт важливості, коефіцієнт збереження важливості, коефіцієнт прискорення і зменшити час виконання завдань і час очікування в порівнянні з відомими процедурами планування.

У **четвертому розділі** наведені рекомендації щодо інтеграції розробленого методу і моделі планування розподілу завдань в системи управління мережевими ресурсами сучасних і перспективних ТКС. В роботі приведена загальна концепція створення управління диспетчеризацією в Grid, яка передбачає наявність чотирьохрівневої ієрархічної структури управління, що поєднує в собі централізоване і децентралізоване управління. При цьому розподіл завдань розглядається на основі принципу роздільного розподілу завдань, але при цьому розподіл за допомогою диспетчера здійснюється не статично, як це робиться в відомих роздільних схемах розподілу завдань, а динамічно і безперервно на основі процедури *D*. В роботі показана архітектура дворівневої Grid-системи та модель взаємодії і структура брокерів верхнього рівня.

**Ключові слова**: Grid, розподілені обчислення, планування, кластер, методи планування виконання завдань, гетерогенні системи, мета-обчислення.

## ABSTRACT

*Kurtsev M.S.* The method of scheduling tasks for management of telecommunication networks based on Boolean solving nonlinear programming. – Qualification Scholarly Manuscript.

Ph.D. Thesis in Engineering Science in the specialty 05.12.02 «Telecommunication systems and networks». – Kharkiv National University of Radio Electronics, Kharkiv, 2018.

The work is devoted to the development of a method for planning tasks for managing telecommunication networks. The purpose of the dissertation research is to increase the efficiency of planning the distribution of management tasks in the clusters of telecommunication systems by developing a method for planning tasks implementation for managing telecommunication networks based on solving nonlinear Boolean programming problems.

**Content of the thesis.** In the **Introduction** the urgency problems solved, formulated goals and tasks of the research, scientific novelty and practical value of the work and presented its general characteristics.

**The first chapter** works On the basis of analysis of the main directions of improvement of clusters of telecommunication systems (TCS) and problems arising in solving tasks of scheduling tasks for management of telecommunication networks in Grid systems, it was established that an important direction of development of modern batch processing systems in TCS clusters is the development of appropriate model methods planning, which allows to increase the efficiency and quality of service maintenance tasks for telecommunication networks characterized by: coefficient the importance of maintaining the importance and the rate of acceleration of the tasks, as well as the expansion of the functional capabilities of distributed TCS based on the use of computing clusters using Grid technologies. It is shown that increasing the efficiency and quality of service tasks is possible due to the development of a method for planning tasks implementation for telecommunication networks management, which is based on the construction of the problem of planning the implementation of tasks for solving the problem of nonlinear boolean programming and developing an effective method for solving this problem, based on a ranked approach.

**The second chapter** of the proposed method of implementation planning tasks packets in a cluster of telecommunication systems on the basis of a sample group of individual segmentation.

Based on the examples shown in the work, we can conclude that the use of the method of sampling individual group segmentation reduces the number of steps to resolve the queue and reduce the frequency of denial of service tasks inlet system at peak load. To ensure the effectiveness of the proposed method should be used as a method of mathematical tools with small time complexity for solving linear and nonlinear programming with Boolean variables.

Therefore, as a planning method is proposed to use rank algorithm for solving nonlinear Boolean programming.

The paper shows the effectiveness rank approach to solving arbitrary tasks Boolean programming based on proposed procedures that address how the task of linear and nonlinear programming with arbitrary nonlinearities, both in functionality and in limitations, algorithms of polynomial complexity with small error.

The results of the pilot study time complexity and uncertainty developed algorithms visually confirm their theoretical assessment. Increased number of restrictions in the problems of nonlinear programming Boolean reduces the error of their decision, with the same degree of nonlinearity not significantly affect the value of the error.

Using the proposed optimization procedures of the second type can significantly increase the efficiency of solving problems Boolean programming and accuracy of their solution, which is essential for the planning tasks in distributed computing systems.

The procedure can be effectively rasparaleleni as forming paths in plural tiers $m_{sj}^{r}$ can be carried out simultaneously. So if you have $n$ - processing elements to form a temporary ways that complexity algorithms based on Procedures not exceed respectively $O\left( pn^{2} \right)$ and $O\left( pn \right)$ If they sell using CUDA technology, it will apply the procedures for managing real-time, distributed computing systems.

**The third chapter** conducted simulations of cluster telecommunication network planning task performance packages based on solving the problem of nonlinear programming Boolean rank method.

Presented simulation model allows the user to manually change the settings of Grid-systems including planning methods, which are the subject of study of this model and receive results in the form of detailed graphs. There are mathematical description of the studied characteristics.

As seen from the simulation results, use as a scheduling algorithm in the proposed method of solving the problem of non-linear Boolean programming can significantly increase such factors as the importance factor coefficient conservation

importance, acceleration rate and reduce the tasks and the waiting time compared with known procedures planning.

**The fourth chapter** the recommendations for the integration of the developed method and planning model distribution tasks in the management of network resources, current and future TCS. The paper shows the general concept of creating control scheduling in Grid, which provides for a four-level hierarchical management structure that combines centralized and decentralized management. This division of tasks is considered on the basis of separate distribution of tasks, but the distribution manager shall not static, as in the known distribution of tasks separate schemes and dynamically and continuously through the procedure D. The work shows the two-tier architecture Grid-system and model structure interaction and brokers upper level.

**Key words**: Grid, distributed computing, scheduling, cluster, methods of planning tasks, heterogeneous systems, meta-calculus.

**Список публікацій здобувача**

1. Listrovoy S.V. A uniform procedure of a system resources interaction in distributed computer media [Text] / S.V. Listrovoy, K.A. Trubchaninova, V.A. Bryksin, M.S. Kurtsev // Bulletin of NTU “KhPI”. Series: Strategic management, portfolio, program and project management. – Kharkiv : NTU “KhPI”, 2017. – No 3(1225). – P. 101-107. Bibliogr.: 10. – ISSN 2311-4738.

2. Listrovaya E.S. Modeling Local Scheduler Operation Based on Solution of Nonlinear Boolean Programming Problems [Text] / E.S. Listrovaya, V.A. Bryksin, M.S. Kurtsev // “Cybernetics and Systems Analysis”. – Springer International Publishing AG. – 2017. – Vol.53. №5. – P. 766-775.

3. Листровой С.В. Математическая и имитационная модель планирования выполнения заданий в кластере Grid-системы [Текст] / С.В. Листровой, М.С. Курцев // Інформаційно-керуючі системи на залізничному транспорті. – Харків: УкрДУЗТ, 2016. – №2(117). – С.61-72.

4. Листровой С.В. Метод и модель планирования распределения пакетов заданий в кластере Grid системы [Текст] / С.В. Листровой, Е.С. Листровая, М.С. Курцев // Международный научно-теоретический журнал «Электронное моделирование». – Институт проблем моделирования в энергетики имени Г.Е. Пухова, 2016. – Т.38, №6. – С. 85-106.

5. Листровой С.В. Подход к организации планирования распределением ресурсов в системах управления железнодорожным транспортом [Текст] / С.В. Листровой, М.С. Курцев // Науково-практичний журнал «Залізничний транспорт України». – Науково-дослідний та конструкторсько-технологічний інститут залізничного транспорту (Філія "НДКТІ" ПАТ "Укрзалізниця"), 2016. – №3-4(118-119). – С.14-22.

6. Приходько С.И. Алгоритм кодирования каскадными кодами в частотной области [Текст] / С.И. Приходько, М.С. Курцев, Хамзе Билал // Інформаційно-керуючі системи на залізничному транспорті. – Х.: УкрДАЗТ, – 2013. – №3. – С. 78 – 82.

7. Приходько С.И. Алгоритм декодирования каскадными кодами в частотной области [Текст] / С.И. Приходько, М.С. Курцев, Хамзе Билал // Системи обробки інформації. – Х.: ХУПС, – 2013. – Вип. 5(112). – С. 132 – 136.

8. Листровой С.В. Ранговый подход к решению задач линейного и нелинейного булевого программирования для планирования и управления в распределенных вычислительных системах [Текст] / С.В. Листровой, Е.С. Листровая, М.С. Курцев // Международный научно-теоретический журнал «Электронное моделирование». – Институт проблем моделирования в энергетики Г.Е. Пухова, 2017. – Т.39. №1. – С. 19-38.

9. Лістровий С.В. Ефективний метод вирішення задачі планування та виконання завдань у розподілених обчислювальних системах на основі нелінійного булевого програмування [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 79 Міжнародної науково-технічної конференції «Розвиток

наукової та інноваційної діяльності на транспорті» 25-27 квітня. - Харків: УкрДУЗТ, 2017. – №169(додаток). – С. 9-11.

10. Листровой С.В. Метод эффективного управления очередью заданий в распределенных вычислительных системах на основе решения задач нелинейного булевого программирования [Текст] / С.В. Листровой, М.С. Курцев // Материали XXII Международной научно-технической конференции «Современные средства связи» 19-20 октября 2017 г. Минск, Республика Беларусь: Белорусская государственная академия связи, 2017. – С.185.

11. Лістровий С.В. Метод планування ресурсів в кластерах Grid-систем на основі рангових алгоритмів вирішення задач нелінійного булевого програмування [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 30 Міжнародної науково-практичної конференції «Інформаційно-керуючі системи на залізничному транспорті» 26-28 вересня. - Харків: УкрДУЗТ, 2017. – №4(додаток). – С. 8.

12. Лістровий С.В. Моделювання роботи Grid системи в телекомунікаційних мережах [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 78 Міжнародної науково-технічної конференції «Розвиток наукової та інноваційної діяльності на транспорті» 26-28 квітня. - Харків: УкрДУЗТ, 2016. – № 4 (додаток). – С.37.

13. Листровой С.В. Планирование выполнения заданий в распределенных вычислительных системах на основе алгоритма, построенного на ранговом подходе к решению задач булевого программирования [Текст] / С.В. Листровой, М.С. Курцев // Міжнародна науково-практична конференція «Інформаційні технології і мехатроніка: освіта, наука та працевлаштування» 20-21 квітня 2016 року: матеріали конференції. - Харків: ХАДІ, 2016. – с.67-69.

14. Лістровий С.В. Загальна концепція створення управління диспетчеризацією в Grid [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 29 Міжнародної науково-практичної конференції «Інформаційно-керуючі

системи на залізничному транспорті» 27-29 вересня. - Харків: УкрДУЗТ, 2016. – №4(додаток). – С.2-3.

15. Лістровий С.В. Загальна концепція створення управління диспетчеризацією в Grid [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 28 Міжнародної науково-практичної конференції «Інформаційно-керуючі системи на залізничному транспорті» 24-25 вересня. - Харків: УкрДУЗТ, 2015. – №4(додаток). – С.43.

# ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ ТА ТЕРМІНІВ

| | | |
|---|---|---|
| ВВ | - | віртуальний вузол |
| ВО | - | віртуальна організація |
| ВР | - | використання ресурсів |
| ВЧ | - | виконання черги |
| ЗНП | - | завдання про найменше покриття |
| ІС | - | інформаційна система |
| ІТ | - | інформаційна технологія |
| КІ | - | коефіцієнт використання |
| ЛСУР | - | локальна система управління ресурсами |
| МКР | - | модель керованого ресурсу |
| МО | - | математичне очікування |
| ОС | - | операційна система |
| ПОС | - | паралельна обчислювальна система |
| ПЕ | - | процесорний елемент |
| РМ | - | ранговий метод |
| РОС | - | розподілена обчислювальна система |
| СПО | - | система пакетної обробки |
| СУБД | - | система управління базами даних |
| СУР | - | система управління ресурсами |
| ЧС | - | частотний метод |
| FCFS | - | First Come First Served |
| NLP | - | None Lineal Programming |
| QoS | - | Quality of service |

# ВСТУП

**Актуальність теми**. Аналіз сучасних систем управління телекомунікаційними мережами, такими як стандарт Telecommunication Management Network (TMN), показує, що останнім часом істотно зросла роль факторів, обумовлених часом, затрачуваним системою на доведення інформації про стан керованого процесу до пунктів управління, на обробку інформації, що надходить, ухвалення рішення на основі прийнятої інформації та доведення прийнятого рішення до виконавчих органів.

При цьому управління телекомунікаційними мережами характеризуються складністю умов, у яких здійснюється управління. До них можна віднести: значно збільшений обсяг виникаючих задач; твердий ліміт часу, що відводиться на прийняття (уточнення) рішення по їхньому виконанню; інтенсивність потоків даних, що збільшується, між різними ланками управління; високий динамізм зміни обстановки; обмеженість ресурсу сил і засобів, призначених для рішення задач з управління телекомунікаційними мережами. Існують об'єкти, у яких час доведення інформації про стан керованого процесу до пунктів управління складає одиниці секунд. У таких системах управління час є найважливішим параметром, від якого залежать вхідна інформація і отримання рішення. Функціонування сучасних телекомунікаційних мереж здійснюється у масштабі реального часу, забезпечуючи швидкий і гнучкий розподіл інформаційних потоків. Тому системи управління ними повинні бути високопродуктивними, відрізнятися простотою й надійністю у експлуатації. Найбільш перспективним для цього, є застосування телекомунікаційної технології Grid разом з використанням обчислювальних кластерів у структурі управління телекомунікаційними мережами.

Основоположниками концепції Grid вважаються Фостер Я. (Ian Foster), Кессельман К. (Carl Kesselman), а також Тьюке С. (Steven Tuecke) та Ник Д. М. (Jeffrey M. Nick). Значний внесок у розвиток технології Grid

належить таким закордонним вченим: Коваленко В.Н., Корягін Д.А., Куссуль Н.Н., Лобунець А.Г., Сухорослов О.В., Хайнос М. (Матт Хайнос), Чайковскі К. (Karl Czajkowski), Фрей Д (Jeffrey Frey), Грехам С. (Steve Graham) та ін. Певний вплив на розвиток Grid в Україні зробили роботи вітчизняних учених: Алексієв В.О., Лістровий С.В., Мінухін С.В., Велічкевич С.В., Петренко А.І.

Ефективність управління в телекомунікаційних мережах залежить від оперативності рішення задач управління потоками інформації в телекомунікаційних мережах, математичними моделями яких є широкий клас задач лінійного булевого програмування і теорії графів, стосовних до NP-повних задач і класу задач нелінійного булевого програмування, для яких ефективні методи рішення в режимі реального часу мало досліджені.

Тому представляється актуальним вирішення науково-практичної задачі, яка полягає в оптимізації процесів управління телекомунікаційними мережами шляхом розробки методу планування виконання завдань з управління телекомунікаційними мережами обчислювальних кластерів із застосуванням Grid-технологій, який базується на зведенні задачі планування виконання завдань до вирішення задачі нелінійного булевого програмування і розробки ефективного методу вирішення даного завдання на основі рангового підходу, що дозволяє підвищити ефективність планування.

**Зв'язок роботи з науковими програмами, планами, темами.** Дисертаційна робота безпосередньо пов'язана з реалізацією основних положень «Концепції національної інформаційної політики», «Концепції Національної програми інформатизації», «Концепції конвергенції телефонних мереж і мереж з пакетною комутацією в Україні», «Основних засад розвитку інформаційного суспільства в Україні на 2007-2015 роки» та «Створення національної Grid-інфраструктури для забезпечення наукових досліджень».

Основні результати роботи одержані при виконанні держбюджетної (Держпрограма МОН України № 23/1-2016Б) НДР по темі «Формування

теоретичних засад підвищення ефективності використання інформаційно-керуючих систем на залізничному транспорті» (ДР№ 0116U000787).

**Мета дисертаційної роботи** полягає у підвищенні оперативності планування розподілу завдань з управління в кластерах телекомунікаційних систем за рахунок розробки методу планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування.

У дисертаційній роботі розв’язані такі окремі задачі дослідження:

1. Аналіз основних напрямків удосконалення кластерів телекомунікаційних систем і проблем, що виникають при вирішенні завдань планування виконання задач з управління телекомунікаційними мережами в кластерах Grid-систем та вибір показників якості їх функціонування.

2. Подальший розвиток методу планування розподілу завдань у кластерах ТКС шляхом удосконалення методу вирішення задач нелінійного булевого програмування на основі рангового підходу.

3. Розробка методу оперативного планування розподілу завдань на основі вирішення задач НБП у кластерах ТКС з використанням Grid-технології на основі використання методу групової вибірки з індивідуальною сегментацією.

4. Розробка моделі функціонування кластера Grid-системи на основі використання для планування виконання завдань вирішення задач НБП.

5. Оцінка ефективності та перевірка адекватності розробленого методу і математичної моделі шляхом проведення їх аналітичного і експериментального дослідження із застосуванням комп'ютерного імітаційного моделювання.

6. Розробка рекомендацій щодо інтеграції розробленого методу і моделі планування розподілу завдань в системи управління сучасних і перспективних ТКМ.

**Об'єкт дослідження:** процес планування розподілу та виконання завдань з управління телекомунікаційними мережами в кластерах телекомунікаційних систем.

**Предмет дослідження:** моделі і методи планування розподілу та виконання завдань з управління телекомунікаційними мережами.

**Методи дослідження.** При розробці у дисертації методів і моделей використано: математичний апарат теорії графів, теорії дослідження операцій – для подальшого розвитку заснованого на ідеї рангового підходу методу вирішення завдань нелінійного булевого програмування (НБП) для підвищення оперативності розподілу завдань в кластерах ТКС; методи нелінійного булевого програмування – для розробки моделі функціонування кластера Grid-системи та методу оперативного планування розподілу завдань на основі вирішення задач НБП у кластерах ТКС з використанням Grid-технології на основі використання методу групової вибірки з індивідуальною сегментацією, а також методи теорії ймовірностей і математичної статистики, математичне та імітаційне комп'ютерне моделювання, програмні та алгоритмічні засоби мови програмування С++ – для оцінки ефективності та перевірка адекватності розробленого методу і математичної моделі.

**Наукова новизна одержаних результатів.** Новизна отриманих наукових результатів полягає в тому, що:

1. Вперше розроблений метод оперативного планування розподілу завдань з управління телекомунікаційними мережами, який дозволяє підвищити значення сумарного коефіцієнту важливості виконаних завдань та зменшити час їх обслуговування шляхом вирішення задач нелінійного булевого програмування у кластерах ТКС з використанням Grid-технології.

2. Вперше створена модель функціонування кластера телекомунікаційної Grid-системи, новизна якої полягає у можливості дослідження ефективності використання розробленого методу оперативного планування розподілу завдань з управління телекомунікаційними мережами при різних законах розподілу потоків завдань та інтенсивності обробки їх в

кластері та, яка базується на основі використання для планування виконання завдань вирішення задач нелінійного булевого програмування.

3. Одержав подальший розвиток метод планування розподілу завдань у кластерах ТКС, який дозволив у порівнянні з існуючими методами дискретної оптимізації суттєво зменшити часову складність планування розподілу завдань у кластерах ТКС, забезпечуючи малу похибку результатів рішення шляхом удосконалення методу вирішення задач нелінійного булевого програмування на основі рангового підходу.

**Обґрунтованість і достовірність** наукових положень та висновків підтверджується даними математичного моделювання та збіганням теоретичних результатів із експериментальними дослідженнями, а також практичним впровадженням результатів роботи (Держпрограма МОН України № 23/1-2016Б) по темі «Формування теоретичних засад підвищення ефективності використання інформаційно-керуючих систем на залізничному транспорті» (ДР№ 0116U000787), акти про використання результатів дисертаційної роботи в навчальному процесі УкрДУЗТ та в розробках ).

**Наукове значення роботи** полягає в подальшому розвитку принципів теорії управління телекомунікаційними мережами, шляхом удосконалення та розробки відповідних методів та моделей, що дозволили ефективно розв’язати задачу оперативного планування розподілу пакетів завдань в обчислювальних кластерах ТКС на основі технології Grid.

**Практичне значення отриманих результатів.** Отримані на основі розроблених у дисертації методів та моделей нові результати мають наступне практичне значення:

1. Створено програмний комплекс, на якому проведено моделювання роботи кластера з використанням розробленого методу оперативного планування розподілу завдань на основі вирішення задач нелінійного булевого програмування. Показано, що розроблений метод, на відміну від методів FCFS, дозволяє зменшити сумарний час виконання на 12-24% та

підвищити значення сумарного коефіцієнту важливості виконаних задач на більш ніж 50%.

2. Побудована модель вирішення задач нелінійного булевого програмування на основі рангового підходу, на якій показано, що в порівнянні з відомими, розроблені наближені процедури оперативного розподілу завдань мають меншу часову та обчислювальну складність, а також малу й асимптотично зменшувану зі зростанням розмірності задачі похибку рішення.

3. Сформульовані практичні рекомендації з інтеграції й ефективного використання розробленого методу оперативного планування в сучасних планувальниках, які дозволили підвищити оперативність виконання завдань кластеру та зменшити час очікування завдань в черзі на більш ніж 20%.

4. Отримані в дисертації практичні результати використані при створенні моделі планувальника завдань системи пакетної обробки кластера ТКС на основі Grid-технології, а розроблені процедури паралельного розподілу завдань використані в моделі супервізора обчислювального кластеру для підсистеми управління цифрової мережі електрозв'язку (підтверджено довідкою про участь у НДР (Держпрограма МОН України № 23/1-2016Б) по темі: «Формування теоретичних засад підвищення ефективності використання інформаційно-керуючих систем на залізничному транспорті» (ДР№ 0116U000787).

5. Результати дисертаційної роботи використані в навчальному процесі УкрДУЗТ при виконанні курсового та дипломного проектування (підтверджено актом впровадження УкрДУЗТ).

**Особистий внесок здобувача.** Усі викладені в дисертаційній роботі результати отримані автором самостійно. В статтях, що виконані у співавторстві здобувачу належить: у роботі [1] здобувачем розроблено єдину процедуру взаємодії системних ресурсів у гетерогенних телекомунікаційних мережах; у роботі [2] – модель роботи локального планувальника на основі рішення задач нелінійного булевого програмування; у роботі [3] –

математична і імітаційна модель планування виконання завдань з управління телекомунікаційними мережами в кластері Grid-системи; у роботі [4] здобувачем розроблено метод і модель планування розподілу пакетів завдань з управління телекомунікаційними мережами в кластері Grid-системи; у роботі [5] – описаний підхід до організації планування розподілом ресурсів в системах управління залізничним транспортом; у роботі [6] – проведений аналіз можливості використання завадостійких каскадних кодів при моніторингу ресурсів Grid-систем; у роботі [7] – особливості методу декодування завадостійкими каскадними кодами в частотній області в умовах використання в гетерогенних телекомунікаційних мережах; у роботі [8] – описаний ранговий підхід до вирішення завдань лінійного та нелінійного булевого програмування для планування і управління в гетерогенних телекомунікаційних мережах.

**Апробація результатів дисертації.** Результати досліджень були висвітлені та одержали схвалення на семи науково-практичних конференціях: 28 Міжнародна науково-практична конференція "Інформаційно-керуючі системи на залізничному транспорті" (м. Харків, вересень 2015 р.) [9]; 78 Міжнародна науково-технічна конференція "Розвиток наукової та інноваційної діяльності на транспорті" (м. Харків, квітень 2016 р.) [10]; Міжнародна науково-практична конференція «Інформаційні технології і мехатроніка: освіта, наука та працевлаштування» (м. Харків, квітень 2016 р.) [11]; 29 Міжнародна науково-практична конференція "Інформаційно-керуючі системи на залізничному транспорті" (м. Харків, вересень 2016 р.) [12]; 79 Міжнародна науково-технічна конференція "Розвиток наукової та інноваційної діяльності на транспорті" (м. Харків, квітень 2017 р.) [13]; 30 Міжнародна науково-практична конференція "Інформаційно-керуючі системи на залізничному транспорті" (м. Харків, вересень 2017 р.) [14]; XXII Международная научно-техническая конференция "Современные средства связи" (м. Мінськ, Республіка Білорусь, жовтень 2017 р.) [15].

**Публікації.** Основні положення та результати дисертаційної роботи викладено у 15 наукових працях, які опубліковані у фахових наукових виданнях, затверджених МОН України, та у виданнях, що індексуються міжнародними наукометричними базами, серед яких 8 статей (1 з них – у журналі, що індексується наукометричними базами Scopus) та 7 матеріалів доповідей на міжнародних науково-практичних конференціях.

**Структура дисертації.** Робота виконана на 265 сторінках, з яких 157 основного тексту, ілюстрована 57 рисунками, має 4 таблиці та складається із вступу, 4 розділів основної частини, загальних висновків, списку використаних джерел, що містить 99 найменувань і 4 додатків. Дисертація написана українською мовою.

# РОЗДІЛ 1

# АНАЛІЗ НАПРЯМІВ ВДОСКОНАЛЕННЯ КЛАСТЕРІВ РОЗПОДІЛЕНИХ ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМ ТА ПОСТАНОВКА ЗАДАЧІ НА ДОСЛІДЖЕННЯ

## 1.1 Напрямок вдосконалення телекомунікаційних систем та мереж

Розширення функціональних можливостей ТКС можливо шляхом вбудовування в їх структуру комп'ютерних кластерів. Застосування в перспективних телекомунікаційних системах комп'ютерних кластерів дає можливість значно ефективніше здійснювати управління телекомунікаційними мережами, підвищити відмовостійкість їх функціонування. Використання технології Grid [10,14] істотно збільшує спектр послуг, що надаються телекомунікаційною системою, розширює коло користувачів і різноманітність завдань, розв'язуваних в областях бізнесу, виробництва, науки і освіти.

Для побудови інтегрованої системи управління різнорідними елементами мережі природно застосувати багаторівневий ієрархічний підхід. Це, в принципі, стандартний підхід для побудови великої системи будь-якого типу і призначення. Стосовно до систем управління мережами найбільш опрацьованим і ефективним для створення багаторівневої ієрархічної системи є стандарт Telecommunication Management Network (TMN), розроблений спільними зусиллями ITU-T, ISO, ANSI і ETSI. Хоча цей стандарт і призначався спочатку для телекомунікаційних мереж, але орієнтація на використання загальних принципів робить його корисним для побудови будь-якої великої інтегрованої системи управління мережами. Стандарти TMN складаються з великої кількості рекомендацій ITU-T, основні принципи моделі TMN описані в рекомендації M.3010.

На кожному рівні ієрархії моделі TMN вирішуються завдання одних і тих же п'яти функціональних груп:

1. *Управління конфігурацією мережі і ім'ям (Configuration Management).* Ці завдання полягають в конфігурації параметрів як елементів мережі (Network Element, NE), так і мережі в цілому. Для елементів мережі, таких як маршрутизатори, мультиплексори і т. п., з допомогою цієї групи завдань визначаються мережні адреси, ідентифікатори (імена), географічне положення тощо. Для мережі в цілому управління конфігурацією зазвичай починається з побудови карти мережі, тобто відображення реальних зв'язків між елементами мережі і зміні зв'язків між елементами мережі - утворення нових фізичних або логічних каналів, зміна таблиць комутації і маршрутизації. Управління конфігурацією можуть виконуватися в автоматичному ручному або напівавтоматичному режимах.

2. *Обробка помилок (Fault Management).* Ця група завдань включає виявлення, визначення та усунення несправностей і відмов в роботі мережі. На цьому рівні виконується не тільки реєстрація повідомлень про помилки, але і їх фільтрація та аналіз. Усунення помилок може бути як автоматичним, так і напівавтоматичним.

3. *Аналіз продуктивності і надійності (Performance Management).* Завдання цієї групи пов'язані з оцінкою на основі накопиченої статистичної інформації таких параметрів, як час реакції системи, пропускна здатність реального або віртуального каналів зв'язку між абонентами мережі, інтенсивність трафіку в окремих каналах мережі, імовірність спотворення даних при їх передачі через мережу, а також коефіцієнт готовності мережі. Функції аналізу продуктивності та надійності потрібні як для оперативного управління мережею, так і для планування розвитку мережі.

4. *Управління безпекою (Security Management).* Завдання цієї групи включають контроль доступу до ресурсів мережі і збереження цілісності даних при їх зберіганні і передачі через мережу. Базовими елементами управління безпекою є процедури аутентифікації користувачів, перевірка

прав доступу до ресурсів, розподіл і підтримка ключів шифрування, управління повноваженнями.

5. *Облік роботи мережі (Accounting Management).* Завдання цієї групи займаються реєстрацією часу використання різних ресурсів мережі - пристроїв, каналів, транспортних служб. Ці завдання мають справу з такими поняттями, як час використання служби і плата за ресурси.

У моделі TMN можна виділити наступні основні рівні:

1. *Рівень елементів мережі (Network Element layer, NE)* - складається з окремих пристроїв мережі: каналів, підсилювачів, кінцевої апаратури, мультиплексорів, комутаторів і т. п. Елементи можуть містити вбудовані засоби для підтримки управління - датчики, інтерфейси управління. Сучасні технології зазвичай мають вбудовані функції управління, які дозволяють виконувати операції по контролю за станом пристроїв і за переданим пристроями трафіком. Подібні функції вбудовані в технології FDDI, ISDN, Frame Relay, SDH. Пристрої, які працюють по протоколах, які не мають вбудованих функцій контролю і управління, забезпечуються окремим блоком управління, який підтримує один з двох найбільш поширених протоколів управління - SNMP або CMIP. Ці протоколи відносяться до прикладного рівня моделі OSI.

2. *Рівень управління елементами мережі (Network Element Management Layer)* - являє собою елементарні системи управління. Елементарні системи управління автономно управляють окремими елементами мережі - контролюють канал зв'язку SDH, управляють комутатором або мультиплексором. Рівень управління елементами ізолює верхні шари від деталей і особливостей управління конкретним обладнанням. Цей рівень є відповідальним за моделювання поведінки обладнання та функціональних ресурсів нижче лежачої мережі. Атрибути цих моделей дозволяють управляти різними аспектами поведінки керованих ресурсів. Прикладами таких систем можуть служити системи управління Cisco View

від Cisco Systems, Optivity від Bay Networks, RAD View від RAD Data Communications і т.д.

3. *Рівень управління мережею (Network management layer).* Цей рівень координує роботу елементарних систем управління, дозволяючи контролювати конфігурацію складових каналів, узгоджувати роботу транспортних підмереж.

4. *Рівень управління послугами (Service Management Layer)* - займається контролем та управлінням за транспортними та інформаційними послугами, які надаються кінцевим користувачем мережі. Формування послуги полягає в фіксації в базі даних значень параметрів послуги, наприклад необхідної пропускної спроможності, коефіцієнтів готовності і т. п. У функції цього рівня входить також видача рівню управління мережею завдання на конфігурацію віртуального або фізичного каналу зв'язку для підтримки послуги. Після формування послуги даний рівень займається контролем за якістю її реалізації.

5. *Рівень бізнес-управління (Business management layer)* займається питаннями довгострокового планування мережі з урахуванням фінансових аспектів діяльності організації, що володіє мережею. Цей рівень є окремим випадком рівня автоматизованої системи управління підприємством (АСУП), в той час як всі нижчі рівні відповідають рівням автоматизованої системи управління технологічними процесами (АСУТП).

Завдання управління в межах кожного рівня вирішуються автономно системою оперативної підтримки (Operational Support System - OSS) через відповідні функції. Реалізація функцій управління на кожному рівні здійснюється підсистемами управління: елементами мережі - ПУЕМ (Element Manager System - EMS), мережею - ПУМ (Net Manager System - NMS), послугами - ПУП (Service Manager System - SMS), а також центрами: технічної експлуатації і технічного обслуговування - ЦТЕ (Operation and Maintenance Centre - OMC), мережею - ЦУМ (Net Manager Centre - NMC), послугами - ЦУП (Service Manager Center - SMC).

Як показано на рис. 1.1, архітектура системи управління телекомунікаційними мережами може містити, як зовнішні (приклади 1, 2 на рис. 1.1) по відношенню до групи керованих об'єктів, так і вбудовані (приклад 3) ПУЕМ. Така гетерогенність використовуваних ПУЕМ викликана тим, що різні виробники телекомунікаційних систем передачі використовують специфічні, що не стандартизовані, засоби управління. ПУМ збирає, організовуючи обмін інформацією через мережу передачі даних, і обробляє дані від усіх ПУЕМ, представляючи їх в інтегральному вигляді для адміністрації мережі і ПУМ.

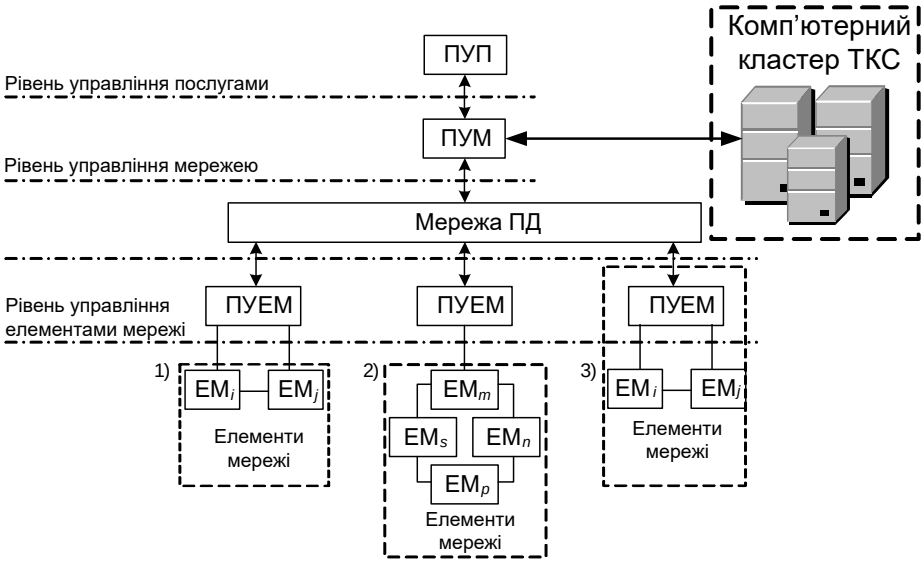



Рис. 1.1. Архітектура управління телекомунікаційною мережею з використанням кластерів

Найбільш доцільним, з позиції підвищення ефективності роботи системи управління телекомунікаційної мережі, представляється використання комп'ютерного кластера на рівні управління мережею спільно з ПУМ. Це дозволить досягти більшої повноти інтегрального уявлення про стан телекомунікаційної мережі, підвищити оперативність і точність виявлення виникаючих в процесі її роботи проблем, поліпшити

відмовостійкість системи і приймати найбільш оптимальні рішення з управління. Використання в ПУМ комп'ютерних кластерів дозволить підвищити оперативність: моніторингу, планування топології і реконфігурування мережі, перерозподілу її ресурсів, а також мінімізувати час відновлення працездатності ТКС при перевантаженні і відмовах її об'єктів. Таким чином, введення в структуру ТКС комп'ютерних кластерів підвищить ефективність передачі і обробки сигналів, розподілу інформації, а також контролю стану і управління ТКС.

Стосовно до загальної структури телекомунікаційної мережі, включення обчислювальних ресурсів кластерів в систему управління може здійснюватися як в розташованих і мережних вузлах, в яких встановлено обладнання системи передачі і здійснюється перемикання каналів (або їх груп), що належать різним системам, так і в мережевих станціях, які є кінцевими пристроями мережі і призначеними для з'єднання до цієї мережі споживачів її телекомунікаційних послуг.

Спектр послуг, що надаються телекомунікаційною мережею з кожним роком зростає, а функції її елементів різноманітніше, що значно ускладнює роботу системи управління мережею і вимагає підключення додаткових обчислювальних потужностей, таких як комп'ютерні кластери.

Функції ПУМ реалізуються центром управління мережею (ЦУМ), який формує уявлення про трафік та експлуатацію телекомунікаційної мережі. Головна функція ЦУМС - оптимізація використання ресурсів телекомунікаційних мереж на основі інформації, одержуваної від ЦТЕ в реальному часі.

В Україні ведеться активна робота по інтеграції інформаційних технологій та телекомунікаційних систем і мереж, головним вектором якої має стати впровадження Grid-технологій, що отримали зараз стрімкий розвиток у всьому світі. Grid – це мережева технологія, яка надає засоби для координації застосування гетерогенних розподілених ресурсів для рішення задач віртуальними організаціями з багатьма учасниками [9]. Глобалізація

бізнесу, поява наукових транснаціональних об’єднань та зростаючі потреби організацій та окремих осіб в надійних, потужних та зручних засобах телекомунікації та обробки інформаційних потоків потребують впровадження технології Grid у телекомунікаційні системи та створення телекомунікаційних Grid-систем (TGrid).

## 1.2 Розподілені телекомунікаційні системи на основі використання Grid-технологій

Сучасні телекомунікаційні системи взаємодіючі з телекомунікаційними мережами, які представляють сукупність вузлів каналів зв'язку, що їх об'єднують, являють собою розподілену середу обробки інформації і обчислень. В процесі збільшення навантаження, в зв'язку з ускладненням різних додатків, що виконуються в розподілених середовищах з одного боку виникають потреби в підвищенні продуктивності систем організації більш ефективного використання ресурсів і організації в розподіленому середовищі обчислень і з іншого боку надання користувачам розподілених обчислювальних систем широкого спектру різних сервісів. Це додатково висуває вимоги до організації розподіленої обробки інформації і обчислень на базі використання кластерів телекомунікаційних систем. Таким чином, сучасні телекомунікаційні системи і мережі являють розподілену обчислювальну середу призначену як для обробки інформації, так і організації обчислень в розподіленої середовищі. Термін «розподілені обчислення» (distributed computing або metacomputing) був введений в 1987 році Ларрі Смарром (Larry Smarr) і Чарльзом Кетлетт (Charles Catlett) [15]. В основі даного підходу є інтеграція обчислювальних машин за допомогою мережі передачі даних і спеціалізованого системного програмного

забезпечення в розподілену обчислювальну систему. Призначення такої системи - організація однакового доступу користувачів до її ресурсів.

Ресурси - сукупність програмних і апаратних засобів для виконання процесів. Приклади ресурсів: процесор, середовище передачі даних, прикладне програмне забезпечення, система зберігання даних тощо. Під процесом розуміється потік інструкцій для процесора обчислювальної машини з єдиним адресним простором, значеннями регістрів процесора, стеком, відкритими файлами, глобальними змінними тощо.

Так як процесори на вузлах розподіленої обчислювальної системи часто керують окремими ресурсами (або пов'язаними тільки з цими процесорами, або розділяються декількома процесорами), то доцільно розглядати узагальнене поняття віртуального вузла (ВВ). Таким чином, ВВ - один обчислювальний елемент (ядро процесора), що розглядається в зв'язці з зіставленними йому ресурсами. Якщо кілька процесорів використовують одні і ті ж ресурси, що характеризуються обсягом (оперативна пам'ять, постійна пам'ять тощо), то будемо вважати, що обсяг цих ресурсів рівномірно ділиться між усіма віртуальними вузлами, відповідними даним процесорам. На рисунку 1.2 схематично зображено співвідношення ВВ і ресурсів в одному обчислювальному вузлі.

Словосполученням «Інші ресурси» на рисунку 1.2 позначені додаткові можливості обчислювальних вузлів. Наприклад, наявність графічних або сигнальних процесорів, спеціалізованих електронних плат прискорення обчислень.

У розподілену обчислювальну систему (РОС) можуть включатися обчислювальні системи різної архітектури. Опис особливостей сучасних обчислювальних архітектур є в [2,15,16,20,32]. Особливості архітектури вітчизняних сімейств, сучасних високопродуктивних систем викладені в [15,20].

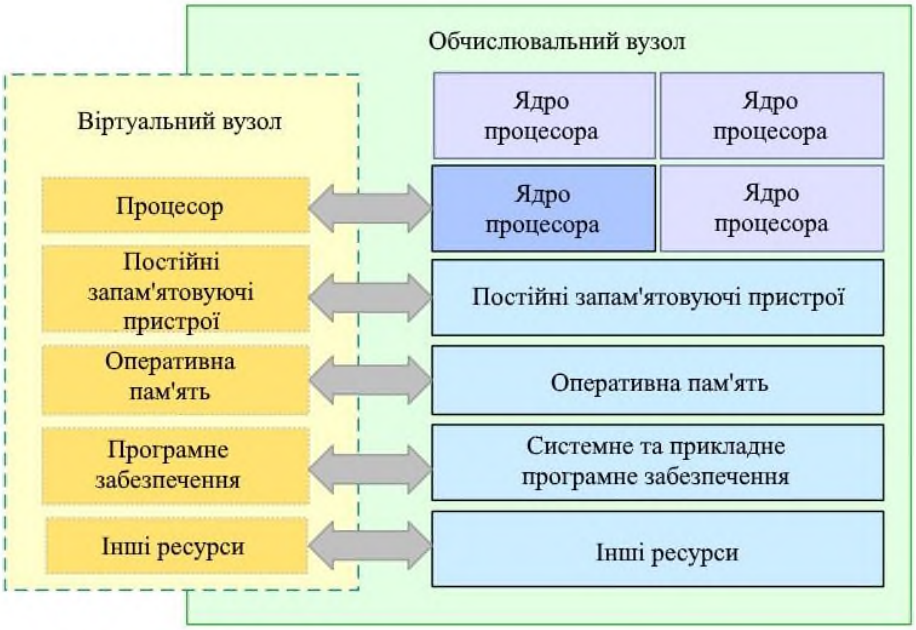



Рис. 1.2. Схема співвідношення віртуального та фізичного вузлів розподіленої обчислювальної системи

Найбільш поширеними реалізаціями РОС є системи, побудовані за технологією Grid [9,85,86]. Визначення терміну Grid на даний момент не є сталим. Будемо дотримуватися визначення Яна Фостера (Ian Foster) і Карла Кессельмана (Carl Kesselman) [87]: "Grid - узгоджене, відкрите й стандартизоване середовище, яке забезпечує гнучкий, безпечний і скоординований розподіл ресурсів в рамках віртуальної організації". Під віртуальною організацією розуміється об'єднання користувачів і реальних організацій, яким є деяка підмножина ресурсів в Grid. Ціллю введення такого об'єднання є необхідність поділу повноважень в Grid між користувачами.

У більшості випадків в якості вузлів Grid виступають обчислювальні кластери [71], розташовані віддалено і відрізняються за своїми характеристиками. У загальному випадку, обчислювальний кластер – паралельна обчислювальна система, що масштабується та включає набір високопродуктивних комп'ютерів (процесорних вузлів кластера), об'єднаних комунікаційними мережами і керованих єдиним диспетчером

завдань [3,34,62]. При інтеграції процесорних вузлів в єдиний кластер міжвузлові комунікації здійснюються за допомогою передачі повідомлень між процесами завдань.

Крім обчислювальних кластерів в Grid можуть бути включені і інші типи вузлів. Наприклад, робочі станції візуалізації або мережеві сховища даних. Термін Grid грунтується па метафорі електричної мережі (electrical power grid), яка надає користувачам електроенергію за запитом без необхідності володіння знаннями - де і як вона генерується.

Доступ до ресурсів в Grid зазвичай здійснюється за допомогою служб [36], присутніх на кожному з вузлів. Служби повинні бути стандартизовані в рамках Grid, що дозволяє одноманітно отримувати доступ до всіх ресурсів, незалежно від його виду і локальних політик на сайті. Стандарт архітектури побудови Grid сервісів OGSA (Open Grid Services Architecture) уніфікує створення, найменування і взаємодію служб.

Фактичним стандартом побудови Grid на базі архітектури OGSA став інструментарій Globus Toolkit. Даний програмний комплекс включає в себе служби управління завданнями (для моніторингу та координації віддаленого виконання завдань), служби збору даних (засновані на використанні протоколу LDAP), служби забезпечення безпеки GSI (Grid Security Infrastructure), компоненти якого засновані на технології електронно-цифрових сертифікатів стандарту X.509) і служби управління даними (дозволяють користувачам отримувати доступ, передавати і керувати розподіленими даними).

Компоненти середовища виконання інструментарію Globus Toolkit включають зовнішні програмні інтерфейси (API і SDK мов програмування Java, Python і C++), середовище виконання як стандартних, так і призначених для користувача служб. Для роботи з сертифікатами застосовується пакет програм GSI - OpenSSH - модифікована версія OpenSSH, що входить до складу інструментарію. Розроблена організацією National Center for Supercomputing Applications (NCSA), вона може використовуватися для

доступу до віддалених систем і для передачі файлів без введення пароля - всі операції аутентифікуються сертифікатами підсистеми GSI. Для цих цілей використовуються модифікації класичних утиліт sftp, scp і ssh: gsisftp, gsiscp і gsisssh відповідно.

Запущені в POC додатки можна розділити на три класи: розподілені, паралельні і непаралельні.

Розподілені Grid додатки (multi-site Grid applications) - клас додатків, які можуть бути запущені на безлічі вузлів Grid. Іншими словами, процеси одного розподіленого додатка можуть бути запущені на декількох обчислювальних кластерах одночасно. Теоретичний розгляд подібного класу додатків проведено в роботі [67].

Розподілені додатки практично не знаходять застосування в Grid через відсутність підтримки з боку Grid-інструментаріїв. Основна складність тут полягає в тому, що обчислювальні кластери контролюються своїми локальними системами управління ресурсами. Таким чином, контроль всіх процесів розподіленого між декількома кластерами завдання ускладнюється.

Іншою причиною малої поширеності даного класу додатків є те, що різниця між швидкісними характеристиками мережі передачі даних між вузлами одного кластера і між декількома кластерами може сильно відрізнятися. Облік даної характеристики мереж передачі даних ускладнює алгоритм розподіленого додатка.

Також перешкодою до розвитку сфери розподілених додатків в деяких випадках може стати різнорідність архітектури процесорів в складі сайтів Grid. При запуску програми корисна компіляція вихідного коду під задану архітектуру процесора вже після складання розкладу і запуску завдання на вузлі Grid. Така компіляція в ряді випадків дозволяє значно прискорити додаток за рахунок оптимізації бінарного коду під дану архітектуру процесора. Очевидно, що в цьому випадку розподілений додаток повинен компілюватися і запускатися в декількох розрізнених копіях, що ускладнює конструювання подібного роду додатків.

У переважній більшості випадків є можливість провести декомпозицію розподіленого алгоритму на кілька паралельних або непаралельних. Після завершення обчислень безлічі таких завдань, що відповідають одній розподіленої задачі, отримані дані збираються і обробляються окремо.

Паралельні додатки відносяться до іншого класу. На відміну від розподілених, додатки даного класу розраховані для виконання на однорідному обчислювальному кластері або багатопроцесорній системі із загальною пам'яттю.

Паралельної будемо називати програму з числом процесів N > 1, одночасно виконуваних на N однорідних процесорах. Процеси паралельної програми потенційно можуть обмінюватися даними між собою як по мережі передачі даних, так і використовуючи механізми взаємодії процесів в рамках однієї операційної системи (IPC, Inter-Process Communication). Всі процеси паралельної програми повинні бути сплановані на одночасний запуск. Такий вид планування називається спільним (gang-scheduling).

У загальному випадку паралельне завдання - це вказівка користувача запустити одну паралельну програму з певними вимогами до ресурсів, передавши їй всі необхідні для розрахунку файли даних. У плануванні паралельних завдань для Grid важливим є те, що процеси однієї програми не виходять за рамки однієї групи ВВ - вузла Grid. Прикладом паралельного завдання може служити виконання MPI-додатку на вузлах обчислювального кластера.

Непаралельною будемо називати програму, що включає лише один обчислювальний процес. Непаралельним завданням буде вказівка користувача виконати одну або кілька непаралельних програм, між якими відсутні взаємодії. Часто таке завдання включає запуск безлічі копій однієї програми з різними вхідними параметрами.

У непаралельних додатках часто використовується керуюча програма, що збирає результати розрахунків і видає нову порцію вхідних даних. Такий вид архітектури носить назву Постачальник - Споживач [12]. Ізольовані

обчислювальні процеси часто полегшують розробку і підтримку додатків. Прикладами алгоритмів з ізольованими процесами можуть служити Map-Reduce [82] і сімейство алгоритмів k-середніх [63], які знаходять застосування в методах аналізу даних, математичній статистиці та інших областях знань.

## 1.3 Особливості функціонування кластерів в Grid-системах

Запропоновані для вирішення великомасштабних обчислювальних задач в науці, техніці і бізнесі глобальні обчислювальні мережі Grid відкривають перспективу одночасного використання тисяч обчислювальних ресурсів, розташованих в різних адміністративних і географічних областях, які належать різним організаціям [3,14,15]. Одним з видів ресурсів Grid є комп'ютерні кластери - група об'єднаних високошвидкісними каналами зв'язку комп'ютерів (обчислювальних вузлів кластера) представляє з точки зору користувача єдиний апаратний ресурс. У класичній схемі при роботі з додатками всі вузли поділяють зовнішню пам'ять на масиви жорстких дисків, використовуючи внутрішні дискові накопичувачі для спеціальних функцій (наприклад, системних).

При цьому, розподіляючи ресурси Grid-системи, виникає необхідність як визначення мінімального числа кластерів, на яких можна виконати задану підмножину задач, так і оптимального розподілу цих завдань всередині самих кластерів між його обчислювальними вузлами. Такий підхід вимагає використання дворівневої структури Grid, яка зображена на рисунку 1.3. На першому рівні кілька незалежних брокерів розподіляють обчислювальні завдання на кластери, а на другому рівні кожен кластер розподіляє завдання, присвоєні йому локальним планувальником.

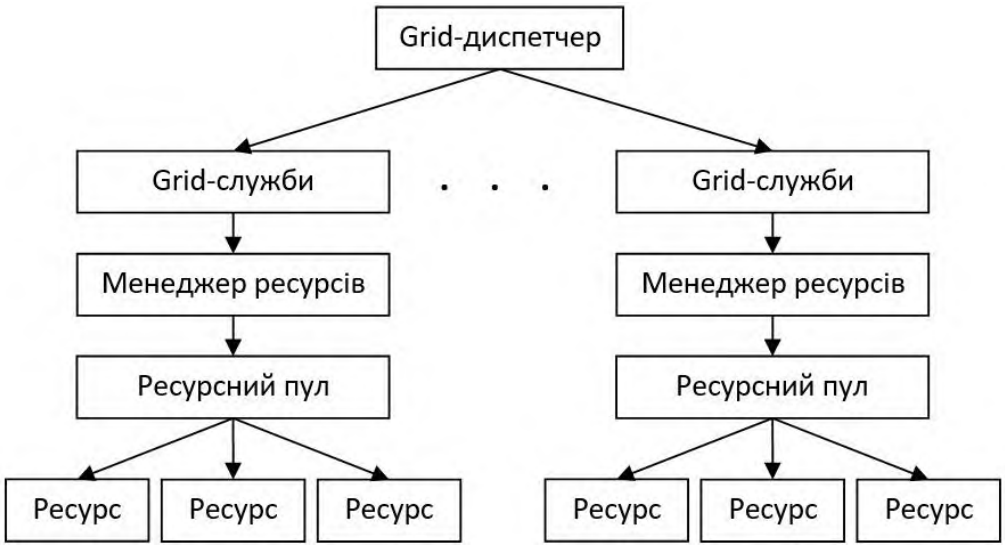



Рис. 1.3. Схема Grid дворівневої структури

Для диспетчеризації завдань в кластері Grid можна використовувати різні підходи [16-19].

Централізована диспетчеризація. При централізованої диспетчеризації всі завдання розподіляються на ресурси однієї виділеної служби. Завдання, які не можуть бути запущені безпосередньо в момент надходження, зберігаються в пулі завдань диспетчера. З одного боку, централізована служба диспетчеризації, володіючи повною інформацією, може здійснювати ефективне планування, але з іншого боку, в разі збою диспетчера, вся система стане непрацездатною.

Децентралізована схема диспетчеризації. У децентралізованій схемі розподілені диспетчери взаємодіють один з одним і відправляють завдання безпосередньо на обчислювальні ресурси. При цьому не існує центрального диспетчера, який здійснює розподіл завдань, а інформація про поточний стан ресурсів не групується в єдиному місці. Тому, притаманна централізованій схемі проблема масштабування відсутня в схемі децентралізованій, а відмова в роботі одного з компонентів мало впливає на роботу всієї системи. Наявність безлічі рівноправних диспетчерів допускає використання різних механізмів їх взаємодії. Виділяють два основних механізми: безпосередню

взаємодію і із допомогою загального пулу [16,18,20]. У першому випадку все диспетчера, маючи список диспетчерів, з якими вони можуть обмінюватися завданнями, посилають і приймають завдання безпосередньо один від одного. Якщо диспетчер не може запустити завдання, починається пошук альтернативного диспетчера, якому пересилаються завдання і всі вихідні дані. У разі взаємодії з використанням загального пулу, такі завдання пересилаються в центральний пул завдань, а не іншому диспетчерові. Із загального пулу, диспетчер вибирає завдання, для яких є вільні обчислювальні ресурси. Однак існує небезпека зависання деяких завдань в загальному пулі. При використанні загального пулу зазвичай всі нові завдання відразу надходять в нього, а звідти вибираються диспетчерами для запуску. Відсутність повної інформації про всі ресурси знижує ефективність децентралізованої схеми при складанні оптимального плану розподілу завдань. Також ускладнюється запуск паралельних завдань, частини яких виконуються на різних ресурсах, оскільки це вимагає злагодженої роботи різних диспетчерів для синхронного запуску частин завдання.

Ієрархічна диспетчеризація. Використовуючи ієрархічну структуру, коли один централізований диспетчер оперує диспетчерами нижчого рівня, можна об'єднати властивості централізованої і децентралізованої систем. При цьому виникає можливість застосування різних політик на центральному диспетчері і на підлеглих йому диспетчерах. У цьому випадку центральний диспетчер виступає в ролі метадіспетчера, розподіляє завдання, що надходять до нього, по диспетчерам нижчого рівня відповідно з деяким набором правил.

Випереджаючий розподіл. Метод попереджувального розподілу грунтується на механізмі прогнозування звільнення-зайнятості ресурсів. Оскільки доставка файлів завдання в вузол комп'ютерного кластера Grid займає багато часу здійснити його запуск безпосередньо в момент звільнення обчислювального ресурсу неможливо. Поки здійснюється доставка вхідних файлів для запланованого завдання, обчислювальний ресурс може зайняти

інше завдання. Тому пересилання повинно здійснюватися з певним випередженням, достатнім для доставки необхідних вхідних файлів.

Використання пріоритетів завдань. Щоб споживач мав можливість впливати на черговість запуску завдання, всі завдання наділяються пріоритетами, в залежності від яких проводиться розподіл завдань по ресурсах. Для демонополізації обчислювальних ресурсів кожному користувачеві виділяється бюджет, в рамках якого він може призначати плату за виконання завдань. В цьому випадку, плата виступає в ролі пріоритетів. Апарат призначення пріоритетів завдань використовується як в глобальній, так і в локальній черзі завдань. Однак метрика пріоритетів, що в них використовується, може відрізнятися. Для перерахунку пріоритетів використовується механізм, що враховує атрибути глобального завдання і параметри конкретного ресурсу. Він заснований на тому, що адміністратор ресурсів надає функцію перерахунку глобального пріоритету в локальний. Це дозволяє з одного боку досягти уніфікації метрик пріоритетів всієї інфраструктури, з іншого боку дає власнику ресурсів можливість управляти інтенсивністю потоку глобальних завдань на свої обчислювальні ресурси. Таким чином, механізм перерахунку пріоритетів надає адміністраторами засоби реалізації стратегій надання Grid ресурсів своїх комп'ютерних кластерів.

Ухвалення угод в системі диспетчеризації. Диспетчер повинен мати можливість отримати інформацію про можливості запуску глобального завдання на ресурсі, не звертаючись до локальної моделі управління ресурсами, оскільки це вимагає значних витрат часу і мережевих ресурсів. Висновок про можливість запуску завдання робиться після перерахунку глобальних пріоритетів в локальні на основі порівняння отриманого результату з пріоритетами завдань, що знаходяться в локальній черзі. Диспетчер зможе бачити, як буде виглядати локальна черга, якщо глобальне завдання послати на обчислювальний кластер. Безліччю ресурсів розподіленої обчислювальної системи завідує сервер - метадіспетчер. На

нього надходить потік завдань від усіх користувачів. Метадіспетчер буферизує завдання в чергу (пул) і далі, керуючись прийнятою стратегією, знаходить ресурси і перенаправляє завдання системі пакетної обробки (СПО) кластера. Тобто метадіспетчер розподіляє завдання по СПО, а вони в свою чергу по обчислювальним ресурсам (вузлам) комп'ютерного кластера, рисунок 1.4. При цьому загальна ефективність управління в значній мірі визначається способом взаємодії СПО і метадіспетчера. СПО спочатку проектувалися як ізольовані системи і володіють інтерфейсами користувача (для запуску та управління завданнями) та адміністратора (для конфігурації, управління чергами та машинами). Цих засобів недостатньо для оптимального планування в масштабі Grid, потрібні інтерфейси глобальної взаємодії і нові механізми управління в самих СПО.

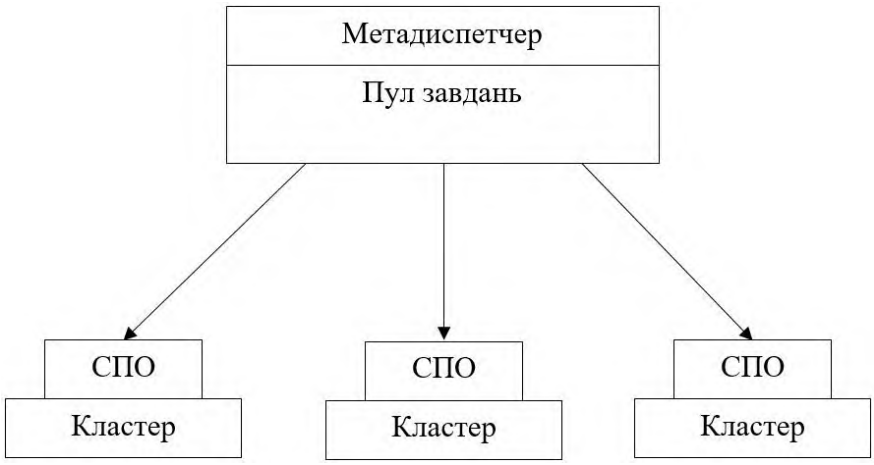



Рис. 1.4. Розподіл завдань на Grid кластери

Спільна робота локального і глобального рівня планування повинна враховувати використовувані в них процедури надання ресурсів. Ця взаємодія має відображати механізм своєчасного виділення достатньої кількості ресурсів під завдання, що знаходяться в черзі метадіспетчера. Тому СПО повинна попередньо резервувати ресурси для конкретного завдання.

Робота систем управління розподіленими обчислювальними мережами грунтується на схемі взаємодії «менеджер - агент», схема якої представлена на рисунку 1.5. Агент є посередником між керованим ресурсом і головною керуючою програмою-менеджером. Щоб один менеджер мав можливість керувати різними реальними ресурсами, створюється модель керованого ресурсу (МКР), яка відображає тільки ті характеристики ресурсу, які необхідні для його контролю і управління. Менеджер отримує від агента тільки ті дані, які описуються моделлю ресурсу. Агент звільняє менеджера від непотрібної інформації про деталі реалізації ресурсу і забезпечує його обробленою і представленою в нормалізованому вигляді інформацією. На основі цієї інформації менеджер приймає рішення по керуванню і узагальнює дані про стан ресурсу. Менеджер і агент повинні мати у своєму розпорядженні однаковою МКР, інакше вони не зможуть зрозуміти один одного.


Рис. 1.5. Схема взаємодії менеджера, агента і керованого ресурсу

На рисунку 1 - інтерфейс менеджера з МКР; 2 - інтерфейс менеджер-агент; 3 - інтерфейс агенту з МКР; 4 - інтерфейс агенту з ресурсом.

У використанні цієї моделі між агентом і менеджером є велика різниця. Агент наповнює МКР поточними значеннями характеристик даного ресурсу, і в зв'язку з цим модель агента називають базою даних керуючої інформації. Менеджер використовує модель, щоб знати характеристики ресурсу, які їх

параметри він може вимагати від агенту і якими параметрами можна керувати.

Схема «менеджер-агент» дозволяє створювати розподілені системи управління складної структури. Така розподілена система включає велику кількість зв'язок менеджер-агент, що доповнюються робочими станціями операторів мережі. З їх допомогою здійснюється доступ до менеджерів. Кожен агент збирає дані і управляє певним елементом мережі. Менеджери, які іноді називають серверами системи управління, збирають дані від своїх агентів і зберігають їх в базі даних. Оператори, що працюють за робочими станціями, можуть з'єднатися з будь-якими з менеджерів і переглянути дані про керовану мережу, а також видати менеджеру директиви з управління мережею і її елементами. Наявність декількох менеджерів дозволяє розподілити між ними навантаження, забезпечуючи масштабованість системи. Більш гнучким є ієрархічна побудова зв'язків між менеджерами. Кожен менеджер нижнього рівня виконує також функції агента для менеджера верхнього рівня. Такий агент працює з укрупненою моделлю своєї частини мережі, в якій збирається саме та інформація, яка потрібна менеджеру верхнього рівня для управління мережею в цілому.

Важливою особливістю функціонування кластерів систем Grid є наявність двох вхідних потоків інформаційно-розрахункових завдань. Перший потік містить чергу локальних завдань кластера, а другий складається із завдань глобальної інфраструктури Grid. Система пакетної обробки потоків завдань в кластерізованій Grid показана на рисунку 1.6.

Обидва потоки, що надходять через інтерфейс системи управління пакетною обробкою (СУПО) кластера формують загальний потік (чергу) інформаційно-розрахункових завдань СПО, що розподіляються по обчислювальним вузлам комп'ютерного кластера. Таким чином, неминуче виникає конкурування локального і глобального потоків завдань за пріоритет володіння ресурсами кластера. В результаті цього систематично відбувається відчуження обчислювальних ресурсів кластера системою Grid, а також їх

відмову через внутрішні причини або через перевантаження - в обох випадках стабільна робота кластера порушується.

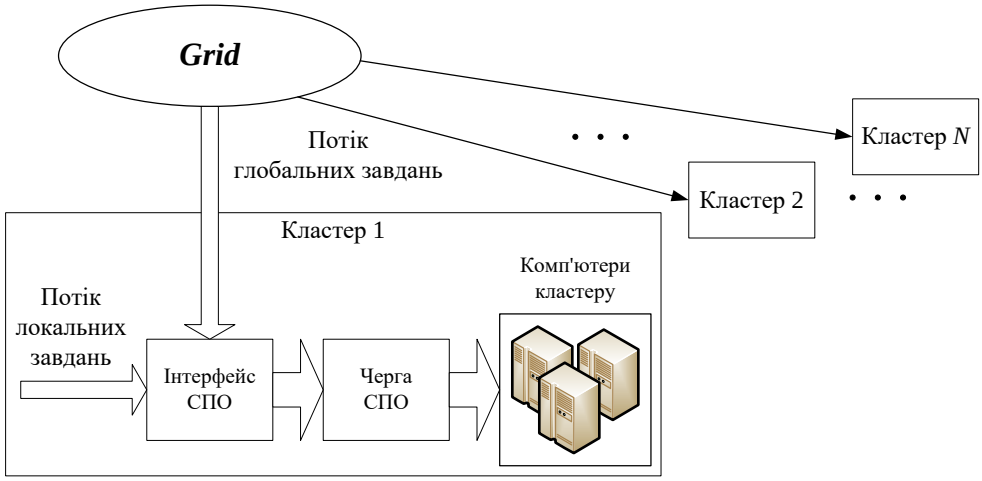



Рис. 1.6. Система пакетної обробки потоків завдань в кластерізованій Grid

Тому потрібен оперативний, в режимі реального масштабу часу, оптимальний перерозподіл завдань із загальної черги СПО по працездатним модулям Grid кластера таким чином, щоб домогтися максимального відновлення його функціональної потужності.

## 1.4 Системи управління ресурсами в розподілених обчислювальних системах

Система управління ресурсами (СУР) є основним компонентом (РОС). В даний час існує велика кількість проектів, орієнтованих на обчислення, в яких реалізовані СУР з різними архітектурами та службами [1-10, 14, 16, 17, 18, 21-23, 26, 27, 36, 45, 46]. В даному підрозділі розглянуті схеми взаємодії користувачів з ресурсами на основі СУР, існуючі класифікації СУР, що дозволяють визначити особливості проектів Grid з точки зору методів

управління ресурсами і планування. Розглянуті класифікації використовуються для опису існуючих СУР в великомасштабних системах розподілених обчислень – Grid - системах і мультикластерних системах.

Система розподілених обчислень - це віртуальна обчислювальна машина, утворена мережевою групою різнотипних ресурсів, призначених для розподілу локальних ресурсів в цих групах між собою. Таким чином, Grid - система являє собою великомасштабну розподілену систему, що функціонує на основі Інтернет-середовища з ресурсами (вузлами), розподіленими між різними організаціями і адміністративними областями. Обробка нових додатків в Grid - системі вимагає підтримки ефективних механізмів управління ресурсами і даними, що використовуються цими додатками. Тому розробка Grid - архітектури, що задовольняє цим вимогам, є складним завданням в силу наступних причин [18, 21-26, 45]:

- необхідності забезпечення адаптованості, розширюваності, масштабованості системи;

- надання можливості систем з різним адміністративним статусом взаємодіяти між собою і, в той же час, зберігати свою автономність;

- ефективного і безпечного спільного використання ресурсів, що розділяються;

- підтримки певного рівня якості служб;

- забезпечення існуючих обмежень на обчислювальні та інші витрати.

Для того щоб Grid - система підтримувала різні додатки, СУР - її центральна компонента - з урахуванням перерахованих причин повинна забезпечувати відмовостійкість і стабільність її роботи, що реалізується шляхом вирішення наведених нижче завдань [21, 22]:

- забезпечення ефективного управління ресурсами, які доступні в Grid - системі (процесорами, оперативної та дискової пам'яттю, пропускною спроможністю каналів зв'язку). Це дає можливість в залежності від специфіки системи визначати ефективні політики для користувачів і системи в цілому;

- забезпечення виробників інформації достовірної та актуальною інформацією, що не призводить до зниження ефективності СУР (наприклад, збільшення витрат на основні операції, які нею проводяться);

- гарантувати управління ресурсами СУР відповідно до існуючих політик. Внаслідок наявності різних адміністративних політик і неоднорідності (гетерогенності) ресурсів Grid - системи локальні СУР можна об'єднувати і використовувати таке об'єднання замість однієї СУР. Це дозволить більш ефективно використовувати локальні системи управління ресурсами (ЛСУР) і планування;

- при наявності обмежень на якість обслуговування СУР повинна забезпечувати і управління ресурсами і підвищення продуктивності СУР.

## 1.5 Задача та методи планування виконання завдань в розподілених обчислювальних системах

Важливим завданням при організації РОС є планування виконання завдань. В даному випадку під плануванням розуміється розподіл завдань, що надходять на вхід на наявні ресурси. Процес планування включає резервування ресурсів для завдання, для чого необхідно знайти підходящі ресурси, вибрати необхідну їх кількість і знайти оптимальний час запуску відповідної програми. Якщо запитувані завданнями ресурси перевищують наявні, це веде до конфліктів, які повинні бути дозволені алгоритмом планування.

Система планування в Grid повинна бути готова приймати потік завдань, генерувати розклади і відповідно до них ставити завдання на виконання. Завдання можуть надходити для планування одночасно. При постановці завдання в чергу потрібен файл опису цього завдання. Різні системи планування можуть працювати з різними форматами файлів опису.

Опис завдання зазвичай включає інформацію наступних категорій:

- Інформація про користувача. Вона може бути використана для визначення пріоритету завдання і доступних для користувача ресурсів. Часто інформація про користувача надходить в систему планування виходячи з даних про відповідну користувачеві системного облікового запису.

- Вимоги завдання до ресурсів. Ці дані конкретизують ресурси, запитувані завданням користувача. Часто ця категорія включає необхідну кількість і тип процесорів, обсяг оперативної пам'яті, а також інші типи ресурсів апаратного і програмного забезпечення, необхідні для виконання завдання. Частина інформації даної категорії може бути визначена наближено, як, наприклад, тривалість виконання завдання.

- Мета планування. Інформація даної категорії в деяких випадках допомагає алгоритму складання розкладу знаходити кращий розклад. Наприклад, користувач може вказати, що йому потрібен результат виконання завдання на певний термін, таким чином підказуючи, що дані йому не знадобляться раніше. У разі, якщо користувачі отримують платний доступ до ресурсів, в даній категорії може вказуватися, що користувач готовий заплатити за прискорення розрахунку.

Для деяких систем планування важливою для складання розкладу інформацією є необхідна кількість ресурсів і передбачувана тривалість виконання. Так, алгоритм зворотного заповнення використовує тривалості виконання завдань для визначення порядку їх запуску. У процесі складання розкладу система планування може масштабувати значення ймовірної тривалості виконання відповідно до характеристик процесорів, на яких завдання буде запущено. Для масштабування даного значення системою планування, користувач повинен вказати параметри процесорів, для яких було отримано значення тривалості виконання.

Також в файлі опису завдання може бути вказана додаткова інформація, така як ім'я завдання, ім'я файлу стандартного виводу помилок,

розташування файлів із вхідними даними. Може бути зазначена явна вимога завдання запускати всі його обчислювальні процеси одночасно.

При розробці і дослідженні алгоритмів в даній роботі передбачалося, що після запуску завдання не буде перервано. Також передбачалося, що міграції завдань з одних ресурсів на інші після їх запуску заборонені. Вимоги завдання до ресурсів також фіксуються і не можуть бути змінені після того, як завдання було сплановано на запуск. Альтернативна модель завдання може припускати, що завдання пристосовані до перезапуску і самостійно (або засобами операційної системи) здатні відновити свій стан, збережений на момент переривання.

Існує безліч форматів опису завдань, але стандартним вважається формат JSDL (Job Submission Description Language), а також його розширення для високопродуктивних систем - JSDL-WG. Стандарти розроблені некомерційною організацією OGF (Open Grid Forum), що представляє спільнота користувачів, розробників програмного забезпечення і організацій, що беруть участь в безпосередньому розвитку та впровадженні в різних сферах діяльності високопродуктивних обчислень. Далі, під стандартом JSDL-WS мається на увазі стандарт версії 1.0, заснований па мові опису даних XML. Перевагою даного формату є те, що як структуру, так і типи даних в XML документі можна перевірити на відповідність заданій схемі. Також XML дані порівняно легко можуть сприйматися і змінюватися людиною.

У загальному випадку система планування включає [13] наступні компоненти:

- політики планування;
- цільова функція;
- алгоритм складання розкладу.

Політики планування - це правила, що встановлюють черговість виділення ресурсів завданням. Часто ресурсів виявляється недостатньо для задоволення потреб всіх завдань негайно. У таких випадках політика

планування покликана вирішити цей конфлікт. При плануванні завдань в РОС будемо дотримуватися правила пріоритетів - процеси, що належать завданням з пріоритетом вище повинні бути запущені не пізніш процесів з пріоритетом нижче, якщо це можливо.

Для оцінки розкладу необхідний метод, що забезпечує однозначне їх порівняння. Цільова функція - функція оцінки розкладу по заданому критерію. Таким чином, є можливість ранжувати розклади по їх ефективності. Часто цільова функція повертає для конкретного розкладу числове значення в діапазоні [0, 1], де значення 1 досягається в разі оптимального розкладу. Цільова функція може містити в собі як один, так і кілька критеріїв [15]. В основі цільової функції може лежати довжина розкладу, середній час виконання всіх завдань, середній час очікування запуску завдань, завантаженість ресурсів та інші [91].

Алгоритм складання розкладу - ключовий компонент системи планування, який на підставі інформації про ресурси і завдання, політик планування і цільової функції будує несуперечливий розклад. Основними властивостями зазначеного алгоритму є: ефективність одержуваних розкладів; час складання розкладу; кількість обчислювальних ресурсів, що необхідні самому алгоритму для складання одного розкладу.

В даний час створюється інфраструктура, яка забезпечує надійний, стійкий і недорогий доступ до високопродуктивних обчислювальних ресурсів. Перед науковими організаціями (Українська академія наук, Міністерство освіти України) ставиться завдання об'єднання національних, регіональних і тематичних Grid-розробок в єдину Grid-інфраструктуру, яка використовувалася б для підтримки наукових досліджень. Одним із завдань, яка знаходиться в рамках цих робіт, є завдання розподілу обчислювальних ресурсів між завданнями, які надходять на виконання.

Для ефективного використання виділених ресурсів необхідно вирішувати задачу планування. Метою планування є визначення ефективності розподілу завдань при використанні різних критеріїв. Процес

планування здійснюється динамічно (в процесі надходження завдань на вхід) спеціальною програмою, яка носить назву планувальник (брокер) [1]. Керує запланованими потоком завдань, що надходять від користувачів, і проводить їх розподіл на наявні ресурси за допомогою інформації, яку надає користувач (опис вимог для коректного виконання завдань - мова JDL, JSDL, TSDL, ClassAds). Планувальник повинен задовольнити всі вимоги користувачів і оптимізувати загальний час виконання завдання, а також вартість використовуваних ресурсів.

Будь-яке завдання, яке надходить в Grid-систему на виконання, описується за допомогою спеціальної мови. Для планувальника важливими параметрами завдання є ті, які визначають вимоги до ресурсів. Виникає питання, який з параметрів є головним, тобто по якому параметру здійснювати оптимізацію. В даний час існує ряд завдань, які необхідно аналізувати не тільки по одному параметру, що в свою чергу накладає труднощі для процесу розподілу. У таких випадках раціонально використовувати методи розв'язання багатокритеріальних задач оптимізації. Застосування цих методів має на увазі об'єднання ряду критеріїв в єдиний. Об'єднання критеріїв, як правило, проводиться евристичними методами, тому з математичної точки зору не існує ідеального рішення таких задач, кожен з них має переваги і недоліки.

Розглянемо набір методів, за допомогою яких Grid-система розподіляє вхідні завдання між ресурсами. У моделі, яка запропонована в даній роботі, існує можливість проводити дослідження із застосуванням методів планування виконання завдань, що описані нижче.

Метод планування MC. Метод заснований на пошуку найменшого числа ресурсів, якими можна вирішити всі завдання, тобто задача про найменше покриття (ЗНП). Для пошуку найменшого числа ресурсів використовується наближений частотний метод [1, 5].

Алгоритм роботи MC: 1) визначаються відповідності між завданнями і ресурсами, які вони використовують; 2) вирішується завдання пошуку

мінімального покриття частотним методом; 3) завдання поміщаються в буфери ресурсів, які увійшли в мінімальне покриття; 4) процедура планування закінчується в разі, якщо помістилися всі завдання, інакше завдання повертаються в пул і виконується перехід до «операції № 2».

У процесі планування завдання, у яких не залишилося вільних ресурсів для їх вирішення, повертаються назад в пул.

Метод планування FCFS. Метод заснований на принципі «першим прийшов, першим обслужився».

Алгоритм роботи FCFS: 1) визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися; 2) вибирається перше по порядку завдання з пулу і поміщається в перший вільний ресурс, яким воно може бути вирішено; якщо таких ресурсів не виявилося, завдання повертається в пул; 3) процедура планування припиняється після обробки в пулі останнього завдання. В даному методі не використовується буфер, завдання відразу надсилаються на рішення ресурсу, якщо він вільний.

## 1.6 Постановка задачі на дослідження

Системи планування виконання завдань забезпечують вирішення проблеми ефективного та гнучкого призначення завдань, що надійшли на доступні обчислювальні ресурси розподілених систем обробки даних (РСОД). При використанні РСОД основною проблемою є трудомісткість налаштування програмного забезпечення, що виконує призначення завдань на обчислювальні ресурси, яка пов'язана з наступними властивостями РСОД:

- різнорідність завдань згідно ресурсним вимогам, апаратна гетерогенність обчислювальних вузлів і різні завантаження вузлів РСОД вимагають спеціального обліку, що веде до створення складних політик планування;

- відсутність повної інформації про ресурсні вимоги завдань ускладнює прийняття інтелектуальних рішень по їх плануванню.

Широке поширення кластерних, Grid і хмарних систем пов'язано зі збільшенням числа розв'язуваних прикладних завдань і значним зростанням навантажень на обчислювальні системи. Постачальники мережевих сервісів, спираючись на великі консолідовані центри обміну даних, особливу увагу стали приділяти вдосконаленню методів планування завдань обробки даних. У загальному випадку схема керування ресурсами в Grid має вигляд, представлений на рисунку 1.7.


Рис. 1.7. Схема керування пакетною обробкою в Grid

Чинними версіями СКПО є: Platform LSF, Windows Compute Cluster Server, Condor, PBS, SGE, TORQUE, LoadLever, MOSIX [1]. Однією з найбільш важливих функцій СКПО є забезпечення механізму планування розподілу завдань. Ця функція може бути реалізована безпосередньо розробниками конкретної СУПО, або за допомогою зовнішніх планувальників - окремо розроблених програмних засобів (наприклад, планувальника Maui). В основі кожного планувальника лежить алгоритм, від якого багато в чому залежить ефективність управління завданнями в цілому.

На рисунку 1.8 наведена концептуальна схема планування пакетів завдань в РОС [52], в якій в якості алгоритму планування використано рішення ЗНП.

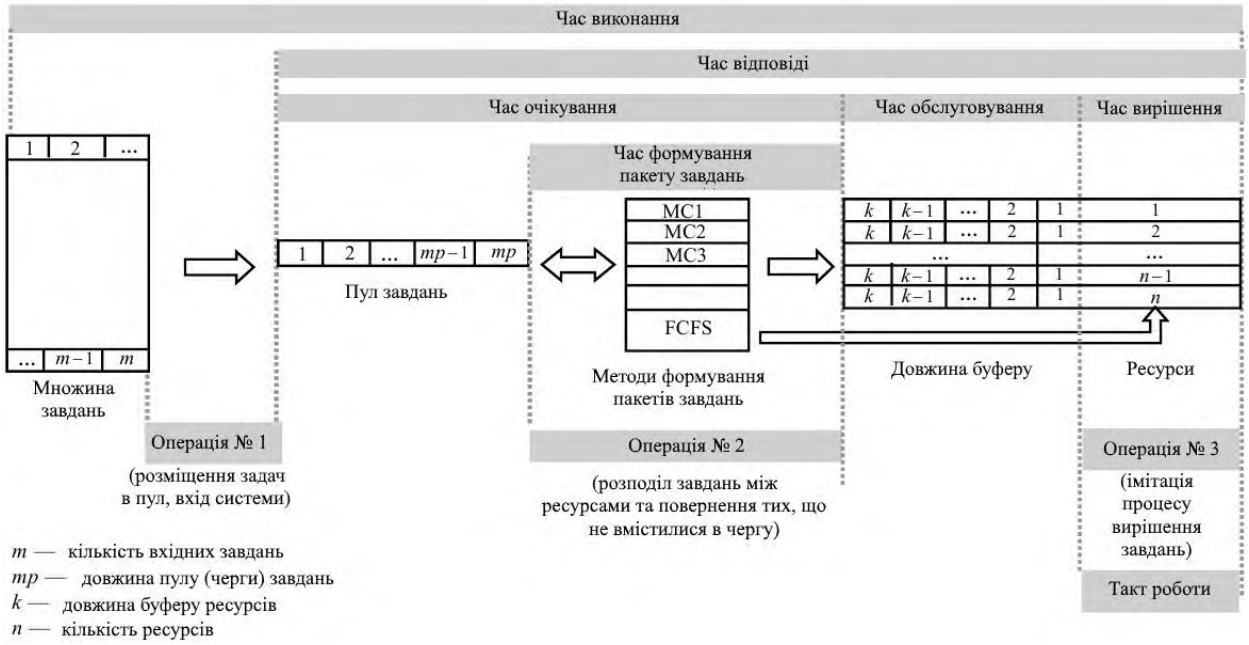



Рис. 1.8. Структурна схема моделі кластера Grid - системи: операція 1 - приміщення завдань в пул, вхід в систему; операція 2 - розподіл завдань між ресурсами і повернення, що не помістилися в чергу; операція 3 - імітація процесу рішення задач; MC1- MC3, FCFS - методи формування пакетів завдань.

Як показано в [86,87], дана концепція планування дозволяє істотно скоротити сумарний час виконання завдань і збільшити коефіцієнт використання ресурсів системи в порівнянні з алгоритмом FCFS (First Come First Serve). Це переліковий алгоритм і алгоритм зворотного заповнення (Backfill), в якому завдання вишиковуються в перелік відповідно до їхнього вступу і як тільки необхідна кількість ресурсів стає доступним завданню, що знаходиться на початку списку, завдання надходить на ресурс. Однак до недоліків даного підходу можна віднести той факт, що в цьому випадку немає можливості забезпечити максимальний сумарний пріоритет

виконуваних завдань на окремих етапах планування в РОС. Пропонується розроблена процедура планування, що дозволяє подолати зазначений недолік, при цьому зменшити сумарний час обробки завдань в системі в порівнянні з процедурою планування на основі рішення ЗНП.

Прагнення до максимальної суми пріоритетів обраних завдань є головним критерієм при виборі завдань з черги, який будемо характеризувати коефіцієнтом важливості і коефіцієнтом збереження важливості. Коефіцієнт важливості - середнє геометричне відносини сумарного пріоритету завдань, поміщених в буфери в результаті планування, до сумарного пріоритету завдань, які перебували в пулі:

$$r_i = \sqrt[p]{\prod_{j=1}^{p} r_{ij}}\ ,\ r_{ij} = \frac{\sum_{k=1}^{m_{pj}} \mathrm{Pr}_k}{\sum_{k=1}^{m_{wj}} \mathrm{Pr}_k}, \qquad (1.1)$$

де $r_{ij}$ - відношення пріоритету завдань, поміщених в буфер ресурсів в результаті планування, до пріоритету завдань, які перебували в пулі системи на $j$ - му плануванні; $m_{pj}$ - число завдань, розпланованих між ресурсами (поміщених в буфер ресурсів) при $j$ - му плануванні; $m_{wj}$ - кількість завдань, що знаходилися в пулі при $j$ - му плануванні; $\mathrm{Pr}_k$ - пріоритет $k$ - ї задачі.

Коефіцієнт збереження важливості - середнє геометричне відносини сумарного пріоритету завдань, поміщених в буфери в результаті планування після відмови випадкового числа ресурсів, до сумарного пріоритету завдань, поміщених в буфери в результаті планування до відмови ресурсів:

$$r_s = \sqrt[p]{\prod_{i=1}^{p} r_{si}}\ ,\ r_{si} = \frac{\sum_{k=1}^{m_{fi}} \mathrm{Pr}_k}{\sum_{k=1}^{m_{pi}} \mathrm{Pr}_k}, \qquad (1.2)$$

де $r_{si}$ - відношення пріоритету завдань, поміщених в буфери ресурсів після планування при випадковій відмові ресурсів, до пріоритету завдань, поміщених в буфери ресурсів після планування до відмови ресурсів на $i$ - му плануванні; $m_{pj}$ - число завдань, розпланованих між ресурсами (поміщених в буфер ресурсів) до відмови випадкового числа ресурсів, на $j$ - му плануванні; $m_{fi}$ - число завдань, розпланованих між ресурсами (поміщених в буфер ресурсів) після відмови випадкового числа ресурсів на $i$ - му плануванні.

Важливою характеристикою також є коефіцієнт прискорення роботи сегмента Grid - системи.

Коефіцієнт прискорення - відношення розрахункового часу виконання завдань розв'язуваних послідовно одним ресурсом з продуктивністю, яка дорівнює середньому значенню продуктивності всіх ресурсів Grid, до часу виконання цих же завдань Grid - системою:

$$r_a = \frac{t_{ов}}{t_в},\ t_{ов} = \frac{S}{\overline{P}}, \tag{1.3}$$

де $t_{ов}$ - розрахунковий час послідовного вирішення завдань одним ресурсом з продуктивністю, яка дорівнює середньому значенню продуктивності ресурсів Grid; $t_в$ - час виконання завдань Grid - системою; $S$ - сумарна складність вирішення всіх завдань; $\overline{P}$ - середня продуктивність всіх ресурсів Grid.

## 1.7 Висновки за розділом

Таким чином, важливим напрямком розвитку сучасних СКПО в кластерах ТКС є розробка відповідних методів моделей планування. Тому метою дисертаційного дослідження є підвищення ефективності та якості обслуговування завдань з управління ТКС, що характеризуються:

коефіцієнтом важливості, коефіцієнтом збереження важливості і коефіцієнтом прискорення виконання завдань, а також розширення функціональних можливостей розподілених ТКС на основі використання обчислювальних кластерів із застосуванням Grid-технологій. Мета полягає у підвищенні оперативності планування розподілу завдань з управління в кластерах телекомунікаційних систем за рахунок розробки методу планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування.

У дисертаційній роботі розв’язані такі окремі задачі дослідження:

1. Аналіз основних напрямків удосконалення кластерів телекомунікаційних систем і проблем, що виникають при вирішенні завдань планування виконання задач з управління телекомунікаційними мережами в кластерах Grid-систем та вибір показників якості їх функціонування.

2. Подальший розвиток методу вирішення завдань нелінійного булевого програмування (НБП) для підвищення оперативності розподілу завдань в кластерах ТКС, заснованого на ідеї рангового підходу.

3. Розробка методу оперативного планування розподілу завдань на основі вирішення задач НБП у кластерах ТКС з використанням Grid-технології на основі використання методу групової вибірки з індивідуальною сегментацією.

4. Розробка моделі функціонування кластера Grid-системи на основі використання для планування виконання завдань вирішення задач НБП.

5. Оцінка ефективності та перевірка адекватності розробленого методу і математичної моделі шляхом проведення їх аналітичного і експериментального дослідження із застосуванням комп'ютерного імітаційного моделювання.

6. Розробка рекомендацій щодо інтеграції розробленого методу і моделі планування розподілу завдань в системи управління сучасних і перспективних ТКС.

# РОЗДІЛ 2

# РОЗРОБКА МЕТОДУ ОПЕРАТИВНОГО ПЛАНУВАННЯ РОЗПОДІЛУ ЗАВДАНЬ З УПРАВЛІННЯ ТЕЛЕКОМУНІКАЦІЙНИМИ МЕРЕЖАМИ НА ОСНОВІ МЕТОДУ ГРУПОВОЇ ВИБІРКИ З ІНДИВІДУАЛЬНОЮ СЕГМЕНТАЦІЄЮ

## 2.1 Формалізація задачі на основі методу групової вибірки з індивідуальною сегментацією

### 2.1.1 Процес обробки запитів в кластері

Завдання Grid являє собою звичайний файл, що виконується (скрипт, програмний код). Службами Grid завдання доставляється на виконавчі ресурси, а виконання відбувається в середовищі операційної системи (ОС) цих ресурсів. Як правило, програма, що підготовлена на певному комп'ютері, не вимагає яких-небудь модифікацій для використання в Grid. Однак будь-яка програма розрахована на певне середовище виконання - ОС, архітектуру комп'ютера, обсяги і характеристики його ресурсів.

Планувальник повинен враховувати, що кластер Grid - системи може бути гетерогенної інфраструктури, до складу якої можуть бути включені комп'ютери з різною архітектурою і комплектацією. Гетерогенність проявляється в тому, що різні класи ресурсів (процесор, основна пам'ять, кеш-пам'ять, дискова пам'ять) розрізняються по типу (процесори - архітектурою) і за характеристиками (процесори - продуктивністю, пам'ять - об'ємом). У зв'язку з цим виконавчі ресурси, що обираються, не можуть бути довільними, а повинні відповідати вимогам завдання.

Слід звернути увагу на параметр ресурсного запиту, який визначає час використання ресурсів, тобто час виконання завдання. Відомо, що

призначена для користувача оцінка цього часу рідко буває точною, проте наявність цього параметра є досить важливим з двох причин. По-перше, навіть приблизна оцінка часу виконання дозволяє використовувати більш ефективні алгоритми планування. По-друге, існує загальноприйнята практика роботи в системах з ресурсами, що розділяються, згідно з якою час виконання (як і ресурсний запит в цілому) служить захистом від програмних помилок, представляючи собою обмеження (за обсягами і за часом) на споживані ресурси, тобто при перевищенні зазначених у запиті лімітів завдання примусово завершується.

Зауважимо, що параметр часу виконання замовляється з розрахунку на певну продуктивність ресурсів. Як в процесі планування, так і при запуску завдання необхідно виконати перерахунок часу відповідно до конкретних ресурсів, на яких воно буде виконуватися. На практиці набули поширення кілька мов ресурсних запитів [2]. Мова RSL, застосовуваний в системі Globus Toolkit, орієнтована на запуск як однопроцесорних, так і багатопроцесорних MPI завдань. Мови ClassAd системи Condor і JDL (WMS) дозволяють визначати альтернативні варіанти ресурсного запиту і уточнювати переваги користувачів при виборі ресурсів. У роботах [1-3] запропоновано суттєві розширення формалізму мови запитів, спрямовані на специфікацію пов'язаних завдань і ланцюжків завдань.

При плануванні відбір ресурсів виконується по інформаційній базі, яка містить відомості про склад і характеристики ресурсів Grid. Поставка цих даних в інформаційну базу здійснюється в оперативному режимі спеціалізованими розподіленими системами моніторингу ресурсів, з яких найбільш поширені MDS і R-GMA.

Розглянемо планування на основі пріоритетів, коли передбачається, що ресурси для більш пріоритетних завдань виділяються раніше, ніж для менш пріоритетних. У цих умовах широко використовуються алгоритми типу FCFS (FIFO), що працюють за принципом виділення звільнившихся ресурсів для самого пріоритетного завдання з черги, яке може на них розміститися. При

цьому велика частина процесорів буде завжди зайнята дрібними завданнями, тобто виникає фрагментація ресурсів. Навіть якщо завдання має найвищий пріоритет, необхідний йому обсяг ресурсів може ніколи не утворитися і, отже, завдання може ніколи не стартувати. Для середовища, яке обслуговує однопроцесорні завдання, цієї проблеми немає. Вона виникає, коли є колективні ресурси, наприклад при обслуговуванні багатопроцесорних завдань. Аналогічна ситуація виникає на машинах із загальною пам'яттю, в середовищі із загальним файловим простором і в інших випадках, коли ресурси діляться між завданнями, а не виділяються під завдання цілком.

Планування в Grid розглядається як циклічний процес обробки фіксованого на момент планування безлічі завдань, що знаходяться в черзі і розподілу їх по ресурсах відповідно з певним часом виділення ресурсів і їх адресою. При виконанні цього процесу здійснюється координація поділу ресурсів між завданнями користувачів. У середовищі такого масштабу, як Grid, планування є найважливішим механізмом забезпечення якості обслуговування, перш за все, забезпечення прийнятного і передбачуваного часу виконання завдань користувача, а також гнучкості політики розподілу ресурсів у відповідність з пріоритетами.

Ресурси, що використовуються в режимі поділу між власниками і користувачами Grid, називають невідчужуваними. Перевага Grid з невідчужуваними ресурсами полягає в тому, що не потрібні витрати на формування спеціальної ресурсної бази і кластер Grid може створюватися динамічно на обмежений період часу з метою вирішення будь-якої великої проблеми, для якої ресурсів окремих членів кооперації не вистачає. З точки зору планування невідчужуваність ресурсів ускладнює ситуацію, так як в цьому випадку на них надходить два потоку завдань: 1) потік з Grid (глобальний), керований планувальником; 2) локальний потік завдань, які запускаються засобами, відмінними від інтерфейсів користувачів Grid. Наприклад, в умовах кластеризованих ресурсів локальні завдання вводяться безпосередньо через інтерфейси СПО. Таким чином, локальні завдання

непідконтрольні планувальнику, хоча створюване ними завантаження ресурсів повинно враховуватися планувальником при розподілі глобальних завдань.

Процес обслуговування стандартизованих запитів на виконання обчислень, оформлених у вигляді завдань для загальнопоширених операційних систем, причому виконання цих завдань проводиться на ресурсах, які вибираються із загального пулу, проводиться наступними основними етапами обробки завдання:

1) присвоєння пріоритетів завданням і моніторинг стану ресурсів системи і стану виконання завдань на основі даних системи спостереження;

2) планування вибірки завдань з черги;

3) виділення із загального пулу виконавчих ресурсів, на яких завдання виконуватиметься;

4) доставка файлів для виконання і вхідних файлів на виконавчі ресурси;

5) виконання завдання;

6) після закінчення виконання завдання доставка файлів з результатами на сервери зберігання (на робоче місце користувача).

Всі перераховані етапи обробки завдання виконуються автоматично без участі суб'єкта, який видав запит (зокрема, користувача, але це може бути і програма). Тому кластери Grid - систем дійсно представляють собою єдину операційну середу.

### 2.1.2 Характеристики процесу обробки запиту в кластері

При наявності обмежень на розмір черги, що зберігається в системі, може виникнути ситуація, коли черговий запит на виконання завдання, що надійшов на вхід системи, не буде прийнятий до обслуговування, що

призведе до його втрати або затримки в обслуговуванні. Тому актуальною є розробка системи з прийнятною (необхідною) швидкістю обслуговування завдань, що надходять із заданою інтенсивністю.

Необхідно врахувати додаткову вимогу до системи управління - вона повинна мати одночасний доступ до будь-якої кількості вільних ресурсів на даний момент часу і на одному ресурсі системи може виконуватися тільки одне завдання.

Під вирішенням черги (ВЧ) будемо розуміти процес поетапної вибірки завдань з черги завдань, які необхідно і можливо виконати на даному етапі. Завдання в чергу надходять в будь-який момент часу, але стан черги визначається тільки між етапами. При виконанні ВЧ завдань необхідно вибрати найбільш прийнятний спосіб вибірки завдань і найкращий метод реалізації обраного способу.

Для оцінки ефективності вибірки завдань будемо використовувати коефіцієнт використання ресурсів $K_{ВР}$, що дозволяє визначити, яка частина із загальної кількості ресурсів, до яких звертаються завдання, що знаходяться в черзі, буде використана. Якщо $K_{ВР} = 0$, то жоден ресурс не буде використаний, якщо $K_{ВР} = 1$, то будуть використані всі ресурси, для яких існують завдання на даний момент. Однак не завжди можна вибрати такі завдання, щоб всі ресурси були задіяні. Тому виникає завдання вибору такого способу ВЧ завдань, при якому $K_{ВР} \rightarrow 1$. У загальному вигляді можна записати:

$$K_{ВР} = \frac{N_B}{N_O}, \tag{2.1}$$

де $N_B$ - кількість ресурсів, які будуть задіяні при реалізації певної вибірки; $N_O$ - кількість ресурсів, до яких існують завдання з черги.

Але в [4] не враховано необхідні пріоритети завдань. Якщо їх врахувати, то значення коефіцієнта $K_{BP}$ буде залежити від способу вибірки (тільки чисельник в (2.1)).

Нехай $Y_i$ максимальна величина пріоритету завдань з черги завдань, які звертаються до ресурсу $R_i$, $M$ - число ресурсів. Тоді знаменник в (2.1) набуде вигляду:

$$\sum_{i=1}^{M} Y_i . \qquad (2.2)$$

Найбільш перспективними способами обслуговування завдань є способи групової вибірки. Спосіб груповий вибірки - це такий спосіб, при реалізації якого з черги завдань обслуговується кілька завдань одночасно. Вибираються завдання, для яких потрібна обробка на різних ресурсах, і сума пріоритетів яких максимальна. У разі наявності рівнозначних завдань вибирають більш «старі».

Нехай $\left\{\vec{X}\right\}$ - множина всіх варіантів вибірки завдань з черги і $\vec{X}$ - один з варіантів вибірки завдань:

$$\vec{X} = \{x_1, x_2, ..., x_p, ..., x_N\},\ p = \overline{1..N}, \qquad (2.3)$$

де $N$ - число завдань в черзі; $x_p$ - булева змінна. Якщо завдання $З_р$ вибрано в цьому варіанті, то $X_р = 1$, якщо немає, то $X_р = 0$.

Нехай $\beta_p$ - пріоритет завдання $З_р$. Тоді чисельник в (2.1) набуде вигляду:

$$\sum_{p=1}^{N} \beta_p x_p . \qquad (2.4)$$

З урахуванням (2.2) і (2.4) отримуємо:

$$K_{ИР} = \frac{\sum_{p=1}^{N} \beta_p x_p}{\sum_{i=1}^{M} Y_i}. \qquad (2.5)$$

Для того щоб коефіцієнт $K_{ВР}$ прийняв одиничне значення, необхідно щоб в (2.5) чисельник дорівнював знаменника (в кращому випадку). У загальному випадку чисельник повинен прагнути до знаменника. Оскільки знаменник для конкретного моменту часу є константою, необхідно зробити таку вибірку завдань з черги, щоб чисельник прийняв максимальне значення, яке не повинно перевищити знаменник. Це означає, що необхідно вибрати з черги якомога більше число завдань, які звертаються до різних ресурсів, сума пріоритетів яких була б максимальна. Прагнення до максимальної суми пріоритетів обраних завдань є головним критерієм при виборі завдань з черги.

### 2.1.3 Математична модель процедури планування

Функція відображення *F* завдань *T* на ресурси *R* системи являє собою матрицю відповідності:

$$F(Matching): T \times R \times ComLinkThroughout \rightarrow R^{+},$$

де *Matching* - матриця відповідності завдань, що плануються *T* ресурсам *R* з урахуванням пропускної здатності безлічі комунікаційних каналів зв'язку *ComLinkThroughout* між спланованими завданнями і

ресурсами. Дана матриця є основою для формування функціоналу і обмежень.

Відповідно до (2.3) для опису суми пріоритетів $\beta_k$ обраних завдань $x_k$ використовуємо функціонал (2.1):

$$F = \sum_{k=1}^{p} \beta_k x_k \rightarrow max. \qquad (2.6)$$

Нехай $A_{kg}$ - булева змінна, що дорівнює одиниці, якщо завдання $З_к$ використовує ресурс $R_g$ і дорівнює нулю, якщо ні; $B_g$ - кількість ресурсів типу $R_g$. Тоді, виходячи з умови, що в будь-який момент часу будь-який ресурс може бути використаний для виконання завдання, отримуємо $M$ обмежень вигляду:

$$\sum_{k=1}^{p} A_{kg} x_k \le B_g, \; g = \overline{1..M}. \qquad (2.7)$$

Отже, необхідно знайти таку вибірку $\vec{X}$ з безлічі $\{\vec{X}\}$, для якої функціонал (2.6) прийме максимальне значення при виконанні всіх обмежень (2.7). Отримано завдання лінійного програмування з булевими змінними. ВЧ запитів при такій формалізації відбувається поетапно. Кожен етап складається зі знаходження оптимальної вибірки $\vec{X}$, її обслуговування і зміни функціонала (2.6) і обмежень (2.7) з урахуванням змін в черзі після обслуговування вибірки.

Приклад. Розглянемо метод групової вибірки при вирішенні наступного завдання. Маємо чергу з семи завдань, кожне з яких має свій пріоритет і вимагає використання ресурсів певного типу. Вихідні дані завдання у вигляді матриці відповідності представлені в наступному вигляді:

| Завдання | $З_1$ | $З_2$ | $З_3$ | $З_4$ | $З_5$ | $З_6$ | $З_7$ |
|---|---|---|---|---|---|---|---|
| Пріоритет | 1 | 3 | 2 | 2 | 4 | 1 | 1 |
| Ресурс | $R_1$, $R_3$ | $R_2$ | $R_1$, $R_4$ | $R_4$ | $R_1$ | $R_5$ | $R_3$, $R_4$ |

За цими даними визначимо, які завдання звертаються до кожного ресурсу:

| Ресурс | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| Завдання | $З_1$, $З_3$, $З_5$ | $З_2$ | $З_1$, $З_7$ | $З_3$, $З_4$, $З_7$ | $З_6$ |

Етапи вирішення задачі:

1. Запишемо функціонал (9), підставивши зазначені вище значення пріоритетів:

$$F = З_1 + 3З_2 + 2З_3 + 2З_4 + 4З_5 + З_6 + З_7 \to \max \ . \qquad (2.8)$$

Обмеження (2.7) матимуть вигляд:

$$З_1 + З_3 + З_5 \le 1, \qquad (2.9)$$

$$З_1 + З_7 \le 1, \qquad (2.10)$$

$$З_3 + З_4 + З_7 \le 1. \qquad (2.11)$$

Обмеження (2.9) складено для ресурсу $R_1$, (2.10) - для $R_3$, а (2.11) - для $R_4$.

Вирішивши завдання лінійного програмування з булевими змінними визначаємо, що на першому етапі можуть бути вирішені завдання $З_2$, $З_4$, $З_5$ та $З_6$.

2. Запишемо функціонал (2.6) з урахуванням результатів першого етапу:

$$F = З_1 + З_3 + З_7 \to \max . \qquad (2.12)$$

Обмеження (2.7) матимуть вигляд:

$$З_1 + З_3 \le 1;\ З_1 + З_7 \le 1;\ З_3 + З_7 \le 1. \qquad (2.13)$$

В результаті рішення даної задачі знаходимо, що необхідно обслужити завдання $З_3$.

3. Аналогічно визначаємо, що на третьому етапі необхідно обслужити завдання $З_1$.

4. На четвертому етапі необхідно обслужити завдання $З_7$.

Отже, в даному випадку ВЧ буде виконано в чотири етапи.

## 2.2 Метод групової вибірки з індивідуальною сегментацією

Метод групової вибірки з індивідуальної сегментацією - це такий метод, при реалізації якого завдання, що знаходяться в черзі, розбиваються на підзадачі. З черги отриманих підзадач вибирають ті, для реалізації яких потрібні ресурси різного типу, і сума пріоритетів яких максимальна. У разі наявності рівнозначних підзапитів вибирають більш «старі».

Підзадача - це частина завдання, для реалізації якої потрібно використання ресурсу конкретного типу. Якщо завдання вимагає $K$ різних ресурсів, то його розбивають на $K$ підзадач. В цьому випадку треба вибрати з черги якомога більше число підзадач, що звертаються до різних ресурсів, сума пріоритетів яких була б максимальна. Прагнення до максимуму суми пріоритетів обраних підзадач є головним критерієм при виборі підзадач з черги.

Нехай $З_{kg}$ - підзадача завдання $З_k$, що звертається до ресурсу $R_g$; $C_{kg}$ - пріоритет підзадачі $З_{kg}$; $\{\vec{X}\}$ - безліч всіх варіантів вибору підзадач з черги; $\vec{X}$ - один з варіантів вибору підзадач $\vec{X}=\{x_{11},x_{12},...,x_{kg},...,x_s\}$, де $k=\overline{1..p}$, $g=\overline{1..M}$; $p$ - число завдань в черзі; $x_{kg}$ - булева змінна, що дорівнює одиниці, якщо обрана відповідна подзадача $З_{kg}$, і рівна нулю, якщо ні. Тоді функціонал (2.6) набуде вигляду:

$$F(x)=\sum_{j=1}^{p_1}C_{1j}S_1(C_n^1)+\sum_{j=1}^{p_2}C_{2j}S_2(C_n^2)+...+\sum_{j=1}^{p_k}C_{kj}S_k(C_n^k)+...+\sum_{j=1}^{p_n}C_{nj}S_n(C_n^n)\to\max, \quad (2.14)$$

де $Sr(C_n^r)=S_1+S_2+...+S_{p_r}$ - сума всіх можливих поєднань добутків змінних, в кожному з яких міститься $Sr=X_pX_к...X_m$ $r$ різних змінних; $p_r=\dfrac{n!}{r!(n-r)!}$; $C_{rj}$ - коефіцієнти в добутках $Sr$, що містять $r$ змінних. Обмеження (2.7) приймуть вигляд:

$$\sum_{k=1}^{p}A_{kg}x_{kg}\le B_g,\ g=\overline{1..M}\ . \quad (2.15)$$

Отримано завдання нелінійного програмування з булевими змінними. Для розглянутого вище прикладу функціонал (2.14) набуде вигляду:

$$F=З_{11}+З_{13}+3З_2+2З_{31}+2З_{34}+2З_4+4З_5+З_6+З_{73}+$$
$$+З_{74}+З_{11}З_{13}+2З_{31}З_{34}+З_{73}З_{74}\to\max, \quad (2.16)$$

а обмеження (2.6) - такий вигляд:

$$З_{11}+З_{31}+З_5\le1;\ З_{13}+З_{73}\le1;\ З_{34}+З_4+З_{74}\le1. \quad (2.17)$$

При використанні методу групової вибірки з індивідуальною сегментацією дане ВЧ буде виконано в три етапи.

З розглянутого прикладу можна зробити наступні висновки:

1. Використання методу групової вибірки з індивідуальною сегментацією дозволяє скоротити число етапів ВЧ і зменшити частоту відмов в обслуговуванні завдань на вході системи при піковому навантаженні;

2. Для забезпечення ефективності запропонованого методу необхідно використовувати в якості математичного апарату методи з малою часовою складністю для вирішення завдань лінійного та нелінійного програмування з булевими змінними.

Тому в якості методу планування пропонується використовувати рангові алгоритми вирішення задач нелінійного булевого програмування [2].

## 2.3 Удосконалення методу розв'язання задачі нелінійного булевого програмування на основі рангового підходу

### 2.3.1 Ранговий підхід до вирішення задачі нелінійного булевого програмування

Завдання булевого лінійного та нелінійного програмування є моделями широкого класу прикладних завдань в теорії побудови складних систем і при цьому завдання булевого лінійного програмування відносяться до класу NP-повних важко вирішуваних завдань [1], а ефективні методи вирішення завдань нелінійного булевого програмування з довільними нелінійностями практично відсутні. В даний час для кожного типу завдання розробляється свій метод вирішення. Є актуальним виробити єдиний підхід до вирішення даного класу задач, що забезпечує їх вирішення з необхідною оперативністю і точністю. Останнє обумовлено тим, що останнім часом набули широкого

поширення розподілені обчислення, які в масштабах глобальної мережі стають одним з ключових напрямків розвитку мереж, що розширяють наше уявлення про способи використання обчислювальних ресурсів і самої мережі в цілому. Про це свідчить поява Grid технологій і хмарних обчислень, що розглядають мережу як один єдиний обчислювальний ресурс. Два основних напрямки, з яких беруть початок Grid технології, це паралельні обчислення та розподілені обчислення. Саме синтез знань цих двох областей і їх застосування необхідні для реалізації концепції розподілених обчислювальних середовищ. Розподілені обчислення в масштабах глобальної мережі стають одним з ключових напрямків розвитку мереж, що розширює наше уявлення про способи використання обчислювальних ресурсів і самої мережі в цілому [29,31]. Розглянемо підхід, що дозволяє вирішувати завдання булевого лінійного та нелінійного програмування з довільними нелінійностями одним і тим же алгоритмом, заснованим на ідеях рангового підходу [44,58]. Суть підходу визначається наступним. Покажемо, що для визначення властивостей довільної складної системи можна використовувати стягнуте дерево всіх шляхів графа, що дозволяє з єдиних позицій вирішувати довільні задачі теорії графів і комбінаторної оптимізації, до яких зводяться моделі підтримки великої кількості різноманітних систем управління. Використання стягнутого дерева шляхів, в свою чергу, дозволить ефективно распаралелити процес їх вирішення на одному типі архітектур паралельних обчислювальних систем (ПОС), що в певному сенсі знімає проблему, пов'язану з тим, що для різних типів завдань необхідно розробляти різні архітектури ПОС.

Розглянемо кінцеву множину довільних об'єктів теорії графів або конфігурацій комбінаторної оптимізації. У загальному випадку об'єкт може бути довільним, але має бути визначено кінцева множина елементів $\Omega = \{\omega_I\}$ або підмножини $L_i \in \Omega$ і правила $R$, що дозволяють формувати об'єкти з вихідних елементів або підмножин $L_i$, що належать множині $\Omega$. Нехай,

задано деякий поділ множини $\Omega$ на сімейства підмножин $\{L_i\}$ таке, що $\bigcup_i L_i = \Omega$ і $L_i$ описують об'єкти, що нас цікавлять, які складаються з базових елементів $\{l_i\}$ таких, що $\bigcup_i l_i = \Omega$ і правила *R*, що дозволяють з базових елементів визначати можливість об'єднання базових елементів $\{l_i\}$ і правила *P*, що визначають вагові характеристики довільних об'єднань $L_k \cup L_p \in \Omega$, що характеризують властивості $\{v\}$ розглянутих об'єктів і потрібно визначити можливість побудови об'єкта з властивістю $v^* \in \{v\}$, що нас цікавить.

Уявімо безліч всіх можливих об'єднань підмножин $L_i$ у вигляді графа $D$, зображеного на рисунку 2.1. Граф з паралельно ярусной структурою, що складається з *n* горизонтальних лінійок з вершинами 1, 2,..., *n* і *n* ярусами, кожен з яких містить всі вершини графа $D$, при цьому кожній вершині графа $D$ поставимо у відповідність базовий елемент $l_i$. Важливою характеристикою шляху в графі *D* є ранг шляху *r* - число ребер, що утворюють шлях. У графі *D* довільна вершина *i* може бути досягнута шляхами рангів *r* = 1, *r* = 2,..., *r* = *n* - 1, а довільному шляху $\mu_{st}$, що задовольняє правилам побудови *R* і проходить через вершини (*j*, *p*,..., *k*, *t*) відповідає об'єднання базових елементів $(1_j \cup 1_p \cup ... \cup 1_k \cup 1_t)$ відповідно до правила *R*, що визначають деякий об'єкт $L_i \in \Omega$. Довжина цього шляху $d(\mu_{st})$ визначається за правилами *P*. Отже, безліч всіх шляхів $m_{si}(r)$ в графі $D$, що задовольняють правилам *R* визначає область допустимих рішень початкового завдання по виділенню об'єкта з властивостями $v^* \in \{v\}$, що нас цікавлять. В якості вихідної вершини в графі $D$ будемо використовувати фіктивну вершину *S*, яку в деяких випадках зручно ототожнювати з нульовим або початковим станом системи, це призводить до того, що максимальний ранг шляху в графі $D$ стає рівним *n*, а додавання вершини *S* до базових елементів системи не змінює їх властивостей, визначених правилами *R*.

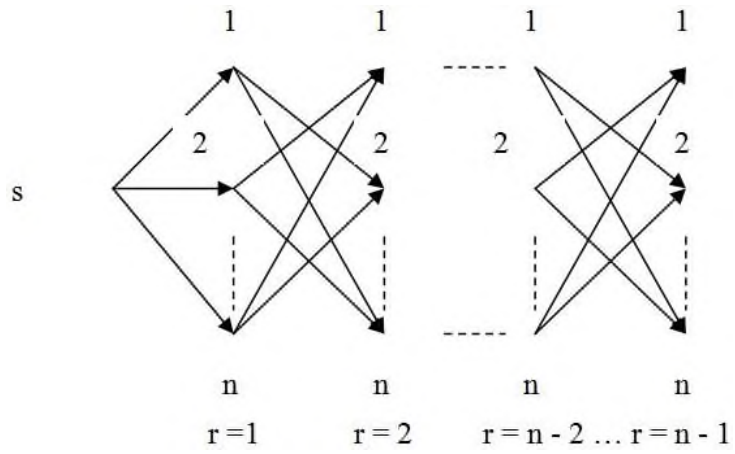



Рис. 2.1. Граф *D*

Безліч складних систем, що володіють різними властивостями $\{F_i\}$, може бути відображено за допомогою деякого підмножини графів $\{G_i\}$. Розглянемо довільний *n* вершинний граф $G(V,E) \in \{G_i\}$, який описує стан системи $F \in \{F_i\}$ з кінцевим числом станів *n*. Вершини $\{i\} \in V$ графа $G(V,E)$ відповідають можливим станам системи. Шляхи в графі $G(V,E)$ визначаються послідовністю проходження вершин $\{v_i\}$ і ребер $\{(i,j)\}$, характеризують можливий порядок досягнення стану $i = p$ з деякого вихідного стану *s*. У графі $G(V,E)$ максимальне значення рангу $r = n-1$ і в загальному випадку ранг довільного шляху $\mu_{sp}$ характеризує суму початкового, кінцевого станів і числа станів передування, через які може бути досягнуто стан *p*, з деякого вихідного стану *s*. Тоді безлічі шляхів $m_{sj}^r$; $j = \overline{(1,n)}$ визначають способи досягнення стану *j*. Поставимо у відповідність кожному базовому елементу з безлічі $\{l_i\}$ вершини графа $G(V,E)$. Тоді об'єктам $\{L_j\}$ будуть відповідати всі множини об'єктів $\Omega$, які можна породити на безлічі *V*, використовуючи правила *R* та *P*. Наприклад, кліки в довільних графах, незалежні множини, цикли графів, вектора і матриці, що

задають певний об'єкт в $G(V,E)$. Кожен об'єкт будемо характеризувати $m+1$ ваговою характеристикою, де $m$ - це деякі другорядні характеристики об'єкта, на які в загальному випадку можуть бути накладені обмеження на те, що вони не повинні перевищувати деяких величин $\{b_i\} i=(1,2,...,m)$ і є один визначальний показник якості об'єкта, побудований з вихідних елементів або їх підмножин, що належить множині $\Omega$ і володіє певною властивістю $v$.

Таким чином, шляху $\mu_{sj}^r$ в графі $D$ відповідає об'єкт $L_j$, який може бути побудований з $r$ базових елементів відповідних вершин $\{v_j\}$, включаючи елемент $j$ на основі правил $R$. А множини шляхів $m_{sj}^r$; $j=(\overline{1,n})$ визначають множину об'єктів $L_j$, які можна побудувати з $r$ базових елементів відповідних вершин $\{v_j\}$, включаючи елемент $j$.

### 2.3.2 Формалізація і постановка задачі нелінійного булевого програмування на основі рангового підходу

Для опису всієї множини адитивних цілочисельних функцій з довільними нелінійностями, що визначаються на множині змінних $\{X_1, X_2...X_n\}$ введемо породжувальну функцію $F(x)$, яка дорівнює:

$$F(x)=\sum_{k=1}^{n}\sum_{j=1}^{p_k} C_{k,j}S_{k,j}\ , \qquad (2.18)$$

де $S_{k,j}$ - добуток $k$ різних змінних; $C_{k,j}$ - цілочисельні коефіцієнти, що стоять при $S_{k,j}$, $p_k=C_n^k=\frac{n!}{k!(n-k!)}(k=1\div n)$. З її допомогою можна задавати і

функціонал і обмеження в задачах нелінійного булевого програмування. Позначимо через $H$ множину функцій з різних доданків в (2.18), які можна отримати вважаючи рівними нулю різні $C_{k,j}$ в (2.18). Множина $H$ є повною в тому сенсі, що містить в собі всі можливі нелінійності, що складаються з усіх можливих поєднань змінних, що утворюють ці нелінійності, які можна взагалі побудувати на основі даної підмножини змінних $\{X_1, X_2 ... X_n\}$. Неважко показати, що потужність даної множини дуже велика, але кінцева і дорівнює $2^{p_\Sigma}$, де

$$p_\Sigma = 1 + \frac{1}{2}\left[\begin{array}{l} \dfrac{n!}{1!(n-1)!}\left(\dfrac{n!}{1!(n-1)!}+1\right) + \dfrac{n!}{2!(n-2)!}\left(\dfrac{n!}{2!(n-2)!}+1\right) + \\ + \dfrac{n!}{k!(n-k)!}\left(\dfrac{n!}{k!(n-k)!}+1\right) + \dfrac{n!}{(n-1)!1!}\left(\dfrac{n!}{(n-1)!1!}+1\right) \end{array}\right].$$

Слід зазначити, що за допомогою співвідношення (2.18) може бути визначено клас задач нелінійного булевого програмування з позитивними цілочисельними коефіцієнтами, в яких рішення визначається тільки поєднанням змінних і не залежить від перестановки змінних в $S_{k,j}$. У загальному випадку завдання булевого програмування можна представити у вигляді:

$$\begin{array}{l} f(X_1, X_2, ..., X_n) = \sum\limits_{k=1}^{n} \sum\limits_{j=1}^{p_k} C_{k,j} S_{k,j} \Rightarrow \max, \\ \textit{при обмеженнях} \\ g_l(X_1, X_2, ..., X_n) = \sum\limits_{k=1}^{n} \sum\limits_{j=1}^{p_k} T_{k,j} S_{k,j} \le b_l; \quad l = (\overline{1,m}), \end{array} \qquad (2.19)$$

де $C_{k,j}; T_{k.j}; b_l$ - позитивні цілі числа.

Розглянемо повнозв'язний граф $G(X,E)$, в якому вершинами графа є змінні $X_i$. Виділимо в графі $G$ довільну кліку $Q = X_p X_r ... X_m$, що складається з $r$ вершин, де $r < n$, і розглянемо включення $Q \subseteq S_{k,j} \in f(X_1, X_2, ..., X_n)$, а також $Q \subseteq S_{k,j} \in g(X_1, X_2, ..., X_n)$. Кожне включення можна охарактеризувати сумами коефіцієнтів $C_{k,j}$, що стоять при $S_{k,j}$ в функціоналі $f(X_1, X_2, ..., X_n)$ і сумами коефіцієнтів $T_{k,j}$, що стоять при $S_{k,j}$ в обмеженнях $g_j(X_1, X_2, ..., X_n)$, при цьому в загальному випадку довільна кліка $Q$ завжди буде характеризуватися відповідною вагою по функціоналу $f(X_1, X_2, ..., X_n)$ і не більше ніж $m$ вагами по обмеженням.

## 2.4 Вирішення задачі нелінійного булевого програмування на основі рангового підходу

Для вирішення завдання скористаємося поданням вихідного графа $G(V, E)$ у вигляді симетричного дерева шляхів $D$ (стягнених дерева шляхів) запропонованого в роботах [69-86]. Сенс такого уявлення полягає в наступному. Нехай всі можливі стану деякої системи визначаються графом $G(V, E)$ з $n$ вершинами, де вершини відповідають можливим станам системи. Перейдемо до простору з $(n - 1)^2$ станами. Для цього кожному з $n$ станів поставимо у відповідність ще $(n - 1)$ стан, що характеризує спосіб досягнення стану з множини $\{1, 2, ..., n\}$. При цьому в якості станів, що додаються, визначимо ранг шляху в графі $G(V, E)$, тобто число ребер, що утворюють даний шлях. Тобто з вершини s графа $G(V, E)$ в довільну вершину $j$ можна потрапити шляхом рангу $r = 1$, використовуючи одне ребро, шляхом рангу $r = 2$, використовуючи 2 ребра тощо. Шляхом рангу $r = n - 1$, використовуючи $n - 1$ ребро. Такий простір станів можна представити у вигляді стягнутого

дерева шляхів *D*, графічно воно може бути зображено як на рисунку 2.1. Дерево всіх шляхів *D* містить (*n* - 1) горизонтальну лінійку і (*n* - 1) ярус. Для прочитання шляхів на кожній горизонтальній лінійці можна бувати тільки один раз. Виходячи зі стягненутого дерева шляхів, для довільної вершини *j* множина шляхів, що ведуть в цю вершину з деякою вершини, можна представити в наступному вигляді:

$$m_s(j) = m_{sj}^{r=1} \cup m_{sj}^{r=2} \cup .... \cup m_{sj}^{r=n-1};\ j = (\overline{1, n-1}), \qquad (2.20)$$

де $m_{sj}^r = \{\mu_{sj}^r\}$ - підмножини шляхів з довільної вершини в деяку вершину графа $G(V,E)$, рангу $r$.

Слід зазначити, що дерево всіх шляхів може бути побудовано і від конкретної вершини *i* графа, в цьому випадку вершина *s* = *i* і при цьому *i* -та горизонтальна лінійка виключається з *D*. Наприклад, при *i* = 2 дерево *D* матиме такий вигляд, як на рисунку 2.2.

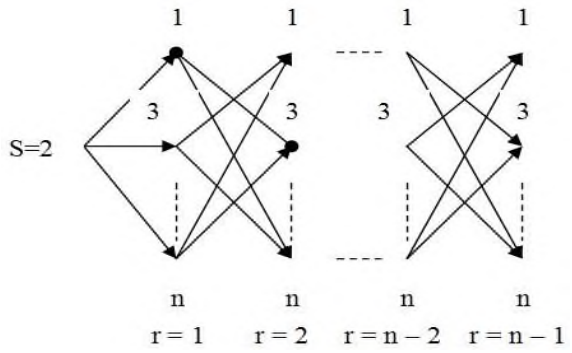



Рис. 2.2. Стягнуте дерево шляхів графа *D* з вершини *S* = 2

Надалі стягнуте дерево шляхів буде використовуватися для побудови однопрохідних алгоритмів розв'язання задачі (2.19), а дерево (рисунок 2.2) для побудови *n*-прохідних алгоритмів, суть побудови яких буде пояснена нижче.

Таким чином, використовуючи граф $D$ і ввівши правила формування шляхів наступного рангу, ми можемо з довільної вершини $s$ поетапно будувати шляхи $\{\mu_{sj}^r\}$ довільного рангу аж до рангу $r=n-1$. У нашій задачі під станом системи ми будемо мати на увазі різні способи об'єднання вершин графа $D$ в кліки. Тоді кожному шляху $\{\mu_{sj}^r\}$ рангу $r$ в графі $D$, що проходить через вершини $\left(v_h, v_k ... v_p\right)$, в вихідному графі $G(V,E)$ розв'язуваної задачі, відповідає кліка з $r=1$ вершини $\left(X_h X_k ... X_p\right)$, що характеризується відповідною вагою по функціоналу $f(X_1, X_2, ..., X_n)$ і не більше ніж $m$ вагами з обмежень $g_l(X_1, X_2, ..., X_n); l=(\overline{1,m})$. Вагові характеристики $\left\{d_{sj}^{r-1}\right\}$ довільної кліки $Q^{r-1}\left(j\right)$, що складається з $r-1$ - вершини і яка визначається одним із шляхів $\mu_{sj}^r \in m_{sj}^r$ рангу $r$ в графі $D$ обчислюються за вагами функціоналу шляхом підсумовування коефіцієнтів підмножини $L_f=\{C_{k,j}\}$, що стоять при доданках $S_{k,j}$ в функціоналі, тобто б відповідала умовам $Q \subseteq S_{k,j} \in f(X_1, X_2, ..., X_n)$. Аналогічно визначаються вагові характеристики по вагах обмежень шляхом підсумовування коефіцієнтів підмножини $L_B=\{T_{k,j}\}$, що стоять при доданках $S_{k,j}$ в обмеженнях, тобто б відповідала умовам $Q \subseteq S_{k,j} \in g(X_1, X_2, ..., X_n)$. Таким чином, вагові характеристики клік $Q^{r-1}\left(j\right)$ характеризуються великою кількістю шляхів $m_{sj}^r$ за вагами функціоналу та обмежень визначаються відповідно виразами:

$$d_{sj}^{f_{r-1}} = \sum_{C_{k,j} \in L_f} C_{k,j} \; ; \; d_{sj}^{B_{r-1}} = \sum_{T_{k,j} \in L_B} T_{k,j} \; . \qquad (2.21)$$

Отже, в графі $D$ кожен шлях має в загальному випадку $m+1$ довжину, одну за вагами функціоналу і $m$ за вагами обмежень і для вирішення поставленого завдання нам в графі $D$ потрібно побудувати шлях максимальної довжини за вагами функціоналу від вершин $1,2,...,n$ до решти

вершин графа і при цьому його довжини по вагам обмежень не повинні перевищувати відповідної величини $b_j$. Якщо на основі підмножин шляхів $m_{sj}^{r=1}$ в графі $D$ будувати підмножини $m_{sj}^{r=2}$ і так далі до шляхів $m_{sj}^{r=n-1}$ рангу $r = n - 1$, то ми змушені будемо побудувати $(n-1)!$ шляхів, тому для формування шляхів вводиться процедура $A$, що дозволяє відсікати неперспективні шляхи. Для відсікання неперспективних варіантів в процедурі $A$ пропонується використовувати принцип оптимізації у напрямку до довільної вершини $p$, при формуванні шляхів наступного рангу $m_{sp}^{r+1}$ на основі шляхів попереднього рангу $m_{sj}^{r}$, запропонованих в роботах [61,75,89], який для даної задачі визначається наступним рекурентним співвідношенням:

$$\mu_{sp}^{r+1} = \max_{j}\{\{\mu_{sj}^{r}\} \cup (j, p)\}; j = (\overline{1,n}); p = (\overline{1,n}); j \neq p, \qquad (2.22)$$

де $(j, p)$ - ребро графа $D$; $n$ - число різних вершин в графі $D$.

Розглянемо можливість побудови $n$ - прохідних і однопрохідних процедур вирішення завдання (2.19) відповідно на стягнутих деревах, наведених на рисунках 2.1 і 2.2.

Процедура $A_1$ з $n$ проходами. Крок 1. Змінній $s := i$ і з вершини $s$ будуються всі можливі шляхи рангу $r = 1$ до всіх вершин графа $D$ (рисунок 2.2), що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = (\overline{1,m}),$ при цьому довжини за вагами функціоналу та обмеженнями обчислюються відповідно до співвідношеннями (2.21).

Крок 2. Використовуючи шляхи поточного рангу $r$, будуються всі можливі шляхи рангу $r := r + 1,$ що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = (\overline{1,m}),$ з використанням рекурентного співвідношення (2.21). При цьому перевірка обмежень і вибір шляху максимального за

вагами функціоналу здійснюється на основі обчислень довжин шляхів за вагами функціоналу та обмежень відповідно до співвідношеннями (2.21). Слід зазначити, що якщо в процесі застосування рекурентного співвідношення (2.22) виникають кілька шляхів однакової довжини, то необхідно все продовжувати на наступному ранзі.

Крок 3. Перевіряємо $m_{sj}^{r+1} = \varnothing$. Якщо так, то шлях $\mu_{sj}^{*r}$ максимальної довжини, отриманий на ранзі $r$, є локальним екстремумом розв'язуваної задачі, інакше виконуємо наступний крок.

Крок 4. Перевіряємо $i := n-1$. Якщо ні, то $i := i+1$ і переходимо до виконання кроку 1, інакше процедура $A_1$ закінчує роботу, при цьому з множини локальних екстремумів $\left\{\mu_{sj}^{*r}\right\}$ вибирається глобальний екстремум $\mu_{sj}^{**r}$, що відповідає оптимальному вирішенню завдання (2.19).

Для зниження часової складності роботи алгоритму можливо використовувати однопрохідний варіант реалізації даної процедури на основі стягнених древа шляхів, наведеного на рисунку 2.1. При цьому однопрохідна процедура $A_2$ має вигляд, описаний нижче.

Процедура $A_2$ з 1 проходом. Крок 1. З вершини $s$ будуються всі можливі шляхи рангу $r=1$ до всіх вершин графа $D$, показаного на рисунку 2.2, що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1,m)}$, при цьому довжини за вагами функціоналу і обмеженням обчислюються відповідно до співвідношення (2.21).

Крок 2. Використовуючи шляху поточного рангу $r$, будуються всі можливі шляхи рангу $r := r+1$, що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1,m)}$, з використанням рекурентного співвідношення (2.22). При цьому перевірка обмежень і вибір шляху максимального за вагами функціоналу здійснюється на основі обчислень довжин шляхів за вагами функціоналу та обмежень відповідно до співвідношення (2.21). Слід зазначити, що якщо в процесі застосування рекурентного співвідношення

(2.22) виникають кілька шляхів однакової довжини, то необхідно їх все продовжувати на наступному ранзі.

Крок 3. Перевіряємо $m_{sj}^{r+1} = \varnothing$. Якщо так, то шлях $\mu_{sj}^{*r}$ максимальної довжини, отриманий на ранзі $r$, є локальним екстремумом розв'язуваної задачі, інакше виконуємо наступний крок.

Крок 4. Перевіряємо рангу $r = n$. Якщо ні, то переходимо до виконання кроку 2, інакше процедура $A_2$ закінчує роботу і шлях $\mu_{sj}^{*r}$ максимальної довжини, отриманий на ранзі $r = n$, відповідає оптимальному вирішенню завдання (2.19).

Ще одним варіантом зменшення часової складності алгоритмів на основі процедур $A_1$ і $A_2$ можуть бути процедури $A'$ і $A''$, що відрізняються від $A_1$ і $A_2$ тим, що на кожному ярусі формування шляхів на основі процедур $A_1$ і $A_2$ локальні екстремуми будуть виділятися не в кожній множині. У цьому випадку виділяються глобальні екстремуми на ярусі, і на основі шляху, відповідні глобального екстремуму на ярусі формуються шляхи наступного ярусу, що задовольняють обмеженням, при цьому процедури $A'$ і $A''$ будуть мати вигляд, наведений нижче.

Процедура $A'$. Перед початком роботи процедури $A'$ змінної $i := 1$.

Крок 1. Змінній $s := i$ і з вершини $s$ будуються всі можливі шляхи рангу $r = 1$ до всіх вершин графа $D$, як показано вище на рисунку 2.2, що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1, m)}$, при цьому довжини за вагами функціоналу і обмеженням обчислюються відповідно до співвідношення (2.21). Далі виділяється найдовший шлях на ярусі.

Крок 2. Використовуючи найдовший шлях поточного рангу $r$, побудований на попередньому кроці, будуються всі можливі шляхи рангу $r := r + 1$, що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1, m)}$, з використанням рекурентного співвідношення (2.22). При цьому перевірка обмежень і вибір шляху максимального за вагами функціоналу здійснюється

на основі обчислень довжин шляхів за вагами функціоналу та обмежень відповідно до співвідношення (2.21).

Крок 3. Перевіряємо $m_{sj}^{r+1} = \varnothing$. Якщо так, то шлях $\mu_{sj}^{*r}$ максимальної довжини, отриманий на ранзі $r$, є локальним екстремумом розв'язуваної задачі, інакше виконуємо наступний крок.

Крок 4. Перевіряємо $i := n-1$. Якщо ні, то $i := i+1$ і переходимо до виконання кроку 1, інакше процедура $A'$ закінчує роботу, при цьому з множини локальних екстремумів, отриманих за один прохід, $\{\mu_{sj}^{*r}\}$ вибирається глобальний, що відповідає оптимальному вирішенню завдання (2.19).

Процедура $A''$. Крок 1. З вершини $s$ будуються всі можливі шляхи рангу $r=1$ до всіх вершин графа $D$, як показано на рисунку 2.1, що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1,m)},$ при цьому довжини за вагами функціоналу і обмеженням обчислюються відповідно до співвідношеннями (2.21). Далі виділяється найдовший шлях на ярусі.

Крок 2. Використовуючи найдовший шлях поточного рангу $r$, побудований на попередньому кроці, будуються всі можливі шляхи рангу $r := r+1,$ що задовольняють обмеженням $g_l(X_1, X_2, ..., X_n); l = \overline{(1,m)},$ з використанням рекурентного співвідношення (24). При цьому перевірка обмежень і вибір шляху максимального за вагами функціоналу здійснюється на основі обчислень довжин шляхів за вагами функціоналу та обмежень відповідно до співвідношення (2.21).

Крок 3. Перевіряємо $m_{sj}^{r+1} = \varnothing$. Якщо так, то шлях $\mu_{sj}^{*r}$ максимальної довжини, отриманий на ранзі $r$, є локальним екстремумів розв'язуваної задачі, інакше виконуємо наступний крок.

Крок 4. Перевіряємо рангу $r = n$. Якщо ні, то переходимо до виконання кроку 2, інакше процедура $A''$ закінчує роботу, і шлях $\mu_{sj}^{*r}$ максимальної

довжини, отриманий на ранзі $r = n$, відповідає оптимальному вирішенню завдання (2.19).

Надалі позначимо через $A_5$ алгоритм, реалізований на основі багатопрохідної процедури, через $A_4$ - алгоритм, реалізований на основі однопрохідної процедури. Через $A_2$ і через $A_3$ - алгоритм, реалізований на основі однопрохідної процедури $A''$, який охоплює всі способи виділення локальних екстремумів.

## 2.5 Оцінка часової складності розроблених процедур $\{A\}$

При роботі процедур $\{A\}$ число шляхів, що будуються на довільному ранзі $r$, не може перевищити $(n-1)(n-1)$. Максимальний ранг $r$ довільного шляху не перевищує $(n-1)$ і число циклів, що виконується процедурою $A_1$ дорівнює $n$. Отже, після $n$ циклів число шляхів, яке побудує процедура $A_1$, не може перевершити $(n-1)(n-1)(n-1)n \approx n^4$, а число оброблених векторів - $n^5$. З урахуванням числа доданків $(k)$ в функціоналі і числа обмежень $(m)$ часова складність алгоритму не перевищить в гіршому випадку $O\left(n^5 k(m+1)\right)$. У разі, коли рішення задачі здійснюється за один прохід процедури $A_2$ або за один прохід процедури $A''$, але з виділенням найбільш довгого шляху на ярусі, складність процедур $A_2$ і $A''$ не перевищить відповідно $O\left(n^4 k(m+1)\right)$ і $O\left(n^3 k(m+1)\right)$. Отже, алгоритми $A_5$, $A_4$, $A_3$ мають відповідно часову складність, що не перевищує в гіршому випадку $O\left(n^5 k(m+1)\right)$, $O\left(n^4 k(m+1)\right)$ і $O\left(n^3 k(m+1)\right)$. Таким чином, запропонований підхід вирішення довільних завдань булевого програмування дозволяє на основі запропонованих алгоритмів вирішувати з єдиних позицій будь-які

завдання лінійного та нелінійного булевого програмування за поліноміальний час з необхідною точністю.

## 2.6 Експериментальне дослідження розроблених процедур

В якості основних характеристик наближених алгоритмів використовуємо кількість елементарних операцій і кількість оброблених векторів, функцію часових витрат і відносну $П$ похибку отриманого наближеного рішення, яка визначається зі співвідношення:

$$П = \frac{|f(x) - f(x^*)|}{f(x^*)}, \tag{2.23}$$

де $f$ - цільова функція, визначена на деякій множині $M$; $x$ - наближене вирішення задачі; $x^*$ - оптимальне вирішення.

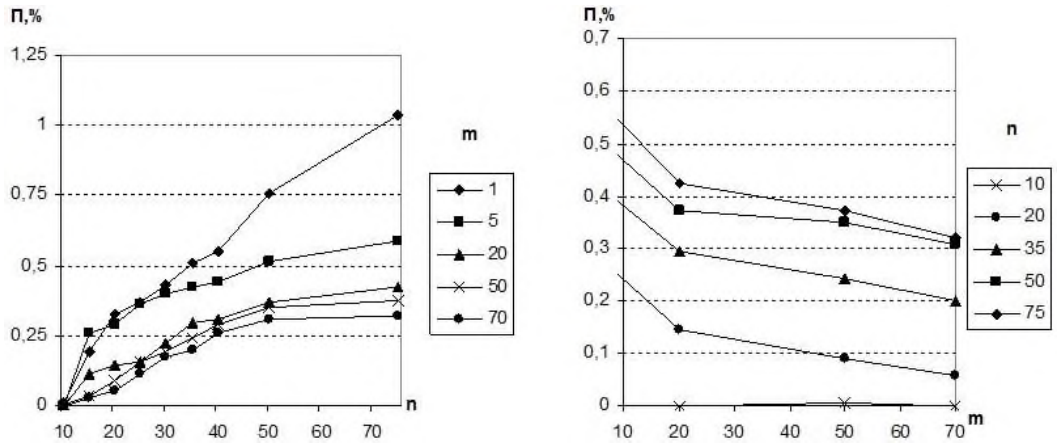
Досліджувалися наступні алгоритми: алгоритм $A_5$ на основі багатопрохідної процедури $A_1$; алгоритм $A_4$ на основі однопрохідної процедури $A_2$ і алгоритм $A_3$ на основі однопрохідної процедури $A''$. При дослідженні коефіцієнти в функціоналі генерувалися по рівномірному закону розподілу в функціоналі в діапазоні від 0 до 10, а в обмеженнях - від 0 до 20. На кожну точку при оцінці часової складності алгоритмів в середньому і похибки алгоритмів вирішувалося не менше 50 тестових завдань, результати отримані з довірливою ймовірністю 0,95. В якості точного алгоритму використовувався розроблений автором алгоритм для задач нелінійного та лінійного програмування, скомбінований на основі використання для прогнозування перспективних рішень ідей рангового підходу, а для відсіву неперспективних шляхів методу гілок і меж, що дозволив зняти похибки для

задач до розмірності, що не перевищує $n = 70$. Графіки залежності похибки від розмірності $(n)$ вирішуваних завдань і від числа обмежень $(m)$ в задачі (2.19) наведені на рисунках 2.3 – 2.6, з яких видно, що похибка алгоритмів зі збільшенням числа обмежень $m$ асимптотично зменшується, зі збільшенням $n$ зростає і $m \geq 50$ , похибка алгоритмів стабілізується і для задач лінійного програмування не перевищує 2%, а для задач квадратичного - 5-10%. Вирішення тестових завдань показало, що збільшення діапазону зміни коефіцієнтів у функціоналі і обмеженнях призводить до різкого зниження похибок алгоритмів, перехід від нелінійностей одного порядку до нелінійностей більш високого порядку може призводити до незначного збільшення похибок при невеликому числі обмежень, а зі зростанням числа обмежень зростання похибки дуже швидко компенсується. Експериментальне дослідження тимчасової складності показало, рисунок 2.3, що число оброблюваних векторів не залежить від числа обмежень і в середньому для алгоритмів $A_5$, $A_4$, $A_3$ тимчасова складність є величиною порядку відповідно $O\left(0{,}1n^{4{,}9}\right)$, $O\left(0{,}3n^{3{,}7}\right)$ і $O\left(0{,}4n^{2{,}8}\right)$.

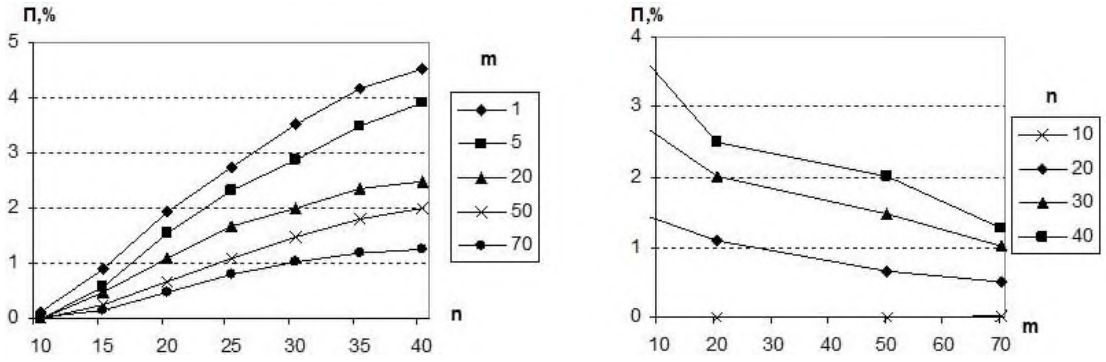
Приклад вирішення задачі.

$$\mathrm{f}(x) = 20\mathrm{x}_1x_2 + 13\mathrm{x}_1x_4 + 17\mathrm{x}_3x_4 + 20\mathrm{x}_3 + 10\mathrm{x}_4 \to \max \qquad (2.24)$$

$$2\mathrm{x}_1x_2 + 10x_3\mathrm{x}_4 + 7\mathrm{x}_1x_4 + 4\mathrm{x}_4 + 3\mathrm{x}_2x_4 \leq 10$$

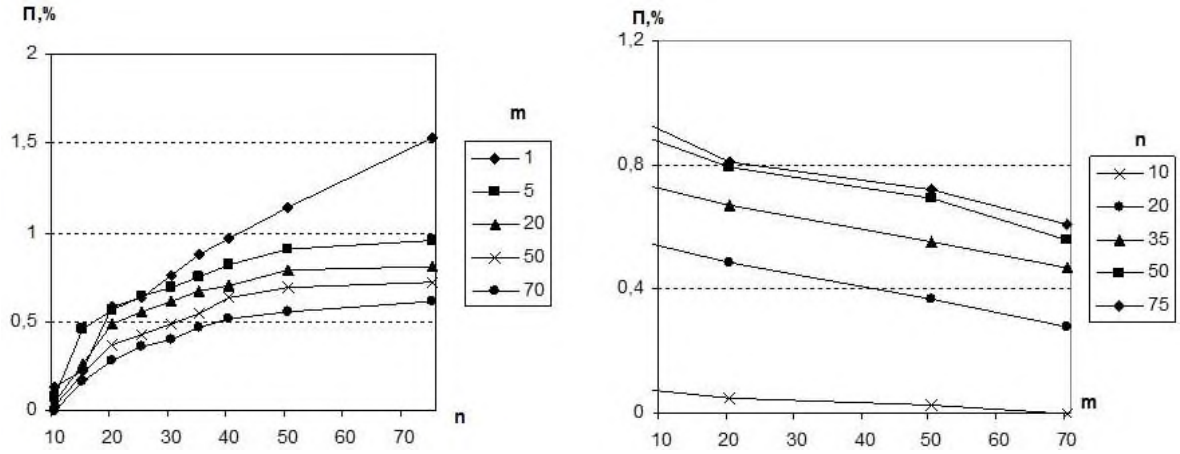



а)

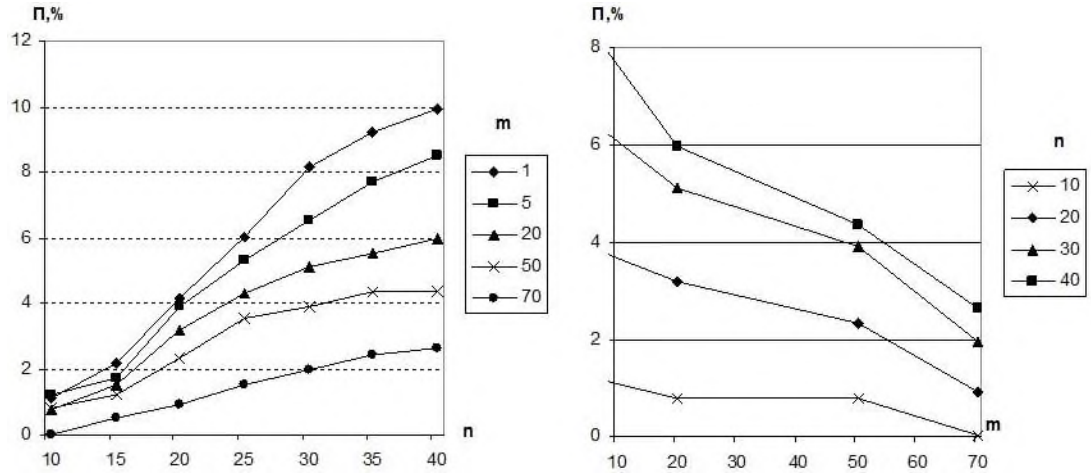



б)

Рис. 2.3. а) Залежність похибки алгоритму $A_5$ від розмірності розв'язуваної задачі лінійного булевого програмування $(n)$ при різній кількості обмежень $(m)$; б) Залежність похибки алгоритму $A_5$ від розмірності розв'язуваної задачі квадратичного булевого програмування $(n)$ при різній кількості обмежень $(m)$

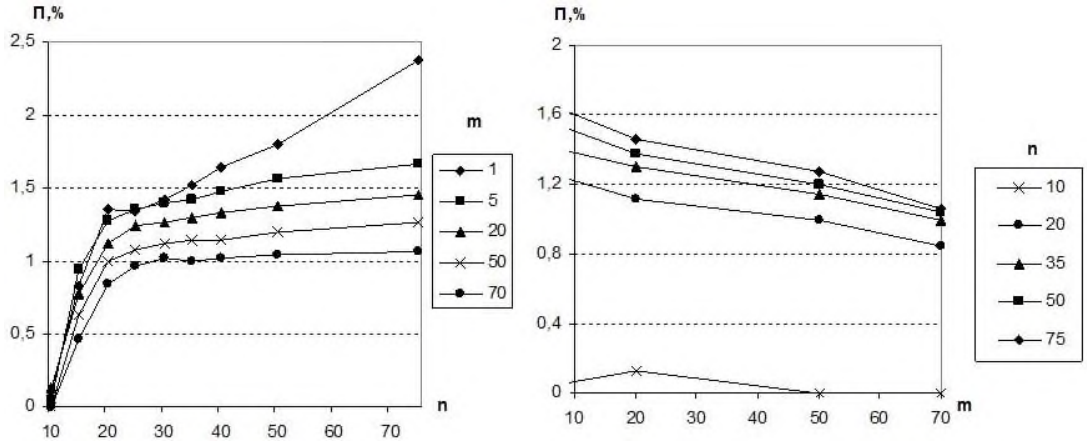



а)

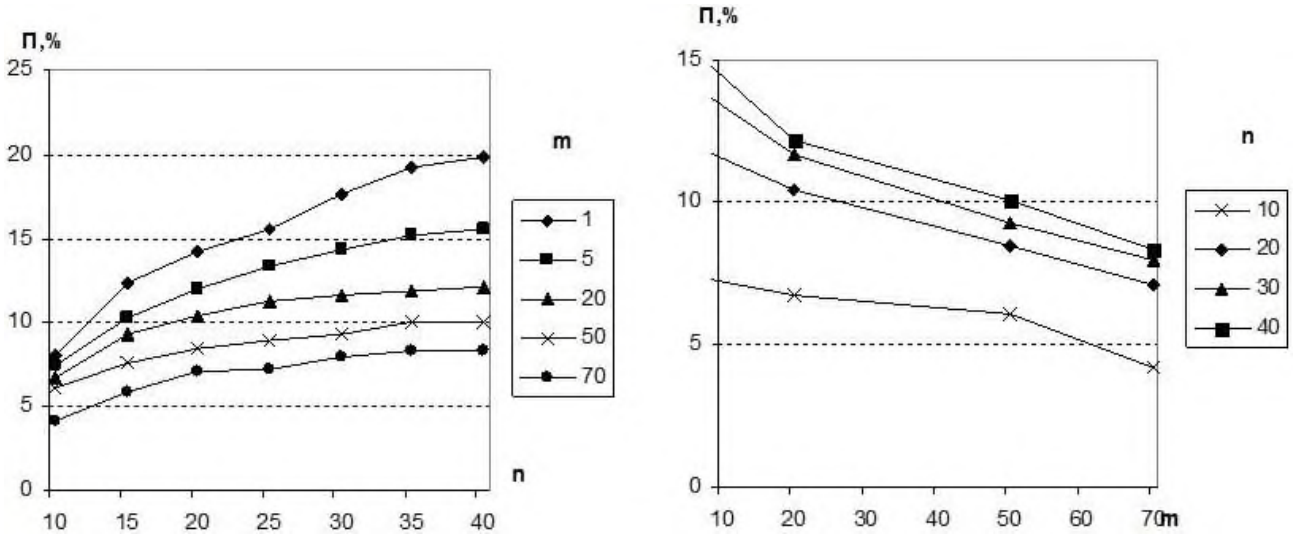



б)

Рис. 2.4. а) Залежність похибки алгоритму $A_4$ від розмірності розв'язуваної задачі лінійного булевого програмування $(n)$ при різній кількості обмежень $(m)$;б) Залежність похибки алгоритму $A_4$ від розмірності розв'язуваної задачі квадратичного булевого програмування $(n)$ при різній кількості обмежень $(m)$




а)




б)

Рис. 2.5. а) Залежність похибки алгоритму $A_3$ від розмірності розв'язуваної задачі лінійного булевого програмування $(n)$ при різній кількості обмежень $(m)$; б) Залежність похибки алгоритму $A_3$ від розмірності розв'язуваної задачі квадратичного булевого програмування $(n)$ при різній кількості обмежень $(m)$

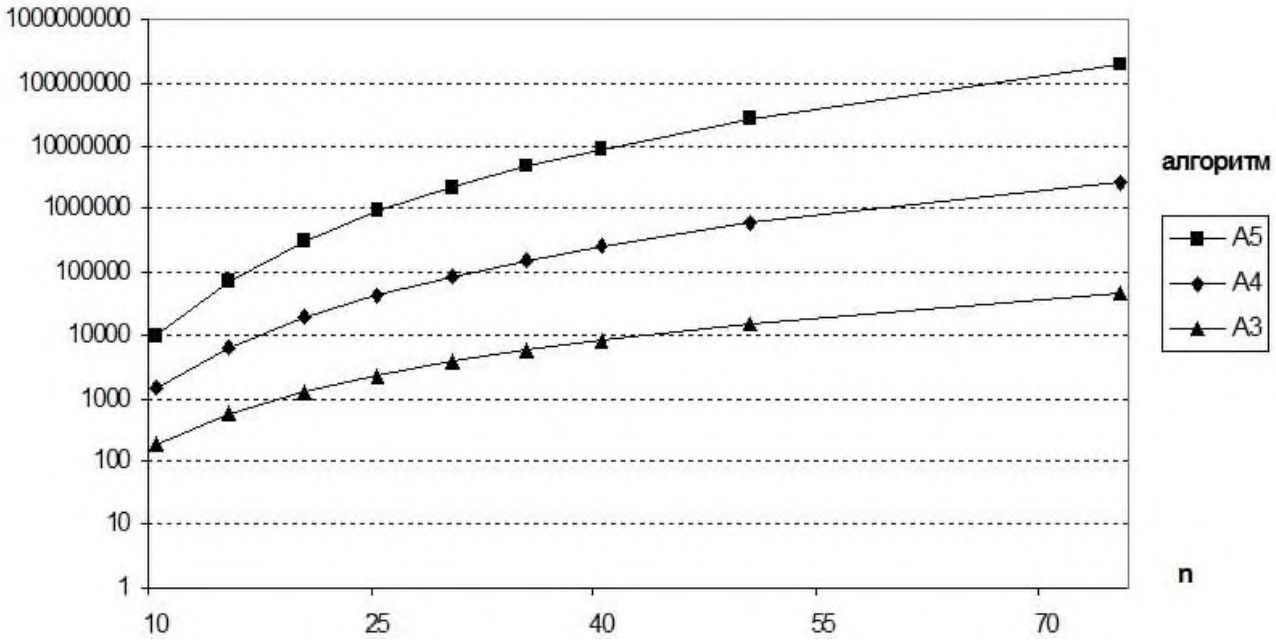



Рис. 2.6. Залежність числа оброблюваних векторів від розмірності розв'язуваної задачі лінійного і квадратичного програмування для алгоритмів $A_5,\ A_4,\ A_3$

Простір можливих рішень задачі представимо у вигляді графа *D*. Процес формування шляхів наведено в таблиці 2.1, де кожній вершині графа *D* відповідає клітка з номером *i*.

Формуємо спочатку всі можливі шляхи з вершини $S$ до вершин 1, 2, 3, 4 - це шляхи $S1,\ S2,\ S3,\ S4$. Кожен шлях характеризується тут двома довжинами. Довжиною по вагах коефіцієнтів у функціоналі (перший в дужках) і довжиною по вагах коефіцієнтів в обмеженні (другий в дужках).

Визначення довжин для шляху $S1$. Перевіряємо, в функціоналі змінна $x_1$ окремо присутня чи ні. Вона не присутня окремо ні в функціоналі, ні в обмеженнях, тому довжини в шляху $S1(0,0)$. Аналогічна ситуація з шляхом $S2(0,0)$. Для шляху $S3$ дивимося, в функціоналі змінна $x_3$ окремо присутня? Так присутня, при ній стоїть коефіцієнт 20, це означає, що довжина шляху по функціоналу дорівнює 20.

Таблиця 2.1

Процес формування шляхів графу

| | | | |
|---|---|---|---|
| S1(0,0)<br>1 | S21(20,2)<br>S31(20,0) 1 | S321(40,2)<br>S231(40,2) 1 | 1 |
| S2(0,0)<br>2 | S12(20,2)<br>S32(20,0)<br>S42(10,7) 2 | S312(40,2)<br>S132(40,2)<br>2 | 2 |
| S3(20,0)<br>3 | S13(20,0)<br>S23(20,0) 3 | S213(40,2)<br>S123(40,2) 3 | 3 |
| S4(10,4) 4 | S24(10,7) 4 | 4 | 4 |

В обмеженні окремо змінна $x_3$ не присутня, отже, довжина шляху по вагам обмежень дорівнює 0, тобто шлях $S3$ має дві довжини (20,0). Для шляху $S4$ перевіряємо, чи окремо присутня змінна $x_4$ в функціоналі? Так присутня, при ній стоїть коефіцієнт 10, отже, довжина шляху по функціоналу дорівнює 10. Перевіряємо, в обмеженні змінна $x_4$ окремо присутня? Так, при ній стоїть коефіцієнт 4, отже, довжина шляху по обмеженню дорівнює 4, тобто шлях $S4$ має дві довжини (10,4). Далі на основі сформованих шляхів $S1$, $S2$, $S3$, $S4$ формуємо всі можливі шляхи наступного рангу до вершин 1, 2, 3, 4, довжини яких за вагами обмежень не перевищують 10. Так з вершини 1 ми можемо потрапити в вершини 2, 3 і 4.

Формуємо шлях $S12$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_1$ і $x_2$ окремо або спільно присутні чи ні? Так присутні, при добутку $x_1 x_2$ стоїть коефіцієнт 20, отже, шлях $S12$ по вагах функціоналу має вагу рівну 20. Перевіряємо в обмеженнях змінна $x_1$ і $x_2$ окремо або спільно присутні чи ні? Так, в обмеженнях змінні $x_1$ і $x_2$ присутні у вигляді добутку $x_1 x_2$, при якому стоїть коефіцієнт 2, отже, шлях $S12$ по вагах обмежень має вагу рівний 4, тобто шлях $S12$ має дві довжини (20,2).

Формуємо шлях $S13$ і визначаємо його довжини. Перевіряємо, в функціоналі змінні $x_1$ і $x_3$ окремо або спільно присутні чи ні? Так, при змінній $x_3$ стоїть коефіцієнт 20, а спільно змінні $x_1$ і $x_3$ в функціоналі не присутні, значить, шлях $S13$ по вагах функціоналу має вагу рівну 20. Перевіряємо в обмеженнях змінні $x_1$ і $x_3$ окремо або спільно присутні чи ні? Ні, отже, довжина шляху по вагах обмежень дорівнює 0, тобто шлях $S13$ має дві довжини (20,0).

Формуємо шлях $S14$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_1$ і $x_4$ окремо або спільно присутні чи ні? Так, при змінної $x_4$, що стоїть окремо, присутній коефіцієнт 10, а при добутку $x_1 x_4$ в функціоналі стоїть коефіцієнт 13, значить, шлях $S14$ по вагах функціоналу має вагу рівний 10 + 13 = 23. Перевіряємо в обмеженнях змінні $x_1$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$ стоїть коефіцієнт 4, а при добутку $x_1 x_4$ в обмеженнях стоїть коефіцієнт 7. Отже, шлях $S14$ по вагах має вагу рівний 7 + 4 = 11. Отже, довжина шляху $S14$ по вагах обмежень дорівнює 11, тобто шлях має дві довжини (23,11), але оскільки довжина по вагах обмежень перевищує праву частину обмеження, то цей шлях виключаємо з подальшого аналізу. Тому в множину 4 на другому ярусі цей шлях не потрапив.

Далі формуємо всі можливі шляхи від вершини 2. Так з вершини 2 ми можемо потрапити в вершини 1,3 і 4.

Формуємо шлях $S21$ і визначаємо його довжини. Перевіряємо, в функціоналі змінні $x_1$ і $x_2$ окремо або спільно присутні чи ні? Так, при добутку $x_1 x_2$ стоїть коефіцієнт 20, отже, шлях $S21$ по вагах функціоналу має вагу рівну 20. Перевіряємо, в обмеженнях змінні $x_1$ і $x_2$ окремо або спільно присутні чи ні? Так, в обмеженнях змінні $x_1$ і $x_2$ присутні у вигляді добутку $x_1 x_2$, при якому стоїть коефіцієнт 2, отже, шлях $S21$ по вагах обмежень має вагу рівну 2, тобто шлях $S21$ має дві довжини (20,2).

Формуємо шлях $S23$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_2$ і $x_3$ окремо або спільно присутні чи ні? Так, при змінній $x_3$ стоїть коефіцієнт 20, а спільно змінні $x_2$ і $x_3$ в функціоналі не присутні, отже, шлях $S23$ по вагах функціоналу має вагу рівну 20. Перевіряємо в обмеженнях змінні $x_2$ і $x_3$ окремо або спільно присутні чи ні? Ні, отже, довжина шляху $S23$ по вагах обмежень дорівнює 0, тобто шлях $S23$ має дві довжини (20,0)

Формуємо шлях $S24$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_2$ і $x_4$ окремо або спільно присутні чи ні? Так, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 10, а добутку $x_2 x_4$ в функціоналі немає, значить, шлях $S24$ по вагах функціоналу має вагу рівну 10. Перевіряємо в обмеженнях змінні $x_2$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінної $x_4$, що стоїть окремо, присутній коефіцієнт 4, а при добутку $x_2$ в обмеженнях стоїть коефіцієнт 3, значить, шлях $S24$ по вагах обмежень має вагу рівну 4 + 3 = 7. Отже, довжина шляху $S24$ по вагах обмежень дорівнює 7, тобто шлях $S24$ має дві довжини (10,7).

Далі формуємо всі можливі шляхи від вершини 3. Так з вершини 3 ми можемо потрапити в вершини 1,2 і 4.

Формуємо шлях $S31$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_1$ і $x_3$ окремо або спільно присутні чи ні? Так, при змінній $x_3$ стоїть коефіцієнт 20, а спільно змінні $x_1$ і $x_3$ в функціоналі не присутні, отже, шлях $S31$ по вагах функціоналу має вагу рівну 20. Перевіряємо в обмеженнях змінні $x_1$ і $x_3$ окремо або спільно присутні чи ні? Ні, отже, довжина шляху $S31$ по вагах обмежень дорівнює 0, тобто шлях $S31$ має дві довжини (20,0).

Формуємо шлях $S32$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_2$ і $x_3$ окремо або спільно присутні чи ні? Так, при змінній $x_3$ стоїть коефіцієнт 20, а спільно змінні $x_2$ і $x_3$ в функціоналі не

присутні, отже, шлях $S32$ по вагах функціоналу має вагу рівний 20. Перевіряємо в обмеженнях змінні $x_2$ і $x_3$ окремо або спільно присутні чи ні? Ні, отже, довжина шляху $S32$ по вагах обмежень дорівнює 0, тобто шлях $S32$ має дві довжини (20,0).

Формуємо шлях $S34$ і визначаємо його довжини: перевіряємо в функціоналі змінні $x_3$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 10, при змінній $x_3$, що стоїть окремо, присутній коефіцієнт 20, а при добутку $x_3 x_4$ в функціоналі стоїть коефіцієнт 17. Отже, шлях $S34$ за вагами функціоналу має вагу рівну 10 + 20 + 17 = 47. Перевіряємо в обмеженнях змінні $x_3$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінної $x_4$, що стоїть окремо, присутній коефіцієнт 4, а при добутку $x_3 x_4$ в обмеженнях стоїть коефіцієнт 10, отже, шлях $S34$ по вагах обмежень має вагу рівну 4 + 10 = 14, отже, довжина шляху $S34$ по вагах обмежень дорівнює 14, тобто шлях $S34$ має дві довжини (47,14). Але оскільки довжина по вагах обмежень перевищує праву частину обмеження, то цей шлях виключаємо з подальшого аналізу. Тому в множину 4 на другому ярусі цей шлях не потрапив.

Далі формуємо всі можливі шляхи від вершини 4. Так з вершини 4 ми можемо потрапити в вершини 1,2 і 3.

Формуємо шлях $S41$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_1$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$, що стоїть окрем, присутній коефіцієнт 10, а при добутку $x_1 x_4$ в функціоналі стоїть коефіцієнт 13. Отже, шлях $S41$ по вагах функціоналу має вагу рівний 10 + 13 = 23. Далі перевіряємо в обмеженнях змінні $x_1$ і $x_4$ окремо або спільно присутні чи ні? Так присутні при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 4, а при добутку $x_1 x_4$ в обмеженнях стоїть коефіцієнт 7. Отже, шлях $S41$ по вагах обмежень має вагу рівний 7 + 4 = 11, тобто шлях $S41$ має дві довжини (23,11), але оскільки довжина по вагам

обмежень перевищує праву частину обмеження, то цей шлях виключаємо з подальшого аналізу. Тому в безліч 4 на другому ярусі цей шлях не потрапив.

Формуємо шлях $S42$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_2$ і $x_4$ окремо або спільно присутні чи ні? Так, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 10, а добутку $x_2 x_4$ в функціоналі немає, отже, шлях по вагах функціоналу має вагу, що дорівнює 10. Перевіряємо в обмеженнях змінні $x_2$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 4, а при добутку $x_2 x_4$ в обмеженнях стоїть коефіцієнт 3. Отже, шлях $S42$ по вагах має вагу обмежень, що дорівнює 4 + 3 = 7, тобто шлях $S42$ має дві довжини (10,7).

Формуємо шлях $S43$ і визначаємо його довжини. Перевіряємо в функціоналі змінні $x_3$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 10, при $x_3$, що стоїть окремо, присутній коефіцієнт 20, а добуток $x_3 x_4$ в функціоналі стоїть коефіцієнт 17. Отже, шлях $S43$ по вагах функціоналу має вагу рівний 10 + 20 + 17 = 47. Перевіряємо в обмеженнях змінні $x_3$ і $x_4$ окремо або спільно присутні чи ні? Так присутні, при змінній $x_4$, що стоїть окремо, присутній коефіцієнт 4, а при добутку $x_3 x_4$ в обмеженнях стоїть коефіцієнт 10, отже, шлях по вагах має вагу рівний 4 + 10 = 14. Отже, довжина шляху $S43$ по вагах обмежень дорівнює 14, тобто шлях має дві довжини (47,14), але оскільки довжина по вагах обмежень перевищує праву частину обмеження, то цей шлях виключаємо з подальшого аналізу. Тому в множину 4 на другому ярусі цей шлях не потрапив.

Таким чином, формування шляхів на другому ярусі завершилося, далі на основі шляхів сформованих на другому ярусі будуємо всі можливі шляхи наступного рангу. При цьому будемо використовувати принцип оптимізації по напрямку, який полягає в тому, що з кожної безлічі шляхів $i$ ми формуємо

в усі підмножини шляху, що залишилися, максимально можливої ваги по функціоналу, довжина яких за вагами обмежень не перевищує правої частини обмеження. Почнемо з множини 1, яка містить два шляхи: $S21(20,2)$ і $S31(20,0)$. З множини 1 ми можемо потрапити в множини 2,3 і 4. Формуємо всі можливі шляхи з множини 1 в 2, використовуючи принцип оптимізації за напрямком: Шлях $S21(20,2)$ в множині 2 не може бути продовжений оскільки вершина 2 вже увійшла в шлях $S21(20,2)$, отже нам залишилося вибрати тільки один шлях, що залишився $S31(20,0)$. В результаті отримаємо шлях $S312$, довжина якого визначається наступним чином: перевіряємо в функціоналі змінні $x_1$, $x_2$ й $x_3$, окремо і в комбінаціях $x_1x_2$; $x_1x_3$; $x_2x_3$ або у вигляді співмножника $x_1x_2x_3$ присутні чи ні? Тобто ми дивимося, чи є добутки зі змінних, які входять в співмножник $x_1x_2x_3$, і коефіцієнти при цих співмножників підсумовуємо. Так, є змінна $x_3$ з коефіцієнтом 20, при цьому є співмножник $x_1x_2$ з коефіцієнтом 20, отже, довжина шляху $S312$, по вагах функціоналу, буде дорівнює 20 + 20 = 40. Аналогічним чином визначається довжина шляху по вагах обмежень. В обмеженнях є такі співмножники, що входять до добутку $x_1x_2x_3$, це тільки $x_1x_2$ з коефіцієнтом при ньому рівним 2, отже, довжина шляху $S312$ по вагах обмежень дорівнює 2, тобто даний шлях $S312$ має дві довжини (40,2). Якби в безлічі 1 залишився не один шлях $S31(20,0)$, а кілька шляхів, які можуть пройти по обмеженням у безліч 2, то серед них ми перетягуємо в 2 тільки один з максимальною вагою по функціоналу.

Формуємо всі можливі шляхи виходу з множини 1 в 3, використовуючи принцип оптимізації за напрямком. Шлях $S31(20,0)$ у множину 3 не може бути продовжений, оскільки 3-тя вершина вже увійшла в шлях $S31(20,0)$, отже, нам залишилося вибрати тільки один шлях $S21(20,0)$, що залишився, в результаті отримаємо шлях $S213$, довжина якого визначається наступним

чином. Перевіряємо в функціоналі змінні $x_1$, $x_2$ і $x_3$ окремо і в комбінаціях $x_1x_2$, $x_1x_3$, $x_2x_3$, або у вигляді співмножника $x_1x_2x_3$ присутні чи ні? Тобто ми дивимося, чи є добутки з змінних, які входять в співмножник $x_1x_2x_3$ і коефіцієнти при цих співмножників підсумовуємо. Так є змінна $x_3$ з коефіцієнтом 20 і є співмножник $x_1x_2$ з коефіцієнтом 20. Отже, довжина шляху $S213$, по вагах функціоналу, буде дорівнює 20 + 20 = 40. Аналогічним чином визначається довжина шляху по вагах обмеження, в обмеженнях є такі співмножники, що входять до добутку $x_1x_2x_3$ це тільки $x_1x_2$ з коефіцієнтом при ньому рівним 2, отже, довжина шляху $S213$ по вагах обмежень дорівнює 2, тобто даний шлях $S213$ має дві довжини (40,2).

Далі формуємо всі можливі шляхи виходу з множини 1 в 4, використовуючи принцип оптимізації за напрямком. Легко перевірити, що жоден із шляхів, які можна побудувати в множині 4, не задовольняють обмеженню і їх всі можна виключити з аналізу. За аналогією формуються шляхи з множини 2 у множини 1, 3, 4 і шляхи виходу з множини 3 у множини 1, 2, 4, а також шляхи з множини 4 у множини 1, 2, 3, причому останні не пройдуть по обмеженням ні в одну множину.

На наступному ранзі нам не вдасться побудувати жодного шляху, оскільки всі вони не будуть задовольняти обмеженню. Таким чином, найдовшим шляхом є шлях $S213$. Він має дві довжини (40,2) і, отже, вектор звертає в максимум функціонал (26) рівний 40 має вигляд X=$\{x_1=1,\ x_2=1,\ x_3=1,\ x_4=0\}$, при цьому значення обмеження не перевищує величини 2. Слід зазначити, що шляхи, що містять однакові підмножини вершин мають однакові довжини за вагами функціоналу і обмеженням, тому вони дублюють один одного на ярусі і тому дублюючі шляхи можна на ярусі прибирати, в цьому випадку процес побудови шляхів матиме вигляд, приведений в таблиці 2.2.

Таблиця 2.2

Процес побудови шляхів графу з однаковими підмножинами вершин

| | | | |
|---|---|---|---|
| S1(0,0) 1 | S31(20,0) 1 | 1 | 1 |
| S2(0,0)<br><br><br>2 | S12(20,2)<br>S32(20,0)<br>S42(10,7) 2 | S312(40,2)<br><br><br>2 | <br><br><br>2 |
| S3(20,0) 3 | 3 | 3 | 3 |
| S4(10,4) 4 | 4 | 4 | 4 |

У разі, коли є не одне, а $m$ обмежень, то кожен шлях характеризується не двома довжинами, а $m+1$ однією довжиною, що визначається аналогічно.

Розглянуті процедури вирішення задач булевого програмування будемо називати першим типом оптимізації і для підвищення ефективності роботи алгоритму введемо другий вид оптимізації, що використовує аналогічну процедуру формування шляхів, але відрізняється від першого тим, що на першому ярусі ми будемо формувати не шляхи рангу 1, а безлічі шляхів, які відповідають складовим в функціоналі. Далі завдання вирішується так само. Рішення завдання (26) в цьому випадку наведено в таблиці 2.3.

Шляхи $S41(33,11)$ і $S43(47,14)$ на першому ярусі не формувалися, так як вони не проходять по обмеженню. Як видно з порівняння таблиць 2.1 і 2.2 з таблицею 2.3 число шляхів у разі використання оптимізації другого типу довелося побудувати менше, що зменшує тимчасову складність роботи запропонованої процедури.

Таблиця 2.3

Вирішення задачі (6)

| | | | | | |
|---|---|---|---|---|---|
| S21(20,2) | 1 | S31(20,0) | 1 | 1 | 1 |
| | 2 | S32(20,0)<br>S42(10,7) | 2 | S312(40,2) 2 | 2 |
| S3(20,0) | 3 | S213(20,2) | 3 | 3 | 3 |
| S4(10,4) | 4 | | 4 | 4 | 4 |

Тому автором було виконано експериментальне дослідження та порівняльний аналіз обох типів оптимізації з точки зору оцінки їх тимчасової складності і похибки, а також значення показника оперативності їх вирішення, яке оцінюється ймовірністю рішення задачі $P(T) = 1 - e^{-\frac{T_д}{T}}$ за деякий допустимий час $T_д$. В якості еталонного точного алгоритму був застосований алгоритм повного перебору з використанням стратегії гілок і меж. При проведенні експерименту кількість обмежень змінювалося від 7 до 68, кількість змінних варіювалася від 2 до 20, кількість доданків в функціоналі і обмеження змінювалися від 2 до 50. Також діапазон зміни коефіцієнтів у функціоналі і обмеженнях змінювався від 0 до 200 за рівномірним законом розподілу. Час рішення приведено в секундах, допустимий час принятий рівним 30 секундам. На кожну точку при оцінці часової складності алгоритмів в середньому і похибки алгоритмів вирішувалося не менше 50 тестових завдань. Результати експериментального дослідження наведені нижче на рисунках 2.7 - 2.12.

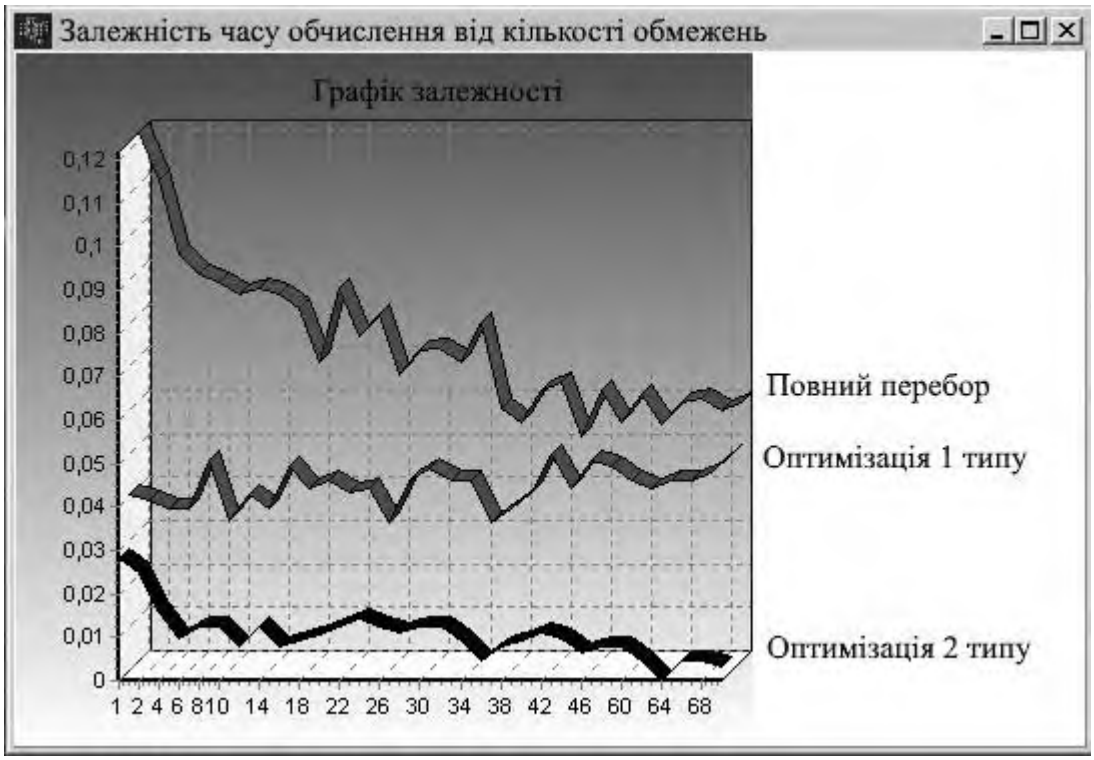



Рис. 2.7. Залежність часу обчислення від кількості обмежень

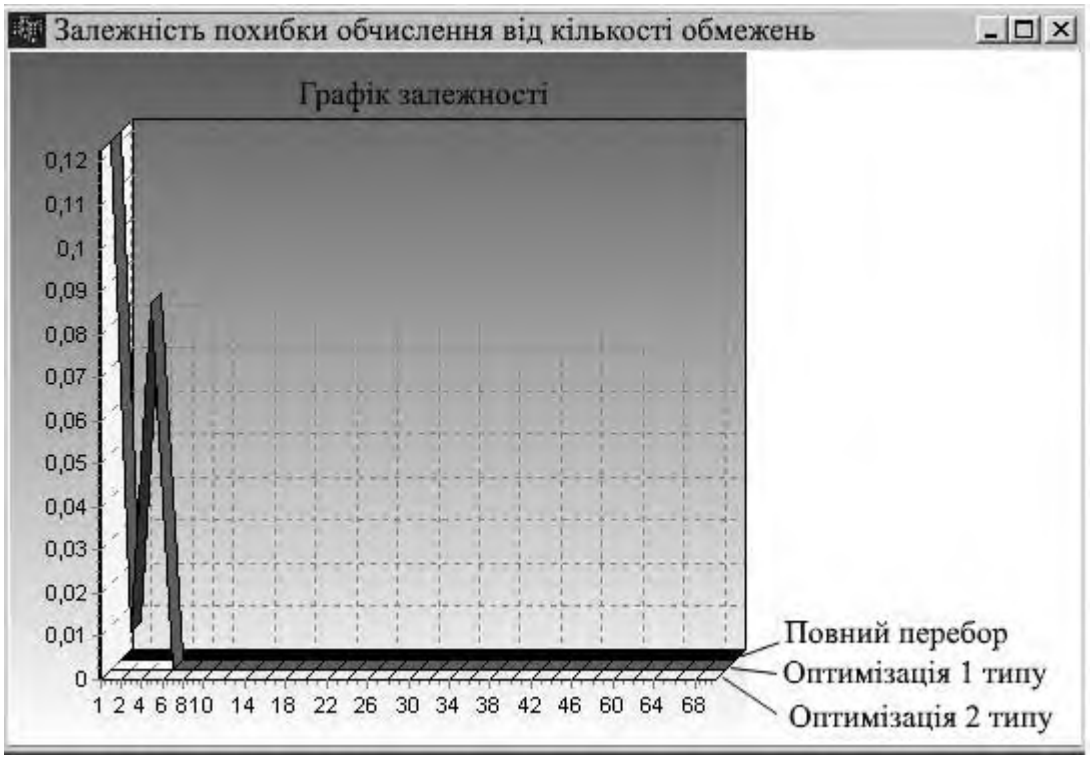



Рис. 2.8. Залежність похибки обчислення від кількості обмежень

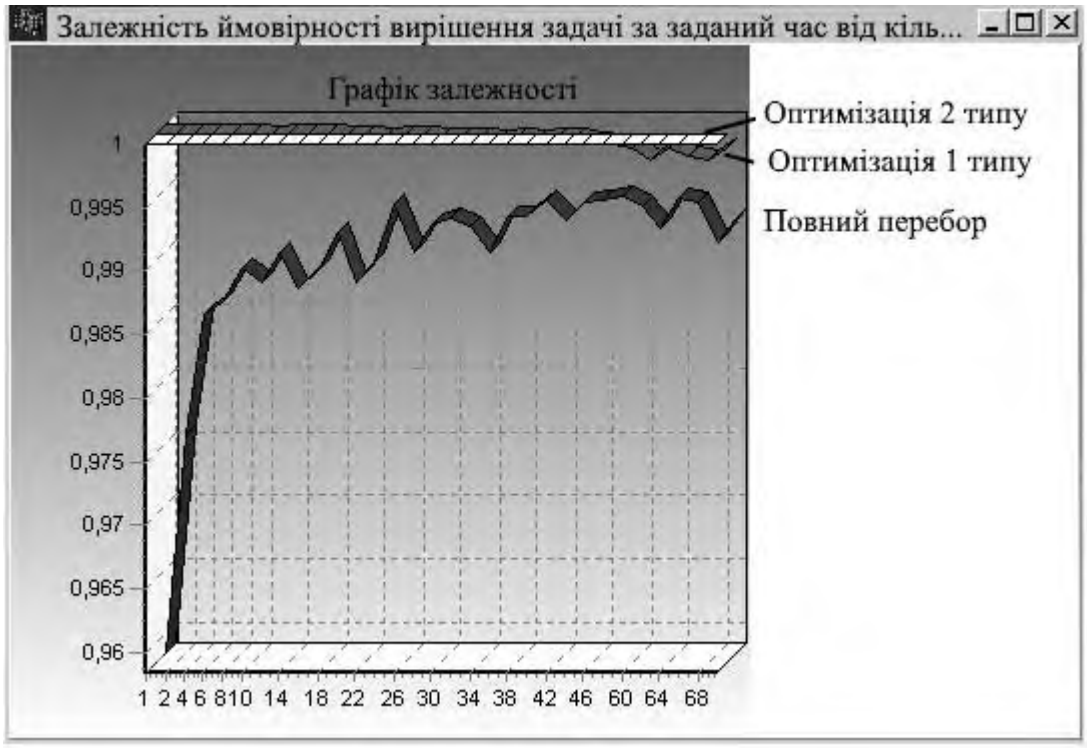



Рис. 2.9. Залежність ймовірності виконання завдання за заданий час від кількості обмежень

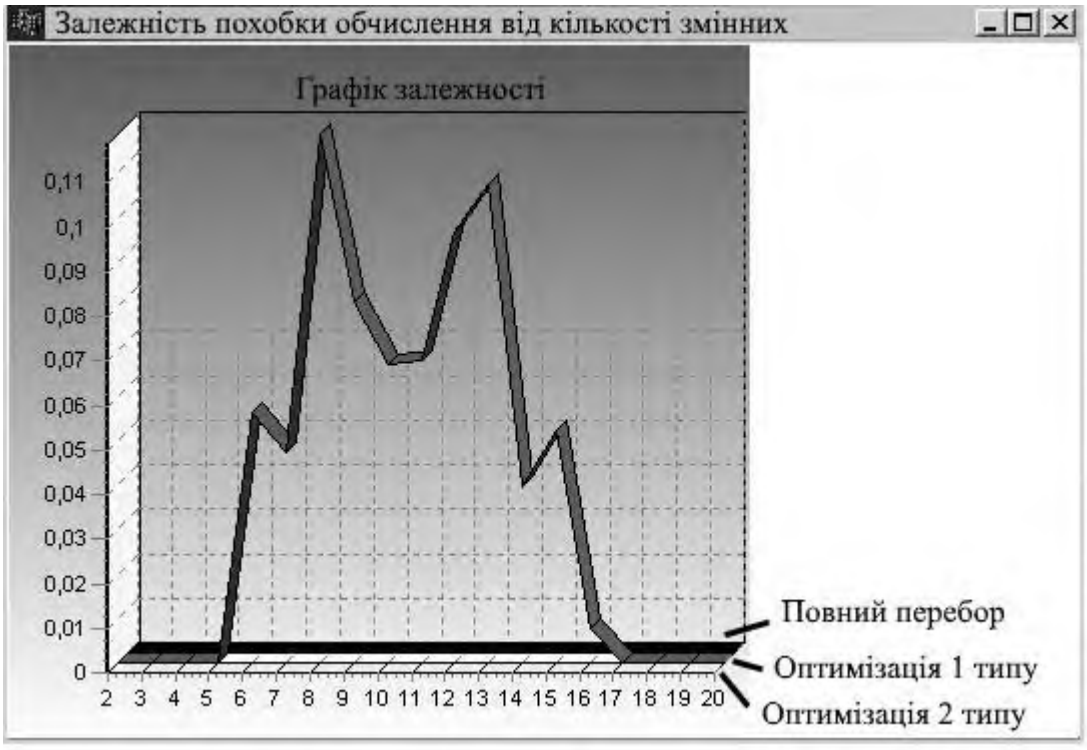



Рис. 2.10. Залежність похибки обчислення від кількості змінних

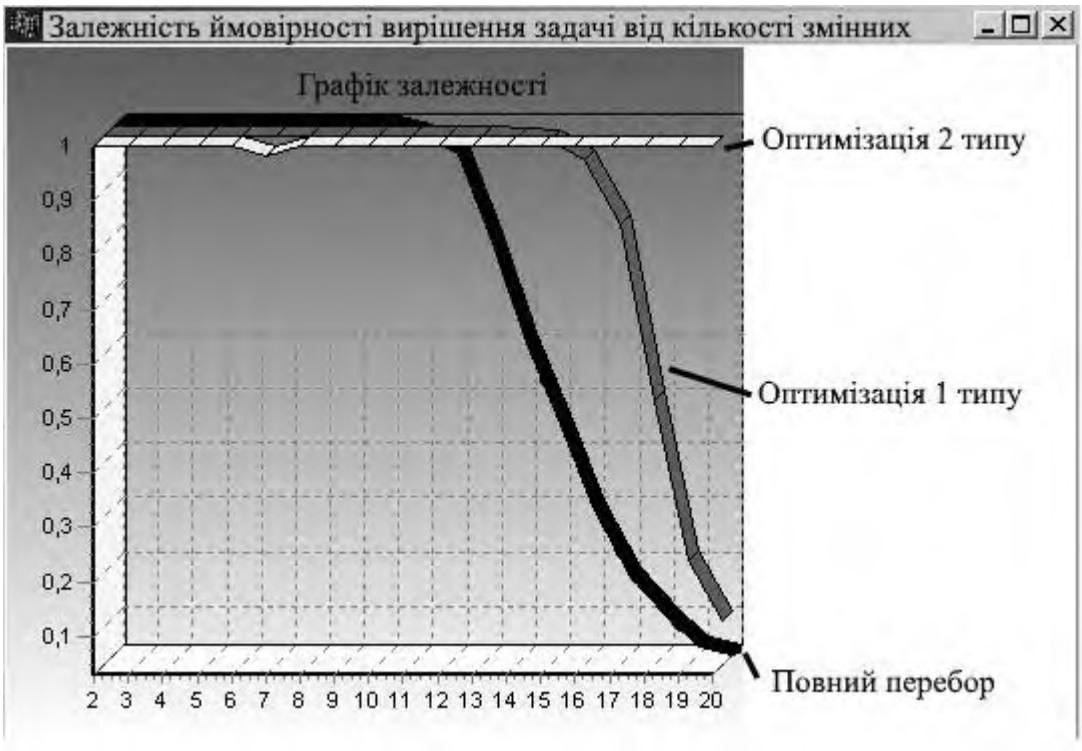



Рис. 2.11. Залежність ймовірності виконання завдання від кількості змінних

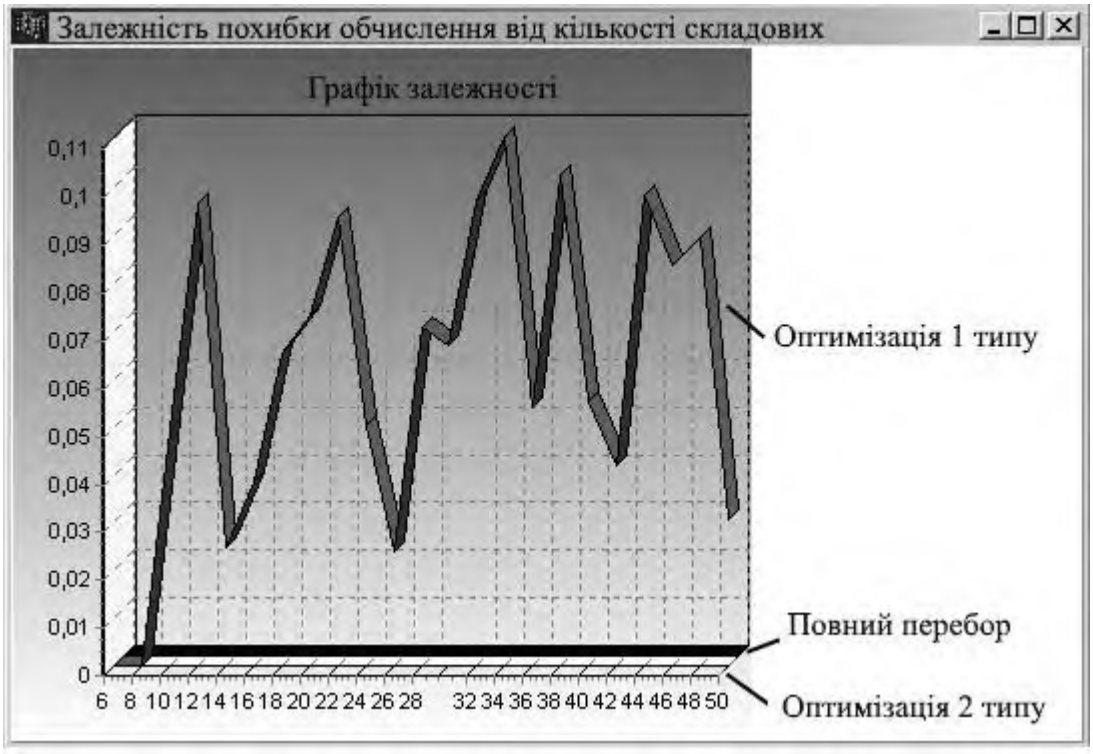



Рис. 2.12. Залежність похибки обчислення від кількості складових

## 2.7 Висновки за розділом

1. Показана ефективність рангового підходу до вирішення довільних завдань булевого програмування на основі запропонованих процедур, що дозволяють вирішувати як завдання лінійного, так і нелінійного програмування, з довільними нелінійностями як в функціоналі, так і в обмеженнях, алгоритмами поліноміальної складності з невеликою похибкою.

2. Результати експериментального дослідження часової складності і похибки розроблених алгоритмів наочно підтверджують їх теоретичну оцінку. Збільшення числа обмежень в задачах нелінійного булевого програмування призводить до зниження похибки їх вирішення, при цьому сама ступінь нелінійності несуттєво впливає на величину похибки.

3. Використання запропонованих процедур оптимізації другого типу дозволяють істотно підвищити оперативність вирішення завдань булевого програмування і точність їх вирішення, що суттєво для процесів планування виконання завдань в розподілених телекомунікаційних системах.

4. Запропоновані процедури ефективно можуть бути распаралелені, оскільки формування шляхів на ярусах у множинах $m_{sj}^{r}$ можна здійснювати одночасно. Таким чином, якщо мати $n$ - процесорних елементів для формування шляхів, то часова складність алгоритмів на основі розглянутих процедур не перевищить відповідно $O\left(pn^2\right)$ і $O\left(pn\right)$, якщо їх реалізовувати з використанням CUDA технологій, то це дозволить застосовувати ці процедури для управління в масштабі реального часу, в розподілених системах обчислення.

# РОЗДІЛ 3

# МОДЕЛЮВАННЯ РОБОТИ КЛАСТЕРА ТЕЛЕКОМУНІКАЦІЙНОЇ МЕРЕЖІ НА ОСНОВІ ВИРІШЕННЯ ЗАДАЧІ НЕЛІНІЙНОГО БУЛЕВОГО ПРОГРАМУВАННЯ РАНГОВИМ МЕТОДОМ

## 3.1 Модель кластера Grid-системи

Процес моделювання роботи кластера Grid-системи складається з покрокового виконання трьох операцій [14-17]:

- Операція №1 - імітація надходження завдань на вхід системи. Із загальної кількості завдань в пул завантажуються перші $mp$ завдань;

- Операція №2 - розподіл завдань з черги пулу між ресурсами і повернення, що не помістилися назад в пул;

- Операції №3 - імітація вирішення завдань ресурсами. Імітацією розв'язання задачі є віднімання від складності рішення задачі значення продуктивності ресурсу, на якому задача вирішується. Дана операція прийнята за умовну одиницю часу - такт. Всі часові процеси в моделюванні роботи кластера вимірюються в цій одиниці. Алгоритм моделювання процесу роботи типового кластера Grid - системи показаний на рис. 3.1. де:

TASK_COUNT - кількість завдань, які будуть подані на вхід системи за весь час моделювання процесу роботи кластера;

TASK_SOLVER - кількість вирішених завдань;

T - такт роботи системи;

T_SH – час, витрачений при формуванні пакета завдань;

FREQ_SH - період формування пакету завдань (період планування).

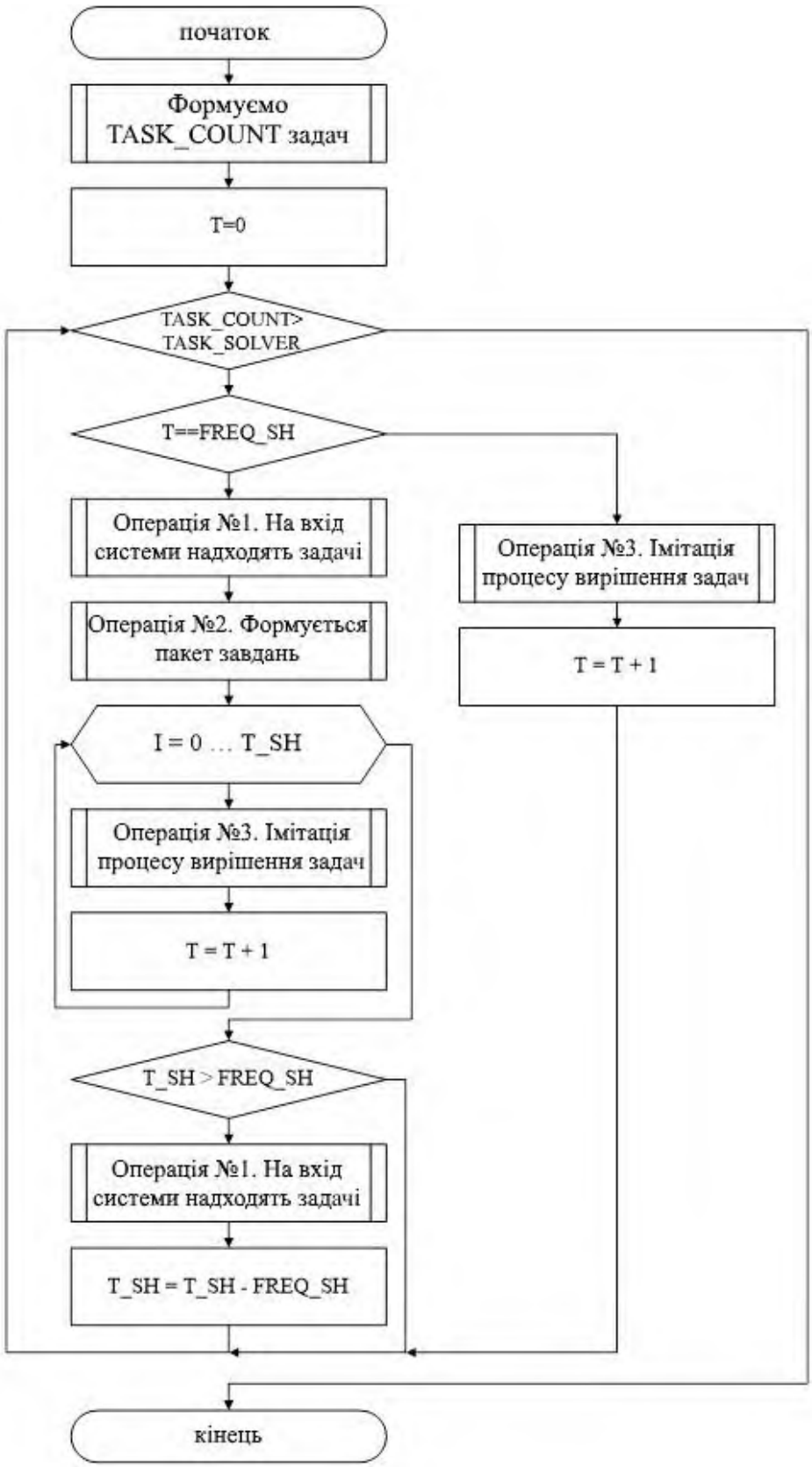



Рис. 3.1. Алгоритм моделювання роботи типового кластера Grid-системи

Моделювання роботи кластера здійснюється наступними кроками:

- Крок 1. Виконується операція №1. На вхід системи подається встановлена кількість завдань.

- Крок 2. Виконується операція №2. Формується пакет завдань встановленим методом.

- Крок 3. Виконується операція №3. Імітується процес вирішення для всіх завдань, які на поточний момент часу вже знаходяться в ресурсі. Виконується така кількість тактів, яка була витрачена на формування пакету завдань.

- Крок 4. Виконується операція №3. Імітується процес вирішення завдань, поки не настає наступний період планування або не будуть вирішені всі завдання.

- Крок 5. Перехід до виконання кроку №1.

### 3.1.1 Елементи моделі

Представлена модель включає в себе наступні елементи:

- множина завдань - це набір завдань, які подаються на Grid-систему для моделювання процесу її роботи;

- множина ресурсів - набір ресурсів, які виконують вирішення множини завдань, поданих на Grid-систему;

- параметри моделювання - набір параметрів, що визначають режим роботи і характеристики моделі Grid-системи;

- методи планування - набір методів, якими Grid-система розподіляє вхідні дзвінки між ресурсами.

### 3.1.2 Множина завдань та їх параметри

Під терміном завдання мається на увазі структура з набором наступних характеристик [26]:

- універсальність;
- типи ресурсів, на яких завдання може бути вирішено;
- складність вирішення;
- пріоритет.

Характеристики завдань визначаються параметрами множини.

Параметри множини завдань. У моделі, для процесу роботи Grid-системи, необхідно встановити такі параметри множини завдань:

- кількість завдань - загальна кількість завдань множини;
- кількість типів ресурсів - максимальне число типів ресурсів, на яких можуть бути вирішені завдання. Для сумісності кількість типів ресурсів в множині завдань має збігатися з одноіменним параметром в множині ресурсів, на яких планується вирішувати дану множину завдань;
- універсальність - максимальне значення універсальності завдань у всій множині. Універсальність визначає кількість типів ресурсів, якими може бути вирішена задача;
- закон розподілу універсальності - випадковий закон, за яким завданням буде призначена універсальність. Максимально можливе значення визначається параметром «Універсальність», мінімальне значення 1;
- кількість унікальних завдань - відсоток завдань, які можуть вирішуватися тільки одним ресурсом щодо загальної кількості завдань. Приймає значення від 0% до 100%;
- складність рішення - максимальне значення складності вирішення завдань у всій множині. Складність вирішення завдання визначає кількість тактів, яке буде витрачено на вирішення завдання ресурсом з продуктивністю 1 такт;

- закон розподілу складності рішення - випадковий закон, за яким завданням буде призначена складність вирішення. Максимально можливе значення визначається параметром «Складність рішення», мінімальне значення - 1;

- сумарна складність вирішення - сума складнощів рішення всіх задач множини. Параметр не задається користувачем. Підраховується при створенні множини завдань;

- пріоритет - максимальне значення пріоритету завдань у всій безлічі. Пріоритет завдання визначає важливість вирішення завдання в умовних одиницях;

- закон розподілу пріоритету - випадковий закон, за яким завданням буде призначений пріоритет. Максимально можливе значення визначається параметром «Пріоритет», мінімальне значення - 1.

### 3.1.3 Створення множини завдань

Для створення множини завдань, яке буде подано на Grid-систему необхідно виконати наведені нижче дії:

1) Поставити параметри множини в таблиці «Вхідні завдання»;

2) Вибираємо команду в пункті меню «Операції», далі «Створити завдання», або використовується поєднання клавіш «Ctrl + M», або натискаємо кнопку створення (з іконкою плюс) на панелі управління внизу списку множин задач.

У списку множин з'явилася назва щойно створеної множини завдань T5000_20_2497543, як показано на рисунку 3.2.

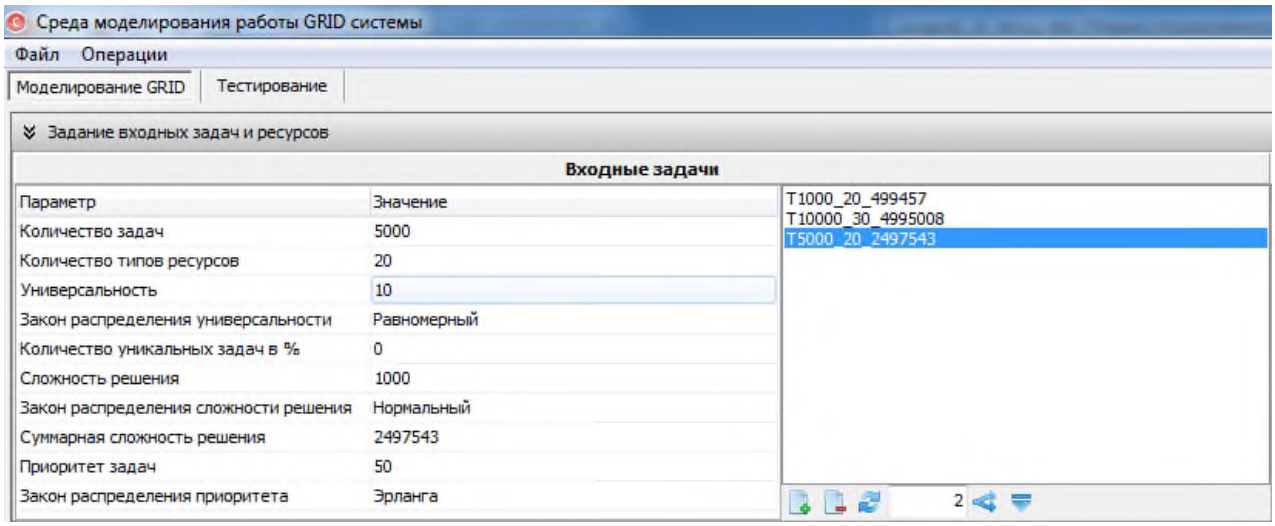
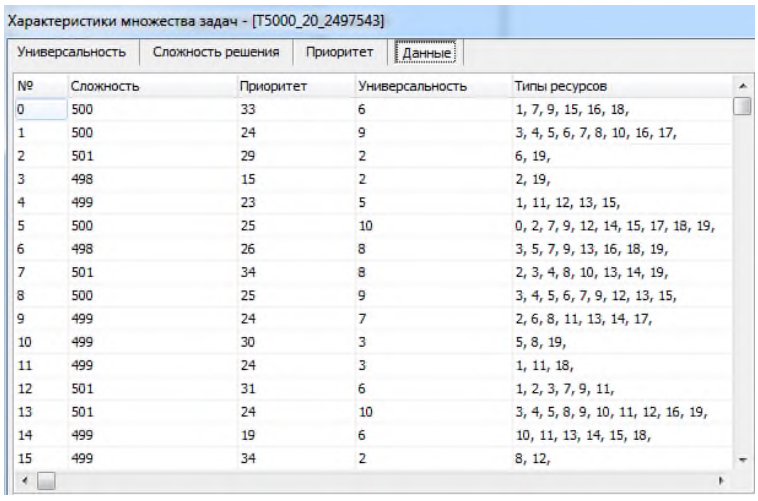
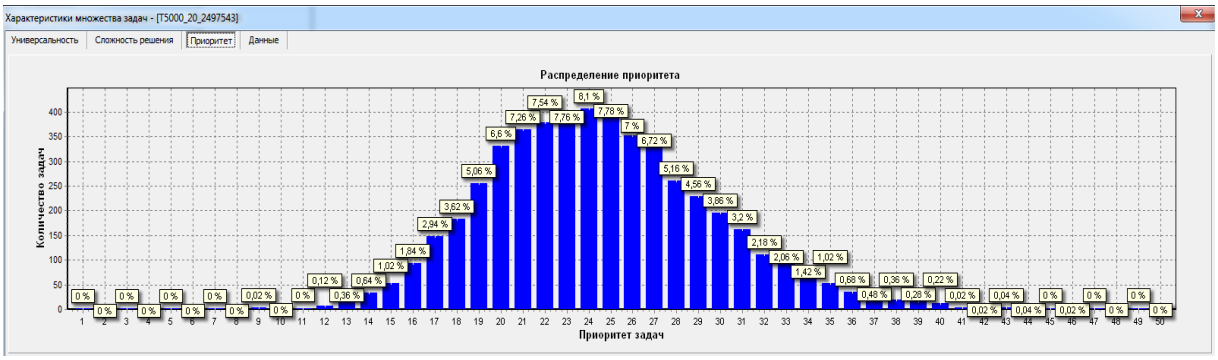



Рис. 3.2. Створення множини завдань. T - Tasks (завдання), 5000 - кількість завдань у множині, 20 - кількість типів ресурсів, 2497543 - сумарна складність вирішення завдань

Клацнувши мишкою на множині в списку відкривається вікно з розгорнутими характеристиками множини, як показано на рисунку 3.3.




А




Б

Рис. 3.3. Розгорнуті характеристики множини завдань: А) список всіх характеристик; Б) пріоритет завдань

Множину завдань можна розбити на підзавдання, при цьому сумарна складність вирішення завдань залишається колишньою, збільшується тільки кількість завдань. Таким чином, моделюється складні завдання, які при вирішенні можна розбити на підзавдання і вирішувати паралельно на різних ресурсах. Для розбиття раніше створеної множини завдань на підзавдання необхідно:

1) В панелі управління під списком множини завдань встановити коефіцієнт;

2) Вибрати команду в пункті меню «Операції», далі «Розмножити завдання», або використовувати поєднання клавіш «Ctrl + O», або натиснути кнопку розбиття множини на підзавдання (з іконкою розгалуження стрілок) на панелі управління внизу списку множин задач.

У списку з'явиться нова множина з великою кількістю завдань, але точно такою ж складністю рішення. Це показано на рисунку 3.4.

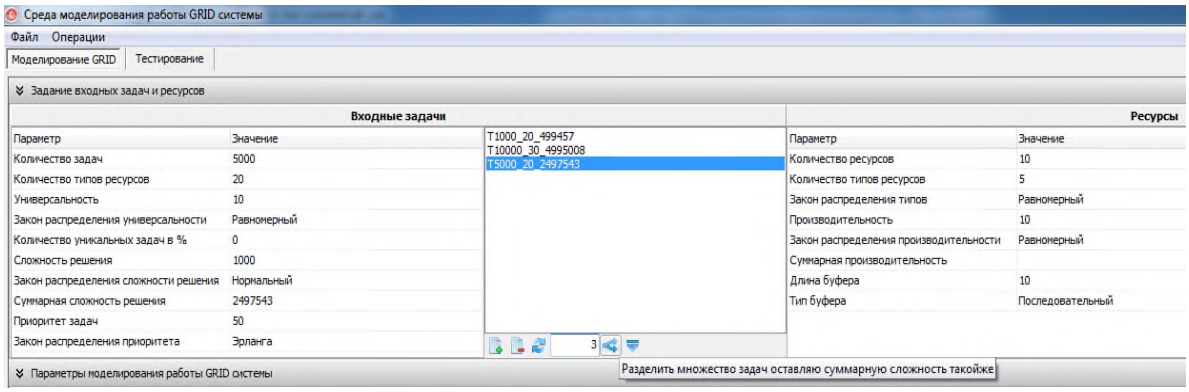



Рис. 3.4. Розподіл множини завдань на підзавдання

Для збереження на жорсткому диску, або переносному накопичувачі, або в хмарі створеної множини завдань потрібно вибрати команду в меню «Файл», далі «Зберегти завдання» або натиснути «Ctrl + T».

### 3.2 Множина ресурсів

Набір ресурсів, які виконують вирішення множини завдань, поданих на Grid-систему. Ресурс є структурою з набором наступних характеристик:

- тип;

- продуктивність.

Характеристики завдань визначаються параметрами множини.

Параметри множини ресурсів. У моделі є можливість задавати такі параметри для множини ресурсів:

- кількість ресурсів - загальна кількість ресурсів в множині;

- кількість типів ресурсів - максимальне число типів ресурсів, позначається номером по порядку;

- закон розподілу типів - випадковий закон, за яким ресурсам буде призначений тип. Максимально можливе значення визначається параметром «Кількість типів ресурсів», мінімальне значення 0;

- продуктивність - максимальне значення продуктивності ресурсу у всій множині. Продуктивність ресурсу визначає кількість тактів складності рішення задачі, яке буде виконано ресурсом за один 1 такт;

- закон розподілу продуктивності - випадковий закон, за яким ресурсам буде призначена продуктивність. Максимально можливе значення визначається параметром «Продуктивність», мінімальне значення 1;

- сумарна продуктивність - сума продуктивностей всіх ресурсів в множині. Параметр не задається користувачем. Підраховується при створенні множини ресурсів;

- довжина буфера - розмір буферів всіх ресурсів множини. Визначає кількість завдань, які можуть бути поставлені в чергу на вирішення до ресурсу;

- тип буфера - характер поведінки буфера в процесі імітації вирішення завдань. Послідовний - завдання послідовник по одному вирішуються ресурсом. Паралельний - все завдання в буфері вирішуються ресурсом одночасно.

### 3.2.1 Створення множини ресурсів

Для створення множини ресурсів, якими буде вирішуватись множина поданих на Grid-систему завдань необхідно виконати наступні дії:

1) Поставити параметри множини в таблиці «Ресурси»;

2) Натиснути кнопку створення (з іконкою плюс) на панелі управління внизу списку множин ресурсів.

У списку множин з'явиться назва щойно створеної множини ресурсів R30_ 121, як показано на рисунку 3.5.

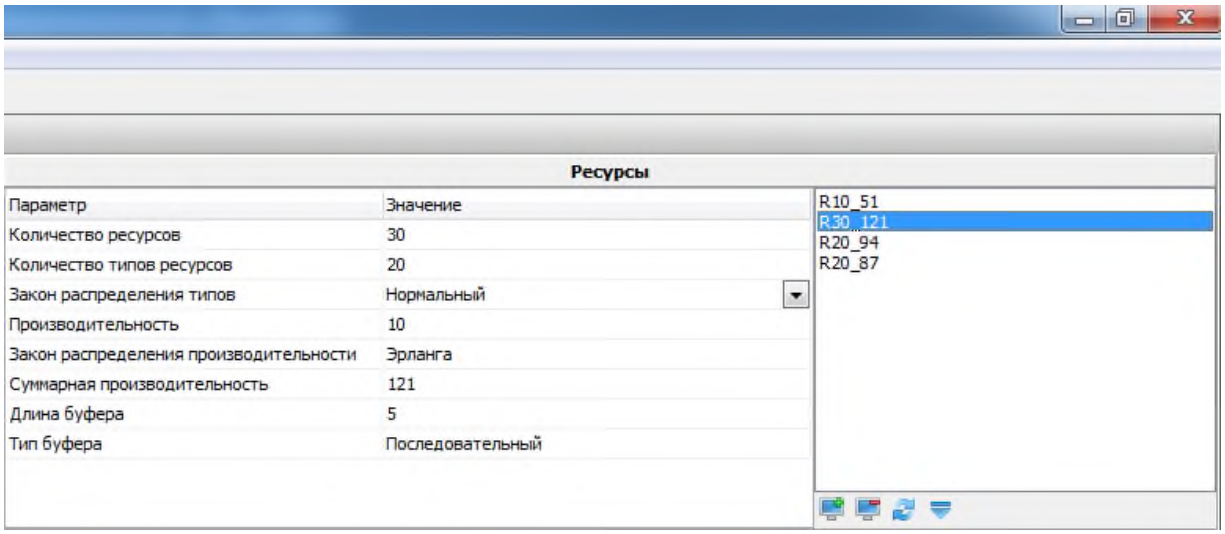
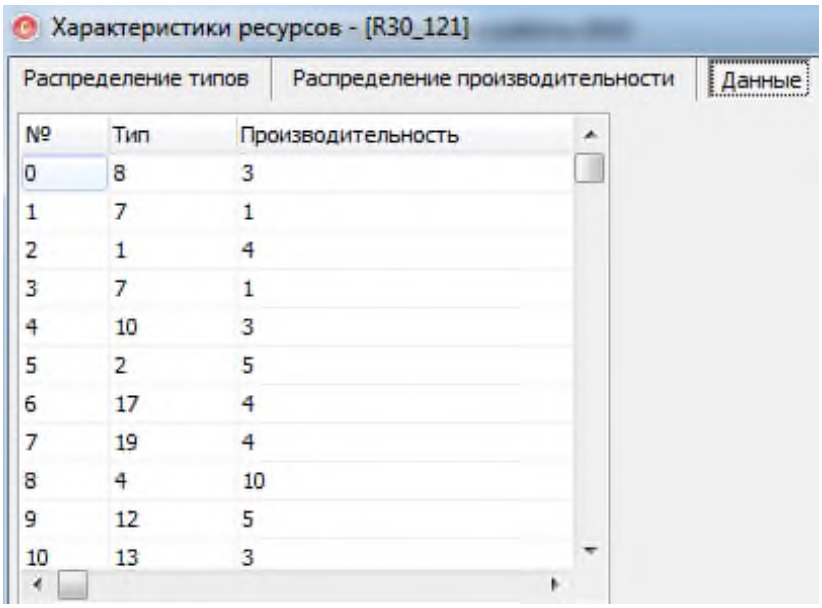



Рис. 3.5. Створення множини ресурсів. R - Resours (ресурси), 30 - кількість ресурсів в множині, 121 - сумарна продуктивність ресурсів

Клацнувши мишкою на множині, в списку відкриється вікно з розгорнутими характеристиками множини, як показано на рисунку 3.6.

Для збереження на жорсткому диску, або переносному накопичувачі, або в хмарі створеної множини ресурсів потрібно вибрати команду в меню «Файл», далі «Зберегти ресурси» або «Ctrl + R».




а)


б)

Рис. 3.6. Розгорнуті характеристики множини ресурсів, а) список всіх характеристик; б) розподіл типів ресурсів

### 3.3 Параметри моделювання

Набір параметрів, що визначає режим роботи і характеристики моделі Grid-системи.

У моделі задаються наступні параметри:

- завдання - множина завдань, яка буде подана на систему в процесі моделювання;

- ресурси - множина ресурсів системи, на якій будуть вирішуватися завдання при моделюванні;

- інтенсивність - максимальна кількість завдань, яка надійде на вхід системи при черговому такті планування;

- закон розподілу інтенсивностей - випадковий закон, за яким завдання будуть розподілені для послідовного надходження на вхід системи. Максимально можливе значення визначається параметром «Інтенсивність», мінімальне значення 0;

- відчуженість - максимальна кількість ресурсів, що вийшли з ладу на черговому такті планування. Задається в процентах від 0% до 100% щодо кількості ресурсів;

- закон розподілу відчуженості - випадковий закон, за яким розподіляється відчуженість для кожного такту планування. Максимальне значення визначається параметром «Відчуженість», мінімальне 0;

- довжина пулу - максимальна кількість завдань, які може помістити система для подальшої обробки;

- затримка - кількість тактів, які витрачає система на передачу завдання з буфера в ресурс на рішення;

- частота планування - кількість тактів, які проходять між плануванням. Так само є кількість тактів між надходженнями нових завдань на вхід системи;

- коефіцієнт планування - коефіцієнт, на який зменшується кількість операцій, виконана планувальником для переводу операцій, витрачених на планування в витрачений час, яке в системі виражається в умовних одиницях - тактах.

### 3.3.1 Встановлення параметрів роботи Grid-системи

Для введення завдань і ресурсів необхідно з списку множин відповідно вибрати ресурси і завдання, як показано на рисунку 3.7 і натиснути кнопку з іконкою у вигляді стрілки спрямованої вниз на панелі управління списків ресурсів і завдань. Одразу імена множин будуть відображені в полях відповідних параметрів у вкладці «Параметри моделювання роботи Grid-системи».

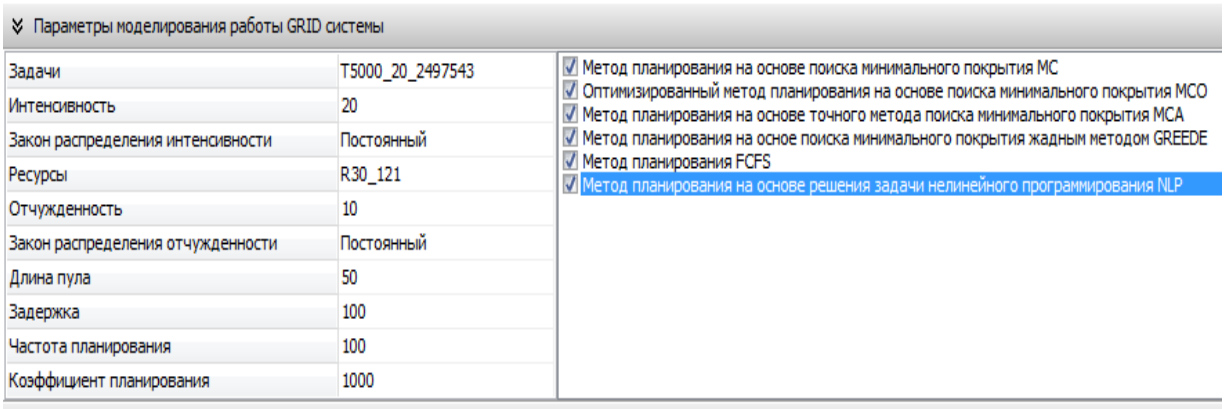



Рис. 3.7. Встановлення параметрів роботи Grid-системи

## 3.4 Методи планування

Набір методів, якими Grid-система розподіляє вхідні завдання між ресурсами. У даній моделі є можливість проводити дослідження, грунтуючись на наступних методах планування виконання завдань:

- метод планування MC. Метод заснований на пошуку найменшого числа ресурсів, якими можна вирішити всі завдання. Для пошуку

найменшого числа ресурсів використовується наближений «Частотний метод».

Алгоритм роботи МС:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Вирішується задача пошуку мінімального покриття частотним методом;

3) Завдання поміщаються в буфери ресурсів, які увійшли в мінімальне покриття;

4) Якщо помістилися всі завдання, закінчується процедура планування. Інакше завдання повертаються в пул і відбувається перехід до «Операції №2».

У процесі планування завдання, у яких не залишилося вільних ресурсів, які можуть їх вирішити, поміщаються назад в пул;

- метод планування МСО. Метод заснований на пошуку найменшого числа ресурсів, якими можна вирішити всі завдання з додаванням операції оптимізації, яка полягає в тому, що в першу чергу завантажуються більш вільні ресурси [3,4]. Для пошуку найменшого числа ресурсів використовується наближений «Частотний метод».

Алгоритм роботи МСО:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Вирішується задача пошуку мінімального покриття частотним методом;

3) Завдання поміщаються в буфери ресурсів, які увійшли в мінімальне покриття, в першу чергу поміщаємо в більш вільні ресурси;

4) Якщо помістили всі завдання, закінчуємо процедуру планування. Інакше завдання повертаються в пул і відбувається перехід до «Операції №2».

У процесі планування завдання, у яких не залишилося вільних ресурсів, які можуть їх вирішити, поміщаються назад в пул;

- метод планування MCA. Метод заснований на пошуку найменшого числа ресурсів, якими можна вирішити всі завдання. Для пошуку найменшого числа ресурсів використовується точний «Ранговий метод».

Алгоритм роботи MCA:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Вирішується задача пошуку мінімального покриття ранговим методом;

3) Завдання поміщаються в буфери ресурсів, які увійшли в мінімальне покриття;

4) Якщо помістили всі завдання, закінчуємо процедуру планування. Інакше повертаємо завдання в пул і відбувається перехід до «Операції №2».

У процесі планування завдання, у яких не залишилося вільних ресурсів, які можуть їх вирішити, поміщаються назад в пул;

- метод планування GREEDE. Метод заснований на пошуку найменшого числа ресурсів, якими можна вирішити всі завдання. Для пошуку найменшого числа ресурсів використовується наближений «Жадібний метод».

Алгоритм роботи GREEDE:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Вирішується задача пошуку мінімального покриття;

3) Завдання поміщаються в буфери ресурсів, які увійшли в мінімальне покриття;

4) Якщо помістили всі завдання, закінчуємо процедуру планування. Інакше повертаємо завдання в пул і відбувається перехід до «Операції №2».

У процесі планування завдання, у яких не залишилося вільних ресурсів, які можуть їх вирішити, поміщаються назад в пул;

- метод планування FCFS. Метод заснований на принципі «першим прийшов, першим обслужився».

Алгоритм роботи FCFS:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Береться перше по порядку завдання з пулу і поміщається в перший вільний ресурс, яким воно може бути вирішено, якщо таких ресурсів не виявилося завдання повертається в пул;

3) Процедура планування припиняється після обробки останнього в пулі завдання.

В даному методі не використовується буфер, завдання відразу надсилаються на рішення ресурсу, якщо він вільний;

- метод планування NLP. Метод заснований на пошуку на вирішенні задачі нелінійного програмування, який полягає в тому, що алгоритм знаходить максимальну кількість завдань з сумарним максимальним пріоритетом, які можуть бути поміщені в буфер вільних ресурсів.

Алгоритм роботи NLP:

1) Визначаються відповідності між завданнями і ресурсами, на яких вони можуть вирішуватися;

2) Вирішується задача нелінійного програмування;

3) Завдання поміщаються в буфери ресурсів, якщо залишилися завдання, для яких ще є вільні ресурси, відбувається перехід до «Операції №2», якщо таких завдань немає, процедура планування закінчується.

У процесі планування завдання, у яких не залишилося вільних ресурсів, які можуть їх вирішити, поміщаються назад в пул.

## 3.5 Характеристики роботи Grid-системи, що досліджуються

У даній моделі, в процесі роботи Grid-системи, досліджуються наступні характеристики:

- час виконання – кількість тактів роботи системи, за яке було вирішено всю множину завдань, поданих на вхід Grid-системи:

$$t_в = t_{end} - t_{begin}, \tag{3.1}$$

де $t_{begin}$ – такт надходження першого завдання в пул системи; $t_{end}$ – такт вирішення системою останнього завдання;

- час відповіді – середня кількість тактів, за яке була вирішена одна задача в режимі моделювання Grid-системи:

$$t_r = \frac{1}{m}\sum_{i=1}^{m} t_{ri}, \tag{3.2}$$

де $m$ – кількість завдань, які були вирішені системою; $t_{ri}$ – кількість тактів $i$-ої задачі з моменту потрапляння в пул системи до моменту її вирішення системою;

- максимальний час відповіді – максимальна кількість тактів, за яку була вирішена одна задача:

$$t_{r\max} = \max_{0<i\le m} t_{ri}, \tag{3.3}$$

де $m$ – кількість завдань, які були вирішені системою; $t_{ri}$ – кількість тактів $i$-ої задачі з моменту потрапляння в пул системи до моменту її вирішення системою;

- час очікування – середня кількість тактів, яку очікували завдання на вході системи до подальшої обробки:

$$t_w = \frac{1}{m}\sum_{i=1}^{m} t_{wi}, \tag{3.4}$$

де $m$ – кількість завдань, які були вирішені системою; $t_{wi}$ – кількість тактів $i$-ої задачі з моменту потрапляння в пул системи до моменту призначення ресурсу, який буде її вирішувати і відправки її в буфер призначеного ресурсу (час знаходження завдання в пулі системи);

- час обслуговування – середня кількість тактів, за яку завдання перебували в буфері призначеного ресурсу до початку їх вирішення даними ресурсом:

$$t_s = \frac{1}{m}\sum_{i=1}^{m} t_{si}, \tag{3.5}$$

де $m$ – кількість завдань, які були вирішені системою; $t_{si}$ – кількість тактів $i$-ої задачі з моменту потрапляння в буфер призначеного ресурсу до початку її рішення даними ресурсом (час знаходження завдання в буфері ресурсу);

- час планування – середня кількість тактів, яка була витрачена системою на планування:

$$t_p = \frac{1}{p}\sum_{i=1}^{p} t_{pi}, \tag{3.6}$$

де $p$ – кількість планувань, вироблених системою за час вирішення всіх завдань; $t_{pi}$ – кількість тактів, витрачених системою, на $i$-е планування;

- коефіцієнт використання – середнє геометричне значення коефіцієнтів використання всіх ресурсів. Коефіцієнт використання ресурсу – відношення суми пріоритетів всіх вирішених завдань цим ресурсом до суми пріоритетів всіх завдань з вхідної множини, які можуть бути вирішені на даному ресурсі:

$$r_u = \sqrt[n]{\prod_{i=1}^{n} r_{ui}}\ , \tag{3.7}$$

$$r_{ui} = \frac{\sum_{j=1}^{m_{si}} \mathrm{Pr}_j}{\sum_{j=1}^{m_i} \mathrm{Pr}_j}, \tag{3.8}$$

де $n$ – кількість ресурсів в системі; $r_{ui}$ – коефіцієнт використання $i$-ого ресурсу; $m_{si}$ – кількість завдань, які були вирішені $i$-м ресурсом; $m_i$ – кількість завдань з загальної множини вхідних завдань, які можуть бути вирішені $i$-м ресурсом; $Pr_j$ – пріоритет $j$-ого завдання;

- коефіцієнт завантаження – середнє геометричне значення коефіцієнтів завантаження всіх ресурсів. Коефіцієнт продуктивності ресурсу – відношення кількості вирішених завдань цим ресурсом до кількості всіх завдань з вхідної множини, яка може бути вирішена на даному ресурсі:

$$r_l = \sqrt[n]{\prod_{i=1}^{n} r_{li}}\ , \tag{3.9}$$

$$r_{li} = \frac{m_{si}}{m_i}, \tag{3.10}$$

де $n$ – кількість ресурсів в системі; $r_{li}$ – коефіцієнт використання $i$-ого ресурсу; $m_{si}$ – кількість завдань, які були вирішені $i$-м ресурсом; $m_i$ –

кількість завдань з загальної множини вхідних завдань, які могли бути вирішені *i*-м ресурсом;

- коефіцієнт важливості – середнє геометричне відносини сумарного пріоритету завдань, які були поміщені в буфери в результаті планування, до сумарного пріоритету завдань, що перебували в пулі:

$$r_i = \sqrt[p]{\prod_{j=1}^{p} r_{ij}}\,, \tag{3.11}$$

$$r_{ij} = \frac{\sum_{k=1}^{m_{pj}} \mathrm{Pr}_k}{\sum_{k=1}^{m_{wj}} \mathrm{Pr}_k}, \tag{3.12}$$

де $p$ – кількість планувань, виконаних системою за час вирішення всіх завдань; $r_{ij}$ – відношення пріоритету завдань поміщених в буфер ресурсів в результаті планування до пріоритету завдань, що перебували в пулі системи, на $j$-му плануванні; $m_{pj}$ – кількість завдань, яка була розпланована між ресурсами (поміщена в буфер ресурсів) при $j$-му плануванні; $m_{wj}$ – кількість завдань які перебували в пулі при $j$-му плануванні; $Pr_k$ – пріоритет $j$-ого завдання;

- коефіцієнт збереження важливості – середнє геометричне відносини сумарного пріоритету завдань, які були поміщені в буфери в результаті планування після відмови випадковоі кількості ресурсів, до сумарного пріоритету завдань, які були поміщені в буфери в результаті планування до відмови ресурсів:

$$r_s = \sqrt[p]{\prod_{i=1}^{p} r_{si}}\,, \tag{3.13}$$

$$r_{si} = \frac{\sum_{k=1}^{m_{fi}} \mathrm{Pr}_k}{\sum_{k=1}^{m_{pi}} \mathrm{Pr}_k}, \tag{3.14}$$

де $p$ – кількість планувань, виконаних системою за час вирішення всіх завдань; $r_{si}$ – відношення пріоритету завдань, поміщених в буфери ресурсів після планування при випадковій відмові ресурсів до пріоритету завдань, поміщених в буфери ресурсів після планування до відмови ресурсів на $i$ -му плануванні; $m_{pj}$ – кількість завдань, які були розплановані між ресурсами (поміщені в буфер ресурсів) до відмови випадкової кількості ресурсів на $j$ -му плануванні; $m_{fi}$ – кількість завдань, які були розплановані між ресурсами (поміщені в буфер ресурсів) після відмови випадкової кількості ресурсів на $i$-му плануванні; $Pr_k$ – пріоритет $k$-ого завдання;

- коефіцієнт прискорення – відношення розрахункового часу виконання завдань, розв'язуваних послідовно одним ресурсом з продуктивністю, яка дорівнює середньому значенню продуктивності всіх ресурсів Grid до часу виконання цих же завдань Grid-системою:

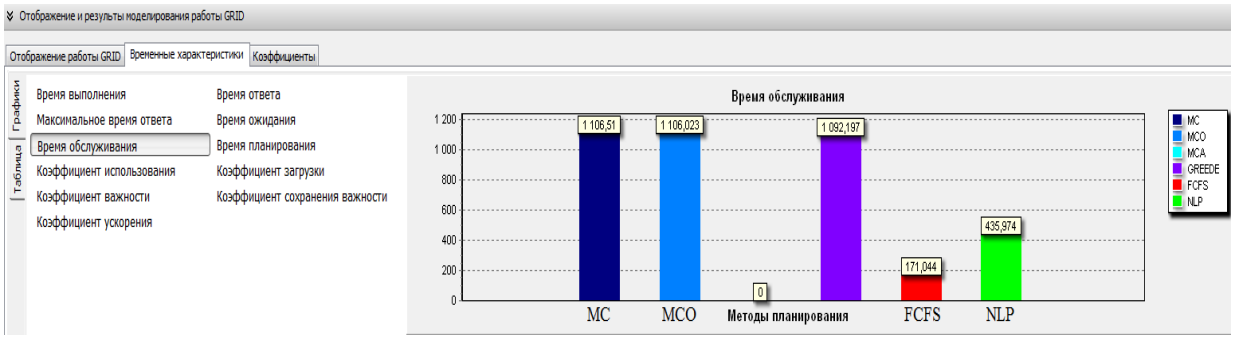
$$r_a = \frac{t_{ов}}{t_в}, \tag{3.15}$$

$$t_{ов} = \frac{S}{\overline{P}}, \tag{3.16}$$

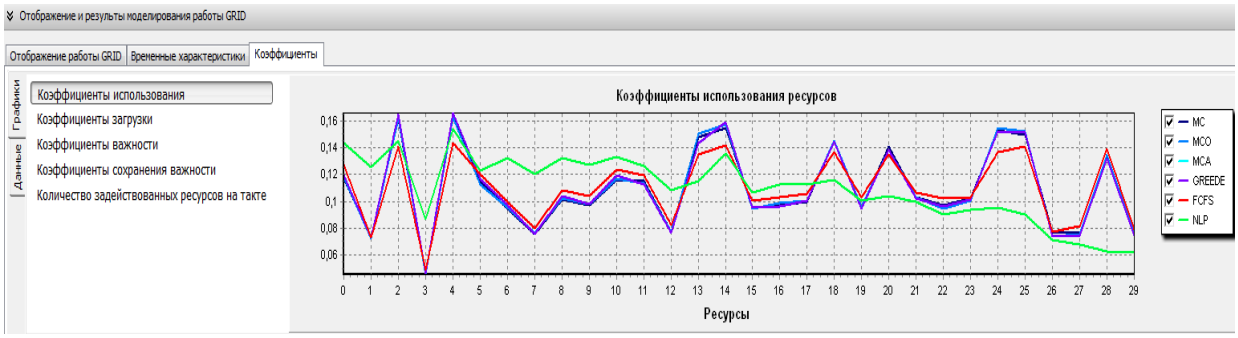
де $t_{ов}$ – розрахунковий час послідовного вирішення завдань одним ресурсом з продуктивністю, яка дорівнює середньому значенню продуктивності ресурсів Grid; $t_в$ – час виконання завдань Grid-системою; $S$ – сумарна складність вирішення всіх завдань; $\overline{P}$ – середня продуктивність всіх ресурсів Grid.

### 3.6 Відображення характеристик, що досліджуються

Досліджувані характеристики виводяться на графіки і в таблицю, які знаходяться у вкладках «Часові характеристики» і «Коефіцієнти», рисунок 3.8.




а)




б)

Рис. 3.8. Висновок досліджуваних характеристик: а) часові характеристики; б) коефіцієнти

Код реалізації моделі наведено в додатку А.

## 3.7 Результати експериментального дослідження методів планування на основі розробленої моделі

Результати моделювання, отримані при заданих значеннях та параметрах моделювання, наведені в таблиці 3.1. Час вказаний в тактах. На рисунках 3.9-3.11 наведено приклад задання множини завдань, ресурсів та встановлення параметрів моделювання.

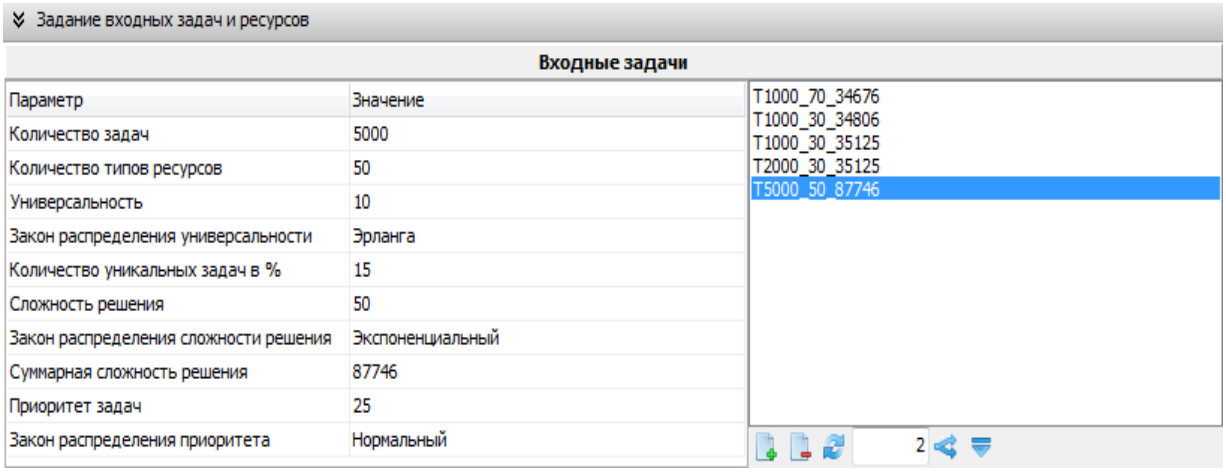



Рис. 3.9. Параметри множини завдань

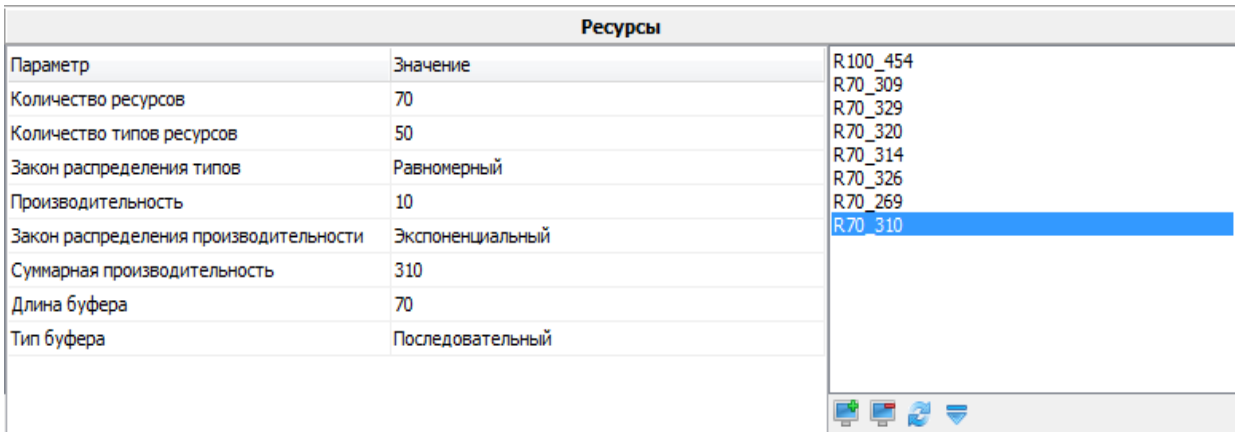



Рис. 3.10. Параметри множини ресурсів

При моделюванні досліджувалися залежності часу виконання всіх завдань глобальної черги в залежності від кількості завдань у черзі при різних складнощах розв'язуваних завдань і різної продуктивності ресурсів, а також залежність коефіцієнта використання ресурсів гетерогенної GRID-системи від кількості розв'язуваних завдань.

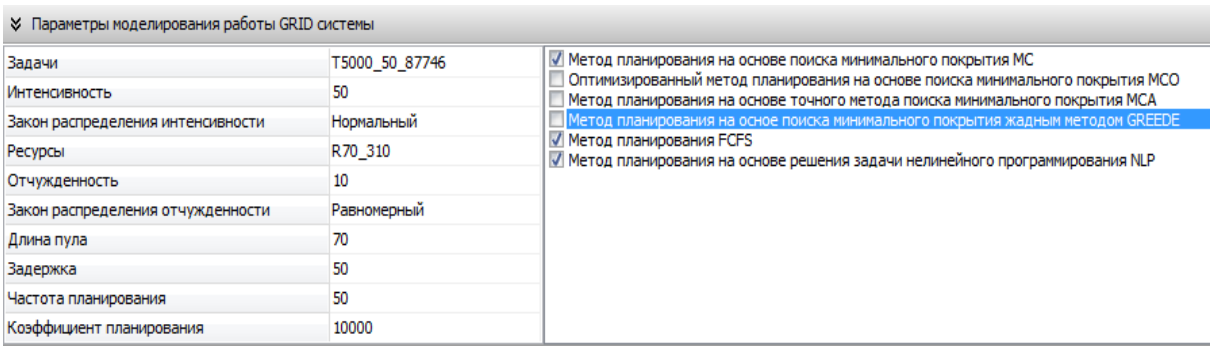



Рис. 3.11. Параметри моделювання

Таблиця 3.1

Результати моделювання роботи Grid-системи при заданих параметрах

| Показник якості планування | Метод планування | | |
|---|---|---|---|
| | MC | FCFC | NLP |
| Час виконання | 16157 | 18762 | 14206 |
| Час очікування | 36 | 196 | 5 |
| Коефіцієнт важливості | 0,626 | 0,188 | 0,999 |
| Коефіцієнт прискорення | 1,226 | 1,056 | 1,395 |
| Коефіцієнт збереження важливості | 0,781 | 0,763 | 0,868 |
| Коефіцієнт завантаження | 0,384 | 0,334 | 0,453 |
| Коефіцієнт використання | 0,38 | 0,334 | 0,454 |
| Максимальний час відповіді | 5347 | 2075 | 4168 |
| Час обслуговування | 2095 | 50 | 833 |

Характеристики, наведені на рис.3.12-3.15, отримані при числі ресурсів, використовуваних в гетерогенної Grid-системі, рівному 10 з довірчою ймовірністю рівною 0,95, при цьому всі параметри Grid-системи змінювалися за законом Пуассона.

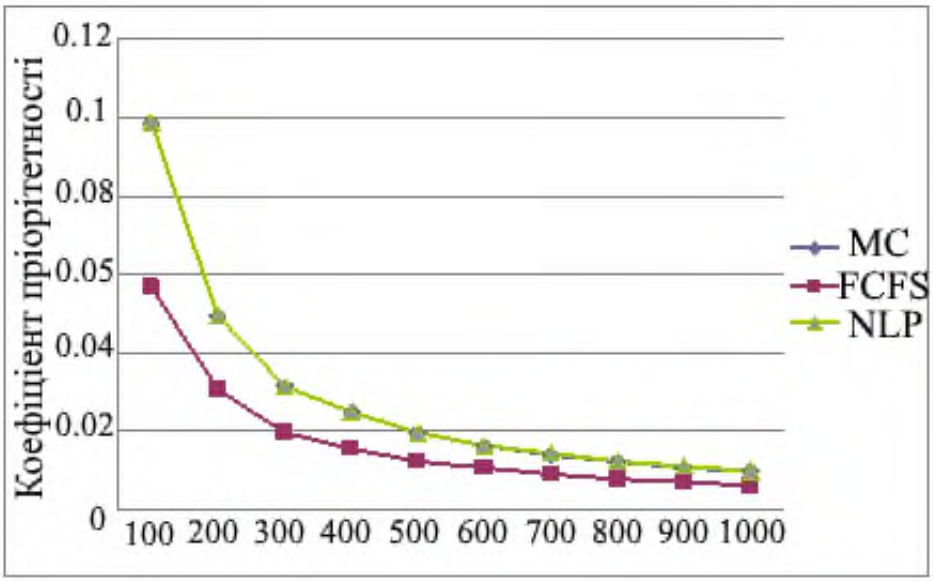



Рис. 3.12. Залежність коефіцієнта пріоритетності від числа завдань у черзі

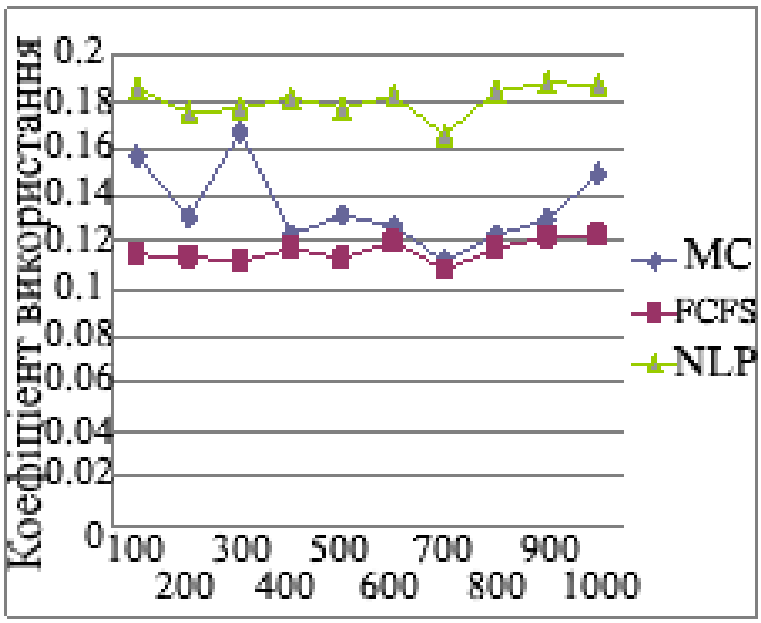



Рис. 3.13. Залежність коефіцієнта використання ресурсів від числа завдань у черзі.

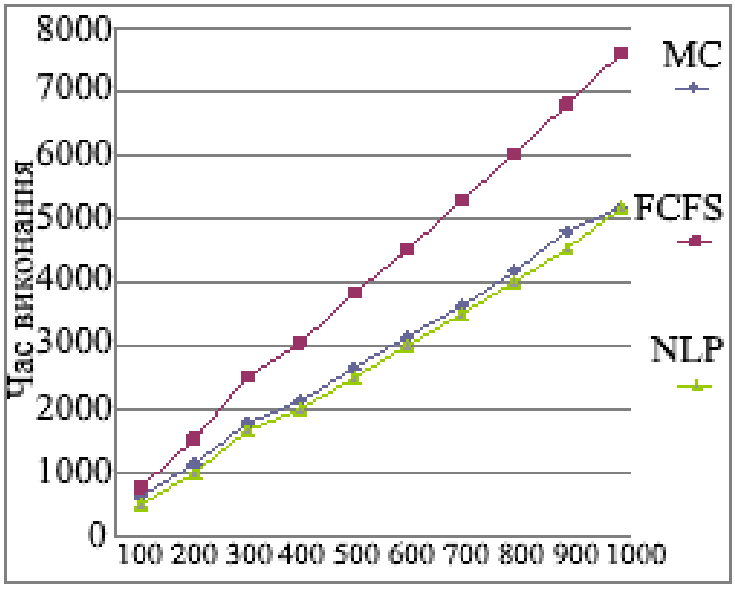



Рис. 3.14. Залежність часу виконання всієї черги завдання від числа завдань у черзі

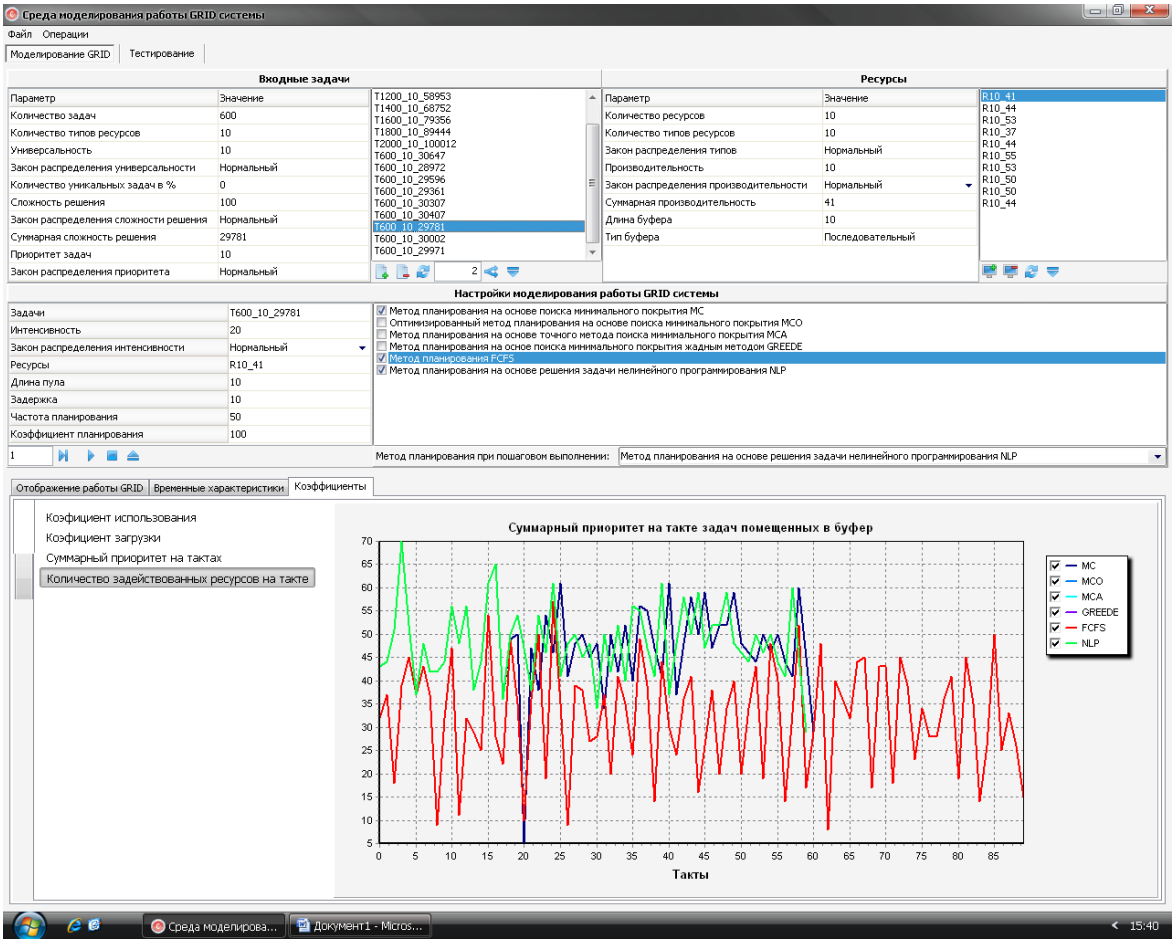



Рис. 3.15. Зміни сумарного пріоритету виконаних завдань на різних тактах планування

З рисунків 3.12-3.14 видно, що при використані для планування процедур, що базуються на вирішенні задачі нелінійного програмування, коефіцієнт використання і коефіцієнт пріоритетності вище ніж у процедур планування на основі рішення ЗНП і процедур FCFS. При цьому оперативність виконання черги завдань вище у процедури NLP.

### 3.8 Висновки за розділом

1. У розділі представлена модель, яка дозволяє користувачеві в ручному режимі змінювати параметри роботи Grid-системи, включаючи методи планування, які і є предметом дослідження даної моделі і отримувати результат у докладної формі з графіками. Також наведено математичний опис досліджуваних характеристик.

2. Як видно з результатів моделювання, використання в якості алгоритму планування запропонованого в роботі методу розв'язання задачі нелінійного булевого програмування дозволяє істотно збільшити такі показники, як коефіцієнт важливості, коефіцієнт збереження важливості, коефіцієнт прискорення і зменшити час виконання завдань і час очікування у порівнянні з відомими процедурами планування.

# РОЗДІЛ 4

# РЕКОМЕНДАЦІЙ ЩОДО ІНТЕГРАЦІЇ РОЗРОБЛЕНОГО МЕТОДУ І МОДЕЛІ ПЛАНУВАННЯ РОЗПОДІЛУ ЗАВДАНЬ В СИСТЕМИ УПРАВЛІННЯ СУЧАСНИХ І ПЕРСПЕКТИВНИХ ТЕЛЕКОМУНІКАЦІЙНИХ МЕРЕЖ

## 4.1 Загальна постановка задачі планування в розподіленому обчислювальному середовищі

Як показано в роботах [22-25] модель планування може бути представлена у вигляді кортежу:

*Schedule = (T, R, ComLinkThroughout, MappingEvent, F(Map), SchedMethod, ResChar, TaskChar, StratSchedul(Focus), StratExec, ObjectFunc, Mode, NoLevel.*

Де *T* – множина завдань глобального потоку; *R* – множина ресурсів; *ComLinkThroughout* – множина комунікаційних каналів зв'язку і їх пропускної спроможності між завданнями і ресурсами; *MappingEvent* – час відображення завдань на ресурси; *F(Map)* – функція відображення завдань на ресурси; *SchedMethod* – метод планування виконання завдань; *ResChar* – множина характеристик ресурсів *R*; *TaskChar* – множина характеристик завдань *T*; *StratSchedul(Focus)* – множина стратегій планування завдань; *StratExec* – множина стратегій виконання завдань на ресурсах; *ObjectFunc* – множина цільових функцій; *Mode* – множина різних режимів планування; *NoLevel* – рівень планування.

Множина *T* визначає множину глобального потоку завдань, що надійшли на обробку в РОС:

$$T=(T_1,T_2,T_3,\ldots,T_n).$$

Множина *R* визначає множину всіх ресурсів РОС:

$$R=(R_1,R_2,R_3,\ldots,R_n).$$

Час відображення *MappingEvent* завдань на ресурси визначається мінімальною кількістю завдань для планування, що визначаються кількістю доступних ресурсів, і/або мінімальним часом для процедури моніторингу стану ресурсів і завдань системи.

Функція відображення $F$ завдань $T$ на ресурси $R$ системи представляє матрицю відповідності:

$$F(Matching): T^*R^* ComLinkThroughout \rightarrow R^+,$$

де *Matching* – матриця відповідності завдань $T$, що плануються ресурсами РОС з урахуванням пропускної здатності безлічі комунікаційних каналів зв'язку *ComLinkThroughout* між спланованими завданнями і ресурсами РОС.

Множина методів планування визначає методи планування завдань на ресурси. Множина стратегій планування завдань визначає стратегії планування, що описуються кортежем:

$$StratSchedul(Focus)=(SystemOrient, UserOrient),$$

де *SystemOrient* – системноорієнтована стратегія планування, орієнтована (сфокусована) на підвищення продуктивності і пропускної здатності РОС (рівень метапланувальника потоків завдань); *UserOrient* – стратегія планування завдань, орієнтована (сфокусована) на користувачів (рівень додатків (локального менеджера ресурсів або локального планувальника)).

Множина стратегій *StratExec* виконання завдань $T$ на виділених для них з множини ресурсів $R$ описується кортежем:

$$StratExec=(Deadline, Budget),$$

де *Deadline* – директивні терміни виконання завдань; *Budget* – обмеження на бюджет (вартість) виконання завдань.

Множина цільових функцій *ObjectFunc*, що характеризують роботу РОС описується кортежем:

$$ObjectFunc=(Utilization, LoadBalance, Time(Cost), Makespan, Penalty),$$

де *Utilization(NoResources, ResUtiliz)* – кількість задіяних ресурсів *NoResources* і значення коефіцієнта використання *ResUtiliz* ресурсів *R* системи; *LoadBalance* – балансування завантаження ресурсів *R*; *Time(Cost)* – час (вартість) виконання завдань глобальної черги; *Makespan* – максимальний час завершення виконання завдань *T* глобальної черги на ресурсах *R*; *Penalty* – розмір штрафів за перевищення допустимого часу виконання або директивного терміну виконання завдань.

Множина режимів планування виконання завдань в РОС описується кортежем:

$$Mode=(Batch, Online),$$

де *Batch* – пакетний режим виконання завдань; *Online* – режим планування та виконання завдань безпосередньо після їх надходження в РОС на обробку.

Множина рівнів планування виконання завдань *NoLevel* описується кортежем:

$$NoLevel=(Global, Local),$$

де *Global* – глобальний рівень планування завдань (рівень метапланувальника потоків завдань); *Local* – локальний рівень планування завдань, рівень локальних планувальників (планувальників систем управління локальними ресурсами) пакетів локальних завдань.

Множина *ResChar* визначає множину характеристик ресурсів *R* системи, що описуються кортежем:

$$ResChar=(NoCores, NoProcessor, MinDisk, MinMemory, OS, MinCPUFreq,$$
$$MaxCPUFreq, MinCPUPower, MaxCPUPower, Architecture,$$
$$ComLinkThroughout\ (R_j)),$$

де *NoCores* – кількість процесорних ядер ресурсу; *NoProcessor* – кількість процесорів ресурсу; *MinDisk* – обсяг дискової пам'яті ресурсу; *MinMemory* – обсяг оперативної пам'яті ресурсу; *OS* – тип операційної системи, що використовується на ресурсі; *MinCPUFreq* – мінімальна частота процесора ресурсу; *MaxCPUFreq* – максимальна частота процесора ресурсу; *MinCPUPower* – мінімальне енергоспоживання процесором; *MaxCPUPower* –

максимальне енергоспоживання процесором; *Architecture* – тип архітектури ресурсу; *ComLinkThroughout* ($R_j$) – пропускна здатність комунікаційних каналів зв'язку до ресурсу $R_j$.

Множина *TaskChar* визначає множину характеристик завдань системи, що описуються кортежем:

*TaskChar=(TaskResourceRequirements(TaskType, NoCores, minDisk, minMemory, Software, Hardware, OS), walltime, deadline, budget),*

де *TaskResourceRequirements* – ресурсні вимоги завдання; *TaskType* – тип завдань, паралельні (*Parallel*), послідовні (*Serial*); *NoCores* – необхідна для виконання кількість ядер; *minDisk* – мінімальний обсяг дискової пам'яті; *minMemory* – мінімальний обсяг оперативної пам'яті; *Software* – необхідне програмне забезпечення; *Hardware* – необхідне апаратне забезпечення; *OS* – необхідна операційна система; *walltime* – очікуваний час виконання завдань на ресурсі; *deadline* – директивний термін виконання завдання; *budget* – обмеження (бюджет) вартості виконання завдання визначається наступним чином:

$$cost(walltime)+penalty(deadline)\leq budget$$

Однією з можливих постановок задач підвищення ефективності функціонування розподіленого обчислювального середовища може бути представлена у наступному варіанті. Потрібно в момент часу *MappingEvent* побудувати *SystemOrient* – розклад виконання глобального потоку завдань $T$ на ресурсах $R_j \subseteq R$, які використовують пакетний режим планування завдань *Batch*, на основі розробленого методу планування *SchedMethod* (метапланувальником) рівня *Global*, яке визначається виразом:

$$F:T\rightarrow R\times ComLinkThroughout,$$

що забезпечує виконання:

$$ResUtiliz\ R_j\rightarrow max, j=\overline{1,n}, \qquad (4.1)$$

за умов:

$$TaskResourceRequirements\subseteq ResChar,$$

$$\sum_{j=1}^{k} R_j \to min,\ \cup R_j \subseteq R.$$

$$r_i \to max,$$

де $r_i$ коефіцієнт важливості завдань, що виконуються в кластерах Grid-системи.

Розглянемо загальну концепцію диспетчеризації в Grid для вирішення даного завдання, що дозволяє на основі розробленого методу планування та відомого [71], що базується на основі рішення задачі про найменше покриття, підвищити ефективність використання ресурсів в глобальному розподіленому обчислювальному середовищі.

## 4.2 Загальна концепція створення управління диспетчеризацією в Grid

Для вирішення завдання (4.1) передбачається наявність чотирьох рівневої ієрархічної структури управління, що поєднує в собі централізоване і децентралізоване управління. Перший рівень управління повинен забезпечити координацію перерозподілу ресурсів регіонів глобальної мережі. Другий рівень повинен забезпечити необхідну якість обслуговування динамічно мінливих віртуальних спільнот в мережі в рамках регіону, надаючи їм можливість використовувати всі ресурси мережі. Третій рівень, незалежно від першого рівня, на основі загального пулу завдань здійснює перерозподіл між кластерами. І четвертий рівень здійснює планування в кластерах Grid-системи. Для цього в мережі доцільно мати центральний пункт управління, основними функціями якого є обслуговування загального пулу завдань і координації роботи диспетчерів другого рівня, не втручаючись в процес планування, але надаючи диспетчерам другого рівня необхідну

інформацію для процесу планування. Диспетчери другого рівня самі вибирають завдання із загального пулу завдань і можуть пересилати їх один одному і відправляти їх на рішення в певний кластер. Після того, як віртуальні спільноти сформовані і відправили в центральний пункт управління свої завдання, він пропонує на певних умовах можливості приєднання додаткових ресурсів мережі, які залишаються не задіяні в мережі. При цьому передбачається, що всі бажаючі надавати свої ресурси сповіщають про це центральний пункт управління. Посередницька діяльність центрального пункту управління дозволить розвантажити диспетчерів другого рівня від процесу аналізу і обробки стану всіх ресурсів мережі і повноцінно займатися тільки плануванням розподілу ресурсів через загальний пул завдань у мережі. На міжрегіональному рівні центральні пункти управління кожного регіону взаємодіють між собою, утворюючи розподілений центральний пункт управління глобальної мережі, що відповідає четвертому рівню управління, який створює потоки інформації. Розглянемо принципи побудови Grid-систем, в яких розподіл завдань здійснюється в гетерогенному обчислювальному середовищі і вони в загальному випадку повинні забезпечити вирішення таких завдань як: визначення призначення кожного компонента архітектури Grid; визначення загальних принципів взаємодії компонентів архітектури Grid; створення математичного та програмного забезпечення гарантує ефективне і надійне функціонування Grid-систем. Будемо розглядати Grid-систему як гетерогенну середу, в якій кожен ресурс $R_i$ характеризується вектором характеристик $(\alpha_1^{R_i}, \alpha_2^{R_i}, ..., \alpha_q^{R_i})$, де кожне завдання $З_i$ теж будемо характеризувати характеристиками $(\alpha_1^{З_i}, \alpha_2^{З_i}, ..., \alpha_q^{З_i})$. Завдання $З_i$ може бути виконане на ресурсі $R_i$, якщо виконуються нерівності $\alpha_1^{R_i} \le \alpha_1^{З_i}; \alpha_2^{R_i} \le \alpha_2^{З_i}; ..., \alpha_q^{R_i} \le \alpha_q^{З_i}$. Розподіл завдань розглянемо на основі принципу роздільного розподілу завдань, схема якого наведена на рисунку 4.1, але при цьому розподіл за допомогою диспетчера здійснюватися буде не статично, як це робиться в відомих роздільних схемах

розподілу завдань, а динамічно і безперервно на основі наступної процедури *D*.

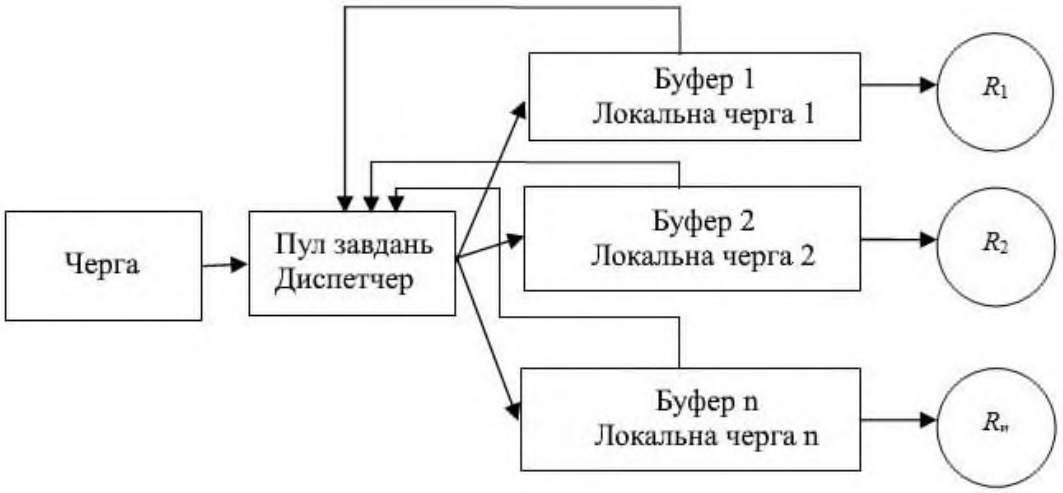



Рис. 4.1. Роздільний розподіл завдань в гетерогенному середовищі

Вихідними даними для функціонування системи є відомості про завдання та вільні ресурси, на основі яких в диспетчері створюється таблиця відповідності або як її називають *Matching* – матриця відповідності завдань *T*, що плануються ресурсами *R* РОС з урахуванням пропускної здатності безлічі комунікаційних каналів зв'язку *ComLinkThroughout* між спланованими завданнями і ресурсами РОС.

Процедура *D*. У диспетчері створюється таблиця відповідності, стовпцям якої відповідають ресурси $\{R_i\}$, рядкам завдання зі сформованої черги завдань. На перетині рядка і стовпця будемо ставити одиницю, якщо з порівняння $(\alpha_1^{R_i},\alpha_2^{R_i},...,\alpha_q^{R_i})$ і $(\alpha_1^{З_i},\alpha_2^{З_i},...,\alpha_q^{З_i})$ витікає, що дана задача може бути реалізована на наявному вільному ресурсі і нуль в іншому випадку.

Надалі при плануванні пропонується використовувати одну з двох наступних процедур. Перша заснована на рішенні задачі про найменше покриття (ЗНП), знаходимо мінімальне число ресурсів, на якому сформована черга завдань може бути виконана і відправляємо завдання на рішення в

виділені ресурси. Далі черга поповнюється новими завданнями і процедура повторюється, ті завдання, для яких ми в таблиці маємо нульові рядки, повертаються назад в чергу і робиться повідомлення адміністратору про неможливість виконання даного завдання ні на одному ресурсі. Математичною моделлю даної процедури є завдання лінійного булевого програмування, при обмеженнях $\sum_{j=1}^{n}\beta_{ji}x_j(t_k)\geq 1; i=(\overline{1,m})$; $\beta_{ji}\in\{0,1\}$; $x_j(t_k)\in\{0,1\}$, де $m$ – кількість завдань, що підлягають плануванню; $n$ – кількість ресурсів системи, доступних і вільних на момент планування $t_k$ $[T_0, T_n]$. Планування здійснюється на інтервалі часу $[T_0, T_n]$, де $T_0$ – час початку планування; $T_n$ – час завершення планування завдань. Завдання можна розглядати як задачу визначення мінімального числа стовпців в булевій матриці $B$, що показує всі рядки даної матриці, елементи якої в контексті вирішення завдання планування інтерпретуються наступним чином: стовпчикам відповідають доступні і вільні на момент планування ресурси розподіленої обчислювальної системи, а рядкам - завдання, що підлягають планування, які повинні бути вирішені на цих ресурсах. Особливістю тут є те, що розстановка одиниць в матриці $B$ динамічно змінюється.

Друга процедура базується на методі групової вибірки з сегментацією, що був розглянутий в розділі 2. Метод базується на основі рішення задачі нелінійного булевого програмування, при реалізації якого з черги завдань обслуговується кілька завдань одночасно. Вибираються завдання, які вимагають для реалізації ресурси різних типів і щоб сума їх пріоритетів була максимальною. У разі наявності рівнозначних завдань вибирають більш «старі». Тому необхідно вибрати з черги якомога більшу кількість завдань, які використовують різні типи ресурсів і сума пріоритетів обраних завдань повинна бути максимальна. Причому прагнення до максимуму суми пріоритетів обраних завдань є головним критерієм при виборі завдань з черги. Фактично за рахунок рішення ЗНП брокери другого рівня забезпечують виконання завдань мінімальним числом ресурсів, а брокери

нижнього рівня забезпечують максимізацію коефіцієнта важливості виконуваних завдань в кластерах Grid-системи. При моніторингу стану ресурсів мережі її доцільно розбити на зони, в межах яких керуючі сервери можуть здійснювати моніторинг вільних ресурсів групи кластерів $\{K_i\}$, рисунок 4.2, підключених до даного керуючого сервера за час $T$, що не перевищує деякий допустимий час $T_\partial$. Черга завдань знаходиться в буфері сервера.

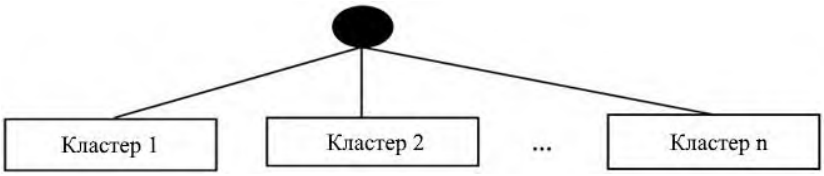



Рис. 4.2. Кластери зони $\{K_i\}$, які можуть бути опитані за час $T \le T_\partial$

Сервери обмінюються інформацією про наявність вільних ресурсів в своїх зонах. Якщо в зоні сервера немає вільних ресурсів в кластерах, то завдання передається на керуючий сервер з максимальним числом вільних ресурсів в кластерах його зони $\{K_i\}$, при цьому пріоритет цих завдань доцільно підвищувати для того, щоб уникнути ситуації, коли деякі завдання можуть дуже довго залишатися не обслугованими. Черга завдань знаходиться в буфері керуючого сервера зони, при цьому кожна задача характеризується вектором показників $(\alpha_1^{З_i}, \alpha_2^{З_i}, ..., \alpha_q^{З_i})$, в якості яких може виступати обсяг пам'яті, необхідний для вирішення завдання, вимоги до операційної системи ресурсу, договірна ціна виконання завдання на ресурсі тощо. Кожен кластер $\{K_i\}$ надає на керуючий сервер зони дані про наявність вільних ресурсів в тому ж форматі, що і для завдань $(\alpha_1^{R_i}, \alpha_2^{R_i}, ..., \alpha_q^{R_i})$. З буфера сервера формується пул завдань, який потрібно відправити на виконання в кластери $\{K_i\}$. На основі відомостей про завдання і вільних ресурсах в базі даних керуючого сервера даної зони створюється таблиця відповідності, стовпцям якої

відповідають кластери $\{K_i\}$, що знаходяться в зоні дії керуючого сервера, а рядкам завдання з сформованого пулу завдань. На перетині рядка і стовпця будемо ставити одиницю, якщо з порівняння характеристик $(\alpha_1^{3_i}, \alpha_2^{3_i}, ..., \alpha_q^{3_i})$ і $(\alpha_1^{R_i}, \alpha_2^{R_i}, ..., \alpha_q^{R_i})$ витікає, що дана задача може бути реалізована на ресурсі кластера $K_i$, що є в наявності і нуль в іншому випадку. Далі на основі рішення ЗНП знаходимо мінімальне число кластерів в зоні, на якому сформований пул завдань може бути виконаний і відправляємо його на рішення в виділені кластери. Керуючий сервер і множини кластерів $\{K_i\}$ в зоні утворюють дворівневу систем, як показано на рисунку 4.3.

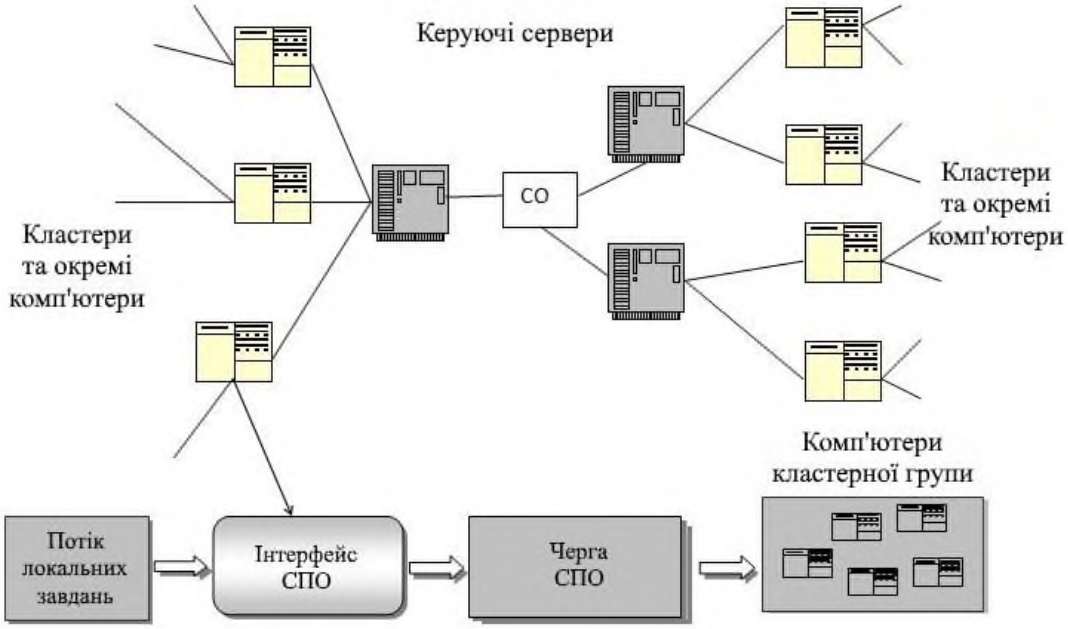



Рис. 4.3. Архітектура дворівневої Grid-системи

На першому рівні керуючі сервери обмінюються інформацією про наявність вільних ресурсів по середовищу обміну інформацією і якщо в зоні сервера немає вільних ресурсів, то він відправляє завдання на рішення на керуючий сервер, в зоні якого є найбільше число вільних ресурсів. Узагальнену структуру взаємодії брокерів верхнього рівня можна представити в наступному вигляді, рисунок 4.4.

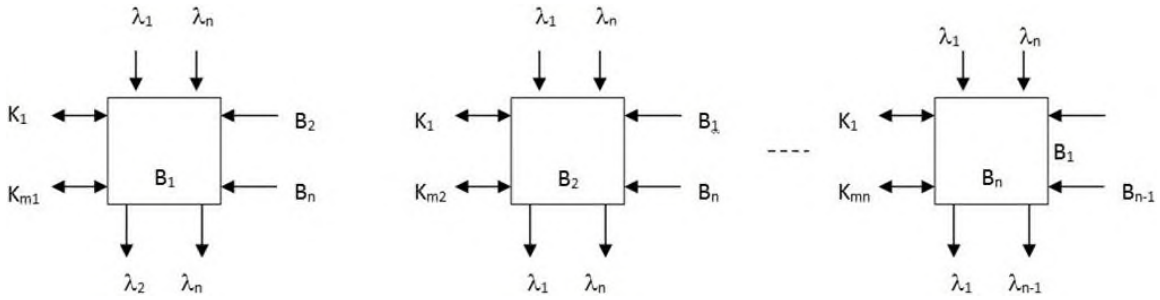



Рис. 4.4. Модель взаємодії брокерів верхнього рівня

Формування завантаження брокерів $\{B_i\}$ здійснюється на основі таблиці відповідності, що зберігається в базі даних (БД), стовпцям якої відповідають типи вільних ресурсів, а рядкам типи завдань, які можуть бути обслужені на даних типах ресурсів. Якщо число завдань в стовпці таблиці перевищує число ресурсів, то в стовпчиках таблиці залишаємо тільки ті завдання, які мають максимальний пріоритет. Особливістю роботи гетерогенних систем є те, що деякі завдання можуть бути виконані тільки на певних типах ресурсів. Тому при довільному розміщенні завдань в черзі за таблицею відповідності може виникнути конкуренція деяких завдань до одного ресурсу, що призводить до нерівномірності завантаження ресурсів і суттєвій затримці виконання конкуруючих завдань. Завдання бажано розміщувати в черзі на попередньому кроці планування таким чином, щоб підготувати на наступний етап максимально більшу кількість ресурсів для виконання наступної підмножини завдань. І тому пропонується процедура перетворення загальної черги в локальні на основі безперервної реалізації процедури *D*. Тобто на кожному кроці планування локальних черг визначається мінімальне число брокерів, на якому можна виконати всі завдання, вміщені диспетчером в пул завдань, що досягається рішенням ЗНП, і в чергу поміщаються завдання до групи брокерів, які можуть виконати весь пул завдань. Причому спочатку заповнюється черга до брокеру, на якому буде вирішуватися більше число завдань і якщо черга починає перевищувати

обсяг буфера, то переміщаємо завдання на наступний брокер даної групи, який може його виконати. Якщо таких брокерів в групі немає, то за таблицею відповідності вибираємо будь-який з брокерів, на якому завдання даного типу може бути виконано, тобто реалізується процедура *D*. Якщо завдання може бути реалізовано тільки на тому брокері, до якого локальна черга перевищує розміри буфера, то завдання відправляється назад в пул завдань. Формування черг до кластерів, підключеним до брокера, здійснюється таким же чином на основі безперервної реалізації процедури *D*. У загальному вигляді структура брокера верхнього рівня має вигляд, як показано на рисунку 4.5.

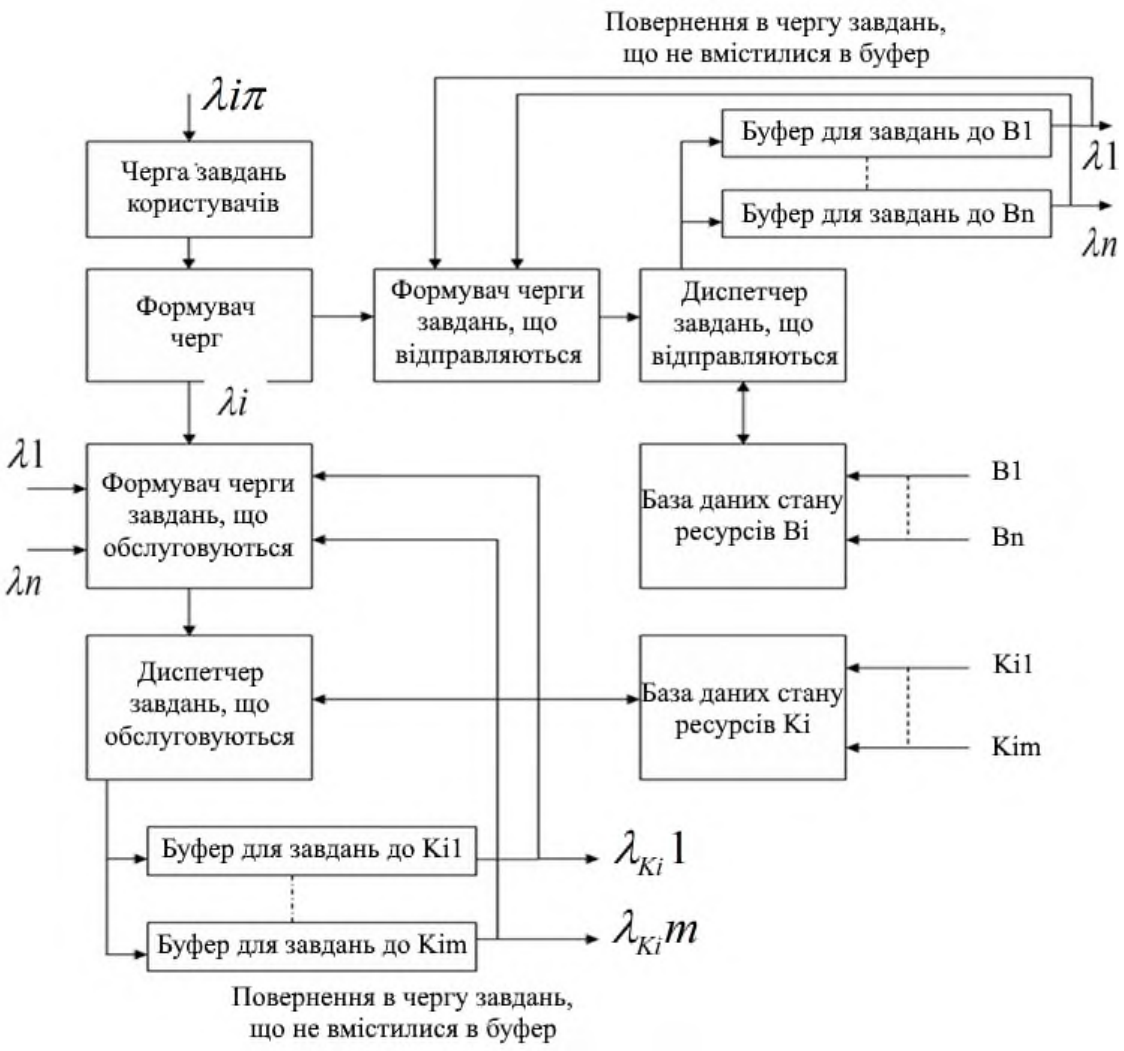



Рис. 4.5. Структура брокера $B_i$ верхнього рівня

Завдання, що надійшли на брокер, розбиваються на дві черги: черга завдань, яка буде вирішуватися кластерами, підключеними до даного брокеру, і черга завдань, які за характеристиками ресурсів і типам вирішуваних завдань, не можуть бути реалізовані на ресурсах кластерів, підключених до даного брокеру. Формування обох черг здійснюється незалежно на основі застосування процедури *D*.

### 4.3 Моніторинг ресурсів в розподілених середовищах

Рішення задач моніторингу РОС є важливою з точки зору забезпечення необхідної при обробці потоків завдань продуктивності і пропускної спроможності. Слід зазначити, що найбільш поширені системи моніторингу Nagios [11], Icinga [24] використовують програмні розширення (агенти), що встановлюються на об'єктах моніторингу для їх віддаленого запуску і реалізують різні сервіси. Для виконання сервісів на вузлах РОС необхідно встановити плагін NPRE (Nagios Remote Plugin Executor), який започатковує роботу програмних агентів. Процедура роботи віддаленого сервісу, що викликається, включає: ініціалізацію командою запуску, здійснюваної агентами NRPE на серверах і вузлах РОС; запуск і виконання сервісу; отримання результатів роботи сервісу; передачу отриманих даних на керуючий вузол (базу даних). Для комплексної оцінки стану програмно-апаратних засобів, комунікаційних компонент і виконуваних завдань РОС слід використовувати сервіси віддаленого доступу, безпосередньо пов'язані з вирішенням завдань планування розподілених обчислень. До них відносяться [35,36]: доступність і рівень завантаження вузлів (в тому числі багатоядерних процесорів) кластерів; доступність і пропускна здатність комунікаційних каналів (включаючи комунікаційні канали кластерів); кількість і доступність вільних вузлів кластерів; стан виконуваних завдань на вузлах; доступність,

поточна продуктивність і завантаження вузлів, використовуваних для зберігання даних моніторингу. При наявності великої кількості необхідних для якісного моніторингу об'єктів РОС різко збільшується навантаження на комунікаційну мережу, обмежену її пропускною спроможністю. Інформація про стан об'єктів моніторингу є фоновою по відношенню до основної – користувальницької і службової (керуючої) інформації, обсяги яких безпосередньо впливають на рівень забезпечення якості обслуговування користувачів – час обслуговування запитів і додатків, сумарне запізнення, вартість обчислень тощо. Модель реалізації множин зазначених сервісів *Rem_Serv* можна представити в наступному вигляді:

$$Rem\_Serv: AN \times AU \times ANet \times AJ \times ADB \rightarrow \{State1, State2, State3, State4\},$$

де *AN* – множина сервісів визначення доступності вузлів; *AU* – безліч сервісів визначення завантаження (використання) вузлів; *AN* – безліч сервісів визначення стану комунікаційних каналів; *AJ* – множина сервісів визначення стану виконуваних завдань; *ADB* – множина сервісів визначення стану БД; $State_i$, $i = \overline{1,4}$ – множина станів об'єкта моніторингу, визначених {*OK, WARNING, CRITICAL, UNKNOWN*}. З огляду на те, що потужності множин розглянутих сервісів для моніторингу складають [11] $|AN| = 2$, $|AU| = 3$, $|ANet| = 3$, $|AJ| = 4$, $|ADB| = 6$, а контролювати системному адміністратору і користувачам віртуальних організацій РОС необхідно, наприклад, тільки два стани об'єктів моніторингу *WARNING* і *CRITICAL*, мінімальна кількість повідомлень, отриманих від віддалених агентів, що беруть участь тільки в одному опитуванні, складе $2^{18}$.

З огляду на те, що розмір одного повідомлення становить близько 4 Кб, такий обсяг службової інформації призводить до значної завантаженості комунікаційних каналів, що призводить до необхідності забезпечення необхідної надійності інформації, що передається за рахунок застосування ефективних методів кодування інформації. Поява конкретних помилок в каналах може істотно знизити якість сервісів, забезпечуваних Grid-

системами. Тому є актуальним розглянути можливості підвищення достовірності передачі інформації при моніторингу ресурсів Grid-систем.

## 4.4 Підвищення достовірності передачі інформації при моніторингу ресурсів Grid-систем

Рішення задач забезпечення завадостійкої і достовірної передачі повідомлень по каналах сучасних телекомунікаційних мереж, таких як Grid, покладається на методи і обчислювальні алгоритми завадостійкого кодування та цифрової обробки сигналів, які повинні функціонувати в умовах обмежених частотно-часових ресурсів і забезпечувати виконання підвищених вимог до якості передачі даних.

Найбільшого поширення в техніці завадостійкого кодування отримали каскадні коди, в конструкції яких використовуються недвійкові коди другого ступеня. Однак математичний апарат багатовимірних спектрів до таких кодів не застосовують, неможливо використовувати і швидкі багатовимірні перетворення Фур'є, тобто отримати той ефект, який дають в техніці завадостійкого кодування перетворення в частотній області. Спектральні властивості узагальнених каскадних кодів на сьогоднішній день також мало досліджені, методи алгебраїчного опису з урахуванням обмежень, що накладаються особливостями дискретних перетворень Фур'є в кінцевих полях Галуа мало вивчені.

Тому для підвищення достовірності передачі інформації при моніторингу ресурсів Grid-систем актуальним є дослідження математичного апарату багатовимірних спектрів для подання каскадних кодових конструкцій в частотній області та реалізації на їх основі ефективних алгоритмів завадостійкого кодування та декодування.

Пропонований у [85] підхід до дослідження спектральних властивостей узагальнених каскадних кодів полягає у використанні обмежень недвійкових кодових слів на двійкове підполе і дослідженні спектрів цих обмежень. При цьому спектральне подання недвійкових кодових слів основного коду і спектральне подання відповідних слів коду-обмеження безпосередньо пов'язані і мають сувору алгебраїчну структуру, яка визначається груповими властивостями елементів кінцевого поля.

Запропонований підхід до визначення спектральних властивостей основного коду за відомим спектром кодових слів його двійкового коду-обмеження є, по суті, подальшим розвитком математичного апарату перетворення Фур'є в кінцевих полях. Його практичне використання дозволяє застосувати багатовимірні спектральні перетворення при дослідженні алгебраїчних і частотних властивостей узагальнених каскадних кодів.

Дійсно, використання багатовимірних спектрів над двійковим поданням кодових слів узагальненого каскадного коду дозволяє сформувати спектр коду-добутку двійкового коду першого ступеня і двійкового коду-обмеження основного недвійковий коду другого ступеня.

***Метод опису каскадних кодів в частотній області***

Для вирішення завдання опису каскадних кодів в частотній області необхідно аналітично зв'язати значення спектральних компонент коду зовнішньої ступені з відповідними спектральними компонентами його обмеження на підполі. Тоді математичний апарат багатовимірних спектрів з урахуванням цієї введеної аналітичної зв'язку, вочевидь, дозволить обчислити кодове слово каскадного коду в частотній області.

Структурна схема пропонованого методу опису каскадних кодів в частотній області представлена на рис. 1 [85].

Метод використовує математичний апарат дискретного перетворення Фур'є в кінцевих полях, лінійної алгебри і полів Галуа. В якості вихідних даних використовується відомий опис каскадних кодів в часовій області

через послідовне обчислення кодових слів коду зовнішньої і внутрішньої ступенів. В основі методу лежать наступні процедури:

• уявлення коду зовнішнього ступеню через безліч обмежень кодових слів на довільне підполе;

• обчислення спектра слів коду-обмеження і дослідження його властивостей;

• висновок взаємно-однозначної функціональної залежності спектра кодових слів зовнішнього ступеню зі спектром слів коду-обмеження;

• висновок аналітичних виразів для опису каскадного коду в частотній області.

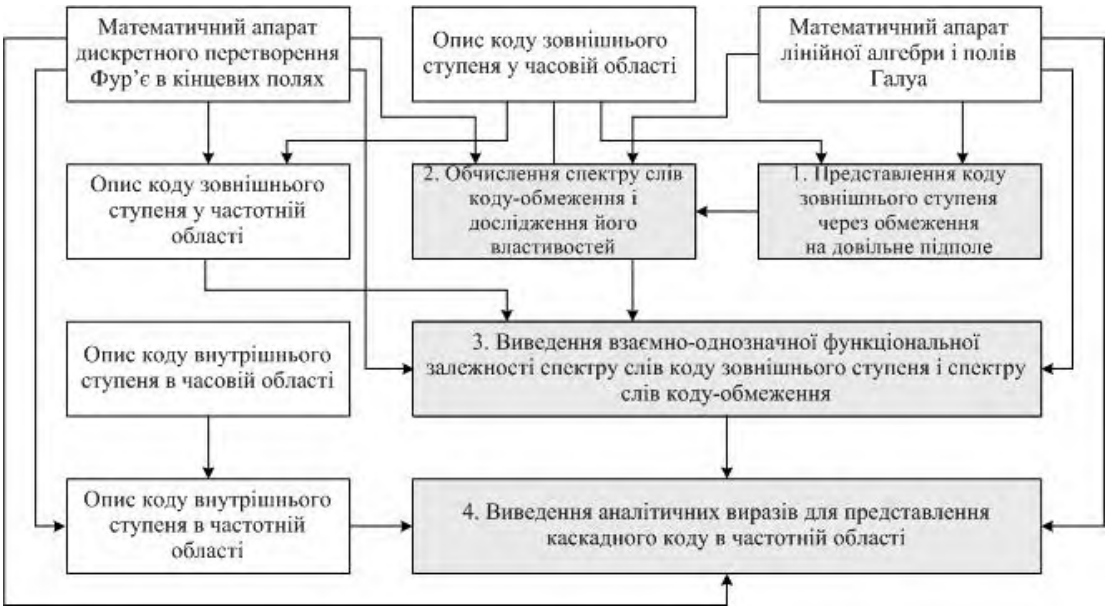



Рис. 4.6. Структурна схема пропонованого методу опису каскадних кодів в частотній області

При вирішенні першого завдання опису каскадних кодів в частотній області встановлено, що спектр $C$ довільного часового вектору $V$ над $GF(q^m)$ є лінійна комбінація спектрів $c_0$, $c_1$,…, $c_{m-1}$ його векторів $v_0$, $v_1$, …, $v_{m-1}$ - обмежень на довільне підполе $GF(q) \subseteq GF(q^m)$. Конкретний вид цієї

лінійної комбінації визначається вибором елемента $\alpha$ - ядра перетворення Фур'є:

$$C = VW = v_0 W + \alpha v_1 W + ... + \alpha^{m-1} v_{m-1} W = c_0 + \alpha c_1 + ... + \alpha^{m-1} c_{m-1}$$

Практично це означає можливість виведення простих (лінійних) виразів для розрахунку спектра довільного часового вектору за відомими спектрами його векторів-обмежень.

Вирішення другого завдання показало, що компоненти спектрів векторів-обмежень довільного часового вектору на довільне підполе будуть виражатися лінійною комбінацією результатів ступеневих відображень компонентів спектра цього вектору.

$$\begin{cases} \left(C_s\right)^{q^m} = c_{0,s} + \alpha^{q^m} c_{1,s} + ... + \alpha^{(m-1)q^m} c_{m-1,s}, \\ \left(C_{sq \bmod N}\right)^{q^{m-1}} = c_{0,s} + \alpha^{q^{m-1}} c_{1,s} + ... + \alpha^{(m-1)q^{m-1}} c_{m-1,s}, \\ ..... \\ \left(C_{sq^{u_s-1} \bmod N}\right)^{q^{m-u_s+1}} = c_{0,s} + \alpha^{q^{m-u_s+1}} c_{1,s} + ... + \alpha^{(m-1)q^{m-u_s+1}} c_{m-1,s}. \end{cases}$$

Дане твердження дозволяє аналітично зв'язати спектр векторів-обмежень довільного кодового слова зі спектром цього кодового слова.

Аналітичне рішення третього завдання встановило такі закономірності:

• кодове слово каскадного коду є лінійна комбінація векторів-обмежень кодового слова зовнішньої ступені;

• багатомірний спектр багатовимірного слова є результат багаторазового обчислення одновимірного спектра до всіх одновимірним уявленням цього слова;

• спектр кодового слова каскадного коду є, в порядковому запису, безліччю результатів дворазового обчислення одновимірного спектра до всіх лінійним комбінаціям векторів-обмежень кодового слова зовнішньої ступені;

• компоненти спектра довільного кодового слова каскадного коду визначаються лінійною комбінацією результатів ступеневих відображень компонентів спектра кодового слова коду зовнішньої ступені.

Результат останнього твердження дозволяє аналітично виразити спектр довільного кодового слова каскадного коду через функціональну відповідність елементам спектра кодового слова коду зовнішньої ступені, тобто вирішити спільне завдання опису каскадних кодів в частотній області.

Таким чином, в результаті проведених досліджень отримано спільне рішення задачі уявлення каскадних кодів в частотній області, що дозволяє, використовуючи виведені аналітичні залежності компонентів багатовимірних спектрів, будувати в частотної області обчислювально ефективні алгоритми кодування і декодування [81]. Найбільш перспективним у цьому сенсі є використання швидких багатовимірних перетворень Фур'є.

**Алгоритми кодування / декодування каскадними кодами в частотній області**

Реалізація алгоритмів кодування і декодування каскадних кодами з використанням швидкого перетворення Фур'є дозволяє істотно скоротити обчислювальну складність перетворень і підвищити, таким чином, оперативність обробки інформації.

В результаті проведених в [81] досліджень було встановлено, що компоненти (в часовій області) кодового слова:

$$v = \begin{pmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,K-1} & v_{0,K} & v_{0,K+1} & \cdots & v_{0,N-1} \\ v_{1,0} & v_{1,1} & \cdots & v_{1,K-1} & v_{1,K} & v_{1,K+1} & \cdots & v_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{m-1,0} & v_{m-1,1} & \cdots & v_{m-1,K-1} & v_{m-1,K} & v_{m-1,K+1} & \cdots & v_{m-1,N-1} \\ v_{m,0} & v_{m,1} & \cdots & v_{m,K-1} & v_{m,K} & v_{m,K+1} & \cdots & v_{m,N-1} \\ v_{m+1,0} & v_{m+1,1} & \cdots & v_{m+1,K-1} & v_{m+1,K} & v_{m+1,K+1} & \cdots & v_{m+1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{n-1,0} & v_{n-1,1} & \cdots & v_{n-1,K-1} & v_{n-1,K} & v_{n-1,K+1} & \cdots & v_{n-1,N-1} \end{pmatrix} \quad (4.2)$$

каскадного $(Nn, Kk, Dd)$ коду над $GF(q)$ визначаються лінійною комбінацією результатів ступеневих відображень компонентів спектра

$$C = (C_0, C_1, C_2, ..., C_{N-1}), \quad (4.3)$$

кодового слова коду зовнішньої ступені, $C_i \in GF(q^m)$.

Фактично це означає, що всі елементи кодового слова каскадного коду можуть бути аналітично обчислені за заданими інформаційними частотами алгебраїчного коду зовнішньої ступені.

Позначимо символом

$$C_{Inf} = \{C_{i_0}, C_{i_1}, C_{i_2}, ..., C_{i_{K-1}}\}, \quad (4.4)$$

безліч інформаційних частот, $C_{i_j} \in GF(q^m)$.

Номера $i_j$ інформаційних частот, що входять в безліч $C_{Inf}$, вибираються, виходячи з алгебри структури коду зовнішньої ступені. Ці $K$ інформаційні частоти задають всі інші компоненти спектра (4.2) кодового слова коду зовнішньої ступені й аналітично задають всі компоненти спектру

$$c = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,K-1} & c_{0,K} & c_{0,K+1} & \cdots & c_{0,N-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,K-1} & c_{1,K} & c_{1,K+1} & \cdots & c_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ c_{m-1,0} & c_{m-1,1} & \cdots & c_{m-1,K-1} & c_{m-1,K} & c_{m-1,K+1} & \cdots & c_{m-1,N-1} \\ c_{m,0} & c_{m,1} & \cdots & c_{m,K-1} & c_{m,K} & c_{m,K+1} & \cdots & c_{m,N-1} \\ c_{m+1,0} & c_{m+1,1} & \cdots & c_{m+1,K-1} & c_{m+1,K} & c_{m+1,K+1} & \cdots & c_{m+1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ c_{n-1,0} & c_{n-1,1} & \cdots & c_{n-1,K-1} & c_{n-1,K} & c_{n-1,K+1} & \cdots & c_{n-1,N-1} \end{pmatrix} \quad (4.5)$$

каскадного $(Nn, Kk, Dd)$ коду і, відповідно, компоненти кодового слова (4.2) в часовій області.

Таким чином, алгоритм обчислення кодового слова (4.2) каскадного коду над $GF(q)$ задаємо такою послідовністю кроків.

***Алгоритм кодування каскадними кодами в частотній області [88]:***

Крок 1. Ввести значення інформаційних частот (4.4).

Крок 2. Сформувати спектр кодового слова коду зовнішньої ступені (4.3).

Крок 2.1. Якщо поле символів коду зовнішньої ступені збігається з полем компонент спектра (наприклад, для РС кодів), тоді всі не інформаційні частоти прийняти рівними нулю.

Крок 2.2. Якщо поле символів коду зовнішньої ступені не збігається з полем компонент спектра (наприклад, для БЧХ кодів), тоді всі перевірочні частоти прийняти рівними нулю, інші частоти вирахувати з використанням обмежень пов'язаності $V^q{}_j = V_{((qj))}$, $j = 0, ..., n-1$.

Крок 3. Обчислити всі компоненти спектру (4.5) каскадного коду.

Крок 3.1. Обчислити елементи $\left(C_{sq^w \bmod N}\right)^{q^{m-w}}$ для всіх $w = 0, ..., u_s - 1$, де $u_s$ - число в хорді $A_s$ поля $GF(q^m)$, $s$ - позитивне ціле, що пробігає всі

ступені примітивного елемента з розкладання поля $GF(q^m)$ на класи $\{\alpha^s, \alpha^{s^q}, ..., \alpha^{s^{q^{u_s}}}\}$ так, що $\sum_s u_s = q^m - 2$, $\#s = u$.

Крок 3.2. Використовуючи лінійну комбінацію $\left(C_{sq^w \bmod N}\right)^{q^{m-w}}$, яка визначається структурою алгебри коду внутрішньої ступені, обчислити всі $c_{i,j}$, $i = 0,1,..., n-1$, $j = 0,1,..., N-1$ в (4.5).

Крок 4. Виконати зворотне двовимірне перетворення Фур'є спектру.

Запропонований алгоритм легко узагальнюється на випадок багатовимірних каскадних кодів. У цьому випадку, після обчислення спектра кодового слова коду зовнішньої ступені і формування ступеневого відображення $\left(C_{sq^w \bmod N}\right)^{q^{m-w}} = \varphi\left(C_{sq^w \bmod N}\right)$, $w = 0,..., u_s - 1$, крок 3.2 повторюється для кожної «мірності» каскадного коду. Сформований таким чином багатовимірний спектр виду:

$$c_{j_1, j_2, ..., j_p} = \sum_{i_1=0}^{n_1-1} \sum_{i_2=0}^{n_2-1} ... \sum_{i_p=0}^{n_p-1} \alpha_1^{i_1 j_1} \alpha_2^{i_2 j_2} ... \alpha_p^{i_p j_p} v_{i_1, i_2, ..., i_p},$$

де $\alpha_1$, $\alpha_2$, …, $\alpha_p$ – елементи кінцевого поля $GF(q^m)$ порядку $n_1$, $n_2$, …, $n_p$, відповідно, за допомогою зворотного багатовимірного перетворення Фур'є перетворюється в багатовимірне кодове слово

$$v_{i_1, i_2, ..., i_p} = \frac{1}{n_1} \frac{1}{n_2} ... \frac{1}{n_p} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} ... \sum_{j_p=0}^{n_p-1} \alpha_1^{-i_1 j_1} \alpha_2^{-i_2 j_2} ... \alpha_p^{-i_p j_p} c_{j_1, j_2, ..., j_p}.$$

Для обґрунтування практичних рекомендацій з використання розробленого алгоритму кодування каскадними кодами в частотної області в даній роботі пропонується структурна схема пристрою, наведена на рис. 4.7.

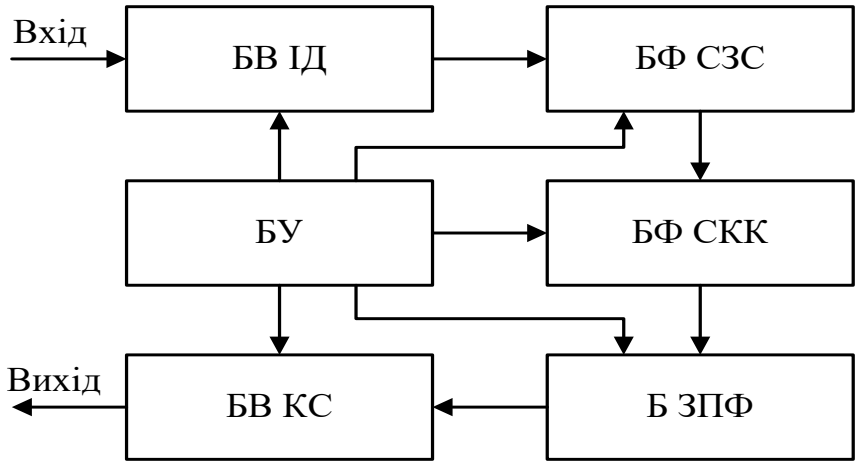



Рис. 4.7. Структурна схема пристрою кодування каскадними кодами в частотній області

Пристрій, схема якого наведена на рис. 4.7, містить:

БВ ІД – блок введення інформаційних даних;

БФ СЗС – блок формування спектра кодового слова зовнішньої ступені;

БФ СКК – блок формування спектра кодового слова каскадного коду;

Б ЗПФ – блок зворотного перетворення Фур'є;

БВ КС – блок виведення кодового слова каскадного коду;

БУ – блок узгодження.

Пристрій побудований таким чином. Вхід пристрою з'єднаний з першим входом БВ ІД, вихід якого з'єднаний з першим входом БФ СЗС. Вихід БФ СЗС з'єднаний з першим входом БФ СКК, вихід якого з'єднаний з першим входом Б ЗПФ. Вихід Б ЗПФ з'єднаний з першим входом БВ КС, вихід якого з'єднаний з входом пристрою. П'ять виходів БУ з'єднані з другими входами БВ ІД, БФ СЗС, БФ СКК, Б ЗПФ, БВ КС, відповідно.

Пристрій працює наступним чином. На вхід пристрою подаються інформаційні дані, що підлягають завадостійкому кодуванню. В БВ ІД подані дані записуються в інформаційні частоти спектра кодового слова зовнішньої ступені, тобто формується безліч (4.4). Сформована безліч інформаційних частот подається на БФ СЗС, в якому формуються елементи спектра, що залишилися (4.3). Сформований спектр (4.3) кодового слова зовнішньої ступені подається на БФ СКК, в якому обчислюються всі компоненти спектру каскадного коду, тобто обчислюється (4.5). Обчислений спектр (4.5) подається на Б ЗПФ, де виконується зворотне перетворення Фур'є і обчислюється, таким чином, кодове слово каскадного коду (4.2) в часовій області. Обчисленне слово (4.2) подається на БВ КС, де завершується формування кодового слова за допомогою форматування його потрібним чином з видачою на вихід пристрою. БУ служить для узгодження функціонування окремих блоків пристрою і управління його роботою. Структура кодового слова каскадного коду в частотній області однозначно задається структурою алгебри складових його кодів першої і другої ступенів. У цьому сенсі, процес декодування каскадних кодів представляється через послідовне декодування в частотної області складових його алгебраїчних кодів першої і другої ступенів.

*Алгоритм декодування каскадних кодів в частотній області [87]*

Крок 1. Ввести значення спотвореного кодового слова (4.2) з символами з $GF(q)$, тобто ввести послідовність $v' = v + e$, де

$$e=\begin{pmatrix} e_{0,0} & e_{0,1} & \dots & e_{0,K-1} & e_{0,K} & e_{0,K+1} & \dots & e_{0,N-1} \\ e_{1,0} & e_{1,1} & \dots & e_{1,K-1} & e_{1,K} & e_{1,K+1} & \dots & e_{1,N-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ e_{m-1,0} & e_{m-1,1} & \dots & e_{m-1,K-1} & e_{m-1,K} & e_{m-1,K+1} & \dots & e_{m-1,N-1} \\ e_{m,0} & e_{m,1} & \dots & e_{m,K-1} & e_{m,K} & e_{m,K+1} & \dots & e_{m,N-1} \\ e_{m+1,0} & e_{m+1,1} & \dots & e_{m+1,K-1} & e_{m+1,K} & e_{m+1,K+1} & \dots & e_{m+1,N-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ e_{n-1,0} & e_{n-1,1} & \dots & e_{n-1,K-1} & e_{n-1,K} & e_{n-1,K+1} & \dots & e_{n-1,N-1} \end{pmatrix}$$

є послідовність помилок, що сталися в каналі зв'язку при передачі кодового слова (4.2).

Крок 2. Розглядаючи стовпці матриці $v'=v+e$ як спотворені кодові слова коду першої ступені декодувати їх в спектральній області.

Крок 2.1. Для всіх $i=0,1,...,N-1$ сформувати інтерполяційний багаточлен таким чином, що б

$$t_i(\alpha^j)=v'_{j,i},\ \deg(t_i(x))<n.$$

Крок 2.2. Для всіх $i=0,1,...,N-1$ знайти такі багаточлени $\lambda_i(x)$ и $q_i(x)$, для яких справедливо порівняння:

$$\lambda_i(x)t_i(x)\equiv q_i(x)\bmod(x^n-1),$$

причому слід знайти багаточлен $q_i(x)$ найбільшого ступеню, так, що

$$\deg(q(x))<\frac{n+k}{2}.$$

Крок 2.3. Для всіх $i=0,1,...,N-1$ обчислити інформаційний спектральний багаточлен

$$M_i^1(x) = \frac{q_i(x)}{\lambda_i(x)}.$$

Коефіцієнти багаточлена $M_i^1(x) = M_{n-1,i}^1 x^{A-1} + ... + M_{1,i}^1 x + M_{0,i}^1$ лежать в полі $GF(q^m)$ й задають, таким чином, відновлені інформаційні частоти $i$-ого кодового слова коду першої ступені.

Крок 2.4. Для всіх $i = 0,1,..., N-1$ відновити інформаційну послідовність $i$-ого кодового слова коду першої ступені. Для цього перетворити (з урахуванням обмежень пов'язаності) ненульові компоненти вектору

$$M_i^1 = (M_{n-1,i}^1,..., M_{1,i}^1, M_{0,i}^1)$$

з елементами з $GF(q^m)$ у вектор

$$\overline{M}_i^1 = (\overline{M}_{m-1,i}^1,..., \overline{M}_{1,i}^1, \overline{M}_{0,i}^1)$$

з елементами з $GF(q)$.

Крок 3. Сформувати спотворене кодове слово коду зовнішньої ступені.

Крок 3.1. З усіх знайдених на попередньому кроці елементів

$$\overline{M}_{j,i}^1,\ j = 0,1,..., m-1,\ i = 0,1,..., N-1,$$

сформувати матрицю

$$\overline{M^1} = \begin{pmatrix} \overline{M}^1_{0,0} & \overline{M}^1_{0,1} & \dots & \overline{M}^1_{0,K-1} & \overline{M}^1_{0,K} & \overline{M}^1_{0,K+1} & \dots & \overline{M}^1_{0,N-1} \\ \overline{M}^1_{1,0} & \overline{M}^1_{1,1} & \dots & \overline{M}^1_{1,K-1} & \overline{M}^1_{1,K} & \overline{M}^1_{1,K+1} & \dots & \overline{M}^1_{1,N-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \overline{M}^1_{m-1,0} & \overline{M}^1_{m-1,1} & \dots & \overline{M}^1_{m-1,K-1} & \overline{M}^1_{m-1,K} & \overline{M}^1_{m-1,K+1} & \dots & \overline{M}^1_{m-1,N-1} \end{pmatrix}.$$

Крок 3.2. Розглядаючи кожен стовпець матриці $\overline{M^1}$, що складається з $m$ символів з $GF(q)$, як один символ з $GF(q^m)$:

$$\overline{M}_i \Rightarrow \begin{pmatrix} \overline{M}^1_{0,i} \\ \overline{M}^1_{1,i} \\ \dots \\ \overline{M}^1_{m-1,i} \end{pmatrix}, \; i = 0,..., N-1,$$

сформувати вектор

$$\overline{M} = (\overline{M_0}, \overline{M_1}..., \overline{M_{K-1}}, \overline{M_K}, \overline{M_{K+1}},..., \overline{M_{N-1}}),$$

тобто для поліноміального уявлення елементів поля маємо:

$$\overline{M}_i(z) = \overline{M}^1_{0,i} + \overline{M}^1_{1,i} z + ... + \overline{M}^1_{m-1,i} z^{m-1}, \; \overline{M}_i(z) \in GF(q^m).$$

Крок 4. Декодувати сформоване на попередньому кроці кодове слово $\overline{M} = (\overline{M_0}, \overline{M_1}..., \overline{M_{K-1}}, \overline{M_K}, \overline{M_{K+1}},..., \overline{M_{N-1}})$ коду зовнішньої ступені через перетворення в спектральної області.

Крок 4.1. Сформувати інтерполяційний багаточлен $T(x)$ таким чином, що б

$$T(\alpha^i) = \overline{M_i}, \; \deg(T(x)) < N.$$

Крок 4.2. Знайти такі багаточлени $\Lambda(x)$ й $Q(x)$, що справедливо порівняння

$$\Lambda(x)T(x) \equiv Q(x) \bmod(x^N - 1),$$

причому слід знайти багаточлен $Q(x)$ найбільшого ступеню, так, що

$$\deg(Q(x)) < \frac{N+K}{2}.$$

Крок 4.3. Обчислити інформаційний спектральний багаточлен

$$M^2(x) = \frac{Q(x)}{\Lambda(x)}.$$

Коефіцієнти $M^2(x)$ задають, таким чином, відновлені інформаційні частоти кодового слова зовнішньої ступені.

Крок 5 (при необхідності). Перетворити ненульові компоненти вектору

$$M^2 = (M^2_{N-1},..., M^2_1, M^2_0)$$

з елементами з $GF(q^m)$ в інформаційний вектор

$$I = (I_{Km-1},..., I_1, I_0)$$

з елементами з $GF(q)$.

Запропонований алгоритм узагальнюється на випадок багатовимірних каскадних кодів. У цьому випадку, на Кроці 1 вводиться багатовимірне кодове слово виду:

$$v_{i_1,i_2,\ldots,i_p} = \frac{1}{n_1}\frac{1}{n_2}\ldots\frac{1}{n_p}\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\ldots\sum_{j_p=0}^{n_p-1}\alpha_1^{-i_1 j_1}\alpha_2^{-i_2 j_2}\ldots\alpha_p^{-i_p j_p}c_{j_1,j_2,\ldots,j_p},$$

а Крок 2 повторюється для кожної «мірності» каскадного коду. Після декодування всіх кодів першої ступені подальші обчислення (крок 3 і далі) виробляються аналогічно.

Розроблений алгоритм декодування каскадних кодів з використанням перетворень в частотної області дозволяє за кінцеве число кроків по заданому кодовому слову з помилками відновити інформаційні символи. При цьому основні обчислення виконуються в арифметиці кінцевих полів. Для обґрунтування практичних рекомендацій з використання розробленого алгоритму декодування в даній роботі пропонується структурна схема пристрою, наведена на рис. 4.8.

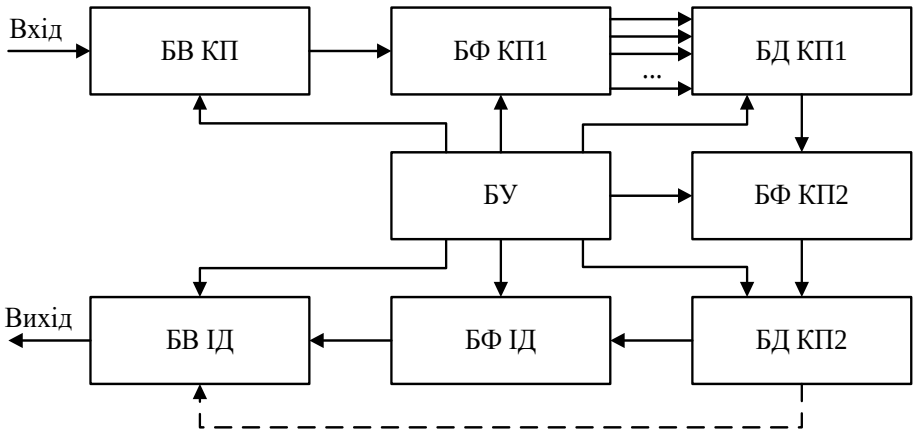



Рис. 4.8. Структурна схема пристрою декодування каскадних кодів в частотній області

Пристрій, схема якого наведена на рис. 4.8, містить:

БВ КП – блок введення кодових даних, тобто кодових слів з можливо спотвореними кодовими символами;

БФ КП1 – блок формування кодових послідовностей першого каскаду;

БД КП1 – блок декодування кодових слів першого каскаду в частотній області;

БФ КП2 – блок формування кодових послідовностей другого каскаду;

БД КП2 – блок декодування кодових слів другого каскаду в частотній області;

БФ ІД – блок формування інформаційних даних;

БВ ІД – блок виведення інформаційних даних;

БУ – блок узгодження.

Пристрій побудований таким чином. Вхід пристрою з'єднаний з першим входом БВ КП, вихід якого з'єднаний з першим входом БФ КП1. Виходи БФ КП1 з'єднані з першим входом БД КП1, вихід якого з'єднаний з першим входом БФ КП2. Вихід БФ КП2 з'єднаний з першим входом БД КП2, вихід якого з'єднаний з першим входом БФ ІД або відразу з першим входом БВ ІД. Вихід БВ ІД з'єднаний з входом пристрою. Сім виходів БУ з'єднані з другими входами БВ КП, БФ КП1, БД КП1, БФ КП2, БД КП2, БФ ІД, БВ ІД, відповідно.

Пристрій працює наступним чином. На вхід пристрою подаються кодові послідовності каскадного коду, що підлягають декодуванню кодування. В БВ КП подані дані формуються у вигляді слова каскадного коду з можливими помилками. Сформовані слова подаються на БФ КП1, де виробляється формування слів коду першої ступені. Сформовані слова подаються (допускається паралельна обробка) на вхід БД КП1, де проводиться їх декодування з перетвореннями в частотної області. Результат декодування (інформаційні частоти слів першої ступені) подається в БФ КП2, де формується кодове слово коду другої ступені, що подається в БД КП2. В БД КП2 кодове слово декодується, отримані інформаційні частоти кодового слова другої ступені подаються на вхід БФ ІД або відразу на вхід БВ ІД. При подачі інформаційних частот на вхід БФ ІД формується інформаційна послідовність у вигляді символів з $GF(q)$, яка подається на вхід БВ ВД, де завершується процес декодування і відформатований відповідним чином

результат роботи пристрою подається на його вихід. При подачі інформаційних частот на вхід БВ ІД на вихід пристрою подається інформаційна послідовність символів з $GF(q^m)$. БУ служить для узгодження функціонування окремих блоків пристрою і управління його роботою.

Проведені дослідження обчислювальної складності розроблених алгоритмів кодування каскадними кодами з перетвореннями в частотній області показали, що для двовимірних $(Nn, Kk, Dd)$ кодів з великим $n$ й $N$ складність обчислень буде мати порядок $n\log n + N\log N$ операцій. Для багатовимірного випадку ця оцінка визначається $n_1\log n_1 + n_2\log n_2 + ... + n_p\log n_p$. Обчислювальна складність алгоритмів декодування становить $Nn(\log n)^2$ операцій для двовимірного каскадного $(Nn, Kk, Dd)$ коду й $n_p\sum_{i=1}^{p-1} n_i(\log n_i)^2$ операцій для багатовимірної каскадної кодової конструкції.

Проведені порівняльні дослідження обчислювальної складності розроблених алгоритмів кодування і декодування показали, що перехід в частотну область перетворення призводить, як правило, до істотного зниження обчислювальної складності. Це зниження можна порівняти з переходом від блокового коду заданої довжини до відповідного каскаду тієї ж довжини. Застосування одночасно і каскадних кодових конструкцій і обчислювальних процедур в частотній області дозволяє забезпечити найменшу обчислювальну складність.

Обчислювальна ефективність розроблених вище алгоритмів кодування узагальненими каскадних кодами в частотній області за рахунок використання спектрального опису узагальнених каскадних кодів без зниження підвищує коректуючу здатність коду.

При декодуванні за рахунок застосування алгоритмів дискретного перетворення Фур'є над кінцевими полями Галуа і перенесення в частотну

область основних етапів виявлення і виправлення помилок в прийнятому кодовому слові вдається підвищити обчислювальну ефективність в 2 - 4 рази.

З урахуванням вищесказаного, можна зробити висновок, що для заданої довжини кодової конструкції та «мірності» каскадного коду використання розроблених і розглянутих в роботі пропозицій забезпечує підвищення обчислювальної ефективності завадостійкого кодування і декодування дискретних повідомлень в телекомунікаційних системах та мережах. З ростом довжини коду, що характерно для сучасних телекомунікаційних систем і мереж, одержуваний виграш зростає. Таким чином, методи і обчислювальні алгоритми кодування / декодування каскадними кодами в частотній області доцільно використовувати для підвищення достовірності передачі даних в сучасних телекомунікаційних системах і мережах, таких як Grid.

### 4.5 Висновки за розділом

1 Розглянута загальна концепція диспетчеризації в Grid для вирішення даного завдання, що дозволяє на основі розробленого методу планування та відомого, що базується на основі рішення задачі про найменше покриття, підвищити ефективність використання ресурсів в глобальному розподіленому обчислювальному середовищі.

2 Сформульовані практичні рекомендації з інтеграції й ефективного використання розробленого методу оперативного планування в сучасних планувальниках, які дозволили підвищити оперативність виконання завдань кластеру та підвищити сумарний коефіцієнт важливості виконаних задач на 20-57%.

Показана актуальність питання підвищення достовірності передачі інформації при моніторингу ресурсів Grid-систем.

# ЗАГАЛЬНІ ВИСНОВКИ

У дисертаційній роботі, на основі теоретичних досліджень на комп'ютерного моделювання вирішена науково-практична задача, яка полягає в оптимізації процесів управління телекомунікаційними мережами шляхом розробки методу планування виконання завдань з управління телекомунікаційними мережами обчислювальних кластерів із застосуванням Grid технологій, який базується на зведенні задачі планування виконання завдань до вирішення задачі нелінійного булевого програмування і розробки ефективного методу вирішення даного завдання на основі рангового підходу, що дозволяє підвищити ефективність планування. За підсумками вирішення поставленого завдання зроблені наступні висновки:

1. На основі проведеного аналізу основних напрямків удосконалення кластерів ТКС і проблем, що виникають при вирішенні завдань планування виконання завдань з управління телекомунікаційними мережами в кластерах Grid-систем установлено, що важливим напрямком розвитку сучасних систем управління пакетною обробкою в кластерах ТКС є розробка відповідних методів моделей планування, які дозволяють підвищити ефективність та якість обслуговування завдань з управління телекомунікаційними мережами, що характеризуються коефіцієнтом важливості, коефіцієнтом збереження важливості і коефіцієнтом прискорення виконання завдань, а також розширення функціональних можливостей розподілених ТКС на основі використання обчислювальних кластерів із застосуванням Grid-технологій. Показано, що підвищення ефективності та якості обслуговування завдань можливо за рахунок розробки методу планування виконання завдань з управління телекомунікаційними мережами, який базується на зведенні задачі планування виконання завдань до вирішення задачі нелінійного булевого програмування і розробки ефективного методу вирішення даного завдання на основі рангового підходу.

2. Вперше розроблений метод оперативного планування розподілу завдань з управління телекомунікаційними мережами, який дозволяє підвищити значення сумарного коефіцієнту важливості виконаних завдань та зменшити час їх обслуговування шляхом вирішення задач нелінійного булевого програмування у кластерах ТКС з використанням Grid-технології. Збільшення числа обмежень в задачах нелінійного булевого програмування призводить до зниження похибки їх вирішення, при цьому сама ступінь нелінійності несуттєво впливає на величину похибки. Тобто, якщо мати $n$-процесорних елементів для формування шляхів, то часова складність алгоритмів на основі розглянутих процедур не перевищить відповідно $O(pn^2)$ і $O(pn)$, якщо їх реалізовувати з використанням CUDA технологій, то це дозволить застосовувати ці процедури для управління в масштабі реального часу, в розподілених телекомунікаційних систмах.

3. Вперше створена модель функціонування кластера телекомунікаційної Grid-системи, новизна якої полягає у можливості дослідження ефективності використання розробленого методу оперативного планування розподілу завдань з управління телекомунікаційними мережами при різних законах розподілу потоків завдань та інтенсивності обробки їх в кластері, та яка базується на основі використання для планування виконання завдань вирішення задач нелінійного булевого програмування. Модель дозволяє користувачеві в ручному режимі змінювати параметри роботи Grid-системи, включаючи методи планування, які і є предметом дослідження даної моделі, і отримувати результат у докладній формі з графіками.

4. Одержав подальший розвиток метод планування розподілу завдань у кластерах ТКС, який дозволив у порівнянні з існуючими методами дискретної оптимізації суттєво зменшити часову складність планування розподілу завдань у кластерах ТКС, забезпечуючи малу похибку результатів рішення шляхом удосконалення методу вирішення задач нелінійного булевого програмування на основі рангового підходу. Метод дозволяє

вирішувати як завдання лінійного, так і нелінійного програмування, з довільними нелінійностями, як в функціоналі, так і в обмеженнях, алгоритмами поліноміальної складності з невеликою похибкою.

5. Побудована імітаційна модель вирішення задач нелінійного булевого програмування на основі рангового підходу, на якій показано, що в порівнянні з відомими, розроблені наближені процедури оперативного розподілу завдань з управління телекомунікаційними мережами мають меншу часову та обчислювальну складність, а також малу й асимптотично зменшувану зі зростанням розмірності задачі похибку рішення.

6. Сформульовані практичні рекомендації з інтеграції й ефективного використання розробленого методу оперативного планування в сучасних планувальниках, які дозволили підвищити оперативність виконання завдань кластеру та зменшити час очікування завдань в черзі на більш ніж 20%.

7. Отримані в дисертації практичні результати використані при створенні моделі планувальника завдань СПО кластера ТКС на основі Grid-технології, а розроблені процедури паралельного розподілу завдань використані в моделі супервізора комп’ютерного кластеру для підсистеми управління цифрової мережі електрозв’язку (підтверджено довідкою про участь у НДР (Держпрограма МОН України № 23/1-2016Б) по темі: «Формування теоретичних засад підвищення ефективності використання інформаційно-керуючих систем на залізничному транспорті» (ДР№ 0116U000787).

Результати дисертаційної роботи використані в навчальному процесі УкрДУЗТ при виконанні курсового та дипломного проектування (підтверджено актом впровадження УкрДУЗТ).

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Foster I. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration [Text] / I. Foster et al. - Argonne National Laboratory, Argonne, Ill., 2002. – 57 P.

2. Foster I. The Grid: Blueprint for a New Computing Infrastructure [Text] / I. Foster, C. Kesselman. - San Francisco, Calif.: Morgan Kaufmann, 1999. – 639 P.

3. Foster I. The Grid: Blueprint for a New Computing Infrastructure (2nd Edition) [Text] / I. Foster, C. Kesselman. - San Francisco, Calif.: Morgan Kaufmann, 2004. – 748 P.

4. Foster I. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems [Text] / I. Foster, N.T. Karonis // Supercomputing. IEEE. –1998.– November.– P. 21-26.

5. Foster I. The Anatomy of the Grid: Enabling Scalable Virtual Organizations [Text] / I. Foster, C. Kesselman, S. Tuecke // International Journal of High Performance Computing Applications. –2000.– Vol.15, № 3.– P. 200-222.

6. Foster I. Modeling Stateful Resources with Web Services [Text] / I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, T. Storey, S. Weerawaranna. - Globus Alliance, 2004. – 24 P.

7. Foster I. Managing multiple communication methods in high-performance networked computing systems [Text] / I. Foster, J. Geisler, C. Kesselman, S. Tuecke // Parallel and Distributed Computing. – 1997.– P. 35–48.

8. Foster I. Strand: New Concepts in Parallel Programming [Text] / I. Foster, S. Taylor. - Prentice-Hall, Inc., 1990. – 378 P.

9. Foster I. Grid Services for Distributed Systems Integration [Text] / I. Foster, C. Kesselman, J.M. Nick, S. Tuecke // IEEE Computer. – 2002.– № 6.– P. 37-46.

10. Martin F. Maldonado Grid Technical Architect [Electronic resource] / F. Martin // Grid computing in higher education: Trends, values and offerings. - December 2004, - P. 29-40. Access mode: http://www-1.ibm.com/grid/pdf/grid_computing_in_higher_ed.pdf. - Title from the screen.

11. Moore R. Virtualization Services for Data Grids [Text] / R. Moore, C. Baru // Grid Computing: Making the Global Infrastructure a Reality. – John Wiley & Sons Ltd., 2003. – P. 398-410.

12. Francine Berman. The grads project: software support for high-level grid application development [Electronic resource] / B. Francine, Andrew Chien et al. // COMPUTING APPLICATIONS. – 2001. - P. 327-344. Access mode: http://hipersoft.cs.rice.edu/grads. - Title from the screen.

13. Pordes Ruth Grid2003 Project. The Grid2003 Production Grid: Principles and Practice / Ruth Pordes // VDGL: Technical Report. – 2004. – 22 p.

14. Foster I. What Is The GRID? A Three Point Checklist. GRID Today [Electronic resource] / I. Foster // July 22, 2002: Vol. 1 No. 6. Access mode: http://www.gridtoday.com/02/0722/100136.html.

15. Коваленко В.Н. Оценка возможностей программных платформ грид, Труды международной конференции "Распределенные вычисления и Грид-технологии в науке и образовании" [Електронний ресурс] / В.Н. Коваленко, Д.А. Корягин // Дубна, 29 июня - 2 июля 2004г., сс. 128-133. Режим доступа: http://www.gridclub.ru/library/publication.2005-03-17.0023427070.

16. Богданов С.А. Метадиспетчер: реализация средствами метакомпьютерной системы Globus [Електронній ресурс] / С.А. Богданов, В.Н. Коваленко, Е.В. Хухлаев, О.Н. Шорин // Препринт ИПМ им. М.В. Келдыша РАН, 2001. – 24 с. Режим доступа: http://www.keldysh.ru/papers/2002/source/prep2002_22.doc.

17. Коваленко В.Н. Метод опережающего планирования для Грид [Текст] / В.Н. Коваленко, Е.И. Коваленко, Д.А Корягин, Э.З. Любимский – Препринт ИПМ им. М.В. Келдыша. – 2005. – 33 с.

18. Коваленко В.Н. Управление заданиями в распределенной среде и протокол резервирования ресурсов [Текст] / В.Н. Коваленко, А.В. Орлов - Препринт ИПМ им. М.В.Келдыша РАН. – М.: 2002, 23 с.

19. Коваленко В.Н. Управление заданиями в распределенной вычислительной среде [Текст] / В.Н. Коваленко, Е.И. Коваленко, Д.А. Корягин, Э.З. Любимский, Е.В. Хухлаев // Открытые системы. – № 5-6. – 2001. – С. 22-28.

20. Коваленко В.Н. Структура и проблемы развития программного обеспечения среды распределенных вычислений Грид [Текст] / В.Н. Коваленко, Е.И. Коваленко, Д.А. Корягин, Э.З. Любимский, А.В. Орлов, Е.В. Хухлаев // М.: 2002, 23 с.

21. Уткин В.Ф. Надежность и эффективность в технике [Текст] / В.Ф.Уткин – М: Машиностроение, 1988. – Т. 3: Эффективность технических систем. – 328 с.

22. Киселев В.Д. Оптимизация восстановительного резервирования информации в сетях ЭВМ [Текст] / В.Д. Киселев, О.В. Ешков // Электронное моделирование – 1995. – т.17, № 3. – С. 53–58.

23. Турута Е.Н. Обеспечение отказоустойчивости управляющих много микропроцессорных систем путем перераспределения задач отказавших модулей [Текст] / Е.Н. Турута // В кн.: Системы управления информационных сетей. М.: Наука, 1983. – С. 187–198.

24. Турута Е.Н. Об одном методе повышения живучести локальных сетей ЭВМ [Текст] / Е.Н. Турута, В.Ш. Ковалев // АВТ, № 5. 1983. – С. 42–44.

25. Самойленко С.И. Сети ЭВМ [Текст] / С.И. Самойленко - М.: Наука, 1986. – 180 с.

26. Харченко В.С. Живучесть и безопасность систем управления летательных аппаратов и комплексов. Ч .1. Основные понятия и модели [Текст] / В.С. Харченко, П.Е. Марков - Учебное пособие, МОУ, 1994. – 68 с.

27. ГОСТ 24.701-86. Надежность автоматизированных систем управления. Основные положения [Текст] М.: Издательство стандартов, 1986. – 78 с.

28. ГОСТ 24.702-85. Эффективность автоматизированных систем управления. Основные положения [Текст] М.: Издательство стандартов, 1985. – 35 с.

29. ГОСТ 24.703-85. Типовые проектные решения в АСУ. Основные положения [Текст] М.: Издательство стандартов, 1985. – 47 с.

30. ГОСТ 27.004-85. Системы технологические. Термины и определения [Текст] М.: Издательство стандартов, 1985. – 68 с.

31. Дьяченко В.Ф. Управление на сетях связи [Текст] / В.Ф. Дьяченко, В.Г. Лазарев, Г.Г. Саввин // М: Изд-во Энергия, 1984. – 277 с.

32. Зиновьев Э.В. Методы управления сетевыми информационными системами [Текст] / Э.В. Зиновьев, А.А. Стрекалев - Рига: Зинатне, 1991. – 308 с.

33. Bochman G. Formal methods in communication Protocol design / G. Bochman, C. Sunshine // IEEE Trans. Commins, 1980, vol. COM-28. – P. 624–631.

34. Borgerson B.R. A reliability model for gracefully degrading and standby-sparing systems [Text] / B.R. Borgerson, R.F. Freitas // IEEE Trans. on Comput., vol. C.-24, May 1975. – P. 517–525.

35. Brown C.W. Adaptive Routing in Centralized Computer [Text] / C.W. Brown, M. Schwartz. // Communication networks. Proc. IEEE International Conference on Communications, San Francisco, June 1975. – P. 47–12 to 47–16.

36. Microsoft Corporation. Компьютерные сети. Учебный курс [Текст] / Пер.с англ. – М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1997. – 696 с.

37. Городнов В.П. Моделирование боевых действий частей, соединений и объединений войск ПВО / В.П. Городнов – Харьков: ВИРТА ПВО, 1987.– 380 с.

38. Lifka D.A. The ANL/IBM SP Scheduling System [Electronic resource] / D.A.Lifka – Access mode: http://www.tc.cornell.edu/~lifka/.

39. Jon B.Weissman Ensemble Scheduling: Resource Co-Allocation on the Computational Grid [Electronic resource] / Access mode: http://www-users.cs.umn.edu/~jon/papers/grid2001_ens.ps

40. The Globus Resource Specification Language RSL v1.0 [Electronic resource] / Access mode: http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1/

41. Globus Toolkit [Electronic resource] / Access mode: htpp://www.globus.org

42. Raman R. An extensible framework for distributed resource management [Text] / R. Raman, M. Livny, and M. Solomon - Cluster Computing, 2(2), 1999.

43. Condor [Electronic resource] / Access mode: http://www.cs.wisc.edu/condor/

44. Job Description Language HowTo [Electronic resource] / Access mode: http://server11.infn.it/workload-grid/docs/DataGrid-01-TEN-0102-Document.doc.

45. Frey J. A computation management agent for multiinstitutional Grids. Cluster Computing [Text] / J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke // 5(3):237–246, 2002.

46. Czajkowski K. Grid information services for distributed resource sharing. In Proceedings of the Tenth [Text] / K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman - IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), August 2001.

47. Relational Grid Monitoring Architecture (R-GMA) [Electronic resource] / A.Cooke et al, UK e-Science All Hands Conference, Nottingham, September 2003. Access mode: http://www.r-gma.org/pub/ah03_148.pdf

48. Пономаренко В.С. Методы и модели планирования ресурсов в Grid – системах [Текст] / В.С. Пономаренко, С.В. Листровой, С.В. Минухин, С.В. Знахур - Монография. Х.: ВД «ІНЖЕК», 2008. – 408 с.

49. Листровой С.В. Метод решения задач целочисленного линейного программирования с булевыми переменными на основе рангового подхода

[Текст] / С.В. Листровой, Д.Ю. Голубничий, Е.С. Листровая // Электрон. моделирование, 1998. — Т.20. №6. – С.14-32.

50. Листровой С.В. Метод решения задачи о минимальном покрытии на основе рангового подхода [Текст] / С.В. Листровой, А.Ю. Гуль // Электрон. моделирование, К.: 1999. – № 1. – С. 58-70.

51. Листровой С.В. Об использовании гарантированных прогнозов в методах решения задач булевого программирования на основе рангового подхода [Текст] / С.В Листровой., О.Н. Симашкевич // Электрон. моделирование. –2003. – Т.25 №4. – С.89-103.

52. Жихарев В.Я. Методы моделирования и дискретной оптимизации вычислительных систем реального времени [Текст] / В.Я. Жихарев, В.М. Илюшко, Л.Г. Кравец, С.В. Листровой, В.С. Харченко - Под ред. В.Я. Жихарева, Харьков–Житомир, ЖГУ, 2004.– 494 с.

53. Балаш Э. Аддитивный алгоритм для решения задач линейного программирования с переменными, принимающими значения 0 или 1 [Текст] / Э. Балаш // Кибернет. сб. - К. - 1969. - Вып.6. - С.217 - 252.

54. Беллман Р. Прикладные задачи динамического программирования [Текст] / Р. Беллман, С. Дрейфус - М.: Наука, 1965. - 458 с.

55. Вагнер Г. Основы исследования операций / Г. Вагнер - М.: Мир, 1973. - 231 с.

56. Зайченко Ю.П. Исследование операций / Ю.П. Зайченко - К.: Вища школа, 1988. - 552 с.

57. Калинин В.Н. Теория систем и оптимального управления. Часть 2. Понятия, модели и алгоритмы оптимального выбора / В.Н. Калинин, Б.А. Резников, Е.И. Варакин - МО СССР, 1987. - 590 с.

58. Кофман А. Методы и модели исследования операций. Целочисленное программирование / А. Кофман, А. Анри-Лабродер - М.: Мир, 1977. - 236 с.

59. Пападимитриу Х. Комбинаторная оптимизация. Алгоритмы и сложность [Текст] / Х. Пападимитриу, К. Стайглиц - М.: Мир, 1985. - 512с.

60. Рейнгольд Э. Комбинаторные алгоритмы. Теория и практика [Текст] / Э. Рейнгольд, Ю. Нивергельт, Н. Део - М.: Мир. 1980. 476 с.

61. Dantzig G.B. Discrete-variable extremum problems [Text] / G.B Dantzig // Oper. Res. - 1957. - № 2. - P.266 - 277.

62. Gomory R.E. Outline of an algorithm for integer solution to linear programs [Text] / R.E. Gomory // Bull.Amer.Math.Soc. - 1958. - №5 - P.275-278.

63. Беллман Р. Динамическое программирование и современная теория управления [Текст] / Р. Беллман, Р. Калаба - М.: Наука, 1969. - 112 с.

64. Листровой С.В. О возможности решения задач оптимального распределения ресурсов при управлении сложными системами в реальном масштабе времени [Текст] / С.В. Листровой, В.Н. Хрин // Изв.АН России. Техническая кибернетика. - 1992. - №4. - С.125 -133.

65. Емеличев В.А. Дискретная оптимизация. Последовательные схемы решения [Текст] / В.А Емеличев // Кибернетика. – 1972. – № 2. – С. 109–121.

66. Емеличев В.А. Метод построения последовательности планов для решения задач дискретной оптимизаци [Текст] / В.А. Емеличев, В.И. Комлик - М.: Наука, 1981. – 208 с.

67. Михалевич В.С. Методы последовательной оптимизации в дискретных сетевых задачах оптимального распределения ресурсов [Текст] / В.С. Михалевич, А.И. Кукса - М.: Наука, 1983. – 208 с.

68. Михалевич В.С. К вопросу оптимизации вычислений [Текст] / В.С. Михалевич, И.В. Сергиенко // Кибернетика и системный анализ. – 1994.– № 2.– С. 65–93.

69. Михалевич В.С. Исследование методов решения оптимизационных задач и их приложения [Текст] / В.С. Михалевич, И.В. Сергиенко, Н.З. Шор // Кибернетика. – 1981. –№ 4. – С. 89–113.

70. Сергиенко И.В. Модели и методы решения на ЭВМ комбинаторных задач оптимизации [Текст] / И.В. Сергиенко, М.Ф. Каспшицкая - К.: Наук.думка, 1981. – 288 с.

71. Сергиенко И.В. Математические модели и методы решения задач дискретной оптимизации [Текст] / И.В. Сергиенко-К.:Наук.думка,1988–472с.

72. Листровой С.В. О возможности распараллеливания комбинаторных задач на дереве путей графа [Текст] / С.В. Листровой, В.Я. Певнев - Всесоюзный семинар Вопросы оптимизации вычислений. докл. Киев: Институт кибернетики им. В.М. Глушкова АН УССР. 1987. – С. 185.

73. Листровой С.В. Вопросы построения параллельных вычислительных систем и параллельный алгоритм для решения задачи о кратчайшем пути [Текст] / С.В. Листровой, В.Я. Певнев // Электрон. Моделирование. – 1990. – Т. 12,№ 1. – С. 14–20.

74. Сергиенко И.В. Приближенные методы решения дискретных задач оптимизации [Текст] / И.В. Сергиенко, Т.Т. Лебедева, В.А. Рощин - К.: Наук. думка. 1980. – 276 с.

75. Олешко М.В. Алгоритм для решения задач целочисленного программирования с булевыми переменными, использующий вероятностные оценки [Текст] / М.В. Олешко, В.П. Шило // Программное обеспечение экстремальных задач и пакеты прикладных программ – К.: Ин-т кибернетики АН УССР, 1982. – С. 110–113.

76. Сергиенко С.В. Полиномиальный ассипмтотический ε – оптимальный алгоритм случайного поиска для задач булевого линейного программирования [Текст] / С.В. Сергиенко, В.П. Шило // Докл. АН УССР. Сер. А. – 1987. - №3. – с. 70-72.

77. Гуляницкий Л.Ф. О методах дискретной оптимизации для многопроцессорных вычислительных комплексов [Текст] / Л.Ф. Гуляницкий, И.В. Сергиенко, А.Н. Ходзинский // Кибернетика. – 1988. – № 4. – С. 26–33.

78. Wah B.W. The status of MANIP – a multicomputer architecture for solving combinatorial extremum – search problems [Text] / B.W. Wah, L.J. Li, C.F. Yu // 11-th Annu. Int. Simp. Comput. Archit. (Ann Arbor, Mich., 5-7 June 1984). – S.I., Silver Springer, 1984. – P. 58-63.

79. Листровой С.В. Создание основанной на идее рангового подхода процедуры оптимального распределения заданий в GRID – системе и исследование эффективности её алгоритма [Текст] / С.В. Листровой, Е.В. Тимошенко // Інформаційно-керуючі системи на залізничному транспорті. - 2007. - № 5,6. – С.44-51.

80. Листровой С.В. Оптимизация распределения заданий в GRID – системе [Текст] / С.В. Листровой, Е.В. Тимошенко // Зб. наук. праць. – Донецьк: ДонІЗТ, 2007. - № 12. – с.71-80.

81. Дэвис Д. Вычислительные сети и сетевые протоколы [Текст] / Д. Дэвис, Д.Барбер, У.Прайс , С. Соломонидес - М.: Мир, 1982. – 562 с.

82. Наумчук О.Ф. Оценка пропускной способности сетей передачи и распределения информации [Текст] / О.Ф. Наумчук, Г.Г. Савин - Сб. Сети передачи информации и их автоматизация. Изд-во Наука, 1965. – 265 с.

83. Лістровий С.В. Оперативне управління телекомунікаційними системами та мережами на основі рангових методів рішення задач мулевого програмування та теорії графів [Текст] / дис. д.т.н.: 05.12.02 - Лістровий Сергій Володимирович. – Харків, 2005. – 392 с.

84. Пономаренко В.С. Цілочисельне програмування в економіці. Наукове видання [Текст] / В.С. Пономаренко, Д.Ю. Голубничий, В.Ф. Третяк - Харків: Вид. ХНЕУ, 2005. – 204 с.

85. Кормен Т. Алгоритмы построение и анализ [Текст] / Т. Кормен., Ч. Лейзерсон., Р. Ривест - М:МЦНМО, 2002. – 955 с.

86. Гэри М. Вычислительные машины и трудно решаемые задачи [Текст] / М. Гэри, Д. Джонсон // М.: Мир, 1982.– 416 с.

87. Mathlab For Deep Learning [Electronic resourse] / Acess mode: www.mathworks.com.

88. Дащенко О.Ф. MATLAB в інженерних та наукових розрахунках [Текст] / О.Ф. Дащенко, В.Х. Кириллов, Л.В. Коломієць, В.Ф. Оробей - Монографія. – Одеса: Астропринт, 2003. – 214 с.

89. Кетков Ю.Л. MATLAB 7: программирование, численные методы [Текст] / Ю.Л.Кетков, А.Ю.Кетков - СПб:БХВ-Петербург, 2005. – 752 с.

90. Слепокуров Ю.С. MATLAB 5. Анализ технических систем [Текст] / Ю.С. Слепокуров - Воронеж: Изд-во ВГТУ, 2001. 167 с.

91. Globus Toolkit [Electronic resourse] / Access mode: http://www.globus.org/developer/news/20011112a.html

92. Globus Toolkit [Electronic resourse] / Access mode:http://www.globus.org

93. PBS Professional Open Source Project [Electronic resourse] / Access mode:http://www.openpbs.org

94. High Throughput Computing [Electronic resourse] / Access mode:http://www.cs.wisc.edu/condor/

95. IT Infostructure Solutions [Electronic resourse] / Access mode:http://www.platform.com/products/LSFfamily/

96. Sun Microsystems [Electronic resourse] / Access mode: http://www.sun.com/software/gridware/sge/

97. Towards “Cluster as Server”: An Integrated Approach to Workload and Systems Management for Compute Clusters, Technical Whitepaper [Electronic resourse] / Access mode: http://www.platform.com/pdfs/whitepapers/

98. Krallmann J. On the design and evaluation of job scheduling systems [Text] / J. Krallmann, U. Schwiegelshohn, and R. Yahyapour // Job Scheduling Strategies for Parallel Processing (Proceedings of the Fifth International JSSPP Workshop; LNCS #1659), pages 17–42. Springer-Verlag, 1999.

99. Hotovy S/ Workload evolution on the Cornell Theory Center IBM SP2 [Electronic resourse] / pp. 27–40, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162. Access mode: http://www.cs.huji.ac.il/~feit/parsched/jsspp96/p-96-2.ps.

# ДОДАТКИ

# ДОДАТОК А.АКТИ ПРО ВПРОВАДЖЕННЯ ДИСЕРТАЦІЙНОЇ РОБОТИ

З А Т В Е Р Д Ж У Ю
Начальник
виробничого підрозділу
"Харківське відділення" філії
"Головний інформаційно-
обчислювальний центр"
публічного акціонерного
товариства "Українська залізниця"
_____ Шепотенко С.О.
"____" _____ 2018г.

АКТ
про впровадження результатів дисертаційної роботи
Курцева Максима Сергійовича
«Метод планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування», поданої на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.12.02 – телекомунікаційні системи та мережі.

Цей акт складений комісією в складі:
Голова комісії:
заступник начальник, Смоловик Г.П.
Члени комісії:
начальник відділу ХКМПД, Давидов І.В.
провідний інженер ХКМПД, Кириченко С.С.

Комісія вивчила матеріали результатів досліджень методів і моделей підвищення ефективності функціонування та якості обслуговування, а також розширення функціональних можливостей телекомунікаційних систем та мереж на основі використання комп'ютерних кластерів із застосуванням Grid - технології, проведених старшим викладачем кафедри спеціалізованих комп'ютерних систем Українського державного університету залізничного транспорту Курцевим М.С., і зазначає наступне:

1. Розроблений у роботі метод оперативного планування розподілу завдань на основі вирішення задач нелінійного булевого програмування у кластерах ТКС з використанням Grid-технології дозволяє, в порівнянні з методами на основі групової вибірки та FIFO, зменшити час обслуговування інформаційно-розрахункових задач та підвищити значення сумарного коефіцієнту важливості виконаних задач, що забезпечує зменшення сумарного часу виконання на 12 та 24% відповідно, підвищення значення сумарного коефіцієнту важливості виконаних задач на більш ніж 50%.

2. Удосконалена модель функціонування кластера Grid-системи на основі використання для планування виконання завдань вирішення задач нелінійного булевого програмування дозволяє досліджувати ефективність використання

розробленого методу при різних законах розподілу потоків завдань та інтенсивність обробки завдань в кластері.

3. Практичну цінність має розроблений у роботі програмний комплекс, написаний на мові програмування С++, що моделює роботу телекомунікаційної інфраструктури з використанням розробленого методу планування виконання завдань на основі вирішення задач нелінійного булевого програмування, який використовується на виробничому підрозділі «Харківське відділення» філії «Головний інформаційно-обчислювальний центр» публічного акціонерного товариства «Українська залізниця» для моделювання навантажень на апаратуру та канали зв'язку при виконанні робіт з проектування систем передачі інформації АСУ УЗ з урахуванням пріоритетів та пропускної здатності інфраструктури.

Голова комісії: Смоловик Г.П.

Члени комісії: Давидов І.В.

Кириченко С.С.

ЗАТВЕРДЖУЮ
Перший проректор
Українського державного
університету залізничного
транспорту
кандидат технічних наук, доцент
_____ В.М. Астахов
«___» _____ 2018 р.

АКТ
впровадження результатів дисертаційної роботи
Курцева Максима Сергійовича
«Метод планування виконання завдань з управління телекомунікаційними мережами на основі вирішення задач нелінійного булевого програмування»
у навчальному процесі Інституту перепідготовки та підвищення кваліфікації кадрів Українського державного університету залізничного транспорту

До основних результатів дисертаційної роботи Курцева М.С., що використовуються у навчальному процесі Інституту перепідготовки та підвищення кваліфікації кадрів Українського державного університету залізничного транспорту, належать:

- розроблений метод оперативного планування розподілу завдань на основі вирішення задач нелінійного булевого програмування у кластерах телекомунікаційних систем та мереж з використанням Grid-технологій, який дозволяє зменшити час обслуговування інформаційно-розрахункових задач та підвищити значення сумарного коефіцієнту важливості виконаних задач;
- модель функціонування кластера Grid-системи на основі використання для планування виконання завдань вирішення задач нелінійного булевого програмування, що дозволяє досліджувати ефективність використання розробленого методу при різних законах розподілу потоків завдань та інтенсивність обробки завдань у кластері;
- програмний комплекс, на якому проведене моделювання роботи кластера в середовищі C++, впроваджений на кафедрі спеціалізованих комп'ютерних систем, що застосовується у навчальному процесі ІППК.

Дані розробки з 2017 року по теперішній час використовуються:

1. При проведенні занять у групах навчально-наукового центру підвищення кваліфікації для фахівців служб сигналізації та зв'язку;
2. При підготовці магістрів денної та заочної форми навчання за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології»:
   - «Комп'ютерні мережі, internet та хмарні сервіси»;
   - «Бази даних сучасних інформаційних систем»;
   - «Апаратне програмне забезпечення комп'ютерних систем загального та спеціального призначення».

Заступник директора ІППК,
к.т.н., доцент Захарченко В.В.

# ДОДАТОК Б. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ ТА ВІДОМОСТІ ПРО АПРОБАЦІЮ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЇ

***Наукові праці в яких опубліковані основні наукові результати дисертації***:

1. Listrovoy S.V. A uniform procedure of a system resources interaction in distributed computer media [Text] / S.V. Listrovoy, K.A. Trubchaninova, V.A. Bryksin, M.S. Kurtsev // Bulletin of NTU “KhPI”. Series: Strategic management, portfolio, program and project management. – Kharkiv : NTU “KhPI”, 2017. – No 3(1225). – P. 101-107. Bibliogr.: 10. – ISSN 2311-4738.

2. Listrovaya E.S. Modeling Local Scheduler Operation Based on Solution of Nonlinear Boolean Programming Problems [Text] / E.S. Listrovaya, V.A. Bryksin, M.S. Kurtsev // “Cybernetics and Systems Analysis”. – Springer International Publishing AG. – 2017. – Vol.53. №5. – P. 766-775.

3. Листровой С.В. Математическая и имитационная модель планирования выполнения заданий в кластере Grid-системы [Текст] / С.В. Листровой, М.С. Курцев // Інформаціно-керуючі системи на залізничному транспорті. – Харків: УкрДУЗТ, 2016. – №2(117). – с.61-72.

4. Листровой С.В. Метод и модель планирования распределения пакетов заданий в кластере Grid системы [Текст] / С.В. Листровой, Е.С. Листровая, М.С. Курцев // Международный научно-теоретический журнал «Электронное моделирование». – Институт проблем моделирования в энергетики Г.Е. Пухова, 2016. – Т.38. №6. – С. 85-106.

5. Листровой С.В. Подход к организации планирования распределением ресурсов в системах управления железнодорожным транспортом [Текст] / С.В. Листровой, М.С. Курцев // Науково-практичний журнал «Залізничний транспорт України». – Науково-дослідний та конструкторсько-технологічний інститут залізничного транспорту (Філія "НДКТІ" ПАТ "Укрзалізниця"), 2016. – №3-4(118-119). – С.14-22.

6. Приходько С.И. Алгоритм кодирования каскадными кодами в частотной области [Текст] / С.И. Приходько, М.С. Курцев, Хамзе Билал // Інформаційно-керуючі системи на залізничному транспорті. – Х.: УкрДАЗТ, – 2013. – №3. – С. 78 – 82.

7. Приходько С.И. Алгоритм декодирования каскадными кодами в частотной области [Текст] / С.И. Приходько, М.С. Курцев, Хамзе Билал // Системи Обробки Інформації. – Х.: ХУПС, – 2013. – Вип. 5(112). – С. 132 – 136.

8. Листровой С.В. Ранговый подход к решению задач линейного и нелинейного булевого программирования для планирования и управления в распределенных вычислительных системах [Текст] / С.В. Листровой, Е.С. Листровая, М.С. Курцев // Международный научно-теоретический журнал «Электронное моделирование». – Институт проблем моделирования в энергетики Г.Е. Пухова, 2017. – Т.39. №1. – С. 19-38.

9. Лістровий С.В. Ефективний метод вирішення задачі планування та виконання завдань у розподілених обчислювальних системах на основі нелінійного булевого програмування [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 79 Міжнародної науково-технічної конференції "Розвиток наукової та інноваційної діяльності на транспорті" 25-27 квітня. - Харків: УкрДУЗТ, 2017. – №169(додаток). – С.9-11.

10. Листровой С.В. Метод эффективного управления очередью заданий в распределенных вычислительных системах на основе решения задач нелинейного булевого программирования [Текст] / С.В. Листровой, М.С. Курцев // Материали XXII Международной научно-технической конференции "Современные средства связи" 19-20 октября 2017 г. - Минск, Республика Беларусь: Белорусская государственная академия связи, 2017. – С.185.

11. Лістровий С.В. Метод планування ресурсів в кластерах Grid-системна основі рангових алгоритмів вирішення задач нелінійного булевого програмування [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 30 Міжнародної науково-практичної конференції "Інформаційно-керуючі системи на залізничному транспорті" 26-28 вересня. - Харків: УкрДУЗТ, 2017. – №4(додаток). – С. 8.

12. Лістровий С.В. Моделювання роботи Grid системи в телекомунікаційних мережах [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 78 Міжнародної науково-технічної конференції "Розвиток наукової та інноваційної діяльності на транспорті" 26-28 квітня. - Харків: УкрДУЗТ, 2016. – № 4 (додаток). – С.37.

13. Листровой С.В. Планирование выполнения заданий в распределенных вычислительных системах на основе алгоритма, построенного на ранговом подходе к решению задач булевого программирования [Текст] / С.В. Листровой, М.С. Курцев // Міжнародна науково-практична конференція «Інформаційні технології і мехатроніка: освіта, наука та працевлаштування» 20-21 квітня 2016 року: матеріали конференції. - Харків: ХАДІ, 2016. – С.67-69.

14. Лістровий С.В. Загальна концепція створення управління диспетчеризацією в Grid [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 29 Міжнародної науково-практичної конференції "Інформаційно-керуючі системи на залізничному транспорті" 27-29 вересня. - Харків: УкрДУЗТ, 2016. – №4(додаток). – С.2-3.

15. Лістровий С.В. Загальна концепція створення управління диспетчеризацією в Grid [Текст] / С.В. Лістровий, М.С. Курцев // Матеріали 28 Лістровий іжнародної науково-практичної конференції "Інформаційно-керуючі системи на залізничному транспорті" 24-25 вересня. - Харків: УкрДУЗТ, 2015. – №4(додаток). – С.43.

# ДОДАТОК В. ІНТЕГРАЦІЯ РОЗРОБЛЕНОГО МЕТОДУ В СИСТЕМУ УПРАВЛІННЯ ПАКЕТНОЮ ОБРОБКОЮ ЗАВДАНЬ В КЛАСТЕРАХ GRID-СИСТЕМ

Інтеграція розробленого методу в систему управління пакетною обробкою завдань в кластерах Grid-систем

Розглянемо технології обробки і планування завдань на локальних ресурсах РОС з використанням ЛСУР і локального планувальника, що дозволили модифікувати програмне забезпечення планувальника Maui шляхом інтеграції нових алгоритмів планування. Розроблено технологію формування бази даних стану завдань і ресурсів обчислювального кластера на основі даних лог-файлів локального планувальника для реалізації запитів з метою отримання оперативної інформації про стан завдань, що обробляються в системі, і ресурсів. Проведено порівняльний аналіз результатів планування завдань на основі розроблених методів з існуючими та запропоновані практичні рекомендації по використанню досліджуваних методів для підвищення продуктивності функціонування розподілених обчислювальних систем.

***Технологія обробки завдань на обчислювальному кластері під управлінням Maui/Torque***

Об'єднання ресурсів в систему управління розподіленими обчисленнями PBS (Portable Batch System) дозволяє використовувати єдиний підхід користувачів до подання завдань в систему для їх виконання. Програмний пакет Torque - локальний менеджер ресурсів є однією з версій PBS і призначений для контролю над ресурсами вузлів кластера і запуску завдань під управлінням операційної системи сімейства Linux. Управління кластером на основі ЛСУР Torque здійснюється на основі сервера і обчислювальних вузлів.

На сервері встановлюється демон сервера Torque (pbs_server), на обчислювальних вузлах - демон pbs_mom, який взаємодіє з демоном сервера Torque. Клієнтські команди для представлення і управління завданнями встановлюються на будь-якому вузлі (в тому числі на вузлах, де не

встановлені демони pbs_server або pbs_mom). Менеджер ресурсів Torque для планування завдань використовує вбудований планувальник завдань pbs_sched. Він визначає момент запуску завдань після звільнення ресурсу (вузла) кластера. Менеджер ресурсів забезпечує виконання наступних низькорівневих функцій: запуск, переривання, скасування і контроль виконання завдання. Для підвищення якості процесів планування завдань на обчислювальному кластері на сервері також встановлюється демон планувальника Torque pbs_sched, який взаємодіє з демоном сервера Torque (pbs_server) для прийняття рішень про політики використання локальних ресурсів і виділення вузлів для виконання на них завдань. Для реалізації політик планування системні адміністратори застосовують планувальники з досить широким функціоналом - наприклад, Maui або Moab [66]. Технологія виконання завдань включає наступні етапи: демон pbs_server отримує нове завдання і інформує про це планувальника; планувальник знаходить вільні вузли для виконання завдання і посилає команди для виконання завдання на вузли зі списку вузлів сервера Torque; демон pbs_server посилає на виконання нове завдання на перший вільний вузол в списку вузлів і дає йому інструкції по запуску завдання на цьому вузлі. Цей вузол визначається як виконавчий хост - MotherSuperior, а інші вузли - як SisterMoms. Локальний планувальник Maui опитує менеджер ресурсів Torque на предмет наявності вільних ресурсів і завдань в черзі, які необхідно виконати. На основі отриманих даних і налаштувань приймається рішення про запуск завдання, після чого надсилається команда сервера Torque для його виконання. Можливості планувальника Maui дозволяють формувати різні стратегії виконання завдань на основі пріоритетів завдань, визначених різними параметрами: кількістю запитуваних ресурсів, потрібним обсягом пам'яті, необхідним часом виконання завдань, приналежністю користувача до якоїсь групи користувачів (профілювання завдань) тощо.

Завдання буде працювати тільки на обчислювальних вузлах, які мають ресурси, необхідні для його виконання.

Від вибору того чи іншого обчислювального вузла залежить час обробки завдання, яке включає: час очікування в черзі системи пакетної обробки завдань (СПО), час доставки файлів завдання на виконання і час виконання. Час виконання завдання задається в ресурсному запиті. При цьому час очікування в рамках використовуваного інтерфейсу СПО є складно прогнозованою величиною, яка залежить від конфігурації СПО: для того щоб оцінити можливий час очікування, потрібно використовувати інформацію про наявні в черзі завданнях, їх пріоритети, поточне завантаження ресурсів, режимах їх розподілу між завданнями і тощо.

Найбільш поширеною технологічною схемою організації роботи планувальника є жорстке виділення (резервування) ресурсів обчислювального кластеру. Ці ресурси є недоступними для їх локального використання на час зовнішнього резервування, і вони будуть простоювати, якщо планувальник їх недовантажить. Основні втрати мають місце через нескоординовану роботу глобального і локального планувальників кластера. Наприклад, незважаючи на ресурси, що простоюють, локальні планувальники не можуть їх використовувати, так як на певний момент (проміжок) часу ці ресурси зарезервовані для виконання завдань рівня Grid-сегмента (зовнішнього вхідного потоку).

Таким чином, забезпечення спільної роботи локального і глобального рівнів планування є актуальним з точки зору обліку використовуваних політик і принципів призначення ресурсів завданням. Така взаємодія має визначатися не тільки виділенням певної кількості ресурсів в фіксований час, а призначенням їх для тих завдань, які вже знаходяться в черзі, в необхідній кількості. Таким чином, СПО повинна гарантувати попереднє резервування ресурсів для виконання конкретного завдання.

Програмний пакет Maui може підключатися до різних СПО, які мають модульну архітектуру. Модульність дозволяє замінювати компоненти пакета в рамках заданих інтерфейсів. При використанні PBS, наприклад, досить

поміняти стандартний планувальник завдань обчислювального кластера на основі методу FCFS на планувальник Maui.

У пакеті Maui реалізована черговість запуску завдань на виконання і відповідні ресурси для тих з них, які можна використовувати в поточний момент часу. При цьому Maui використовує нові механізми, засновані на прогнозуванні часу виконання завдань і призначені для оптимізації процесів управління виконанням завдань [35].

Для підвищення ефективності планування в Maui слід виділити наступні дані, які відрізняють цей пакет від інших планувальників, компоненти, технології та методи планування, що дозволяють в подальшому використовувати їх для запропонованої модифікації програмного забезпечення:

1) Алгоритми зворотного заповнення (Backfill) і справедливого розподілу ресурсів (fairshare), що використовуються для підвищення ефективності роботи кластера та зменшення часу очікування завдань в черзі;

2) Система автоматичного визначення пріоритетів завдань, що дозволяє користувачам (різним профілям і класам завдань) надавати певні переваги при розподілі наявних ресурсів;

3) Покращена діагностика проблем із завданнями, вузлами і програмними компонентами СПО;

4) Можливість отримання хронологічної статистичної інформації про виконувані завдання, чергах завдань, стан ресурсів тощо;

5) Можливість моделювання роботи СПО, за допомогою якої можна перевірити політики планування шляхом настройки конфігураційного файлу планувальника Maui. Це дозволяє підібрати таку конфігурацію системи, яка найбільш повно відповідає вимогам проведення обчислень – профілям завдань, обмеженням на час їх виконання або директивний термін, обмеженням на необхідну пам'ять тощо.

В Maui є інтерфейс для зовнішнього адміністративного резервування, недоліком якого є необхідність координувати дії щодо резервування, з

управління завданнями, чергами завдань і вузлами, що здійснюються за допомогою Maui.

Технологія резервування реалізована в Maui на основі відповідного типу об'єктів. Об'єкт «резервування» включає:

1) ACL (AccessControlList, список контролю доступу) – набір параметрів, використовуючи які планувальник визначає, для яких завдань доступні зарезервовані ресурси;

2) Список зарезервованих ресурсів;

3) Час дії резервування (слот).

На рівні реалізації кожне резервування фізично являє собою запис в базу даних Maui. Об'єкт резервування пов'язаний, з одного боку, з майбутнім часом; з іншого боку, резервування передбачає аналіз станів ресурсів на момент початку резервування: для кожного ресурсу в часі визначаються мітки – яким завданням в даний момент ресурси можуть бути доступні, сукупність яких і утворює резервування.

Резервування здійснюється на основі команди «setres», дозволеної адміністратору, керуючому роботою планувальника Maui. Слід зазначити, що попереднє резервування здійснюється під завдання, яке не обробляється в СПО. Тому повинен бути розроблений механізм, що дозволяє зв'язати завдання, що виконуються зі зробленими резервуваннями. Об'єкт резервування має список контролю доступу ACL, і саме по ньому відбувається прив'язка до ресурсів. Зарезервовані ресурси призначаються тільки тим завданням, які мають хоча б один з наступних параметрів, використовуваних в команді «setres»: «GROUPLIST», «USERLIST», «ACCOUNTLIST», «CLASSLIST» і «QOSLIST», які збігаються з відповідними атрибутами ACL-резервування . Таким чином, прив'язка може здійснюватися по імені користувача, його групі, класу і рівню якості обслуговування.

Технологія настройки списків доступу (ACL) в Maui дозволяє, зокрема, домогтися того, щоб зарезервовані ресурси були доступні тільки для певного

завдання (певного профілю завдання) навіть в тому випадку, якщо воно ще не надійшло в чергу на виконання.

Maui дозволяє зарезервувати 4 типу ресурсів:

1) «PROCS = <INTEGER>» - кількість процесорів;

2) «MEM = <INTEGER>» - обсяг оперативної пам'яті;

3) «DISK = <INTEGER>» - дисковий обсяг пам'яті;

4) «SWAP = <INTEGER>» - обсяг пам'яті файлу підкачки.

Кількість необхідних ресурсів визначається в ресурсному запиті завдання. В завдання може входити кілька задач, кожна з яких виконується на одному вузлі кластера. При здійсненні резервування в ресурсному запиті вказуються ресурси, необхідні для виконання одного завдання, і кількість завдань. В Maui використовуються засоби для операцій з вже існуючими резервуваннями: наприклад, за допомогою команди «showres» можна отримати інформацію про те, де, для яких завдань, на який час і яку кількість ресурсів зарезервовано, причому доступний також пошук як по резервуванню, так і по вузлах.

Планувальник Maui працює ітераційно, при цьому кожен цикл починається при здійсненні одного з наступних подій:

- змінюється стан завдання або ресурсу;

- досягнуто межа резервування;

- отримана зовнішня команда;

- з початку попереднього циклу минув деякий пороговий час, визначений як максимальний.

У процесі планування застосовується ще один, внутрішній тип резервування Maui – job-резервування. Цей тип резервування використовується для забезпечення доступності ресурсів для виконання завдання. Як тільки завдання запускається, то на весь час його виконання створюється job-резервування, що блокує доступ до ресурсів, необхідних для виконання цього завдання. Після закінчення виконання завдання завершується (можливо, раніше, ніж через встановлене користувачем час

виконання або після закінчення директивного терміну), job-резервування скасовується, і ресурси, виділені для цього завдання, стають вільними для призначення інших завдань.

Процедура «job-резервування» використовується для того, щоб завдання з більш високим пріоритетом не були затримані виконанням менш пріоритетних завдань, що має місце при використанні алгоритму BackFill.

У загальному випадку використовується технологія запуску завдань, що зводиться до наступного: впорядковані за пріоритетом завдання запускаються, і при цьому створюється job-резервування; якщо чергове завдання не можна запустити через відсутність необхідних ресурсів, визначається найближчий час, коли можна буде здійснити його запуск, і, починаючи з цього часу, створюється job-резервування необхідних ресурсів на час виконання завдання; після того як здійснено деяку сконфігуровану кількість зарезервованих завдань, починається робота алгоритму планування BackFill. Результатом його роботи є визначення кількості вузлів, які є вільними, і проміжку часу, починаючи з поточного, протягом якого вузли є вільними; вільні вузли об'єднуються в тимчасові «вікна» (слоти). При цьому сумарна ширина «вікон» (слотів) може виявитися більше, ніж кількість вільних на даний момент вузлів, так як деякі вузли можуть входити в кілька тимчасових «вікон»; з усіх «вікон» вибирається одне - найширше, і з завдань вибирається і запускається те завдання, час виконання якого найбільш точно відповідає величині цього «вікна». Основною відмінністю процесу «job-резервування» від адміністративного є те, що job-резервування здійснюється самим планувальником і їм же воно скасовується перед початком чергового кроку планування. На відміну від адміністративного резервування, job-резервування не гарантує запуск завдання в зазначений час. Таким чином, job-резервування (для незапущених завдань) є міткою, що дозволяє враховувати в алгоритмі Backfill пріоритети завдань. Резервування засноване на прогнозі майбутніх станів СПО і, таким чином, дозволяє оцінити час старту завдань. При наявності оцінки (прогнозу) часу старту планувальником

не потрібно детальна інформація про завантаження окремих процесорів, кількості завдань в чергах тощо. Для реалізації даної схеми планувальник використовує дві функції, які повинні підтримуватися СПО:

1) Функція оцінки часу запуску завдання. При цьому вхідними параметрами є ресурсний запит і час розрахунку, вихідними – склад слотів часу запуску завдання;

2) Функція резервування ресурсів. При цьому вхідними параметрами є ресурсний запит, час початку резервування, тривалість резервування; вихідними – час створення резервування.

***Технологія планування виконання завдань з використанням локального планувальника Maui***

Для подальшої модифікації програмного забезпечення локального планувальника Maui застосовується технологія, побудована на використанні функціональності вихідного коду програмної реалізації планувальника Maui [61]. При першому старті планувальник проводить ініціалізацію всіх сутностей, якими він оперує в процесі планування, а саме менеджера ресурсів, користувачів, черг завдань, класів завдань, розділів тощо. Після ініціалізації сутностей починається перша ітерація планування. Функція MSchedProcessJobs реалізує управління кожної ітерації планування. Повний цикл планування проходить кожен розділ (Partition) окремо. Спочатку планування здійснюється по розділу, який визначений як розділ за замовчуванням, а потім - по іншим. Розглянемо ітерацію планування на прикладі одного розділу. Планування по розділах реалізовано в функції m_schedule_on_partitions. Одним з параметрів (параметр 2) цієї функції є логічне значення, яке описує, чи буде Maui використовувати алгоритм Backfill при плануванні чи ні, а також супутні йому методи - BestFit, Greedy тощо. Архітектура планувальника побудована таким чином, що алгоритм Backfill не бере участі в плануванні розділу за замовчуванням. Перший параметр функції m_schedule_on_partitions визначає, з якого розділу буде

здійснюватися планування. Якщо значення першого параметра приймає значення TRUE, то планування здійснюється по розділу, який визначений як розділ за замовчуванням. В іншому випадку (перший параметр приймає значення FALSE) планування буде здійснюватися за всіма іншими розділами з використанням алгоритму Backfill. Фрагмент програми виклику функцій по розділах наведено на рисунку В.1.

```
/* schedule priority jobs */
if (CurrentQ[0] != -1)
    {
    /* schedule jobs on their default partitions; skip backfilling */

    m_schedule_on_partitions (TRUE, FALSE, CurrentQ);

    /* schedule jobs on all partitions; do backfilling */

    m_schedule_on_partitions (FALSE, TRUE, CurrentQ);
    }       /*   END  if  (GlobalSQ[0]  != -1) */
```

Рис. В.1. Виклик функцій m_schedule_on_partitions

Далі розглянемо варіант ітерації планування з використанням алгоритму Backfill. В Maui використовується технологія реалізації основних функцій, якими Maui оперує в ході планування. Так, наприклад, функція MLocalQueueScheduleJobs дозволяє реалізувати всі механізми планування, надаючи основні інтерфейси для взаємодії з планувальником (такі функції у вихідному коді Maui ідентифіковані як MLocal). Кожна ітерація планування починається з запиту до менеджера ресурсів для отримання даних про стан вузлів, завданнях, політик, QoS тощо. На наступному кроці здійснюється ініціалізація внутрішньої черги завдань Maui, якою планувальник безпосередньо оперує, після чого відбувається вибірка всіх завдань в цю чергу з подальшою їх пріоритезацію на основі функції MQueuePrioritizeJobs, рисунок В.2.

```
int MQueuePrioritizeJobs (
mjob_t **Q,     /* I: lise    of  jobs    to  be prioritized (optional) */
int *JobIndex)  /* 0: hi-low priority sorted array of job indexes */
{
double tmpD;
int jindex;

long MaxIdleStartPriority = 0;

mjob_t *J;

if (Joblndex •= NULL)
    jindex =0;

if ((Q – NULL) ||
    (Q[0] = NULL) ||
    (Q[0]->Next = NULL) ||
    (Q[0]->Next – Q[0]))
  {
  /* no gueue specified, search full job table */
  for (jindex = 0;Joblndex[jindex] != -l;jindex++)
    {
    J = MJob[Joblndex[jindex]];

    MJobGetStartPriority(J,C,StmpD, ,NULL,NULL);
```

Рис. В.2. Реалізація функції приоритезації завдань

Безпосереднє планування по розділах здійснюється наступним чином: відбувається ініціалізація черги для розділу, в яку йде вибірка завдань для розділу з глобальної черги тих завдань, які плануються. На цьому етапі кожне завдання з черги проходить перевірку всіх політик, обмежень і залежностей.

При цьому спочатку функція MqueueScheduleJobs оновлює кеш ресурсів, після чого формується список вузлів, в який включаються ті вузли, на яких можуть виконуватися завдання. Після цього перевіряються всі можливі політики планування, а також здійснюється резервування.

На наступному кроці управління передається функції MQueueBackfill, код реалізації якої представлено на рисунку В.3, яка реалізує алгоритм Backfill. Далі створюється локальна чергу, в яку вибираються завдання з глобальної черги – ті, які підлягають плануванню.

```
int MQueueBackFill(int *BFQueue, int PLevel, mpar_t *P)
  {
  {int BFNodeCount;///variables
  DBG(1,fSCHED) DPrint("%s (BFQueue, %s, %s)\n",
    FName,
    MPolicyMode[PLevel],
    (P != NULL) ? P->Name : "NULL");
  if (P = NULL)
    {
    return(FAILURE);
    }
  memset(&DRes,0,sizeof(DRes));
  DRes.Procs = 1;
  /* backfill partition */
#ifdef  __MLONGESTBFWINDOWFIRST
  OBFTime = MAX_MTIME;
#else /* __ML0NGESTBFWIND0WFIRST */
  OBFTime = 0;
#endif /*  MLONGESTBFWINDOWFIRST */
  /* process normal backfill */
  while (MBFGetWindow( &BFNodeCount, &BFProcCount, BFNodeList, &BFTime, OBFTime,
                       P, ALL, ALL, ALL, 0, 0, 1, &DRes, NULL, NULL, NULL, NULL) = SUCCESS)
    {
    DBG(3,fSCHED) DPrint("INFO: backfill window obtained [%d nodes/%d procs : %s]\n",
      BFNodeCount,
      BFProcCount,
      MULToTString(BFTime));

    OBFTime = BFTime;
```

Рис. В.3. Код реалізації функції MQueueBackfill

Після цього MQueueBackfill отримує за допомогою виклику функції MBFGetWindow, код реалізації якої представлено на рисунку В.4, «вікно», в яке за алгоритмами BestFit, Gready та інші, потрапляють завдання, що виконуються. Після цього для завдання відбувається виділення ресурсів і здійснюється його запуск з подальшою обробкою цього завдання в резервуваннях. Всі розглянуті кроки кожне завдання проходить в розділі.

```
int M3FGetWindow(

  int   *BFNodeCount,       /*  O:  nodes available                */
  int   *BFTaskCount ,      /*  O:  tasks available                */
  nodelist_t BFNodeList,    /*  O:  nodelist available             */
  long  BFTime,             /*  O:  duration available             */
  long  SMinTime,           /*  I:  duration required              */
  mpar_t *P,                /*  I:  partition                      */
  char  *UName,             /*  I:  user name of window requestor  */
  char  *GName,             /*  I:  group name of window requestor */
  char  *AName,             /*  I:  account of window requestor    */
  int   MemCmp,             /*  I:  memory comparison required     */
  int   Memory,             /*  I:  memory required on node        */
  unsigned long WCLimit,
  mcres_t *DRes,            /*  I:  dedicated resources required per task */
  char  *ClassString,
  char  *FeatureString,     /*  I   */
  char  *QOSName,
  char  *Msg)               /*  O:  descriptive message            */
```

Рис. 4. Функція MBFGetWindow

Враховуючи особливості побудови кластерних архітектур РОС, представляється доцільним інтегрувати в планувальник завдань Maui новий алгоритм планування, який буде більш ефективним в умовах гетерогенності вузлів кластера, гетерогенності завдань і високої інтенсивності їх надходження в систему на обробку, що є актуальним завданням в сучасних умовах роботи із завданнями в РОС. Важливим завданням є забезпечення «безпечної» інтеграції нового алгоритму в існуюче ПЗ планувальника: інтеграція повинна бути проведена без зміни ключових механізмів функціонування планувальника Maui. Основна ідея, покладена в основу нового алгоритму планування завдань, полягає в використанні методу планування на основі рішення ЗНП або NLP.

Вхідними даними для вирішення ЗНП або NLP є матриця відповідності, що відображає інформацію про те, які завдання і на яких ресурсах кластеру можуть бути виконані з урахуванням того, що одне завдання може бути виконано відповідно до ресурсного запиту на декількох вузлах кластера. Після формування матриці відповідності вирішується ЗНП

або NLP, в результаті чого визначається множина вузлів кластера, на яких можуть виконуватися призначені на них завдання. Для забезпечення гнучкості роботи планувальника включення нового алгоритму планування пропонується реалізувати на основі використання конфігураційного файлу Maui і параметра NODEALLOCATIONPOLICY, налаштування якого показано на рисунку В.5. За замовчуванням цьому параметру присвоєно значення MINRESOURCE. Для включення роботи нового алгоритму на основі рішення ЗНП (МС) або NLP використовується значення параметра LOCAL, що дозволяє використовувати функції для перевизначення поведінки і механізмів роботи планувальника. До того ж, щоб в роботу алгоритмів МС або NLP не вносити евристику, значення параметра BACKFILLPOLICY встановлюється в значення NONE, що забезпечує блокування роботи алгоритму Backfill. Для інтеграції нового алгоритму після процесу налагодження вихідного коду визначаються функції планувальника, які будуть використані для коректної інтеграції нового алгоритму планування в ПО планувальника Maui. Основною функцією для вирішення цього завдання є функція MQueueScheduleJobs, реалізована в файлі MQueue.c в каталозі дистрибутива maui-3.*.*\Src\moab\.

Для реалізації цієї функції потрібно перевірити параметр NODEALLOCATIONPOLICY, як показано на рисунку В.6, в залежності від значення якого виконується той чи інший блок коду програми. Якщо значення параметра дорівнює LOCAL, то буде виконуватися функція MLocalQueueScheduleJobs, реалізована в файлі MLocal.c, який знаходиться в каталозі дистрибутива maui-3.*.*\Src\moab\., призначеного для модифікації механізмів роботи планувальника. На рисунку В.7. показана функція MLocalQueueScheduleJobs до модифікації вихідного коду програми, а на рисунку В.8 - після модифікації коду.

```
# maui.cfg @PACKAGE_VERSION@
SERVERHOST            @HOSTNAME@
# primary admin must be first in list
ADMIN1                @USER@
# Resource Manager Definition
RMCFG[@MACHINE@] TYPE=@RMTYPE@@RMHOST@@RMPORT@@RMVERSION@@RMWIREPROTOCOL@@RMSOCKETPROTOCOL@
# Allocation Manager Definition
AMCFG[bank]  TYPE=@AMTYPE@@AMHOST@@AMPORT@@AMSOCKETPROTOCOL@@AMWIREPROTOCOL@@AMOTHER@
# full parameter docs at http://supercluster.org/mauidocs/a.fparameters.html
# use the 'schedctl -l' command to display current configuration
RMPOLLINTERVAL        00:00:30
SERVERPORT            42559
SERVERMODE            NORMAL
# Admin: http://supercluster.org/mauidocs/a.esecurity.html
LOGFILE               maui.log
LOGFILEMAXSIZE        10000000
LOGLEVEL              3
# Job Priority: http://supercluster.org/mauidocs/5.1jobprioritization.html
QUEUETIMEWEIGHT       1
# FairShare: http://supercluster.org/mauidocs/6.3fairshare.html
#FSPOLICY              PSDEDICATED
#FSDEPTH               7
#FSINTERVAL            86400
#FSDECAY               0.80
# Throttling Policies: http://supercluster.org/mauidocs/6.2throttlingpolicies.html
# NONE SPECIFIED
# Backfill: http://supercluster.org/mauidocs/8.2backfill.html
BACKFILLPOLICY        NONE
RESERVATIONPOLICY     CURRENTHIGHEST
# Node Allocation: http://supercluster.org/mauidocs/5.2nodeallocation.html
NODEALLOCATIONPOLICY  LOCAL
```

Рис. В.5. Налаштування параметрів конфігураційного файлу Maui

```
for (jindex = 0;Q[jindex] <= 1;jindex++)
  {
  J = MJob[Q[jindex]];
  tmpNAPolicy = (J->Req[0]->NAllocPolicy != NULL) ?
    J->Req[0]->NAllocPolicy->NAllocPolicy :
    MPar[J->Req[0]->PtIndex].NAllocPolicy;
}
switch (tmpNAPolicy) {
  tmpNAPolicy = (J->Req[0]->NAllocPolicy != NULL) ?
    J->Req[0]->NAllocPolicy->NAllocPolicy :
    MPar[J->Req[0]->PtIndex].NAllocPolicy;
}
switch (tmpNAPolicy) {
    case mnalLocal:
        if (MLocalQueueScheduleIJobs(Q, P) != SUCCESS)
        {
            DBG(3,fSCHED) DPrint("DEBUG:     NAPolicy - Local" +
            "Function MLocalQueueScheduleIJobs FAILURE!, Go to MinResource NAPolicy \n");
        }
        else
        {
            return (SUCCESS);
            break;
        }

    case mnalMinResource:

  /* ---------------------------------------- */
```

Рис. В.6. Перевірка конфігураційного файлу і запуск фрагмента коду, що реалізує функціональність нового алгоритму планування на основі використання конфігураційного файлу Maui і параметра NODEALLOCATIONPOLICY

```
int MLocalQueueScheduleIJobs(
  int    *Q,
  mpar_t *P)
  {
  mjob_t *J;
  int     jindex;
  if ((Q == NULL) || (P == NULL))
    {
    return(FAILURE);
    }
  /* NOTE:  insert call to scheduling algorithm here */
  for (jindex = 0;Q[jindex] != -1;jindex++)
    {
    J = MJob[Q[jindex]];
    /* NYI */
    DBG(7,fSCHED) DPrint("INFO:     checking job '%s'\n",
      J->Name);
    }  /* END for (jindex) */
  return(FAILURE);
  }  /* END MLocalQueueScheduleIJobs() */
```

Рис. В.7. Функція MLocalQueueScheduleJobs до модифікації коду

```
int MLocalQueueScheduleIJobs(int *Q, par_t *P){
 { int      jindex; //variables
  if ((Q == NULL) || (P == NULL))
    {
    return(FAILURE);
    }
 if (Q[0] == -1)     {
    DBG(2,fSCHED) DPrint("INFO:     no jobs in queue\n");
    return(FAILURE);
    }
  IdleJobFound = FALSE;
  SchedCount = 0;
  memset(PResCount,0,sizeof(PResCount));
  for (jindex = 0;Q[jindex] != -1;jindex++)    {
    J = MJob[Q[jindex]];
    MTRAPJOB(J,FName);
    DBG(7,fSCHED) DPrint("INFO:     checking job '%s'\n",
      J->Name);

    if ((J->EState == mjsStarting) ||
        (J->State == mjsStarting) ||
        (J->EState == mjsRunning) ||
        (J->State == mjsRunning) ||
        (J->State == mjsSuspended))
      {
```

Рис. В.8. Функція MLocalQueueScheduleJobs після модифікації коду

Функція MLocalQueueScheduleJobs використовує два параметри: перший - це глобальна черга завдань, а другий - це розділ (Partition). Як зазначалося раніше, в кожному розділі планування здійснюється окремо.

Так як MQueueScheduleJobs є основною функцією планування, то при її модифікації буде перевизначатися весь функціонал, але при цьому у функції MLocalQueueScheduleJobs буде закладена аналогічна логіка, але з певними змінами. Дана логіка використана в MLocalQueueScheduleJobs для забезпечення політики «поділ відповідальності».

Функція MlocalQueueScheduleJobs є аналогічною функції MQueueScheduleJobs. У цих функціях реалізована логіка, яка визначає, що кожне завдання проходить повний цикл планування окремо, після чого відбувається його запуск. Такий підхід неприйнятний для використання в новому алгоритмі планування – алгоритмів МС або NLP, так як цей алгоритм обробляє і відправляє завдання на виконання пакетами, тому в функції MLocalQueueScheduleJobs після проходження завданням перевірки всіх обмежень і політик цикл планування переривається і формується пакет завдань для їх подальшої обробки .

Цей функціонал реалізований у функції MJobAllocMC, що є модифікованою версією функції MJobAllocMNL, яка знаходиться в файлі MSched.c каталогу дистрибутива maui-3.*.*\Src\moab\. вихідного коду планувальника. Необхідно відзначити, що головна функція MQueueScheduleJobs (основна, немодифікована), яка реалізує цикл планування, викликає в своєму коді функцію MJobAllocMNL, призначену для призначення вузлів. Для роботи з модифікованою на основі функції MQueueScheduleJobs функцією MLocalQueueScheduleJobs, код якої представлено на рисунку В.9, для забезпечення гнучкості роботи планувальника і поділу відповідальності необхідно викликати модифіковану на основі функції MJobAllocMNL функцію MJobAllocMC, рисунок В.10.

```
int MJobAllocMNL(

  mjob_t      *J,             /* I: job requesting resources */
  mnodelist_t  MFeasibleList, /* I: feasible nodes           */
  char        *NodeMap,
  mnodelist_t  MOutList,      /* O: allocated nodes          */
  int          NAPolicy,      /* I: node allocation policy   */
  long         StartTime)     /* I: time job must start      */
```

```
int MJobAllocMC(

  mjob_t      *J,             /* I: job requesting resources */
  mnodelist_t  MFeasibleList, /* I: feasible nodes           */
  char        *NodeMap,
  mnodelist_t  MOutList,      /* O: allocated nodes          */
  int          NAPolicy,      /* I: node allocation policy   */
  long         StartTime,     /* I: time job must start      */
  int jindex,                 /* I: job index      */
  int *Q)
```

Рис. В.9. Функції MJobAllocMNL і MJobAllocMC

```
if (MJobAllocMNL(
      J,
      MNodeList,
      NodeMap,
      NULL,
      NAPolicy,
      MSched.Time) == FAILURE)
  {
```

```
if (MJobAllocMC(
      J,
      MNodeList,
      NodeMap,
      NULL,
      NAPolicy,
      MSched.Time,
      jindex,
      Q
      ) == FAILURE)
  {
    continue;
```

Рис. В.10. Виклик функцій MJobAllocMNL і MJobAllocMC

Таким чином, для реалізації нового алгоритму планування в функції MLocalQueueScheduleJobs виклик функції MJobAllocMNL замінюється викликом модифікованої функції MJobAllocMC, яка формує пакет завдань на основі алгоритму MC і його відправку на виконання. Виклик MJobAllocMC є останнім викликом в циклі планування перед запуском завдання, після якого цикл планування переривається, і функцією MJobAllocMC формується пакет завдань, який відправляється на обработку алгоритмом NLP і далі на їх виконання.

Таким чином, функція MJobAllocMC реалізує функціонал призначення вузлів кластера для виконання завдань. У цій функції даний процес також залежить від параметра NODEALLOCATIONPOLICY: якщо значення цього параметра дорівнює LOCAL, то буде використовуватися алгоритм MC або

NLP (функція MLocalJobAllocateResources). Формування пакету завдань і його відправку на обробку алгоритмом NLP показано на рисунку В.11.

```
switch (tmpNAPolicy)
  {
  case mnalLocal:
     if (Q[jindex + 1] == -1) {
        DBG(3, fSCHED) DPrint("DEBUG:     The job '%s' is the last in the GlobalQueue;" +
                              "Send jobs to MC Algorithm\n", J->Name);
        MCQ[jindex] = Q[jindex];
        MJob[MCQ[jindex]] = J;
        for(int i = 0; RQ->NodeList[i].N != NULL; i++)
        {
            MCRQ[jindex][i] = RQ->NodeList[i].N;
        }
        MLocalJobAllocateResources( MCQ, MCRQ, NodeList, StartTime, rqindex, MinTPN, MaxTPN,
                                    NodeMap, AffinityLevel, NodeIndex, BestList, TaskCount, NodeCount);
     } else {
        MCQ[jindex] = Q[jindex];
        MJob[MCQ[jindex]] = J;
        for(int i = 0; RQ->NodeList[i].N != NULL; i++)
        {
            MCRQ[jindex][i] = RQ->NodeList[i].N;
        }
        DBG(3, fSCHED) DPrint("DEBUG:    The job '%s' in GlobalQueue is not last; continue scheduling cycle\n",
              J->Name);
        return FAILURE;
     }
    break;

  case mnalContiguous:
```

Рис. В.11. Формування пакету завдань і його відправку на обробку алгоритмом NLP

Функція MLocalJobAllocateResources (реалізована в файлі MLocal.c в каталозі дистрибутива maui-3.*.*\Src\moab\.) одним з параметрів (MCQ) приймає пакет завдань для їх обробки алгоритмом МС або NLP. У цій функції викликається функція MauiToCppAdapter (реалізована в файлі MCScheduler.cpp в каталозі дистрибутива maui-3.*.*\contrib\sched), яка розміщена в файлі MCScheduler.cpp і необхідна для перетворення типів даних для реалізації програми алгоритму МС або NLP, написаного на мові С++ з використанням бібліотеки STL та інших визначених функцій. Функція-адаптер для реалізації алгоритму NLP представлена на рис. В.12.

```
int MauiToCppAdapter(int MCQ, mnalloc_t NodeList[], mreq_t RQ) {

    vector<int> *vMCQ;
    vector<int> *vNodeList;

    *vMCQ = MCQToVector(MCQ);
    *vNodeList = NodelistToVector(NodeList);

    int COUNT_RES = vnodelist.size() - 1;
    vector<vector<int> > buffer(COUNT_RES + 1, vector<int>(1, 0));
    BackOffToPool(vMCQ, *_backoff);
    BackOffToMassive();
    if (SchedulerMC(
        vMCQ,
        vNodeList,
        *_backoff,
        buffer) == SUCCESS)
    {
        MCQ = vectorToMcq(*vMCQ);
        NodeList = vectorToNodeList(*vNodeList);
        return SUCCESS;
    }
    else
    {
        return FAILURE;
    }
}
```

Рис. В.12. Функція-адаптер для реалізації алгоритму NLP

Функція Maui ToCppAdapter виконує перетворення типів, а також викликає функцію (SchedulerMC), яка реалізує функціональність алгоритму MC або NLP, і додаткові функції, що забезпечують роботу даного алгоритму. На рисунку В.13 показаний визов функції-адаптера для реалізації алгоритму NLP.

Функція SchedulerMC, реалізована в файлі MCScheduler.cpp в каталозі дистрибутива maui-3.*.*\Contrib\sched, рисунок 14, повністю виконує алгоритм, включаючи процедуру побудови матриці відповідності. У функції SchedulerMC, рисунок В.14, в якості змінної cover оголошена змінна, яку далі буде присвоєно результат рішення ЗНП або NLP (функція SearchCover, рисунок В.15.

Функція SearchCover реалізована в файлі MCScheduler.cpp в каталозі дистрибутива maui-3.*.*\Contrib\sched. Якщо результат виконання функції SchedulerMC є успішним, то буде виконано зворотне перетворення типі, для того щоб Maui міг інтерпретувати результат планування, отриманий з використанням алгоритмів MC або NLP.

```
#include "../../contrib/sched/MCScheduler.cpp"

int MLocalJobAllocateResources(

  int *MCQ[],                /* I:  job allocating nodes                          */
  mnode_t *RQ[],               /* I:  req allocating nodes                          */
  mnalloc_t NodeList[],      /* I:  eligible nodes                                */
  mulong  StartTime,         /* I                                                 */
  int     RQIndex,           /* I:  index of job req to evaluate                  */
  int     MinTPN[],          /* I:  min tasks per node allowed                    */
  int     MaxTPN[],          /* I:  max tasks per node allowed                    */
  char    NodeMap[],         /* I:  array of node alloc states                    */
  int     AffinityLevel,     /* I:  current reservation affinity level to evaluate */
  int     NodeIndex[],       /* I/OUT:  index of next node to find in BestList    */
  mnalloc_t *BestList[MAX_MREQ_PER_JOB], /* I/OUT:    list of selected nodes      */
  int     TaskCount[],       /* I/OUT:  total tasks allocated to job req          */
  int     NodeCount[])       /* I/OUT:  total nodes allocated to job req          */

  {
    if (MauiToCppAdapter(
        MCQ,
        NodeList,
        RQ) == SUCCESS)
        {
            return SUCCESS;
        }
        else
        {
            return FAILURE;
        }
}
```

Рис. В.13. Виклик функції-адаптера для реалізації алгоритму NLP

```
int ShedulerMC(v pool, v nodelist, s *_backoff, vv _buffer) {

    int count_task_pool = pool.size() - 1;
    int COUNT_RES = nodelist.size() - 1;
    set<int> use_res;
    set<int> setTask;
    set<int> setRes;
    vector<set<int> > incRes;
    vector<set<int> > incTask;

    vector<int> cover;
    vector<int> i_dr(COUNT_RES + 1, 0);
    _backoff.clear();

    for (int i = 1; i <= count_task_pool; i++) setTask.insert(i);
    for (int i = 1; i <= COUNT_RES; i++) setRes.insert(i);

    MatrixToVectorSet(pool, incRes, incTask);

    while ((setTask.size())&&(setRes.size())) {
        set<int> copySetRes(setRes);
        set<int>::iterator it_res;
        for (it_res = copySetRes.begin(); it_res != copySetRes.end(); it_res++) {
            STEP_SHEDULER++;
            if (_buffer[*it_res].size() == BUFFER_SIZE + 1) {
                set<int>::iterator it_task;
                for (it_task = incRes[*it_res].begin(); it_task != incRes[*it_res].end(); it_task++) {
                    STEP_SHEDULER++;
                    incTask[*it_task].erase(*it_res);
                    if (incTask[*it_task].size() == 0) {
                        STEP_SHEDULER++;
                        _backoff.insert(pool[*it_task]);
                        setTask.erase(*it_task);
                        /*-------------------------------------*/
        }
        cover = SearchCover(setRes, setTask, incRes, incTask);
```

Рис. В.14. Функція SchedulerMC (NLP)

Далі здійснюється запуск спланованих завдань. У структурі, яка описує завдання в Maui, використовується поле для запису вузлів, на яких може виконатися завдання. На виході алгоритму кожне завдання має цей запис, дані якої передаються завданням на виконання за допомогою функції MJobStart (реалізована в файлі MJob.c в каталозі дистрибутива maui-3.*.*\Src\moab\.).

Функція MJobStart викликається після останнього в циклі роботи із завданнями виклику функції MJobAllocNLP, а саме після завершення самого циклу.

```
vector<int> SearchCover(s col, s row, vs vcol, vs vrow) {
    vector<int> cover;
    if (col.size() == 0) {
        return cover;
    }
    CALL_COVER++;
    bool seach;
    set<int>::iterator it_col;
    set<int>::iterator it_row;

    do {
        do {
            seach = false;
            set<int> row_(row);
            for (it_row = row_.begin(); it_row != row_.end(); it_row++) {
                STEP_SHEDULER++;
                if (vrow[*it_row].size() == 1) {
                    cover.push_back(*vrow[*it_row].begin());
                    Multiply(*vrow[*it_row].begin(), row, col, vrow, vcol);
                    if (row.size() < 1) return cover;
                    seach = true;
                }
            }
        } while (seach);
        do {
            seach = false;
            set<int> col_(col);
            for (it_col = col_.begin(); it_col != col_.end(); it_col++) {
                STEP_SHEDULER++;
                if (vcol[*it_col].size() == 1)
                    if (vrow[*vcol[*it_col].begin()].size() == 2)
                    {
                        /*----------------------------------------*/
```

Рис. В.15. Функція SearchCover

В результаті проведеного дослідження отримано гнучке і безпечне з точки зору роботи механізмів Maui (реалізації політик планування) рішення, що дозволяє вибирати метод планування і перемикатися між стандартним алгоритмом планування, використовуваним в планувальнику Maui, і новим алгоритмом планування на основі рішення NLP або (MC) з використанням конфігураційного файлу Maui.

# ДОДАТОК Г. ЛІСТІНГ ПРОГРАМИ МОДЕЛЮВАННЯ РОБОТИ ПЛАНУВАЛЬНИКА ЗАВДАНЬ СПО КЛАСТЕРА ТКС НА ОСНОВІ GRID-ТЕХНОЛОГІЇ

Лістінг програми моделювання роботи планувальника завдань СПО кластера ТКС на основі Grid-технології

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit_ThreadWork.h"
#include "Unit_FormMain.h"
#include "Unit_FormCluster.h"
#include "Unit_FormTest.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------

//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(&UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall ThreadWork::UpdateCaption()
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall ThreadWork::ThreadWork(bool CreateSuspended)
        : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall ThreadWork::LogBegin()
{
        OUTFILE.open("log_work.txt");

        OUTFILE<<"Входные задачи"<<endl;
        for(int i = 1; i < INC.size(); ++i)
        {
                OUTFILE<<"задача "<<i<<"   : сложность "<<SOLVER[i]<<"
        размер "<<VOLUME[i]<<"     res ";
```

```
                for(int j = 1; j < INC[i].size(); ++j) if(INC[i][j] == 1)
OUTFILE<<j<<" ";
                OUTFILE<<endl;
        }

        OUTFILE<<endl;
        OUTFILE<<"Ресурсы                              ";
        for(int i = 1; i < SPEED.size(); ++i)  OUTFILE<<i<<"    ";
        OUTFILE<<endl;

        OUTFILE<<"Производительность   ";
        for(int i = 1; i < PRODUC.size(); ++i)  OUTFILE<<PRODUC[i]<<" ";
        OUTFILE<<endl;

        OUTFILE<<"Пропускная способность      ";
        for(int i = 1; i < SPEED.size(); ++i)  OUTFILE<<SPEED[i]<<"         ";
        OUTFILE<<endl;

        OUTFILE<<"Поток       ";
        for(int i = 0; i < FLOW.size(); ++i)  OUTFILE<<FLOW[i]<<" ";
        OUTFILE<<endl;
}
//---------------------------------------------------------------------------

void __fastcall ThreadWork::LogStep()
{
        OUTFILE<<endl;
        OUTFILE<<"--------------------------------- IIIаг "<<STEP<<" ----------------
----------------"<<endl;
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::LogPool(AnsiString comments,const v &pool)
{
        OUTFILE<<endl;
        OUTFILE<<comments.c_str()<<" ("<<pool.size() - 1<<") : ";
        for(int i = 1; i < pool.size(); ++i)
OUTFILE<<pool[i]<<"("<<SW[pool[i]]<<") ";
        OUTFILE<<endl;
}
//---------------------------------------------------------------------------
```

```
void __fastcall ThreadWork::LogBuffer(const vv &buffer)
{
        OUTFILE<<endl;
        OUTFILE<<"Количество задач в буфере : "<<buffer[0][0]<<endl;
        for(int j = 1; j < buffer.size(); ++j)
        {
                OUTFILE<<"буфер ресурса "<<j<<" : ";
                for(int i = 1; i < buffer[j].size(); ++i)
                        OUTFILE<<buffer[j][i]<<"("<<SS[buffer[j][i]]<<") ";
                OUTFILE<<endl;
        }
}
//---------------------------------------------------------------------------

void __fastcall ThreadWork::LogRes(const v &res_number,const v
&res_delay,const v &res_solver)
{
        OUTFILE<<endl;
        OUTFILE<<"Номера ресурсов                                       : ";
        for(int i = 1; i < res_number.size(); ++i)  OUTFILE<<i<<"       ";
        OUTFILE<<endl;
        OUTFILE<<"Задача в ресурсе                                      : ";
        for(int i = 1; i < res_number.size(); ++i)  OUTFILE<<res_number[i]<<"
        ";
        OUTFILE<<endl;

        OUTFILE<<"Задержка                                              : ";
        for(int i = 1; i < res_delay.size(); ++i)  OUTFILE<<res_delay[i]<<"   ";
        OUTFILE<<endl;

        OUTFILE<<"Количество тактов до решения     : ";
        for(int i = 1; i < res_solver.size(); ++i)  OUTFILE<<res_solver[i]<<" ";
        OUTFILE<<endl;

        OUTFILE<<endl;
        OUTFILE<<"Количество решенных задач :
"<<COUNT_SOLVER_TASK<<endl;
}
//---------------------------------------------------------------------------
void __fastcall ThreadWork::LogEnd()
{
        OUTFILE<<endl;
        OUTFILE<<"Результаты работы:"<<endl;
```

```
	OUTFILE<<"время выполнения системой всех задач:
"<<STEP_EXECUTION<<endl;
	OUTFILE<<"среднее время ответа системы:
"<<STEP_RESPEND<<endl;
	OUTFILE<<"среднее время ожидания(нахождение задачи в пулле):
"<<STEP_WAIT<<endl;
	OUTFILE<<"среднее время обслуживания(нахождение задачи в
буффере): "<<STEP_SERVICE<<endl;
	OUTFILE<<"среднее время решения: "<<STEP_SOLVER<<endl;
	OUTFILE<<"среднее время затраченное на планирование одной задачи:
"<<STEP_PLANNING<<endl;

	OUTFILE<<"средний коэффициент использования ресурсов:
"<<RATE_UTILIZATION<<endl;
	OUTFILE<<"средний коэффициент загрузки ресурсов:
"<<RATE_LOAD<<endl;
	OUTFILE<<"коэффициент ускорения: "<<RATE_ACCELERAT<<endl;

	OUTFILE<<"количество решений задач о покрытии:
"<<CALL_COVER<<endl;
	OUTFILE<<"количество планирований:
"<<COUNT_SHEDULER<<endl;

	OUTFILE<<"вектор коэффициентов использвания ресурсов: "<<endl;
	for(int i = 1; i < RU.size(); ++i) OUTFILE<<RU[i]<<" ";

	OUTFILE<<endl;
	OUTFILE<<"вектор коэффициентов загрузки ресурсов: "<<endl;
	for(int i = 1; i < RL.size(); ++i) OUTFILE<<RL[i]<<" ";

	OUTFILE<<endl;
	OUTFILE<<"количество задач вошедших в покрытие на тактах
планирования: "<<endl;
	for(int i = 1; i < COUNT_COVER.size(); ++i)
OUTFILE<<COUNT_COVER[i]<<" ";

	OUTFILE<<endl;
	OUTFILE<<"вектор распределения задач по ресурсам: "<<endl;
	for(int i = 1; i < DR.size(); ++i)
	{
		for(int j = 1; j < DR[j].size(); ++j) OUTFILE<<DR[i][j]<<"	";
		OUTFILE<<endl;
	}
```

```
        OUTFILE.close();
}
//---------------------------------------------------------------------------


//функция отображает ход решения задач в режиме моделирования типового кластера
void __fastcall ThreadWork::ProgressFormCluster()
{
        FormCluster = dynamic_cast<TFormCluster*>(FormMain->ActiveMDIChild);
        FormCluster->ProgressBar->Position = COUNT_SOLVER_TASK;
        Application->ProcessMessages();
}
//---------------------------------------------------------------------------


//функция отображает метод которы производится планирование, в режиме тестирования
void __fastcall ThreadWork::ProgressFormTest()
{
        FormTest = dynamic_cast<TFormTest*>(FormMain->ActiveMDIChild);
        String method_working;

        switch(METHOD)
        {
                case 0: method_working = "Идет решение задач(MC1)...";
                                break;

                case 1: method_working = "Идет решение задач(MC2)...";
                                break;

                case 2: method_working = "Идет решение задач(MC3)...";
                                break;

                case 3: method_working = "Идет решение задач(FCFS)...";
        }

        FormTest->GridProgress->Cells[5][1] = method_working;
        Application->ProcessMessages();
}
//---------------------------------------------------------------------------
```

```
//функция отображает ход решения задач в режиме тестирования
void __fastcall ThreadWork::ProgressFormTestGuadge()
{
        FormTest = dynamic_cast<TFormTest*>(FormMain->ActiveMDIChild);

        FormTest->Gauge->Progress = COUNT_SOLVER_TASK;

        Application->ProcessMessages();
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::Execute()
{
        if (LOG)
                LogBegin();

        if (CALL_TEST)
           Synchronize(ProgressFormTest);

        COUNT_COVER.clear();

        STEP = 0;
        COUNT_SOLVER_TASK = 0;
        LAST_SEND_TASK = 1;

        int freq_sheduler   = 1;    //частота планирования

        int freq_send               = 0;
                                    //частота поступления задач на вход системы
        int count_task_pool = 0;
                                //количество задач введенных в пулл кластера
        int i_send                  = 0;
                                    //индек вектора посылки заданий при
случайном количестве поступления заданий на вход

        vector<int>           pool(1,0);
                                //пул - задачи поступившие на вход системы за
период планирования
        vector<vector<int> > buffer(COUNT_RES + 1,vector<int>(1,0));
                //буферы ресурсов
        vector<int>                 res_number(COUNT_RES + 1,0);
                                //текущее состояние решения задачи на ресурсе
```

```
	vector<int>		res_delay(COUNT_RES + 1,0);
		//текущая задержка на ресурсе
	vector<int>		res_solver(COUNT_RES + 1,0);
	set<int>		backoff;
				//множество возвращенных задач из буффера на
i-ом такте

	STEP_PLANNING = 0;					//среднее
время планирование
	COUNT_SHEDULER	= 0;
					//количество проведенных планирований
за все время решения задач
	CALL_COVER = 0;
	SR	= vector<int>(COUNT_TASK + 1,0);
	//вектор времени ответа системы
	SW	= vector<int>(COUNT_TASK + 1,0);
	//вектор времени ожидания задачи в пулле
	SS	= vector<int>(COUNT_TASK + 1,0);
	//вектор времени обслуживания задачи  в буффере
	SSO = vector<int>(COUNT_TASK + 1,0);
	//вектор времени решения задачи
	ST  = vector<int>(COUNT_TASK + 1,0);
	//вектор времени передачи задачи
	RU	= vector<double>(COUNT_RES + 1,0);
	//вектор коєфициентов использования ресурсов
	RL	= vector<double>(COUNT_RES + 1,0);
	//вектор коєфициентов производительности ресурсов
	DR	= vector<vector<int> >();

	SR.reserve(COUNT_TASK + 1);
	SW.reserve(COUNT_TASK + 1);
	SS.reserve(COUNT_TASK + 1);
	SSO.reserve(COUNT_TASK + 1);
	ST.reserve(COUNT_TASK + 1);
	RU.reserve(COUNT_RES + 1);
	RL.reserve(COUNT_RES + 1);

	while(COUNT_SOLVER_TASK < COUNT_TASK)
	{
		if (LOG)
			LogStep();

		if(freq_send == 0) {
			TaskToPool(STEP,i_send,pool);
```

```
                freq_send = FREQ_SHEDULER;
                if (LOG)
                        LogPool("Задачи в пуле до планирования",pool);
        }

        if(freq_sheduler <= 0) {
                freq_sheduler = FREQ_SHEDULER;
                if(pool.size() > 1) {
                        STEP_SHEDULER = 0;          //обнуляем счетчик
тактов потраченных на планирование
                        switch(METHOD) {
                        case DEF_MC1:
                                switch(MC1_TYPE) {
                                case 0:
                                        ShedulerMCRang(pool,backoff,buffer);
                                        break;
                                case 1:
                                        ShedulerMC(pool,backoff,buffer);
                                        break;
                                case 2:
                                        ShedulerMCFree(pool,backoff,buffer);
                                        break;
                                case 3:
                                        ShedulerMCProduce(pool,backoff,buffer);
                                        break;
                                case 4:
                                        ShedulerGREEDE(pool,backoff,buffer);
                                        break;
                                }
                                break;
                        case DEF_MC2:
                                switch(MC2_TYPE) {
                                case 0:
                                        ShedulerMCRang(pool,backoff,buffer);
                                        break;
                                case 1:
                                        ShedulerMC(pool,backoff,buffer);
                                        break;
                                case 2:
                                        ShedulerMCFree(pool,backoff,buffer);
                                        break;
                                case 3:
                                        ShedulerMCProduce(pool,backoff,buffer);
                                        break;
```

```
                    case 4:
                        ShedulerGREEDE(pool,backoff,buffer);
                        break;
                    }
                    break;
                case DEF_MC3:
                    switch (MC3_TYPE) {
                    case 0:
                        ShedulerMCRang(pool,backoff,buffer);
                        break;
                    case 1:
                        ShedulerMC(pool,backoff,buffer);
                        break;
                    case 2:
                        ShedulerMCFree(pool,backoff,buffer);
                        break;
                    case 3:
                        ShedulerMCProduce(pool,backoff,buffer);
                        break;
                    case 4:
                        ShedulerGREEDE(pool,backoff,buffer);
                        break;
                    }
                    break;
                case DEF_FCFS:
                    ShedulerFCFS(pool,buffer);
                    freq_sheduler = 1;
                }

                //возвращаем в пул задания которые не поместились в буффер, кроме FCFS
                if (METHOD != DEF_FCFS)
                    TaskBackoffToPool(pool,backoff);

                //время планирования уменьшаем на порядок
                STEP_SHEDULER = STEP_SHEDULER / DIVIDE_SP == 0 ? 1 : STEP_SHEDULER / DIVIDE_SP;

                //считаем среднее количество тактов потраченное системой на планирование
                STEP_PLANNING += STEP_SHEDULER;

                COUNT_SHEDULER++;
```

```
                        while(STEP_SHEDULER > freq_sheduler) {
                            if(METHOD == 3)
                                freq_sheduler++;
                            else
                               freq_sheduler += FREQ_SHEDULER;
                        }

                        if(LOG) LogPool("Задачи в пуле после
планирования",pool);
                    }
                }

                TaskSolver(buffer,res_number,res_delay,res_solver);

                freq_send--;
                freq_sheduler--;

                STEP++;

                if(CALL_TEST)
                    Synchronize(ProgressFormTestGuadge);
                else
                    Synchronize(ProgressFormCluster);

                if (LOG)
                    LogBuffer(buffer);

                if (LOG)
                    LogRes(res_number,res_delay,res_solver);

                if (Terminated)
                    return;
            }

            //вычисляем коэффициенты производительности и использования
            CalculateRate();

            //находим средние значения временных характеристик работы кластера
            STEP_PLANNING = (double) STEP_PLANNING / COUNT_SHEDULER;
            STEP_RESPEND = 0;
            STEP_WAIT = 0;
            STEP_SERVICE = 0;
            STEP_SOLVER = 0;
```

```
	for(int i = 1; i <= COUNT_TASK; ++i) {
		STEP_RESPEND += SR[i];
		STEP_WAIT += SW[i];
		STEP_SERVICE += SS[i];
		STEP_SOLVER += SSO[i];
		STEP_TRANSFER += ST[i];
	}

	STEP_EXECUTION = STEP;
	STEP_RESPEND = (double) STEP_RESPEND / COUNT_TASK;
	STEP_WAIT = (double) STEP_WAIT / COUNT_TASK;
	STEP_SERVICE = (double) STEP_SERVICE / COUNT_TASK;
	STEP_SOLVER = (double) STEP_SOLVER / COUNT_TASK;
	STEP_TRANSFER = (double) STEP_TRANSFER / COUNT_TASK;

	DENSIT_COVER = (double) DENSIT_COVER / COUNT_SHEDULER;

	for(int i = 0; i < DR.size(); ++i)
		for(int j = 1; j <= COUNT_RES; ++j) DR[i][0] += DR[i][j];

	if(LOG) LogEnd();
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::TaskToPool(int step,int& _i,v &_pool)
{
	if(_i > FLOW.size() - 1) return;

	while((FLOW[_i] > 0) && (_pool.size() < POOL_SIZE))
	{
		_pool.push_back(LAST_SEND_TASK);
		SR[LAST_SEND_TASK] = step;

		LAST_SEND_TASK++;
		FLOW[_i]--;
	}

	if(FLOW[_i] == 0) _i++;
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::TaskBackoffToPool(v &_pool,s &_backoff)
```

```
{
	_pool = vector<int>(1,0);

	for(set<int>::iterator it =_backoff.begin(); it != _backoff.end(); it++)
_pool.push_back(*it);

	_backoff.clear();
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::ShedulerMC(const v &pool,s &_backoff,vv
&_buffer)
{
	int count_task_pool = pool.size() - 1;//число задач в пулле

	//множество номеров ресурсов в которые была помещенна хотя бы
одна задача
	set<int> use_res;
	//множество не распределенных задач
	set<int> setTask;
	//множество свободных ресурсов
	set<int> setRes;

	vector<set<int> > incRes;	//десятичное представление соответствия
ресурсам задач, которые могут решаться на нем
	vector<set<int> > incTask;	//десятичное представление соответствия
задачам ресурсов, которыми она может решится
	vector<int> cover; //минимальное вершинное покрытие
	vector<int> i_dr(COUNT_RES + 1,0);

	//очищаем множество возвращенных задач
	_backoff.clear();

	for(int i = 1; i <= count_task_pool; ++i) setTask.insert(i);   //формируем
множество не распределенных задач
	for(int i = 1; i <= COUNT_RES; ++i) setRes.insert(i);   //формируем
множество свободных ресурсов

	//формирование десятичного представления соответствий задач в пулле
	MatrixToVectorSet(pool,incRes,incTask);

	int freeRes = countFreeRes(setRes,_buffer);
```

```
	while((setTask.size())&&(setRes.size()))
	{
		//исключение занятых ресурсов

	deleteOccupRes(setRes,setRes,setTask,incRes,incTask,_backoff,pool,_buffe
r);

		//поиск минимального вершинного покрытия
		cover = SearchCover(setRes,setTask,incRes,incTask);

		if(ERROR_COVER != 0)
		{
			AddError(cover,setRes);
			sort(cover.begin(),cover.end());
		}

		//добавляем задачи в буфер ресурсов вошедших в минимальное
покрытие
		for(int i = 0; i < cover.size(); ++i)
		{
			STEP_SHEDULER++;

			while((_buffer[cover[i]].size() < BUFFER_SIZE +
1)&&(incRes[cover[i]].size()))
			{
					STEP_SHEDULER++;

					int task = *incRes[cover[i]].begin();
					_buffer[cover[i]].push_back(pool[task]);
					_buffer[0][0]++;

					SW[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP) - SR[pool[task]];
					SS[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP);
					use_res.insert(cover[i]);
					i_dr[cover[i]]++;

					//удаление уже помещенной в буфер задачи из
других ресурсов
					for(set<int>::iterator it_res = incTask[task].begin();
						it_res != incTask[task].end(); it_res++) {
						incRes[*it_res].erase(task);
					}
```

```
                                setTask.erase(task);
                        }
                }
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);

        if (freeRes > 0)
                DENSIT_COVER += (double) use_res.size() / freeRes;
}
//---------------------------------------------------------------------------


 //функция загружает в начале более свободные ресурсы вощедшие в
покрытие
void __fastcall ThreadWork::ShedulerMCFree(const v &pool,s &_backoff,vv
&_buffer)
{
        int count_task_pool = pool.size()-1;//число задач в пулле

        //множество номеров ресурсов в которые была помещенна хотя бы
одна задача
        set<int> use_res;
        //множество не распределенных задач
        set<int> setTask;
        //множество свободных ресурсов
        set<int> setRes;

        vector<set<int> > incRes;       //десятичное представление соответствия
ресурсам задач, которые могут решаться на нем
        vector<set<int> > incTask;      //десятичное представление соответствия
задачам ресурсов, которыми она может решится
        vector<int> cover; //минимальное вершинное покрытие
        vector<int> i_dr(COUNT_RES + 1,0);

        //очищаем множество возвращенных задач
        _backoff.clear();

        for(int i = 1; i <= count_task_pool; ++i) setTask.insert(i);   //формируем
множество не распределенных задач
        for(int i = 1; i <= COUNT_RES; ++i) setRes.insert(i);   //формируем
множество свободных ресурсов
```

```
	//формирование десятичного представления соответствий задач в пулле
	MatrixToVectorSet(pool,incRes,incTask);

	int freeRes = countFreeRes(setRes,_buffer);

	while((setTask.size())&&(setRes.size()))
	{
		//исключение занятых ресурсов

	deleteOccupRes(setRes,setRes,setTask,incRes,incTask,_backoff,pool,_buffe
r);

		//поиск минимального вершинного покрытия
		cover = SearchCoverFree(setRes,setTask,incRes,incTask,_buffer);

		if(ERROR_COVER != 0)
		{
			AddError(cover,setRes);
			sort(cover.begin(),cover.end());
		}

		//добавляем задачи в буфер ресурсов вошедших в минимальное
покрытие
		int indexRes;
		int numberTask;

		STEP_SHEDULER++;

		while(cover.size() > 0)
		{
			indexRes = MaxFree(cover,_buffer);

			STEP_SHEDULER++;

			while((_buffer[cover[indexRes]].size() < BUFFER_SIZE +
1)&&(incRes[cover[indexRes]].size()))
			{
					STEP_SHEDULER++;

					numberTask = *incRes[cover[indexRes]].begin();

	_buffer[cover[indexRes]].push_back(pool[numberTask]);
					_buffer[0][0]++;
```

```
                                //если задача первая в очереди, определяем
задержку передачи ее на ресурс
                                //if(_buffer[cover[indexRes]].size() == 2)
_res_delay[cover[indexRes]] = VOLUME[pool[numberTask]] /
NT[cover[indexRes]];

                                SW[pool[numberTask]] = STEP +
(STEP_SHEDULER / DIVIDE_SP) - SR[pool[numberTask]];
                                SS[pool[numberTask]] = STEP +
(STEP_SHEDULER / DIVIDE_SP);
                                use_res.insert(cover[indexRes]);
                                i_dr[cover[indexRes]]++;

                                //удаление уже помещенной в буфер задачи из
других ресурсов
                                for(set<int>::iterator it_res =
incTask[numberTask].begin();
                                        it_res != incTask[numberTask].end();
it_res++) {

                                        incRes[*it_res].erase(numberTask);
                                }

                                setTask.erase(numberTask);
                        }

                        //удаляем ресурс из покрытия
                        cover.erase(cover.begin() + indexRes);
                }
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);

        if (freeRes > 0)
                DENSIT_COVER += (double) use_res.size() / freeRes;
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::ShedulerMCProduce(const v &pool,s &_backoff,vv
&_buffer)
{
```

```
	int count_task_pool = pool.size()-1;//число задач в пулле

	//множество номеров ресурсов в которые была помещенна хотя бы одна задача
	set<int> use_res;
	//множество не распределенных задач
	set<int> setTask;
	//множество свободных ресурсов
	set<int> setRes;

	vector<set<int> > incRes;		//десятичное представление соответствия ресурсам задач, которые могут решаться на нем
	vector<set<int> > incTask;		//десятичное представление соответствия задачам ресурсов, которыми она может решится
	vector<int> cover; //минимальное вершинное покрытие
	vector<int> i_dr(COUNT_RES + 1,0);

	//очищаем множество возвращенных задач
	_backoff.clear();

	for(int i = 1; i <= count_task_pool; ++i) setTask.insert(i);   //формируем множество не распределенных задач
	for(int i = 1; i <= COUNT_RES; ++i) setRes.insert(i);   //формируем множество свободных ресурсов

	//формирование десятичного представления соответствий задач в пулле
	MatrixToVectorSet(pool,incRes,incTask);

	int freeRes = countFreeRes(setRes,_buffer);

	while((setTask.size())&&(setRes.size()))
	{
		//исключение занятых ресурсов
		//исключение занятых ресурсов

	deleteOccupRes(setRes,setRes,setTask,incRes,incTask,_backoff,pool,_buffer);

			//поиск минимального вершинного покрытия
			cover = SearchCoverProduce(setRes,setTask,incRes,incTask);

			if(ERROR_COVER != 0)
			{
				AddError(cover,setRes);
```

```
                sort(cover.begin(),cover.end());
        }

        //добавляем задачи в буфер ресурсов вошедших в минимальное
покрытие
        int indexRes;
        int numberTask;

        STEP_SHEDULER++;

        while(cover.size() > 0)
        {
                indexRes = MaxProduce(cover);

                STEP_SHEDULER++;

                while((_buffer[cover[indexRes]].size() < BUFFER_SIZE +
1)&&(incRes[cover[indexRes]].size()))
                {
                        STEP_SHEDULER++;

                        numberTask = *incRes[cover[indexRes]].begin();

      _buffer[cover[indexRes]].push_back(pool[numberTask]);
                        _buffer[0][0]++;

                        //если задача первая в очереди, определяем
задержку передачи ее на ресурс
                        //if(_buffer[cover[indexRes]].size() == 2)
_res_delay[cover[indexRes]] = VOLUME[pool[numberTask]] /
NT[cover[indexRes]];

                        SW[pool[numberTask]] = STEP +
(STEP_SHEDULER / DIVIDE_SP) - SR[pool[numberTask]];
                        SS[pool[numberTask]] = STEP +
(STEP_SHEDULER / DIVIDE_SP);
                        use_res.insert(cover[indexRes]);
                        i_dr[cover[indexRes]]++;

                        //удаление уже помещенной в буфер задачи из
других ресурсов
                        for(set<int>::iterator it_res =
incTask[numberTask].begin();
```

```
                                        it_res != incTask[numberTask].end();
it_res++) {

                                        incRes[*it_res].erase(numberTask);
                                }

                                setTask.erase(numberTask);
                        }

                        //удаляем ресурс из покрытия
                        cover.erase(cover.begin() + indexRes);
                }
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);

        if (freeRes > 0)
                DENSIT_COVER += (double) use_res.size() / freeRes;
}
//---------------------------------------------------------------------------


//метод планирования основанный на поиски минимального вершинного
покрытия точным методом Search Min Cover Accurate
void __fastcall ThreadWork::ShedulerMCRang(const v &pool,s &_backoff,vv
&_buffer)
{
        int count_task = pool.size() - 1;//число задач в пулле
        int count_res = COUNT_RES;

        //множество не распределенных задач
        set<int> setTask;
        //множество свободных ресурсов
        set<int> setRes;

        set<int>::iterator itn;
        set<int>::iterator itm;

        vector<set<int> > incRes;       //десятичное представление соответствия
ресурсам задач, которые могут решаться на нем
        vector<set<int> > incTask;      //десятичное представление соответствия
задачам ресурсов, которыми она может решится
        vector<int> cover; //минимальное вершинное покрытие
```

```
        set<int> use_res;
        vector<int> i_dr(COUNT_RES + 1,0);

        //очищаем множество возвращенных задач
        _backoff.clear();

        for(int i = 1; i <= count_task; ++i) setTask.insert(i);   //формируем
множество не распределенных задач
        for(int i = 1; i <= count_res; ++i) setRes.insert(i);   //формируем
множество свободных ресурсов

        //формирование десятичного представления соответствий задач в пулле
        MatrixToVectorSet(pool,incRes,incTask);

        int freeRes = countFreeRes(setRes,_buffer);

        while((setTask.size())&&(setRes.size()))
        {

                //исключение занятых ресурсов и формируем вектор новой
нумерации ресурсов
                //так как нам нужно исключить из матрицы занятые ресурсы
                unsigned new_count_res = 0; //новый порядковый номер ресурса
                vector<int> newNumRes(count_res + 1,0); //вектор ресурсов новых
номеров
                vector<int> oldNumRes(1,0);//новый вектор ресурсов содержащий
старые номера

                unsigned new_count_task = 0; //новый порядковый номер ресурса
                vector<int> newNumTask(count_task + 1,0); //вектор ресурсов
новых номеров
                vector<int> oldNumTask(1,0);//новый вектор ресурсов
содержащий старые номера

                set<int> _setRes(setRes);

                //удаляем пустые и занятые ресурсы и задачи у которых все
ресурсы заняты
                for(itn = _setRes.begin(); itn != _setRes.end(); itn++)
                {
                  STEP_SHEDULER++;
```

```
                if( (_buffer[*itn].size() == BUFFER_SIZE + 1) ||
(incRes[*itn].size() == 0) )//удаляем либо занятые ресурсы,либо те у которых
нет решаемых задач из текущего пула
                {
                        for(itm = incRes[*itn].begin(); itm != incRes[*itn].end();
itm++)
                        {
                                STEP_SHEDULER++;

                                //удаляем занятый ресурс из задачи
                                incTask[*itm].erase(*itn);

                                //если у задачи нет свободных ресурсов
добавляем ее в возврат
                                if(incTask[*itm].size() == 0)
                                {
                                        STEP_SHEDULER++;

                                        _backoff.insert(pool[*itm]);
                                        setTask.erase(*itm);
                                }
                        }

                        //удаляем ресурс и очищаем множество задач которое
может решать ресурс
                        setRes.erase(*itn);
                        incRes[*itn].clear();
                  }
                }

                //формируем новую порядковую нумерацию ресурсов и
запоминаем их старые номера
                for(itn = setRes.begin(); itn != setRes.end(); itn++)
                {
                        STEP_SHEDULER++;

                        new_count_res++;
                        newNumRes[*itn] = new_count_res;
                        oldNumRes.push_back(*itn);
                }

                //формируем новую порядковую нумерацию задач и запоминаем
их старые номера
                for(itm = setTask.begin(); itm != setTask.end(); itm++)
```

```
		{
			STEP_SHEDULER++;

			new_count_task++;
			newNumTask[*itm] = new_count_task;
			oldNumTask.push_back(*itm);
		}


		//формируем матрицу двоичного представления задач в пулле(матрицу инцидентности)
		vector<int> vecNull(new_count_res + 1,0);
		vector<vector<int> > b(new_count_task + 1,vecNull);//матрица инцидентности

		for(itm = setTask.begin(); itm != setTask.end(); itm++)
			for(itn = incTask[*itm].begin(); itn != incTask[*itm].end(); itn++)
			{
				b[newNumTask[*itm]][newNumRes[*itn]] = 1;
			}

		for(int i = 1; i <= new_count_res; ++i)
		{
			b[0][i] = incRes[oldNumRes[i]].size();
		}

		//поиск минимального вершинного покрытия
		cover = SearchCoverAccurate(b);

		if(ERROR_COVER != 0)
		{
			AddError(cover,setRes);
			sort(cover.begin(),cover.end());
		}

		//добавляем задачи в буфер ресурсов вошедших в минимальное покрытие
		set<int>::iterator _itn;
		for(int i =0; i < cover.size(); ++i)
		{
			STEP_SHEDULER++;

			int task;
```

```
                        while((_buffer[oldNumRes[cover[i]]].size() < BUFFER_SIZE
+ 1)&&(incRes[oldNumRes[cover[i]]].size() != 0))
                        {
                                STEP_SHEDULER++;

                                task = *incRes[oldNumRes[cover[i]]].begin();

        _buffer[oldNumRes[cover[i]]].push_back(pool[task]);
                                _buffer[0][0]++;

                                SW[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP) - SR[pool[task]];
                                SS[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP);
                                use_res.insert(oldNumRes[cover[i]]);
                                i_dr[oldNumRes[cover[i]]]++;

                                for(set<int>::iterator it_res = incTask[task].begin();
it_res != incTask[task].end(); it_res++)
                                        incRes[*it_res].erase(task);

                                STEP_SHEDULER++;

                                setTask.erase(task);
                        }
                }

                cover.clear();
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);

        if (freeRes > 0)
                DENSIT_COVER += (double) use_res.size() / freeRes;
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::ShedulerFCFSB(const v &pool,s &_backoff,vv
&_buffer)
{
        int count_task_pool = pool.size() - 1;//число задач в пулле
```

```
	//множество номеров ресурсов в которые была помещенна хотя бы одна задача
	set<int> use_res;
	vector<int> i_dr(COUNT_RES + 1,0);
	//десятичное представление соответствия ресурсам задач, которые могут решаться на нем
	vector<set<int> > incRes;
	//десятичное представление соответствия задачам ресурсов, которыми она может решится
	vector<set<int> > incTask;

	//очистка множества возвращенных задач
	_backoff.clear();

	//формирование десятичного представления соответствий задач в пулле
	MatrixToVectorSet(pool,incRes,incTask);

	//помещение задач  в буфферы
	for(int i = 1; i <= count_task_pool; ++i)
	{
		STEP_SHEDULER++;

		_backoff.insert(pool[i]);//добавляем задачу в возврат

	  for(set<int>::iterator it = incTask[i].begin(); it != incTask[i].end(); it++)
	  {
			STEP_SHEDULER++;

			//проверка свободен ли буффер
			if(_buffer[*it].size() < BUFFER_SIZE + 1)
			{
				STEP_SHEDULER++;

				_buffer[*it].push_back(pool[i]);//добавляем задачу в буфер ресурса
				_backoff.erase(pool[i]);//удаляем задачу помещенную в буфер из возврата

				SW[pool[i]] = STEP + (STEP_SHEDULER / DIVIDE_SP) - SR[pool[i]];
				SS[pool[i]] = STEP + (STEP_SHEDULER / DIVIDE_SP);
				use_res.insert(*it);//запоминаем номер ресурса в который была размещена задача
```

```
                    i_dr[*it]++;

                    //переходим к следующей задаче в пуле
                    break;
                }
            }
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::ShedulerFCFS(v &_pool,vv &_buffer)
{
        //множество номеров ресурсов в которые была помещенна хотя бы
одна задача
        set<int> use_res;
        vector<int> i_dr(COUNT_RES + 1,0);

        for(int resi = 1; resi <= COUNT_RES; ++resi) {

                STEP_SHEDULER++;

                //проверяем может ли решатся задача на данном ресурсе
                if (INC[_pool[1]][resi] == 1) {
                        //проверяем свободен ли буффер
                        if (_buffer[resi].size() < 2) {
                                _buffer[resi].push_back(_pool[1]);//добавляем задачу в
буфер ресурса
                                _pool.erase(_pool.begin() + 1);

                                SW[_pool[1]] = STEP + (STEP_SHEDULER /
DIVIDE_SP) - SR[_pool[1]];
                                SS[_pool[1]] = STEP + (STEP_SHEDULER /
DIVIDE_SP);
                                use_res.insert(resi);//запоминаем номер ресурса в
который была размещена задача
                                i_dr[resi]++;

                                COUNT_COVER.push_back(1);
                                DR.push_back(i_dr);
                                return;
```

```
                        }
                    }
                }
            }
//---------------------------------------------------------------------------


void __fastcall ThreadWork::ShedulerGREEDE(const v &pool,s &_backoff,vv
&_buffer)
{
        int count_task_pool = pool.size()-1;//число задач в пулле

        //множество номеров ресурсов в которые была помещенна хотя бы
одна задача
        set<int> use_res;
        vector<int> i_dr(COUNT_RES + 1,0);
        //множество не распределенных задач
        set<int> setTask;
        //множество свободных ресурсов
        set<int> setRes;

        vector<set<int> > incRes;        //десятичное представление соответствия
ресурсам задач, которые могут решаться на нем
        vector<set<int> > incTask;       //десятичное представление соответствия
задачам ресурсов, которыми она может решится
        vector<int> cover; //минимальное вершинное покрытие

        //очищаем множество возвращенных задач
        _backoff.clear();

        for(int i = 1; i <= count_task_pool; ++i) setTask.insert(i);   //формируем
множество не распределенных задач
        for(int i = 1; i <= COUNT_RES; ++i) setRes.insert(i);   //формируем
множество свободных ресурсов

        //формирование десятичного представления соответствий задач в пулле
        MatrixToVectorSet(pool,incRes,incTask);

        int freeRes = countFreeRes(setRes,_buffer);

        while((setTask.size())&&(setRes.size()))
        {
                //исключение занятых ресурсов
                //исключение занятых ресурсов
```

```
deleteOccupRes(setRes,setRes,setTask,incRes,incTask,_backoff,pool,_buffe
r);

		//поиск минимального вершинного покрытия
		cover = SearchCoverGreed(setTask.size(),setRes,incRes);

		if(ERROR_COVER != 0)
		{
			AddError(cover,setRes);
			sort(cover.begin(),cover.end());
		}

		//добавляем задачи в буфер ресурсов вошедших в минимальное
покрытие
		for(int i = 0; i < cover.size(); ++i)
		{
			STEP_SHEDULER++;

			while((_buffer[cover[i]].size() < BUFFER_SIZE +
1)&&(incRes[cover[i]].size()))
			{
					STEP_SHEDULER++;

					int task = *incRes[cover[i]].begin();
					_buffer[cover[i]].push_back(pool[task]);
					_buffer[0][0]++;

					SW[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP) - SR[pool[task]];
					SS[pool[task]] = STEP + (STEP_SHEDULER /
DIVIDE_SP);
					use_res.insert(cover[i]);
					i_dr[cover[i]]++;

					//удаление уже помещенной в буфер задачи из
других ресурсов
					for(set<int>::iterator it_res = incTask[task].begin();
						it_res != incTask[task].end(); it_res++) {

						incRes[*it_res].erase(task);
					}

					setTask.erase(task);
```

```
                }
            }
        }

        COUNT_COVER.push_back(use_res.size());
        DR.push_back(i_dr);

        if (freeRes > 0)
            DENSIT_COVER += (double) use_res.size() / freeRes;
}
//---------------------------------------------------------------------------


//считает количество занятых ресурсов
int __fastcall ThreadWork::countFreeRes(const s &res,const vv &buffer) {

        int freeRes = 0;
        for(set<int>::const_iterator it_res = res.begin();
            it_res != res.end(); ++it_res) {

            if (buffer[*it_res].size() == BUFFER_SIZE + 1)
                ++freeRes;
        }

        return freeRes;
}
//---------------------------------------------------------------------------


//исключение занятых ресурсов
void __fastcall ThreadWork::deleteOccupRes(const s &res,s &_res,s &_task,
                                           vs &_incRes,vs &_incTask,s
&_backoff,
                                    const v &pool,const vv &buffer) {

        for(set<int>::const_iterator it_res = res.begin();
            it_res != res.end(); ++it_res) {

            ++STEP_SHEDULER;

            if (buffer[*it_res].size() == BUFFER_SIZE + 1) {
                for (set<int>::iterator it_task = _incRes[*it_res].begin();
                    it_task != _incRes[*it_res].end(); ++it_task) {
```

```
                        ++STEP_SHEDULER;

                        _incTask[*it_task].erase(*it_res);

                        //если у задачи нет свободных ресурсов добавляем ее
в возврат и удаляем из множества
                        if (_incTask[*it_task].size() == 0) {
                                _backoff.insert(pool[*it_task]);
                                _task.erase(*it_task);
                        }
                }

                //удаляем занятый ресурс
                _incRes[*it_res].clear();
                _res.erase(*it_res);
        }
    }
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::TaskSolver(vv &_buffer,v &_res_number,v
&_res_delay,v &_res_solver)
{
    for(int i = 1; i <= COUNT_RES; ++i)
    {
        if(_res_number[i] > 0)
        {
            if(_res_delay[i] == 0)
            {
                int first_in_buf = _buffer[i].size() > 1 ? _buffer[i][1] : -1;
    //вначале убедимся что в буфере есть хотя бы одна задача

                if(_res_number[i] == first_in_buf)  //если ресурс на
этом такте поступил на ресурс, удаляем его из буфера и запоминаем время
нахождения задачи в буфере
                {
            ST[_res_number[i]] = STEP - ST[_res_number[i]];
                        SSO[_res_number[i]] = STEP;
                        SS[_res_number[i]] = STEP - SS[_res_number[i]];
                        _buffer[i].erase(_buffer[i].begin() + 1);
                        _res_solver[i] = SOLVER[_res_number[i]];
                }
                else
```

```
                                if(_res_solver[i] <= 0)
                                {
                                        SR[_res_number[i]] = STEP -
SR[_res_number[i]];          //количество тактов за которое была решена
задача
                                        SSO[_res_number[i]] = STEP -
SSO[_res_number[i]];

                                        RL[i]++;
                                                //количество решенных задач
ресурсом
                                        COUNT_SOLVER_TASK++;
//количество задач которые уже решены системой

                                        _res_number[i] = 0;

                                        if(_buffer[i].size() > 1)
                                                if(SS[_buffer[i][1]] <= STEP)
                                                {
                                                        _res_number[i] = _buffer[i][1];
                                                        _res_delay[i]  =
VOLUME[_res_number[i]] / SPEED[i];
                                                        ST[_res_number[i]] = STEP;
                                                }
                                }
                                else //имитируем решение задачи и
подсчитываем количество тактов которые работал ресурс
                                {
                                        _res_solver[i] -= PRODUC[i];
                                        RU[i]++;
                                }
                        }
                        else // иметируем задержку в линии передачи задачи из
буфера в ресурс
                        {
                                _res_delay[i]--;
                        }
                }
                else //если в буфере ресурса есть задача отправляем ее на решение
                {
                        if(_buffer[i].size() > 1)
                                if(SS[_buffer[i][1]] <= STEP)
                                {
                                        _res_number[i] = _buffer[i][1];
```

```
                                        _res_delay[i]  = VOLUME[_res_number[i]] /
SPEED[i];
                                        ST[_res_number[i]] = STEP;
                                }
                        }
                }
        }
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::MatrixToVectorSet(const v &pool,vs &_incRes,vs
&_incTask)
{
  //преобразовываем матрицу инцидентности в десятичное представление
соответствий задач ресурсам

  //создаем промежуточные переменные
  int count_res = INC[0].size() - 1;//количество ресурсов
  int count_task_pool = pool.size() - 1;//количество задач в пулле
  set<int> set_int;

  //формируем вектор множеств ресурсов, в которых указаны какие задачи
может решать ресурс
  _incRes.reserve(count_res + 1);
  _incRes = vector<set<int> >(count_res + 1,set_int);

  //формируем вектор множеств задач, в которых указыны какими ресурсами
может решаться задача
  _incTask.reserve(count_task_pool + 1);
  _incTask = vector<set<int> >(count_task_pool + 1,set_int);

  //перебираем все значени матрицы инциндетности
  for(int i = 1; i <= count_task_pool; ++i)
  {
        for(int j = 1; j <= count_res; ++j)
        {
                STEP_SHEDULER++;

                if(INC[pool[i]][j] == 1)
                {
                        //если i-й задаче соответствует j-й ресурс то
                        //в множество задач добавляем номер этого ресурса а в
множество ресурса добавляем номер задачи
                        _incTask[i].insert(j);
```

```
                        _incRes[j].insert(i);
                }
        }
  }
}
//---------------------------------------------------------------------------


vector<int> __fastcall ThreadWork::SearchCover(s col,s row,vs vcol,vs vrow)
{
  vector<int> cover;//множество столбцов являющихся минимальным
покрытием

        if(col.size() == 0)
        {
                return cover;
        }

        CALL_COVER++;

  //признак того что были найдены строки или сталбцы с заданными
характеристиками
  bool seach;

  set<int>::iterator it_col;
  set<int>::iterator it_row;

  do
  {
        //поиск строк которые покрываются одним столбцом и добавление
этих столбцов в покрытие
        do
        {
                seach = false;
                set<int> row_(row);

                for(it_row = row_.begin(); it_row != row_.end(); it_row++)
                {
                        STEP_SHEDULER++;

                        if(vrow[*it_row].size() == 1)
                        {
                                //добавляем единственный столбец покрывающий
*it_row строку в покрытие
```

```
                        cover.push_back(*vrow[*it_row].begin());
                        //производим операцию умножения матрици на
столбец вошедший в покрытие
                        //эта операция удалит из матрица все строки которые
покрывает столбец,
                        //удалит номера этих строк из столбцов которые их
тоже покрывали, также удалится и сам столбец из матрицы
                        Multiply(*vrow[*it_row].begin(),row,col,vrow,vcol);

                        //если больше не осталось строк которые нужно
покрыть, значит мы нашил покрытие
                        if(row.size() < 1) return cover;

                        //даем понять алгоритму что была найдена хотя бы
одна строка которая покрывалась одним столбцом
                        //над матрицей провелись преобразования в
результате которых могли появится такие же строки
                        //поэтому нужно повторить поиск повторно
                        seach = true;
                    }
                }
            }
            while(seach);

            //поиск столбцов смежных с висячими(столбцы связаные со столбцом
покрывающим одну строку)
            do
            {
                seach = false;
                set<int> col_(col);

                for(it_col = col_.begin(); it_col != col_.end(); it_col++)
                {
                    STEP_SHEDULER++;

                    //вначале ищим висячий столбец
                    if(vcol[*it_col].size() == 1)
                      if(vrow[*vcol[*it_col].begin()].size() == 2)//если смежный
столбец один добавляем его в покрытие
                        {
                            //удаление висячего столбца(столбец который
покрывает одну строку)
                            vrow[*vcol[*it_col].begin()].erase(*it_col);
                            col.erase(*it_col);
```

```
                        //добавляем в покрытие столбец смежный с висячим
cover.push_back(*vrow[*vcol[*it_col].begin()].begin());

                        //производим операцию умножения матрици на
столбец вошедший в покрытие
Multiply(*vrow[*vcol[*it_col].begin()].begin(),row,col,vrow,vcol);

                        //если больше не осталось строк которые нужно
покрыть, значит мы нашил покрытие
                        if(row.size() < 1) return cover;

                        //даем понять алгоритму что был найден хотя бы
один столбец смежный с висячим, над матрицей
                        //провелись преобразования в результате которых
могли появится такие же столбци, поэтому нужно повторить поиск
                        seach = true;
                    }
                    else //смежных столбцов несколько, удаляем висячий
столбец из множества
                    {
                        vrow[*vcol[*it_col].begin()].erase(*it_col);
                        vcol[*it_col].clear();
                        col.erase(*it_col);
                    }
                }
            }
            while(seach);

            //поиск лучшего столбца по заданному критерию
            int max_col = MaxCol(col,vcol);

            cover.push_back(max_col);

            //производим операцию умножения матрици на столбец вошедший в
покрытие
            //эта операция удалит из матрица все строки которые покрывает
столбец,
            //удалит номера этих строк из столбцов которые их тоже покрывали,
также удалится и сам столбец из матрицы
            Multiply(max_col,row,col,vrow,vcol);
```

```
        STEP_SHEDULER++;

        //если больше не осталось строк которые нужно покрыть, значит мы нашил покрытие
        if(row.size() < 1) return cover;
   }
   while(true);
}
//---------------------------------------------------------------------------


vector<int> __fastcall ThreadWork::SearchCoverFree(s col,s row,vs vcol,vs vrow,const vv &buffer)
{
   vector<int> cover;//множество столбцов являющихся минимальным покрытием

        if(col.size() == 0)
        {
                return cover;
        }

        CALL_COVER++;

   //признак того что были найдены строки или сталбцы с заданными характеристиками
   bool seach;

   set<int>::iterator it_col;
   set<int>::iterator it_row;

   do
   {
        //поиск строк которые покрываются одним столбцом и добавление этих столбцов в покрытие
        do
        {
                seach = false;
                set<int> row_(row);

                for(it_row = row_.begin(); it_row != row_.end(); it_row++)
                {
                        STEP_SHEDULER++;
```

```
                    if(vrow[*it_row].size() == 1)
                    {
                        //добавляем единственный столбец покрывающий
*it_row строку в покрытие
                        cover.push_back(*vrow[*it_row].begin());
                        //производим операцию умножения матрици на
столбец вошедший в покрытие
                        //эта операция удалит из матрица все строки которые
покрывает столбец,
                        //удалит номера этих строк из столбцов которые их
тоже покрывали, также удалится и сам столбец из матрицы
                        Multiply(*vrow[*it_row].begin(),row,col,vrow,vcol);

                        STEP_SHEDULER++;

                        //если больше не осталось строк которые нужно
покрыть, значит мы нашил покрытие
                        if(row.size() < 1) return cover;

                        //даем понять алгоритму что была найдена хотя бы
одна строка которая покрывалась одним столбцом
                        //над матрицей провелись преобразования в
результате которых могли появится такие же строки
                        //поэтому нужно повторить поиск повторно
                        seach = true;
                    }
                }
            }
            while(seach);

            //поиск столбцов смежных с висячими(столбцы связаные со столбцом
покрывающим одну строку)
            do
            {
                seach = false;
                set<int> col_(col);

                for(it_col = col_.begin(); it_col != col_.end(); it_col++)
                {
                    STEP_SHEDULER++;

                    //вначале ищим висячий столбец
                    if(vcol[*it_col].size() == 1)
```

```
                        if(vrow[*vcol[*it_col].begin()].size() == 2)//если смежный
столбец один добавляем его в покрытие
                        {
                                STEP_SHEDULER += 4;

                                //удаление висячего столбца(столбец который
покрывает одну строку)
                                vrow[*vcol[*it_col].begin()].erase(*it_col);
                                col.erase(*it_col);

                                //добавляем в покрытие столбец смежный с висячим

cover.push_back(*vrow[*vcol[*it_col].begin()].begin());

                                //производим операцию умножения матрици на
столбец вошедший в покрытие

Multiply(*vrow[*vcol[*it_col].begin()].begin(),row,col,vrow,vcol);

                                //если больше не осталось строк которые нужно
покрыть, значит мы нашил покрытие
                                if(row.size() < 1) return cover;

                                //даем понять алгоритму что был найден хотя бы
один столбец смежный с висячим, над матрицей
                                //провелись преобразования в результате которых
могли появится такие же столбци, поэтому нужно повторить поиск
                                seach = true;
                        }
                        else //смежных столбцов несколько, удаляем висячий
столбец из множества
                        {
                                STEP_SHEDULER += 3;

                                vrow[*vcol[*it_col].begin()].erase(*it_col);
                                vcol[*it_col].clear();
                                col.erase(*it_col);
                        }
                }
        }
        while(seach);

        //поиск лучшего столбца по заданному критерию
        int max_col = MaxColFree(col,vcol,buffer);
```

```
        cover.push_back(max_col);

        //производим операцию умножения матрици на столбец вошедший в
покрытие
        //эта операция удалит из матрица все строки которые покрывает
столбец,
        //удалит номера этих строк из столбцов которые их тоже покрывали,
также удалится и сам столбец из матрицы
        Multiply(max_col,row,col,vrow,vcol);

        STEP_SHEDULER++;

        //если больше не осталось строк которые нужно покрыть, значит мы
нашил покрытие
        if(row.size() < 1) return cover;
   }
   while(true);
}
//---------------------------------------------------------------------------


vector<int> __fastcall ThreadWork::SearchCoverProduce(s col,s row,vs vcol,vs
vrow)
{
   vector<int> cover;//множество столбцов являющихся минимальным
покрытием

        if(col.size() == 0)
        {
                return cover;
        }

        CALL_COVER++;

   //признак того что были найдены строки или сталбцы с заданными
характеристиками
   bool seach;

   set<int>::iterator it_col;
   set<int>::iterator it_row;

   do
   {
```

```
	//поиск строк которые покрываются одним столбцом и добавление этих столбцов в покрытие
	do
	{
		seach = false;
		set<int> row_(row);

		for(it_row = row_.begin(); it_row != row_.end(); it_row++)
		{
			STEP_SHEDULER++;

			if(vrow[*it_row].size() == 1)
			{
				//добавляем единственный столбец покрывающий *it_row строку в покрытие
				cover.push_back(*vrow[*it_row].begin());
				//производим операцию умножения матрици на столбец вошедший в покрытие
				//эта операция удалит из матрица все строки которые покрывает столбец,
				//удалит номера этих строк из столбцов которые их тоже покрывали, также удалится и сам столбец из матрицы
				Multiply(*vrow[*it_row].begin(),row,col,vrow,vcol);

				STEP_SHEDULER++;

				//если больше не осталось строк которые нужно покрыть, значит мы нашил покрытие
				if(row.size() < 1) return cover;

				//даем понять алгоритму что была найдена хотя бы одна строка которая покрывалась одним столбцом
				//над матрицей провелись преобразования в результате которых могли появится такие же строки
				//поэтому нужно повторить поиск повторно
				seach = true;
			}
		}
	}
	while(seach);

	//поиск столбцов смежных с висячими(столбцы связаные со столбцом покрывающим одну строку)
	do
```

```
	{
		seach = false;
		set<int> col_(col);

		for(it_col = col_.begin(); it_col != col_.end(); it_col++)
		{
			STEP_SHEDULER++;

			//вначале ищим висячий столбец
			if(vcol[*it_col].size() == 1)
			  if(vrow[*vcol[*it_col].begin()].size() == 2)//если смежный столбец один добавляем его в покрытие
				{
					STEP_SHEDULER += 4;

					//удаление висячего столбца(столбец который покрывает одну строку)
					vrow[*vcol[*it_col].begin()].erase(*it_col);
					col.erase(*it_col);

					//добавляем в покрытие столбец смежный с висячим

cover.push_back(*vrow[*vcol[*it_col].begin()].begin());

					//производим операцию умножения матрици на столбец вошедший в покрытие

Multiply(*vrow[*vcol[*it_col].begin()].begin(),row,col,vrow,vcol);

					STEP_SHEDULER++;

					//если больше не осталось строк которые нужно покрыть, значит мы нашил покрытие
					if(row.size() < 1) return cover;

					//даем понять алгоритму что был найден хотя бы один столбец смежный с висячим, над матрицей
					//провелись преобразования в результате которых могли появится такие же столбци, поэтому нужно повторить поиск
					seach = true;
				}
				else //смежных столбцов несколько, удаляем висячий столбец из множества
				{
```

```
                                STEP_SHEDULER += 3;

                                vrow[*vcol[*it_col].begin()].erase(*it_col);
                                vcol[*it_col].clear();
                                col.erase(*it_col);
                        }
                }
        }
        while(seach);

        //поиск лучшего столбца по заданному критерию
        int max_col = MaxColProduce(col,vcol);

        cover.push_back(max_col);

        //производим операцию умножения матрици на столбец вошедший в
покрытие
        //эта операция удалит из матрица все строки которые покрывает
столбец,
        //удалит номера этих строк из столбцов которые их тоже покрывали,
также удалится и сам столбец из матрицы
        Multiply(max_col,row,col,vrow,vcol);

        STEP_SHEDULER++;

        //если больше не осталось строк которые нужно покрыть, значит мы
нашил покрытие
        if(row.size() < 1) return cover;
   }
   while(true);
}
//---------------------------------------------------------------------------


vector<int> __fastcall ThreadWork::SearchCoverGreed(int row_count,s
col_numbers,const vs &col_inc)
{
        CALL_COVER++;

   vector<int> cover;//множество столбцов являющихся минимальным
покрытием

        //находим столбец который покрывает максимальное количество строк
        int col_max_increment = 0;
```

```
	int max_increment = 0;

	for(int i = 1; i < col_inc.size(); ++i)
	{
		STEP_SHEDULER++;

		if(max_increment < col_inc[i].size())
		{
			col_max_increment = i;
			max_increment = col_inc[i].size();
		}
	}

	//добавляем столбец с максимальной степенью в покрытие
	cover.push_back(col_max_increment);

	col_numbers.erase(col_max_increment);

	//формируем множество покрытых строк
	set<int> row_cover;

	row_cover = col_inc[col_max_increment];

	//включаем в покрытие столбцы которые дают максимальный прирост к покрытию строк, пока не покроем все строки
	while(row_cover.size() < row_count)
	{
		max_increment = 0;

	   STEP_SHEDULER++;

		for(set<int>::iterator it_col = col_numbers.begin(); it_col != col_numbers.end(); it_col++)
		{
			set<int> next_row_cover;


	set_union(row_cover.begin(),row_cover.end(),col_inc[*it_col].begin(),col_inc[*it_col].end(),

inserter(next_row_cover,next_row_cover.begin()));

			STEP_SHEDULER += next_row_cover.size();
```

```
                STEP_SHEDULER++;

                if(max_increment < next_row_cover.size())
                {
                        col_max_increment = *it_col;
                        max_increment = next_row_cover.size();
                }
        }

        cover.push_back(col_max_increment);
        col_numbers.erase(col_max_increment);

        STEP_SHEDULER += row_cover.size();


    set_union(row_cover.begin(),row_cover.end(),col_inc[col_max_increment].
begin(),col_inc[col_max_increment].end(),
                        inserter(row_cover,row_cover.begin()));
    }

    return cover;
}
//---------------------------------------------------------------------------


void __fastcall ThreadWork::Multiply(int coli,s &_row,s &_col,vs &_vrow,vs
&_vcol)
{
    //функция удалит все строки которые покрывает столбец coli
    //удалит эти строки из столбцов которые тоже их покрывали
    //удалит из матрици сам coli-й столбец

    //создаем промежуточные переменные
    set<int>::iterator it_row;//ссылка множества строк
    set<int>::iterator it_col;//ссылка множества столбцов

  for(it_row = _vcol[coli].begin(); it_row != _vcol[coli].end(); it_row++)
  {
      //вначале удаляем из строки столбец на которую идет умножение
      //что бы избежать повторного удаления из стобца
      _vrow[*it_row].erase(coli);

      STEP_SHEDULER += _vrow[*it_row].size();
```

```
        //удаление номера поглощенной строки из столбцов
        for(it_col = _vrow[*it_row].begin(); it_col != _vrow[*it_row].end();
it_col++)
        {
                _vcol[*it_col].erase(*it_row);
        }

        //удаление строки содержащей заданую вершину
        _vrow[*it_row].clear();
        _row.erase(*it_row);
  }

  //удаление стобца
  _vcol[coli].clear();
  _col.erase(coli);
}
//---------------------------------------------------------------------------


int __fastcall ThreadWork::MaxCol(const s &col,const vs &vcol)
{
  //создаем промежуточные переменные
  int num_col = 0;//принимаем что первый столбец покрывает максимальное
количество строк
  int max_degree = 0;//максимальное количество покрываемых строк

  //ищим столбец который покрывает максиальное количество строк
  set<int>::const_iterator it_col;
  for(it_col = col.begin(); it_col != col.end(); it_col++)
  {
        STEP_SHEDULER++;

        if(vcol[*it_col].size() > max_degree)
        {
                max_degree = vcol[*it_col].size();
                num_col = *it_col;
        }
  }

  return num_col;
}
//---------------------------------------------------------------------------
```

```
int __fastcall ThreadWork::MaxColFree(const s &col,const vs &vcol,const vv
&buffer)
{
  //создаем промежуточные переменные
  int num_col = 0;//принимаем что первый столбец покрывает максимальное
количество строк
  int max_degree = 0;//максимальное количество покрываемых строк

  //ищим столбец который покрывает максиальное количество строк
        set<int>::const_iterator it_col;
        for(it_col = col.begin(); it_col != col.end(); it_col++)
        {
                STEP_SHEDULER++;

                if(vcol[*it_col].size() > max_degree)
                {
                        max_degree = vcol[*it_col].size();
                        num_col = *it_col;
                }
                else
                {
                        STEP_SHEDULER++;

                        if(vcol[*it_col].size() == max_degree)
                        {
                                STEP_SHEDULER++;

                                if(buffer[*it_col].size() < buffer[num_col].size())
                                {
                                        max_degree = vcol[*it_col].size();
                                        num_col = *it_col;
                                }
                        }
                }
  }

  return num_col;
}
//---------------------------------------------------------------------------


int __fastcall ThreadWork::MaxColProduce(const s &col,const vs &vcol)
{
  //создаем промежуточные переменные
```

```
  int num_col = 0;//принимаем что первый столбец покрывает максимальное
количество строк
  int max_degree = 0;//максимальное количество покрываемых строк

  //ищим столбец который покрывает максиальное количество строк
        set<int>::const_iterator it_col;
        for(it_col = col.begin(); it_col != col.end(); it_col++)
        {
                STEP_SHEDULER++;

                if(vcol[*it_col].size() > max_degree)
                {
                        max_degree = vcol[*it_col].size();
                        num_col = *it_col;
                }
                else
                {
                        STEP_SHEDULER++;

                        if(vcol[*it_col].size() == max_degree)
                        {
                                STEP_SHEDULER++;

                                if(PRODUC[*it_col] > PRODUC[num_col])
                                {
                                        max_degree = vcol[*it_col].size();
                                        num_col = *it_col;
                                }
                        }
                }
  }

  return num_col;
}
//---------------------------------------------------------------------------


int __fastcall ThreadWork::MaxColFreeAndProduce(const s &col,const vs
&vcol,const vv &buffer)
{
  //создаем промежуточные переменные
  int num_col = 0;//принимаем что первый столбец покрывает максимальное
количество строк
  int max_degree = 0;//максимальное количество покрываемых строк
```

```
//ищим столбец который покрывает максиальное количество строк
	set<int>::const_iterator it_col;
	for(it_col = col.begin(); it_col != col.end(); it_col++)
	{
		STEP_SHEDULER++;

		if(vcol[*it_col].size() > max_degree)
		{
			max_degree = vcol[*it_col].size();
			num_col = *it_col;
		}
		else
		{
			STEP_SHEDULER++;

			if(vcol[*it_col].size() == max_degree)
			{
				STEP_SHEDULER++;

				if(buffer[*it_col].size() < buffer[num_col].size())
				{
					max_degree = vcol[*it_col].size();
					num_col = *it_col;
				}
				else
				{
					STEP_SHEDULER++;

					if(buffer[*it_col].size() == buffer[num_col].size())
					{
						STEP_SHEDULER++;

						if(PRODUC[*it_col] > PRODUC[num_col])
						{
							max_degree = vcol[*it_col].size();
							num_col = *it_col;
						}
					}
				}
			}
		}
}
```

```
  return num_col;
}
//---------------------------------------------------------------------------


//Функция поиска минимального вершинного покрытия ранговым методом
vector<int> ThreadWork::SearchCoverAccurate(const vv &b)
{
	CALL_COVER++;

	vector<int> min_cover;

	//количество строк в матрице инциденций
	int row_count = b.size() - 1;

	//количество столбцов в матрице инциденций
	int col_count = b[0].size() - 1;

	if(col_count <= 0) return min_cover;

	//вектор столбцов упорядоченных в порядке убывания
степеней(количетсво покрываемых строк)
	pair<int,int> pair_null(0,0);
	vector<pair<int,int> > col_sort(col_count + 1,pair_null);
	for(int i = 1; i <= col_count; ++i)
	{
		col_sort[i].first = b[0][i]; //в b[0][i] содержится степень i-го столбца
		col_sort[i].second = i; // i - номер столбца
	}

	//сортируем столбцы в порядке убывания степени
	sort(col_sort.begin(),col_sort.end());

	//вектор калибровочных векторов
	vector<int> vec_null(row_count + 1,0);
	vector<vector<int> > h(col_count + 1,vec_null);

	//вектор сочетаний калибровочных векторов - калибровочные вектора
путей (сочетания калибровочных векторов столбцов)
	vector<vector<int> > vec_vec_null(1,vec_null);
	vector<vector<vector<int> > > h_way(col_count + 1,vec_vec_null);

	int value = 0;
	int count_uncover = 0;
```

```
//создаем калибровочный вектор последнего столбца
h_way[col_count].push_back(vec_null);
for(int i = 1; i <= row_count; ++i)
{
	STEP_SHEDULER++;

	if(!b[i][col_sort[col_count].second])
	{
		h[col_count][i] = 99999;
		h_way[col_count][1][i] = 99999;
		count_uncover++;
	}
}

if(count_uncover == 0)
{
	min_cover.push_back(col_sort[col_count].second);
	return min_cover;
}

count_uncover = 0;

//создаем калибровочные вектора остальных столбцов
for(int i = col_count - 1; i >= 1; i--)
{
	h_way[i].push_back(vec_null);

	for(int j = 1; j <= row_count; ++j)
	{
		STEP_SHEDULER++;

		if(b[j][col_sort[i].second])
			value = 0;
		else
		{
			count_uncover++;

			STEP_SHEDULER++;

			if(h[i+1][j] == 99999)
				value = 99999;
			else
				value = h[i + 1][j] + 1;
```

```
                }

                h[i][j] = value;
                h_way[i][1][j] = value;
        }

        if(count_uncover == 0)
        {
                min_cover.push_back(col_sort[col_count].second);
                return min_cover;
        }
}

//вектор путей (сочетаний столбцов)
set<int> set_null;
vector<set<int> > vec_set_null(1,set_null);
vector<vector<set<int> > > col_way(col_count + 1,vec_set_null);

//создаем начальные пути для каждого столбца
for(int i = 1; i < col_way.size(); ++i)
{
        col_way[i].push_back(set_null);
        col_way[i][1].insert(col_sort[i].second);
}

//минимальная степень калибровочного вектора пути, равна количеству различных значений в векторе
//если есть хотя бы один такой элемент вектора равен 99999 равна 99999 - это означает что с этого пути не существует столбца
//который даст покрытие всех строк, и равна 0 если все элементы вектора равны 0, это означает что путь пкорыл все строки матрицы
//путь у которого калибровочный вектор имеет наименьшую степерь, является минимальным покрытием
int dmin = 99999;

for(int step = 1; step <= col_count; step++)
{
        int dmin_new = 99999;

        vector<vector<vector<int> > > h_way_new(col_count + 1,vec_vec_null);
        vector<vector<set<int> > > col_way_new(col_count + 1,vec_set_null);
```

```
for(int i = step + 1; i <= col_count; ++i)
{
	//индекс для новых путей
	int k_new = 0;

	for(int j = step; j < i; ++j)
	{
		for(int k = 1; k < h_way[j].size(); k++)
		{
			STEP_SHEDULER++;

			//если степень калибровочного вектора меньша или равна минимальной создаем с ним новое сочетание
			if(h_way[j][k][0] <= dmin)
			{
				k_new++;
				h_way_new[i].push_back(vec_null);
				col_way_new[i].push_back(col_way[j][k]);

	col_way_new[i][k_new].insert(col_sort[i].second);

				//множество значений калибровочного вектора не равных нулю или бесконечности - 99999
				set<int> h_values;

				int count_uncover = 0;
				int value = 0;

				for(int l=1; l < h_way[j][k].size(); l++)
				{
					STEP_SHEDULER++;

					if(h_way[j][k][l] != 0)
					{
						value = h[i][l];
						if(h[i][l] != 0) count_uncover++;
					}
					else value = 0;

					STEP_SHEDULER++;

					if(value == 99999)
					{
						h_way_new[i].pop_back();
```

```
                                        col_way_new[i].pop_back();
                                        k_new--;
                                        break;
                                }

                                h_way_new[i][k_new][l] = value;
                                h_values.insert(value);
                        }

                        STEP_SHEDULER++;

                        if(count_uncover == 0)
                        {
                                set<int>::iterator it;
                                for(it =
col_way_new[i][k_new].begin(); it != col_way_new[i][k_new].end(); it++)
min_cover.push_back(*it);

                                return min_cover;
                        }

                        STEP_SHEDULER++;

                        //для путей у которых степень
колибровочного вектора не равна безконечности
                        //определяем минимальное количество
столбцов которое необходимо добавить до текущего пути что бы получить
покрытие всех строк
                        if(value != 99999)
                        {

                                int col = col_count;       //начинаем
счет с последнего столбца
                                int col_degree = 0;
        //степень k-го столбца
                                int col_count_cover = 0;
//количество столбцов которое необходимо добавить к текущему пути для
получения покрытия
                                do
                                {
                                        col_count_cover++;
                                        col_degree = col_degree +
col_sort[col].first;

                                        col--;
```

```
                                        }
                                        while((col_degree <
count_uncover)&&(col > 0));

                                        h_way_new[i][k_new][0] =
col_count_cover;

                                        if(dmin_new > h_values.size() - 1)
dmin_new = h_values.size() - 1;

                                    }   // конец цикла по переменной l
                                }       // конец границы условия
h_way[j][k][0] <= dmin
                            }           // конец цикла по переменной k
                        }               // конец цикла по переменной j
                }

                dmin = dmin_new;
                h_way = h_way_new;
                col_way = col_way_new;
        }                               // конец цикла по переменной step

        return min_cover;
}
//---------------------------------------------------------------------------


//функция возвращает номер в векторе самого свободного ресурса
int ThreadWork::MaxFree(const v &numberRes,const vv &buffer)
{
        int number_best = 0;
        int best = 9999999;

        for(int i = 0; i < numberRes.size(); ++i)
        {
                STEP_SHEDULER++;

                if(buffer[numberRes[i]].size() < best)
                {
                        best = buffer[numberRes[i]].size();
                        number_best = i;
                }
        }
```

```
	return number_best;
}
//---------------------------------------------------------------------------


int ThreadWork::MaxProduce(const v &numberRes)
{
	int number_best = 0;
	int best = 0;

	for(int i = 0; i < numberRes.size(); ++i)
	{
		STEP_SHEDULER++;

		if(PRODUC[numberRes[i]] > best)
		{
			best = PRODUC[numberRes[i]];
			number_best = i;
		}
	}

	return number_best;
}
//---------------------------------------------------------------------------


int ThreadWork::MaxFreeAndProduce(const v &numberRes,const vv &buffer)
{
	int number_best = 0;
	int best = 9999999;

	for(int i = 0; i < numberRes.size(); ++i)
	{
		STEP_SHEDULER++;

		if(buffer[numberRes[i]].size() < best)
		{
			best = buffer[numberRes[i]].size();
			number_best = i;
		}
		else
		{
			STEP_SHEDULER++;
```

```
                    if(buffer[numberRes[i]].size() == best)
                    {
                        STEP_SHEDULER++;

                        if(PRODUC[numberRes[i]] > best)
                        {
                            best = PRODUC[numberRes[i]];
                            number_best = i;
                        }
                    }
                }
            }

            return number_best;
}
//---------------------------------------------------------------------------


void ThreadWork::AddError(v &_cover,const s &setRes)
{
        int count_add = (setRes.size() - _cover.size())* ERROR_COVER / 100;

        set<int>::const_iterator it = setRes.begin();

        while((count_add > 0)&&(it != setRes.end()))
        {
                _cover.push_back(*it);

                it++;
                count_add--;
        }

        return;
}
//---------------------------------------------------------------------------


void ThreadWork::CalculateRate()
{
        RATE_UTILIZATION = 1;
        RATE_LOAD                = 1;

        // определяем количесвто задач которые были назначены каждому
ресурсу
```

```
	vector<int> assign_task(COUNT_RES,0);

	for(int i = 1; i <= COUNT_RES; ++i)
		for(int j = 1; j <= COUNT_TASK; ++j)
			if(INC[j][i] == 1) assign_task[i]++;

	for(int i = 1; i <= COUNT_RES; ++i)
	{
		RU[i] = (double) RU[i] / STEP;
		RL[i] = (double) RL[i] / assign_task[i];
	}

	RATE_UTILIZATION = (long double) exp((1.0 / COUNT_RES) *
SumLog(RU));
	RATE_LOAD = (long double) exp((1.0 / COUNT_RES) * SumLog(RL));

	double produce = 0;
	for(int i = 1; i < PRODUC.size(); ++i) produce += PRODUC[i];

	RATE_ACCELERAT = (long double) SOLVER_SUM / (STEP * (long
double)(produce / COUNT_RES));
}
//---------------------------------------------------------------------------


double ThreadWork::SumLog(const vd& x)
{
	long double sum = 0;

	for(int i = 0; i < x.size(); ++i)
	{
		if(x[i] > 0)
		{
			sum += log(x[i]);
		}
		else
		{
			sum += log(NULL_VALUE);
		}
	}

	return sum;
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------

#ifndef Unit_ThreadWorkH
#define Unit_ThreadWorkH
//---------------------------------------------------------------------------
#include <System.Classes.hpp>
#include "Unit_Typedef.h"
#include "Unit_ClassResults.h"

const double NULL_VALUE = 0.000000000001;
//---------------------------------------------------------------------------
class ThreadWork : public TThread
{
private:

        int STEP;               //такт работы кластера
        int STEP_SHEDULER; //количество тактов произведенное при
планировании

        int COUNT_SOLVER_TASK;//количество решеных задач
        int LAST_SEND_TASK;//номер последней задачи загруженной в пул

        vector<int> SR;    // STEP RESPOND
        vector<int> SW;            // STEP_WAIT
        vector<int> SS;            // STEP_SERVICE
        vector<int> SSO;  // STEP_SOLVER
        vector<int> ST;        // STEP_TRANSFER

        ofstream OUTFILE;

protected:

        void __fastcall Execute();
        void __fastcall ProgressFormTest();
        void __fastcall ProgressFormTestGuadge();
        void __fastcall ProgressFormCluster();
        void __fastcall ProgressSolver();

        void __fastcall LogBegin();
        void __fastcall LogEnd();
        void __fastcall LogStep();
        void __fastcall LogPool(AnsiString comments,const v &_pool);
```

```
        void __fastcall LogBuffer(const vv &buffer);
        void __fastcall LogRes(const v &res_number,const v &res_delay,const v
&res_solver);

        //функция выполняет помещение входных задач в пулл
        void __fastcall TaskToPool(int step,int& _i,v &_pool);
        //функция помещает задачи не вошедшие в буффер обратно в пулл
        void __fastcall TaskBackoffToPool(v &_pool,s &_backoff);

        //функции распределения(планирования) задач, находящихся в пулле
между свободными ресурсами,различными методами
        //планируем точным методом минимального покрытия
        void __fastcall ShedulerMCRang(const v &pull,s &_backoff,vv &_buffer);
        //планируем минимальным покрытием
        void __fastcall ShedulerMC(const v &pool,s &_backoff,vv &_buffer);
        //планируем минимальным покрытием помещая задачи с меньшей
универсальностью в первую очередь
        void __fastcall ShedulerMCFree(const v &pool,s &_backoff,vv &_buffer);
        //планируем минимальным покрытием с оптимизацией по
производительности
        void __fastcall ShedulerMCProduce(const v &pool,s &_backoff,vv
&_buffer);

        void __fastcall ShedulerFCFS(v &_pool,vv &_buffer);
        //планировщик FCFS с буфером
        void __fastcall ShedulerFCFSB(const v &pool,s &_backoff,vv &_buffer);

        void __fastcall ShedulerGREEDE(const v &pool,s &_backoff,vv &_buffer);

        //функция выполняет потактовое решение задач,помещает в
освободвшиеся или свободные ресурсы новые задачи из очереди буффера
        void __fastcall TaskSolver(vv &_buffer,v &_res_number,v &_res_delay,v
&_res_solver);

        //функция преобразует двоичное представление задач в десятичное
        void __fastcall MatrixToVectorSet(const v &pool,vs &_vset_n,vs
&_vset_m);

        //функция находит минимальное вершинное множество
        vector<int> __fastcall SearchCover(s col,s row,vs vcol,vs vrow);
        vector<int> __fastcall SearchCoverFree(s col,s row,vs vcol,vs vrow,const vv
&buffer);
        vector<int> __fastcall SearchCoverProduce(s col,s row,vs vcol,vs vrow);
```

```
        vector<int> __fastcall SearchCoverGreed(int row_count,s
col_numbers,const vs &col_inc);

        //функция удяляет из множества все занятые ресурсы и задания для
которых нет свободных ресурсов
        void __fastcall deleteOccupRes(const s &res,s &_new_res,s &_task,
                                                    vs &_incRes,vs &_incTask,s
&_backoff,
                                                    const v &pool,const vv
&buffer);

    int __fastcall countFreeRes(const s &res,const vv &buffer);

        //функция умножает матрицу инциндетности на столбец
        void __fastcall Multiply(int coli,s &_row,s &_col,vs &_vrow,vs &_vcol);

        //функция возвращает номер первого столбца,который покрывает
наибольшее количество строк
        int  __fastcall MaxCol(const s &col,const vs &vcol);
        int  __fastcall MaxColFree(const s &col,const vs &vcol,const vv &buffer);
        int  __fastcall MaxColProduce(const s &col,const vs &vcol);
        int  __fastcall MaxColFreeAndProduce(const s &col,const vs &vcol,const
vv &buffer);

        //Функция поиска минимального вершинного покрытия ранговым
методом
        vector<int> SearchCoverAccurate(const vv &b);

        int MaxFree(const v &numberRes,const vv &buffer);      //функция
возвращает номер в векторе самого свободного ресурса
        int MaxProduce(const v &numberRes);
        int MaxFreeAndProduce(const v &numberRes,const vv &buffer);
        void AddError(v &_cover,const s &setRes);

        void CalculateRate();
        double SumLog(const vd& x);

public:
        __fastcall ThreadWork(bool CreateSuspended);
        __property Terminated;
                                    //если вызвано прерывание потока принимает
значение true
```

```
        bool   CALL_TEST;
                                        //поток вызван в режиме тестирования
        bool   LOG;
                                //сохранить файл отчета работы системы

        int    COUNT_TASK;
                                        //количество задач
        int    COUNT_RES;
                                        //количество рессурсов
        int    POOL_SIZE;
                                        //размер пулла
        int    BUFFER_SIZE;
                                //длина буффера
        int    FREQ_SHEDULER;
                                        //периодичность планирования
        int    METHOD;
                                //метод планирования задач
        int    SOLVER_SUM;
                                        //суммарная сложность решения задач

        int    MC1_TYPE;
        int    MC2_TYPE;
        int    MC3_TYPE;
        int    DIVIDE_SP;
        int            ERROR_COVER;

        vector<vector<int> >    INC;
                        //матрица инциденций(соответствий) задач ресурсам
        vector<int>             SOLVER;
                                //сложность решения задач
        vector<int>             VOLUME;
                                //размер задач
        vector<int>             FLOW;
                                //количество заданий приходящие на очередном
периоде планирования
        vector<int>             PRODUC;
                                //вектор производиетльности ресурсов
        vector<int>             SPEED;
                                //вектор пропускной способности ресурсов

        long double STEP_EXECUTION;
                                                //время выполнения системой всех
задач в тактах
```

```
        long double STEP_RESPEND;
                                        //среднее время ответа системы в тактах
        long double STEP_WAIT;
                                            //среднее время
ожидания(нахождение задачи в пулле) в тактах
        long double STEP_SERVICE;
                                        //среднее время
обслуживания(нахождение задачи в буффере) в тактах
        long double STEP_SOLVER;
        long double STEP_PLANNING;
                                            //среднее время затраченное на
планирование одной задачи
        long double STEP_TRANSFER;                                  //среднее
время передачи задания на ресурс

        long double RATE_UTILIZATION;
                                        //средний коэффициент использования
ресурсов
        long double RATE_LOAD;
                                            //средний коэффициент загрузки
ресурсов
        long double RATE_ACCELERAT;
                                            //коэффициент усиления

        long double COUNT_SHEDULER;
        long double CALL_COVER;
        long double DENSIT_COVER;

        vector<double> RU;
                                        //вектор коэффициентов испольвания
ресурсов Ratio Utilization
        vector<double> RL;
                                        //вектор коэффициентов загрузки
ресурсов Ratio Parfomance

        vector<int> COUNT_COVER;
        vector<vector<int> > DR;
                                    //вектор распределения задач по ресурсам
};
//---------------------------------------------------------------------------
#endif
```