Зміст

[ВСТУП 3](#_Toc421634517)

[1. АЛГОРИТМИ ГЕНЕРУВАННЯ ЗАДАЧ ТА ПЛАНУВАННЯ 4](#_Toc421634518)

[1.1. Генерування графу задачі 4](#_Toc421634519)

[1.1.1. Алгоритм генерування випадкового графу задачі з заданою кореляцією 5](#_Toc421634520)

[1.2. Методи формування черг обчислювальних робіт 5](#_Toc421634521)

[1.2.1. Алгоритм пошуку критичного шляху кінця 6](#_Toc421634522)

[1.2.2. Алгоритм пошуку критичного шляху початку 7](#_Toc421634523)

[1.2.3. Способи представлення графу 7](#_Toc421634524)

[1.2.4. Визначення кількості дуг та зв’язності 8](#_Toc421634525)

[1.3. Призначення обчислювальних робіт 8](#_Toc421634526)

[1.3.1. Алгоритм перевірки графу на ациклічність 10](#_Toc421634527)

[1.3.2. Алгоритм перевірки графу на зв’язність 10](#_Toc421634528)

[1.3.3. Опис алгоритму призначення 10](#_Toc421634529)

[2. СТРУКТУРА РОЗРОБЛЕНОЇ ПРОГРАМНОЇ МОДЕЛІ 12](#_Toc421634530)

[2.1. Редактор графів 12](#_Toc421634531)

[2.1.1. Логічне представлення графу 12](#_Toc421634532)

[2.1.2. Графічне представлення графу 12](#_Toc421634533)

[2.1.3. Збереження графів 14](#_Toc421634534)

[2.2. Генератор графів 14](#_Toc421634535)

[2.3. Формування черги 15](#_Toc421634536)

[2.4. Призначення обчислювальних робіт 16](#_Toc421634537)

[2.4.1. Сутності, що використовуються для призначення обчислювальних робіт 16](#_Toc421634538)

[2.4.2. Моделювання пересилок 18](#_Toc421634539)

[2.4.3. Відображення діаграми Ганта 19](#_Toc421634540)

[2.5. Обчислення показників ефективності алгоритмів планування. 19](#_Toc421634541)

[3. ПОРІВНЯННЯ АЛГОРИТМІВ ПЛАНУВАННЯ 21](#_Toc421634542)

[3.1. Порівняння алгоритмів формування та призначення 23](#_Toc421634543)

[3.1.1. Порівняння ефективності та коефіцієнту прискорення 23](#_Toc421634544)

[3.1.2. Порівняння часу виконання обчислень 27](#_Toc421634545)

[3.2. Дослідження додаткових параметрів системи 30](#_Toc421634546)

[3.2.1. Дослідження впливу кількості фізичних зв’язків 30](#_Toc421634547)

[3.2.2. Дослідження впливу контролеру вводу-виводу 35](#_Toc421634548)

[3.2.3. Дослідження впливу дуплексності зв’язків 38](#_Toc421634549)

[ВИСНОВКИ 41](#_Toc421634550)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 42](#_Toc421634551)

[Додаток А 43](#_Toc421634552)

# ВСТУП

Останні десятиріччя пов’язані з швидким розвитком паралельних комп’ютерних систем (ПКС). Паралельна обробка інформації створює передумови для суттєвого підвищення продуктивності засобів обчислювальної техніки. Зараз у світі існує велика кількість архітектурних рішень ПКС і потреба у їх використанні весь час зростає. Сьогодні однією з найскладніших проблем розвитку ПКС є підвищення ефективності їх використання при вирішенні задач користувача. Основні напрямки підвищення ефективності ПКС пов’язані з досконалим використанням методів та засобів програмування, компіляції та організації паралельних обчислювальних процесів. Курсова робота пов’язана з дослідженням різних алгоритмів планування обчислень для комп’ютерної системи із заданою архітектурою. Отримані результати можуть бути використані для підвищення реальної продуктивності паралельних систем за рахунок ефективного планування обчислень.

Таким чином, метою даної курсової роботи є дослідження методів та засобів забезпечення максимальної реальної продуктивності сучасних паралельних комп’ютерних систем з урахуванням специфіки архітектури при вирішенні задач користувача. Завдання курсового проектування полягає у порівнянні дев’яти реалізованих алгоритмів планування з точки зору ефективності, а також порівнянні додаткових параметрів системи на кращих з цих алгоритмів. Порівняння алгоритмів планування необхідно виконувати на різноманітних, випадково згенерованих, графах задач та конкретному типі системи. В рамках даної курсової роботи розглядається система у вигляді тору. Тор - це тривимірна решітка, розміром 3\*3\*3, у кожному вузлі якої міститься 6 зв’язків, тобто по 2 зв’язки з сусідніми вершинами у кожній площині. Додатковими параметрами системи, що необхідно дослідити у рамках цієї лабораторної роботи є кількість фізичних зв’язків, наявність контролеру вводу-виводу та наявність дуплексності у каналах зв’язків системи.

# АЛГОРИТМИ ГЕНЕРУВАННЯ ЗАДАЧ ТА ПЛАНУВАННЯ

Основні алгоритми, що використовуються у даній курсовій роботі пов’язані з генеруванням випадкового графу задач та планування обчислювальних робіт. Процедура планування обчислювальних робіт, в свою чергу можна поділити на два етапи: формування черги готових обчислювальних робіт та призначення обчислювальних робіт на процесори.

## Генерування графу задачі

Для генерування графу задачі задаються наступні його параметри:

* кількість вершин графу; (nCount)
* мінімальна вага вершини; (minWeight)
* максимальна вага вершини; (maxWeight)
* кореляція; (cor)

Кореляція визначається за формулою:

де W – сумарна вага вершин графу задачі, а L – сумарна вага дуг графу задачі.

Крім цього, для генерування графу необхідно визначити кількість дуг. Визначити кількість дуг можна кількома способами:

* задати точне значення кількості дуг;
* задати кількість дуг через відсоток від максимальної кількості.
* задати таку кількість дуг, яка залежить від кореляції (чим менше кореляція тим більше кількість дуг, і навпаки).

Останній варіант фактично дозволяє звільнитися від необхідності кожен раз, при генеруванні графу, визначати зайвий параметр, тобто кількість дуг. Натомість, необхідно визначити граничне значення, при якому буде генеруватись граф мінімальної зв’язності, тобто, якщо зобразити вершини графа у вигляді паралельно-ярусної форми, то кожна вершина буде мати дуги до усіх вершин, що розташовані на нижніх рівнях. Назвемо цей параметр мінімальна кореляція (minCor). Для певного значення кореляції, мінімальна кореляція дозволяє регулювати кількість дуг, розподіл ваг дуг графу між ними. При дуже малих значеннях мінімальної кореляції кількість дуг менше, а вага дуг більше, при великих значеннях – навпаки.

### Алгоритм генерування випадкового графу задачі з заданою кореляцією

1. Генерація nCount вершин з вагами у межах [minWeight; maxWeight].
2. Визначення суми ваг вершин та підрахунок значення суми ваг дуг, щоб задовольнити нерівність:
3. Визначення максимально допустимої ваги дуг, яка може бути отримана при значенні minCor на графі з тими ж вершинами:

Максимальна кількість дуг:

Середня вага дуг в такому графі складе meanEdgeWeight:

1. Доки сумарна вага дуг :
   1. Генеруємо випадкову вагу
   2. Зменшуємо на значення .
   3. Додаємо до випадкової дуги вагу .
2. Якщо , додаємо до останньої дуги значення .

## Методи формування черг обчислювальних робіт

Формування черг обчислювальних робіт може виконуватись різними евристичними методами. В даній роботі розроблено формування черг 1, 4 та 12 методами. Для реалізації даних методів необхідно обчислити критичні значення для кожної вершини, по часу і по кількості, а також, підрахувати кількість вихідних дуг та зв’язність (сумарна кількість вхідних та вихідних) вершин.

Формування черги задач полягає у сортуванні списку задач за певним критерієм. Отримати відсортований список задач можна з використанням одного із алгоритмів сортування, наприклад quicksort, або побудувавши бінарне дерево пошуку, наприклад червоно-чорне дерево.

|  |  |  |
| --- | --- | --- |
| Номер | Метод формування черг | Використанні характеристики графа |
| 1 | У порядку спадання пронормованої суми критичних по часу і по кількості вершин шляхів до кінця графа задачі. | Критичний шлях графа та вершин по часу. |
| 4 | У порядку спадання критичного по кількості вершин шляхів до кінця графа задачі, а при рівних значеннях – в порядку спадання зв’язності вершин. | Критичний шлях по кількості вершин та їх зв’язність |
| 12 | У порядку спадання кількості вихідних дуг вершин | Кількість вихідних дуг вершин |

Таблиця 1. Методи формування черг

### Алгоритм пошуку критичного шляху кінця

Для підрахунку критичних значень використовується рекурсивні алгоритми.

1. Задамо критичне значення кінця всіх вершин 0.
2. Проходимо по усім вершинам v \in V.
   1. Якщо критичне значення кінця v дорівнює 0, виконуємо для неї процедуру *CalculateCWE*(v).

Процедура *CalculateCWE* (параметр — вершина u \in V)

1. Встановлюємо значення змінної max в 0.
2. Проходимо по усім вершинам w суміжним з u, тобто є дуга u → w.
   1. Якщо критичне значення кінця w дорівнює 0, виконуємо для неї *CalculateCWE* (w).
   2. Якщо критичне значення кінця w більше max, задаємо max критичне значення кінця w.
3. Встановлюємо критичне значення u:
   1. max + u.weight, у випадку пошуку критичного по часу кінця.
   2. max + 1, у випадку пошуку критичного по кількості кінця.

### Алгоритм пошуку критичного шляху початку

Для визначення критичного шляху до початку графа використовується алгоритм аналогічний пошуку критичного шляху кінця, з єдиним виключенням: вага поточної вершини не враховується у критичне значення початку.

1. Задамо критичне значення кінця всіх вершин -1.
2. Проходимо по усім вершинам v \in V.
   1. Якщо критичне значення кінця v дорівнює -1, виконуємо для неї процедуру *CalculateCWB* (v).

Процедура *CalculateCWB* (параметр — вершина u \in V)

1. Встановлюємо значення змінної max в 0.
2. Проходимо по усім вершинам w, таким, для яких u суміжним з w, тобто існує дуга w → u.
   1. Якщо критичне значення початку w дорівнює -1, виконуємо для неї *CalculateCWB* (w).
   2. Якщо критичне значення початку w більше max, задаємо max:
      1. критичне значення початку w + w.weight, у випадку критичного по часу початку.
      2. критичне значення початку w + 1, у випадку критичного по кількості початку.
3. Встановлюємо критичне значення u = max.

### Способи представлення графу

Алгоритм визначення кількості вихідних дуг та зв’язності вершин залежить від способу представлення графу.

Найбільш розповсюджені методи представлення графів:

* матриця суміжності – таблиця, у якій стовбці та строки відповідають вершинам графу, а у комірках записується число, що визначає наявність зв’язку між двома вершинами.
* матриця інцидентності – таблиця, у якій строки відповідають вершинам графу, стовбці – ребрам, а у комірку <i,j> записується:
  + значення 1, якщо j-й зв’язок виходить з комірки i,
  + значення -1, якщо j-й зв’язок входить у комірку i.
  + інакше 0.
* список суміжності – список у кожній вершині графу, який зберігає суміжні вершини.
* список ребер – список, у якому кожному ребру відповідає строчка, що зберігає дві вершини, інцидентні ребру.

У матриць суміжності та інцидентності наявний істотній недолік: вони надзвичайно вимогливі до пам’яті. А при додаванні чи видаленні вершини/ребра із графу, необхідно проводити велику кількість операцій по зміненню розмірів цих матриць, тому вони також вимогливі і до процесорного часу, і тому краще скористатись списками для представлення графу.

### Визначення кількості дуг та зв’язності

У випадку використання списку суміжності, для визначення кількості вихідних вершин достатньо підрахувати кількість записів у цьому списку, але визначення зв’язності вершини являє собою складну задачу, оскільки для цього необхідно обійти усі інші вершини і підрахувати кількість входжень у їхні списки суміжності даної вершини. Для уникнення такої складності обчислень можна використовувати 2 списки суміжності (для попередніх та наступних вершин), або комбінований варіант який полягає у використанні стандартного списку суміжності (для наступних вершин), та списку ребер (як вхідних, так і вихідних) у межах кожної вершини. У цьому випадку для визначення кількості вихідних вершин необхідно підрахувати кількість записів у списку суміжності, а для визначення зв’язності вершини – кількість записів у списку ребер.

## Призначення обчислювальних робіт

В даній курсовій роботі реалізовано 3 алгоритми призначення: 3, 5 та 7.

1. Алгоритм призначення з урахуванням пріоритетів процесорів (по зв'язності). Даний алгоритм може використовуватись як для всіх обчислювальних робіт графа задачі, так і для призначення обчислювальних робіт першого ярусу графа задачі у поєднанні з алгоритмами «сусіднього» призначення
2. Алгоритм «сусіднього» призначення із пересилками «з попередженням». У даному випадку використовується, як і в аналогічному алгоритмі для SMP систем, комунікаційна модель, коли дані передаються асинхронно відразу після їх формування.
3. Алгоритм оптимізованого «сусіднього» призначення, у якому враховуються всі процесори незалежно від їх стану в момент призначення (зайнятий або вільний).

Наявність декількох фізичних зв’язків дозволяє виконувати передачу даних по декільком логічним зв’язкам одночасно. Це призводить до збільшення продуктивності системи, якщо кореляція системи невелика, а кількість дуг (зв'язність) велика. Проте занадто велика кількість фізичних дуг може не надати ефекту прискорення, а також може мати негативний ефект: збільшення часу необхідного для призначення обчислювальних робіт, при певних алгоритмах призначення.

Наявність процесору вводу-виводу дозволяє значно підвищити продуктивність системи, оскільки дозволяє виконувати передачу даних між процесорами, незалежно від обчислювань на процесорі. У разі відсутності контролеру вводу-виводу, дорогоцінний процесорний час буде використовуватись на виконання пересилок даних між процесорами.

Дуплексність, як і наявність декількох фізичних зв’язків, може покращити продуктивність систему у випадку низької кореляції системи та великої зв'язності.

Вхідними даними для цього етапу є:

* граф задачі;
* граф КС;
* черга обчислювальних робіт, отримана на попередньому етапі з використанням 1, 4 або 12 алгоритмів;
* параметри процесорів КС, які включають:
* число фізичних зв’язків;
* наявність процесорів вводу-виводу;
* дуплексність або напівдуплексність зв’язків;

Граф задачі повинен бути ациклічним, а граф системи - зв'язний.

### Алгоритм перевірки графу на ациклічність

1. Всі вершини помітимо білим кольором (стан 0).
2. Проходимо по усім вершинам v \in V.
   1. Якщо вершина v біла(0), виконуємо для неї *DFS*(v).

Процедура *DFS* (параметр — вершина u \in V)

1. Перефарбовуємо вершину u у сірий колір (стан 1).
2. Для кожної вершини w, суміжною з вершиною u перевіряємо колір:
   1. Якщо вершина w біла (0), виконуємо процедуру *DFS*(w).
   2. Якщо вершина w сіра (1), то це означає що **отримали цикл**.
   3. Якщо вершина w чорна (2), переходимо до наступної вершини.
3. Перефарбовуємо вершину u у чорний колір (стан 2).

### Алгоритм перевірки графу на зв’язність

1. Всі вершини помітимо білим (стан 0) кольором.
2. Обираємо будь-яку вершину v \in V, та помічаємо її сірим (стан 1) кольором.
3. До тих пір, поки є сірі (стан 1) вершини:
   1. Помічаємо наступну сіру вершину u \in V чорним (стан 2) кольором.
   2. Усі вершини w, білого (стан 0) кольору, суміжні з даною, помічаємо сірим (стан 1) кольором.
4. Якщо кількість вершин, помічених білим (стан 0) кольором немає, то **граф зв’язний.**

### Опис алгоритму призначення

1. Призначення обчислювальних робіт починається із сортування процесорів за зв’язністю, або вагою у випадку співпадаючої зв’язності, а потім і сортування задач з використанням одного з алгоритмів формування черги.
2. Задачі, які не мають вхідних задач, тобто в ярусно-паралельній формі знаходяться на першому рівні, додаються до черги задач.
3. Доки є вільні процесори та задачі в черзі задач, найбільш пріоритетна задача виймаються з черги та назначається на виконання на найбільш пріоритетний вільний процесор.
   1. При назначені задачі на вільний процесор, цей процесор додається до списку зайнятих процесорів.
4. Доки всі процесори не звільняться, обираємо той процесор зі списку зайнятих процесорів, який завершить виконання наступної задачі першим.
   1. Задача, яка виконувалась на даному процесорі повідомляє наступні задачі (тобто ті, що знаходяться нижче у ЯПФ) про своє завершення.
   2. Якщо задача, яка отримала повідомлення про завершення не має попередніх незавершених задач, вона додає себе до черги задач.
   3. Якщо черга задач готових до виконання порожня, повертаємось до пункту 1.
   4. Обираємо з черги готових задач найбільш пріоритетний. Знаходимо зв’язки між попередніми задачами та поточним.
   5. Обираємо процесор для моделювання згідно з алгоритмом призначення.
   6. Для кожного з обраних процесорів виконуємо моделювання, та обираємо той процесор, який почне виконання поточної задачі раніше за інших.
   7. Назначаємо виконання поточної задачі на обраний процесор.
   8. Додаємо процесор до списку зайнятих процесорів.

Моделювання пересилок та виконання обчислень на процесорах, зазначене вище, описане у підрозділі 2.3.2.

# СТРУКТУРА РОЗРОБЛЕНОЇ ПРОГРАМНОЇ МОДЕЛІ

Програмне забезпечення розроблялось в кілька етапів. На першому етапі було створено редактор графів, на другому – генератор графів, наступний етап – розробка 3 алгоритмів формування черг та 3 алгоритмів призначення та останній етап – порівняння алгоритмів планування.

ПЗ створено з використанням Qt. Qt – кросплатформений інструментарій розробки ПЗ на мові програмування C++.

## Редактор графів

Для більшої гнучкості логічне та графічне представлення графу виділені в окремі класи.

### Логічне представлення графу

Логічне представлення графу складається з власне графу(клас Graph), вершин(клас Node) та дуг(клас Edge). Graph зберігає список всіх Nod’ів та Edg’ів. Graph також використовується як фабрика для створення Nod’ів та Edg’ів. Спосіб представлення графу являю собою комбінацію двох способів представлення: списку суміжності та списку ребер. Це дозволяє виконувати пошук необхідної інформації у графі значно швидше за рахунок оперативної пам’яті необхідної для зберігання графу. Список суміжності – список у кожній вершині графу який зберігає суміжні вершини. Для направленого графу у списку суміжності зберігаються лише вихідні Nod’и. Список ребер – спосіб представлення, коли кожен Edge графу зберігає показники на вхідний та вихідний Nod’и, для направленого графу, або просто суміжні вершини, для ненаправленого. Оскільки граф може бути направленим(граф задачі) або ненаправленим(граф системи), то класи Graph та Edge є віртуальні, а дочірніми класами є DirectedGraph та UndirectedGraph, DirectedEdge та UndirectedEdge, відповідно.

### Графічне представлення графу

Графічне представлення базується на фреймворку графічного представлення, як частині Qt. Основними класами цього фреймворку є QGraphicsView, QGraphicsScene та QGraphicsItem. QGraphicsView використовується у якості вікна для відображення QGraphicsScene. QGraphicsScene – це елемент, призначений для зображення двовимірних зображень. QGraphicsScene містить елементи QGraphicsItem, які утворюють зображення. Кожен з основних класів має свої дочірні класи для створення графічного представлення графу. Дочірніми класами є GraphView, GraphScene, QGraphicsNode та QGraphicsEdge. Ці класи тісно взаємодіють з відповідними класами логічного представлення. QGraphicsNode та QGraphicsEdge мають своїх нащадків. Для створення особливих форм вершин та дуг графу, необхідно створити інших нащадків цих класів та перевизначити ключові віртуальні методи. Клас WeightTextItem показує вагу вершин та дуг графу забезпечує інтерактивність для QGraphicsNode та QGraphicsEdge.

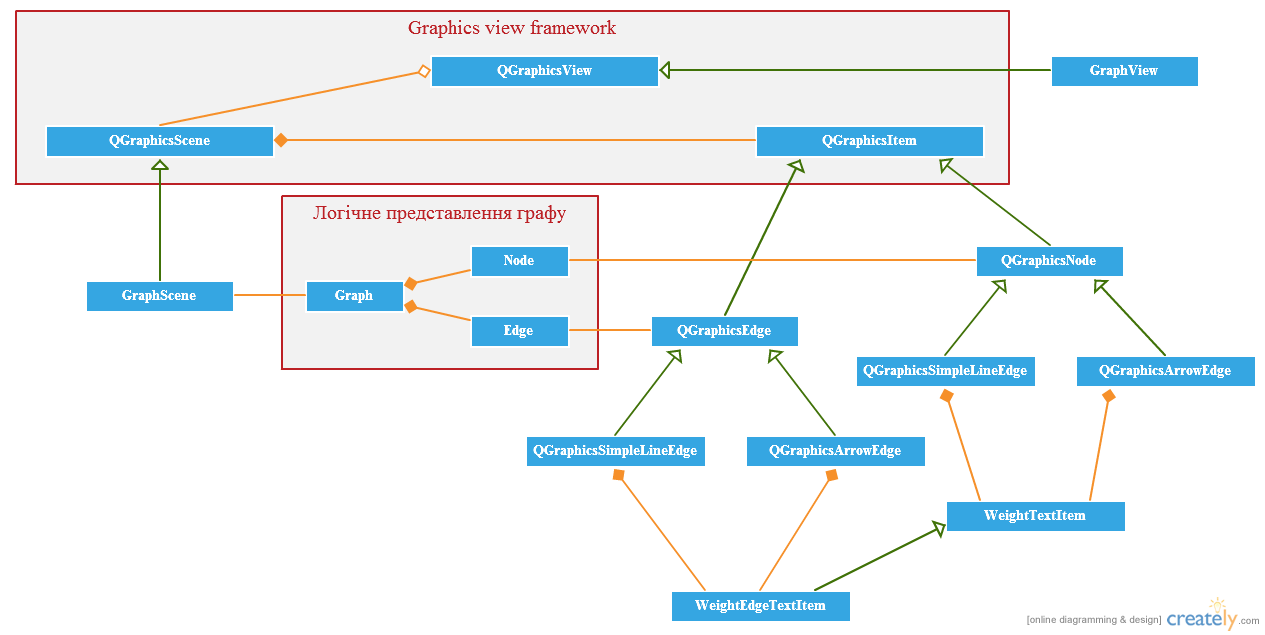


Рис. 2.1. Діаграма класів редактору графів

Для моделювання та дослідження алгоритмів планування обчислень необхідно використовувати графи які задовольняють певним вимогам. Так, для MPP систем, граф задачі має бути ациклічним, а граф системи – зв’язним.

Система спроектована так, що якщо граф системи буде незв’язний, то компоненти зв’язності будуть відображатись різними кольорами. Якщо граф задачі буде містити цикли, то вони будуть виділятися червоним кольором. Приведений у підрозділі 1.3.1 алгоритм не дозволяє знайти всі цикли одразу, а знаходить лише перший(якщо цикли взагалі існують) і відображає його, але у пошуку всіх циклів немає потреби, достатньо лише перевірити чи є взагалі цикли.

### Збереження графів

Редактор графу також містить функціонал пов’язаний із роботою з файлами, а саме збереження та відкриття вже збережених графів. Графи зберігаються у вигляді звичайного текстового файлу з розширенням “.pzksg”. Ці файли мають наступний формат:

<Тип графу>

n <id вершини> <вага вершини> <координати вершини>

e <вага дуги> <id початкової вершини> < id кінцевої вершини>

Тип графу – назва класу, дочірнього від Graph. Кожен рядок починається з символів n або e, що означає Node або Edge відповідно, та закінчується символів переносу рядку. Координати вершини задаються у вигляді “X Y” (через пробіл).

## Генератор графів

Генератор графів дозволяє створювати графи заданих розмірів за допомогою псевдовипадкових чисел. Для генерування графу необхідно ввести вхідні дані, описані у підрозділі 1.1. Алгоритм, за яким відбувається генерування випадкових графів, описаний у підрозділі 1.1.1.

Підставимо значення L, maxL та maxECount з формул (1.2), (1.3) та (1.4) у цю формулу (2.1):

Формула (2.3) дозволяє обчислити середню кількість дуг у згенерованому графі задачі.

Нижче наведений графік, що показує кількість дуг (у відсотках від максимальної кількості дуг), в залежності від кореляції, при значенні мінімальної кореляції 0.2. Так, наприклад, при коефіцієнті кореляції 0.4, у графі в середньому буде 37,5% дуг,а при кореляції 0.5 – в середньому 25% дуг. Якщо значення кореляції буде знаходитись у граничних значеннях, тобто дорівнювати мінімальні кореляції, або 1, то у графі будуть майже всі дуги, або взагалі не буде дуг, відповідно.

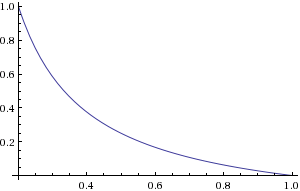


Рис. 2.2. Залежність кількості дуг, від кореляції (при мінімальній кореляції 0.2).

Для виведення графу на екран, вершини графу впорядковуються згідно паралельно-ярусної форми. На першому ярусі знаходяться вершини, що не мають входів, на другому-ті що мають вхід лише з вершин 1 ярусу, на третьому – ті, що мають вхід з другого і можливо з першого і т.д.

## Формування черги

Формування черги обчислювальних робіт (саме сортування) виконується за допомогою QMap класу, який базується на красно-чорному дереві і забезпечує зберігання вхідних даних у відсортованому вигляді. QMap зберігає дані у вигляді пар ключ-значення, при цьому дані сортуються за значенням ключів, тому ключем повинні бути:

* сума нормованого критичного шляху по часу та нормованого критичного шляху по кількості вершин до кінця графа задачі, для 1 алгоритму.
* пара значень: критичний шлях по кількості вершин до кінця графа задачі, та зв’язність вершини, для 4 алгоритму. Пара значень сортується спочатку по 1 значенню, а при рівних перших значеннях – по другому.
* кількість вихідних дуг вершин, для 12 алгоритму.

Зв’язність вершин та кількість вихідних дуг не потребують обчислень, оскільки постійно зберігаються у самій вершині.

## Призначення обчислювальних робіт

Наступний етап планування обчислювальних робіт – призначення обчислювальних робіт по процесорах заданої КС.

### Сутності, що використовуються для призначення обчислювальних робіт

TimeSpan – клас, який містить проміжок часу та метод draw(), який відображає цей проміжок відповідною лінією на діаграмі Ганта.

Клас Task відповідає вершині графу задачі, містить список вхідних та вихідних Task`ів, тобто тих, які мають бути виконані раніше або пізніше. Task`и з’єднуються за допомогою класу Connection. Кожен Task також містить критичні значення початку та кінця, необхідні для планування. В ході виконання планування, кожен Task назначається на певний Processor, на якому він має виконуватись. Після призначення, можна отримати Processor, на якому виконується даний Task через метод executor().

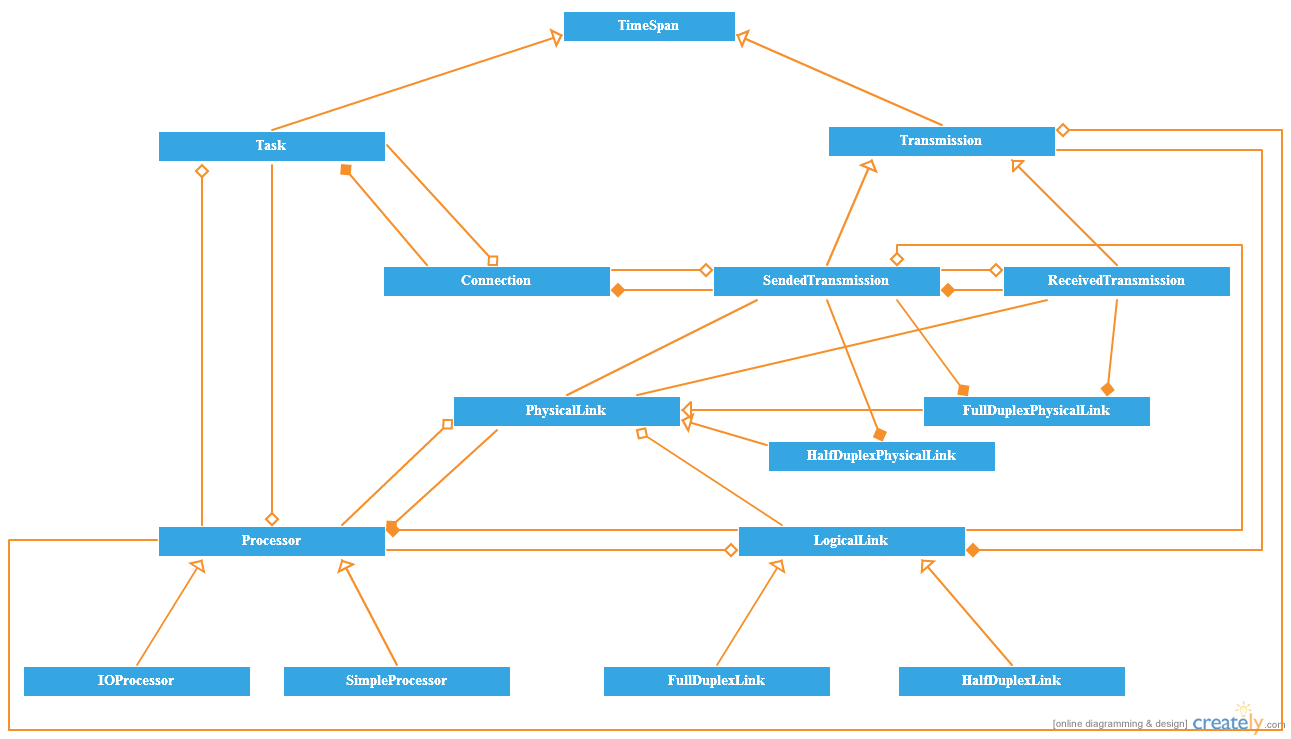


Рис. 2.3. Діаграма класів, що використовуються для призначення обчислювальних робіт.

Клас Connection відповідає дузі графу задачі, показує яка задача повинна виконуватись після закінчення поточної задачі. Як і дуга графу задачі, містить показник на два Task’и: попередній та наступний.

Кожна передача даних між процесором, на якому завершилось виконання попереднього Task`у та наступним Task’ом може містити декілька проміжних передач між іншими Processor`ами, які зв’язують ці Processor`и. Для моделювання цих передач використовується клас Transmission. Клас Connection містить стек Transmission’ів. При виконанні передачі між двома проміжними Processor`ами, до стеку додається новий Transmission, за допомогою методу emplace(), а за допомогою методу popAndDelete() зі стеку видаляється останній доданий Transmission. Усі Transmission’и, які зберігаються у Connection’і мають однакову довжину на діаграмі Ганта, яка дорівнює вазі Connection’а. Якщо певний Task буде виконуватись на тому ж процесорі, на якому виконувався попередній Task, тобто ніяких передач між Processor`ами не треба виконувати, Connection буде містити порожній стек.

Transmission являє собою абсолютно абстрактний клас (інтерфейс), а його конкретними реалізаціями є SendedTransmission та ReceivedTransmission. Вони відповідають за передачу та прийом даних, відповідно, між двома суміжними процесорами. Причому ReceivedTransmission являється приватним вкладеним класом класу SendedTransmission. SendedTransmission сам потурбується про створення та видалення ReceivedTransmission.

Клас Processor відповідає вершині графу системи. Для вирішення питань маршрутизації, кожен Processor містить вектор дистанцій, на зразок вектору дистанцій у мережевому протоколі OSPF. Вектор дистанцій для кожного цільового Processor’у ставить у відповідність пару значень: мінімальна відстань до цільового Processor’у та перелік суміжних процесорів через які можна добратись до цільового Processor’у за мінімальну кількість переходів. Окрім вектору дистанцій, кожен процесор зберігає список Task’ів, які на ньому будуть виконуватись, а також списки фізичних та логічних зв’язків, PhysicalLink та LogicalLink, відповідно. Клас Processor являється абстрактним. Його дочірніми класами є IOProcessor та SimpleProcessor. IOProcessor відповідає Processor’у, який містить контролер вводу-виводу, а SimpleProcessor - Processor’у без контролеру вводу-виводу.

Клас LogicalLink відповідає дузі між двома вершинами у графі системи, тобто каналу зв’язку між Processor’ами. У LogicalLink’у зберігається інформація про суміжні Processor`и. Так як канали зв’язку можуть бути дуплексними або напівдуплексними, були розроблені класи FullDuplexLink та HalfDuplexLink. Ключовою відмінністю між ними є те, що FullDuplexLink зберігає два списки Transmission’ів, в обидві сторони, а HalfDuplexLink зберігає Transmission’и в одному списку.

Клас PhysicalLink містить список LogicalLink’ів, по яким може бути виконана передача. Цей список співпадає зі списком, що зберігається у Processor’і, якому належить цей PhysicalLink. Так як PhysicalLink тісно пов’язаний із структурою LogicalLink, клас PhysicalLink також поділений на два класи, для урахування дуплексної та напівдуплексної передачі: FullDuplexPhysicalLink та HalfDuplexPhysicalLink. Кожен PhysicalLink може виконувати передачу даних у певний час по певному LogicalLink’у. У випадку використання дуплексної передачі неможливо виконувати передачу даних по одному LogicalLink’у, а прийом по іншому LogicalLink’у, з використанням одного PhysicalLink’у. У цьому випадку, необхідно використовувати різні PhysicalLink’и, або зачекати, коли PhysicalLink звільниться.

### Моделювання пересилок

Для кожного з Processor’ів, на яких виконувались попередні Task’и виконуємо моделювання передачі по кожному з можливих шляхів, які зберігаються у векторі дистанцій цього процесора. Іншими словами, виконуємо передачу по кожному з можливих шляхів і обираємо той, на якому час прийому усіх даних Processor`ом, необхідних для виконання поточного Task’у буде найменшим. Виконання передачі між Processor’ами полягає у пошуку часу, коли може бути виконана передача даних між Processor’ами, тобто пошуку часу, коли виконуються усі наступні умови:

* Processor з якого виконується передача даних не виконує обчислень (у разі використання SimpleProcessor'у).
* Processor на якому виконується прийом даних не виконує обчислень (у разі використання SimpleProcessor'у).
* На Processor’і з якого виконується передача даних існує вільний PhysiclLink, який може виконати передачу даних необхідного розміру.
* На Processor’і на якому виконується прийом даних існує вільний PhysiclLink, який може виконати прийом даних необхідного розміру.
* На LogicalLink’у, що з’єднує два зазначених вище Processor`и може бути виконана передача даних.

### Відображення діаграми Ганта

Наприкінці роботи алгоритму призначення отримуємо послідовності Task'ів, які виконуються на кожному процесорі, та список Transmission’ів, які виконуються на кожному з PhysicalLink’ів, які необхідно представити у вигляді діаграми Ганта. Для виводу діаграми Ганту на екран використовується бібліотека QCustomPlot. QCustomPlot – це Qt C++ віджет для візуалізації графіків та різних наборів даних. TimeSpan містить віртуальний метод draw(). Дочірні класи SendedTransmission, ReceivedTransmission та Task мають його перевизначити для коректного виведення інформації на QCustomPlot.

## Обчислення показників ефективності алгоритмів планування.

Завершальний етап даної роботи – порівняння алгоритмів планування та дослідження впливу кількості фізичних зв’язків, дуплексності та наявності контролеру вводу-виводу на процесорі.

Для порівняння алгоритмів планування генерується багато графів різної зв'язності, з різною кількістю вершин. Чим більше кількість графів певної зв'язності з певною кількістю вершин, тим результат аналізу буде більш точним. Після генерації графів, для кожного з них обчислюються діаграми Ганта з використанням різних алгоритмів, збирається статистика, щодо ефективності та часу виконання кожного алгоритму. За допомогою QCustomPlot, виводяться графіки зі статистичними даними кожного алгоритму.

Для дослідження впливу кількості фізичних зв’язків, дуплексності та наявності контролеру вводу-виводу на процесорі використовуються ті ж самі графи, що були створені раніше. Але цього разу обчислюються діаграми Ганта для обраних алгоритмів планування в комбінації з:

* різною кількістю фізичних зв’язків;
* наявність/відсутність дуплексності;
* наявність/відсутність контролеру вводу виводу.

Так як кількість обчислень для порівнянні різних алгоритмів планування дуже висока, було вирішено використати паралельні обчислення. Задача має високу ступінь придатності для виконання на багатопроцесорних системах. При порівняння алгоритмів планування, кожен алгоритм планування запускається у окремому потоці виконання. При виконанні обчислень до даної курсової роботи створюється 9 потоків виконання. Якщо для дослідження впливу кількості фізичних зв’язків, дуплексності та наявності контролеру вводу-виводу використовувати той же самий алгоритм розподілення обчислень, то кількість потоків може перевищити 100 потоків. Для уникнення такої кількості потоків, вирішено відвести під кожен потік виконання задачі більшого розміру. Тому для даного дослідження в потік виконання виконує обчислення всіх обраних алгоритмів планування для однієї з комбінацій параметрів системи.

На початку планування на діалоговому вікні з’являється progress bar, який показує кількість обчислених діаграмі Ганта по відношенню до загальної кількості діаграм Ганта, які необхідно обчислити, тобто прогрес виконання обчислень.

# ПОРІВНЯННЯ АЛГОРИТМІВ ПЛАНУВАННЯ

Для порівняння алгоритмів планування необхідно визначити наступні параметри:

* граф системи;
* кількість вершин (мінімальну, максимальну та крок нарощування);
* вагу вершин (мінімальну та максимальну);
* зв'язність (початкову, кінцеву та крок нарощування);
* мінімальну зв'язність;
* кількість прогонів.

Система має форму тору розміром 3\*3\*3. Таким чином, система складається з 27 вершин, кожна з яких має по 6 зв’язків.

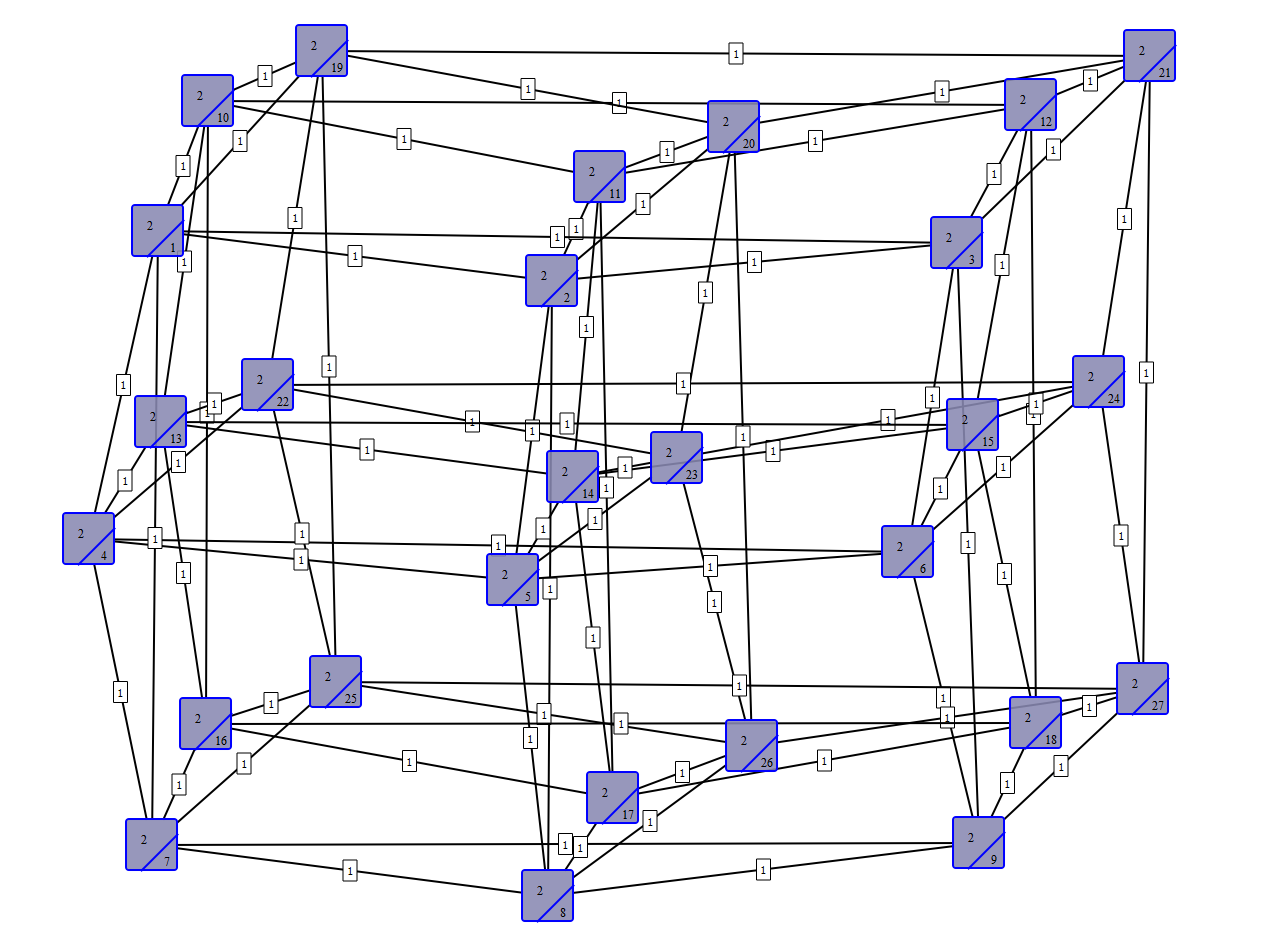


Рис 3.1. Граф системи (тор, розміром 3\*3\*3).

Проведемо порівняння для кількості вершин у графі задачі, яка дорівнює, вдвічі більша та втричі більша кількості вершин у графі системи. Тобто, для графа системи у вигляді тору, який складається з 27 вершин, задамо мінімальну кількість вершин 27, максимальну – 81, а крок нарощування – 27.

Вагу вершин визначимо в межах 1 – 50, це значить, що під задачі які необхідно виконати можуть різнитись в десятки разів.

При зменшенні значення зв’язності, кількість дуг у графі збільшується, внаслідок цього, збільшується і час, необхідний для обчислення. Це NP-повна задача, тому при дуже великій кількості дуг, обчислення діаграми Ганта може зайняти роки. Задамо початкову зв’язність 0.2, кінцеву – 1.0, крок нарощування – 0.1 та мінімальну зв’язність – 0.01. Для 81 вершини кількість дуг в залежності від зв’язності зображена на рисунку 3.1

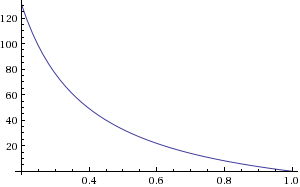


Рис. 3.2. Кількість дуг у графі задачі в залежності від зв’язності.

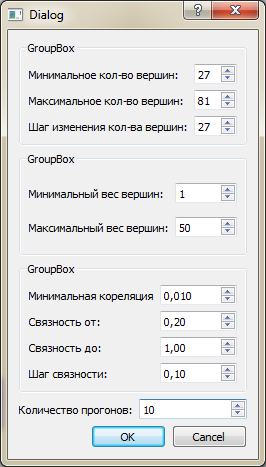


Рис. 3.3. Параметри, задані для порівняння алгоритмів.

Кількість прогонів визначає кількість випадкових графів певної зв’язності, які будуть моделюватись. Для отримання більш точних значень обчислень необхідно задати більшу кількість прогонів. В рамках даного порівняння, кількість прогонів обрано 10.

При цих значеннях, моделювання займає близько години на чотириядерному процесорі AMD Phenom(tm) II X4 B50, з тактовою частотою 3.2 GHz.

## Порівняння алгоритмів формування та призначення

Введемо скорочену форму запису алгоритмів формування та призначення. Будемо записувати у форматі X/Y, де X – алгоритм призначення, а Y – алгоритм формування черги.

### Порівняння ефективності та коефіцієнту прискорення

Коефіцієнт прискорення та ефективність пов’язані між собою формулою:

В рамках даної курсової роботи, кількість процесорів залишається незмінною, значення цих показників будуть повторювати друг друга з єдиною відмінністю, вони будуть відрізнятися в 27 разів. Тому має сенс виконувати порівняння одного з цих показників і розповсюдити висновки на інший.

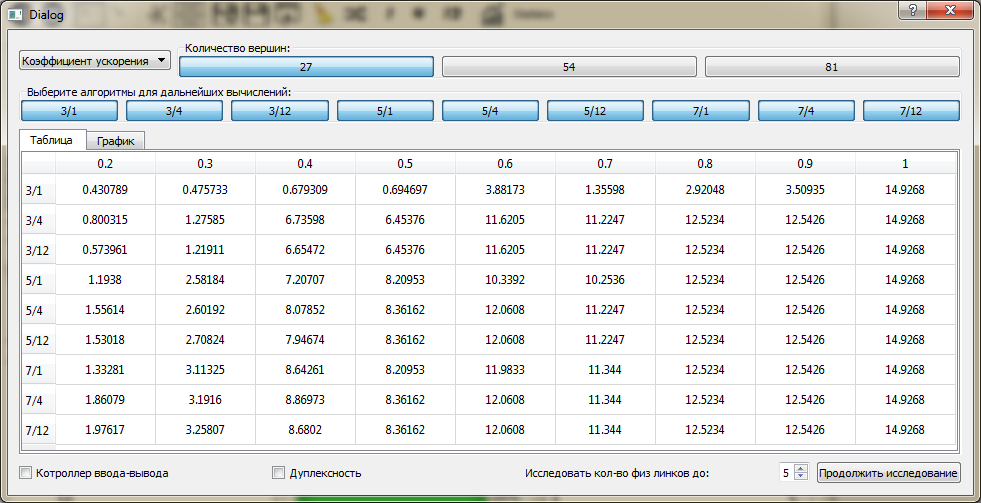


Рис. 3.4. Таблиця коефіцієнту прискорення для задач розміром 27 вершин.

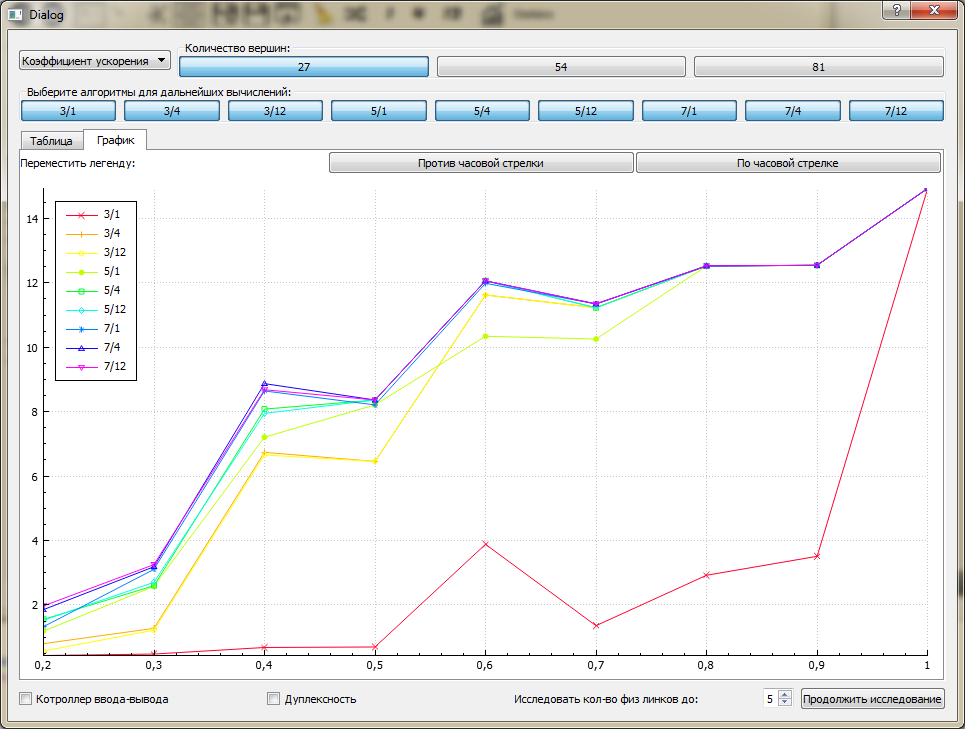


Рис. 3.5. Графік коефіцієнту прискорення для задач розміром 27 вершин.

Згідно рисунків 3.4 та 3.5, на 27 вершинах у графі задач, різниця в ефективності роботи алгоритмів майже не помітна, за виключенням третього алгоритму призначення, який виявився гіршим за інші. А алгоритм 3/1 взагалі непридатний. Ефективність значно гірше за інші алгоритми.

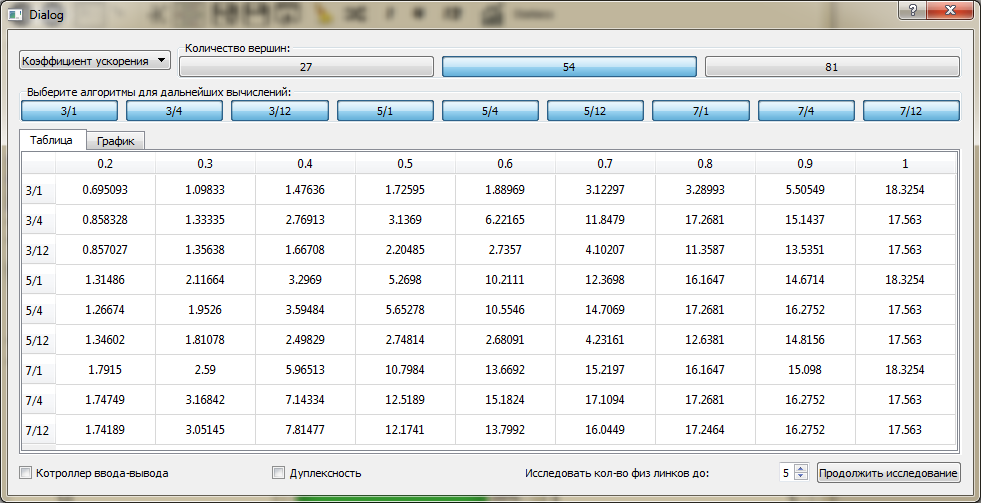


Рис. 3.6. Таблиця коефіцієнту прискорення для задач розміром 54 вершини.

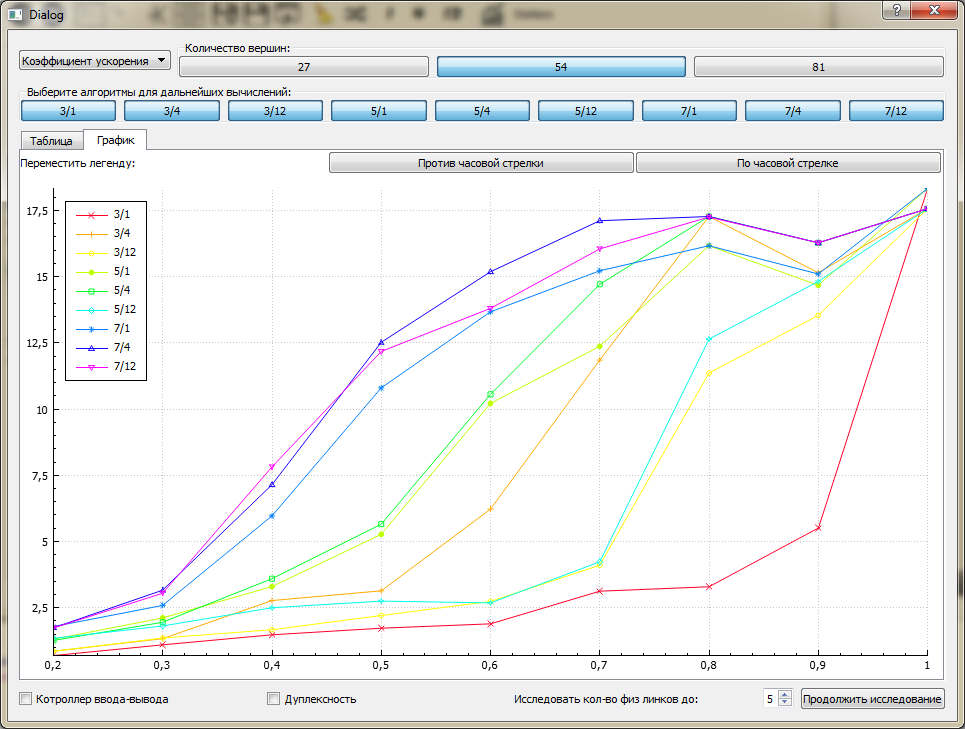


Рис. 3.7. Графік коефіцієнту прискорення для задач розміром 54 вершини.

При 54 вершинах у графі задач, різниця між ефективністю різних алгоритмів становиться більш явною. Це особливо помітно при значеннях зв’язності в межах 0.4 – 0.7.

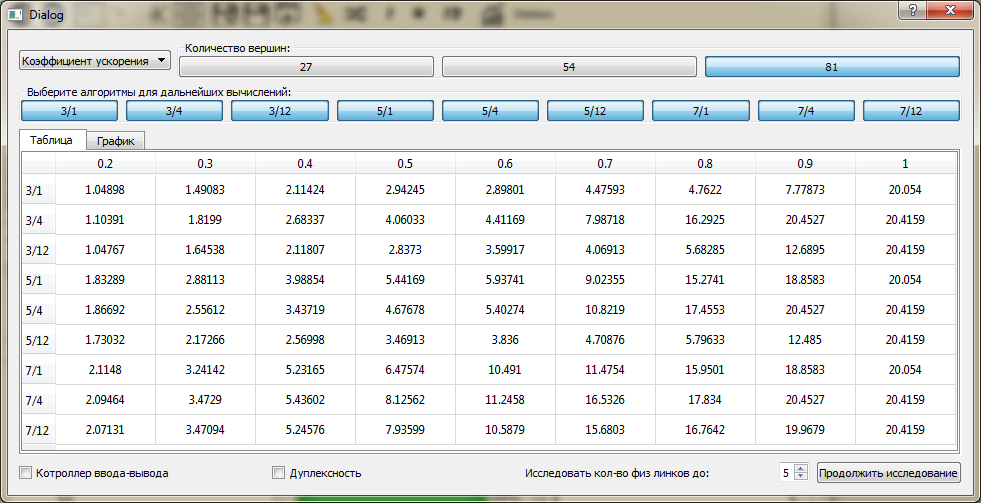


Рис. 3.8. Таблиця коефіцієнту прискорення для задач розміром 81 вершину.

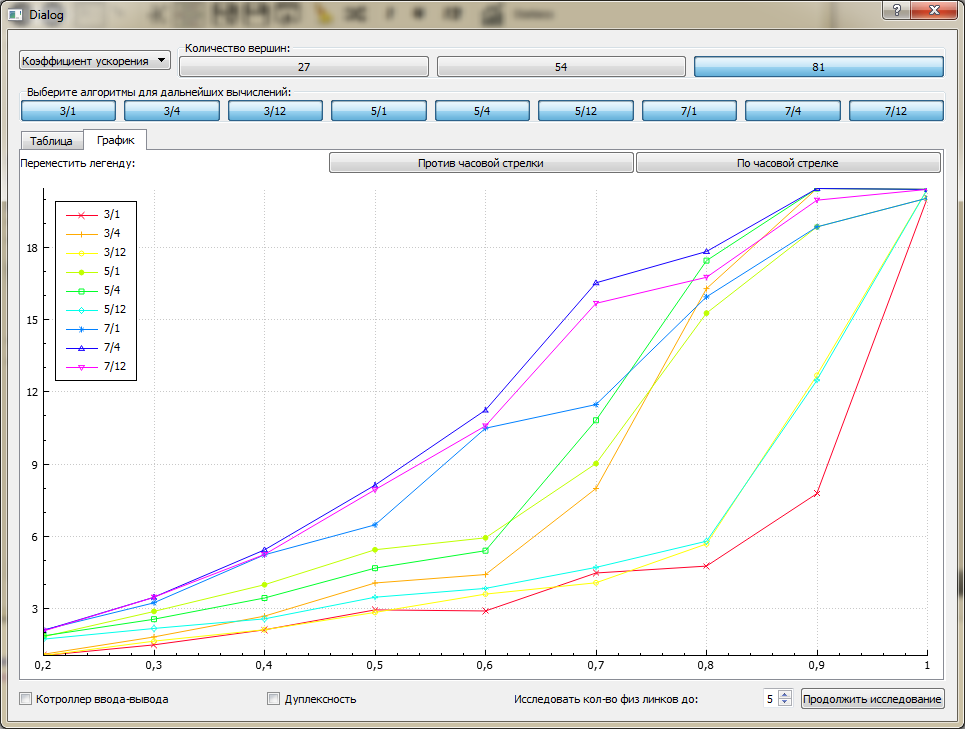


Рис. 3.9. Графік коефіцієнту прискорення для задач розміром 81 вершину.

Ефективності алгоритмів планування для графів задачі, що складаються з 54 та з 81 вершин мають спільні мають багато спільного. Згідно рисунків 3.7 та 3.9, можна виділити 3 категорії алгоритмів за ефективністю:

1. Найбільш ефективні: 7/1, 7/12, 7/4, тобто 7 алгоритм призначення.
2. Достатньо ефективні: 3/4, 5/1, 5/4.
3. Неефективні: 3/1, 3/12, 5/12.

Ефективність 7 алгоритму призначення базується на підвищеній складності самого алгоритму. Для планування задач з використанням 7 алгоритму необхідно виконати значно більше обчислень і витратити значно більшу кількість часу.

3 алгоритм призначення виконує найменшу кількість обчислень, тому отримує найменш ефективні результати. Не зважаючи на це, алгоритм 5/12 виявився значно гіршим за алгоритм 3/4.

### Порівняння часу виконання обчислень

Час виконання експотенційно збільшується зі зменшенням зв’язності, оскільки зменшення зв’язності призводить до збільшення кількості дуг у графі задачі. Так, наприклад, для моделювання зв’язності 0.3 достатньо хвилини, а на моделювання зв’язності 0.2 знадобилась година процесорного часу.

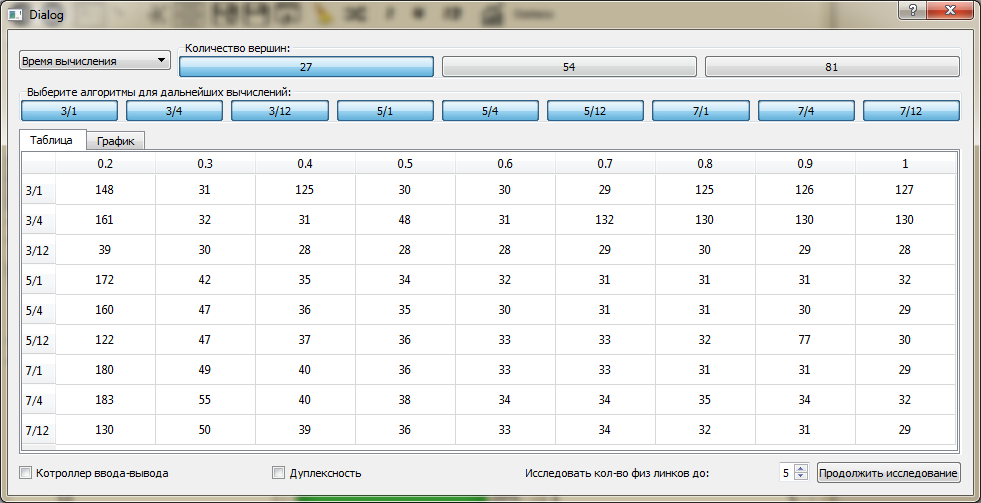


Рис. 3.10. Таблиця часу обчислення для задач розміром 27 вершин.

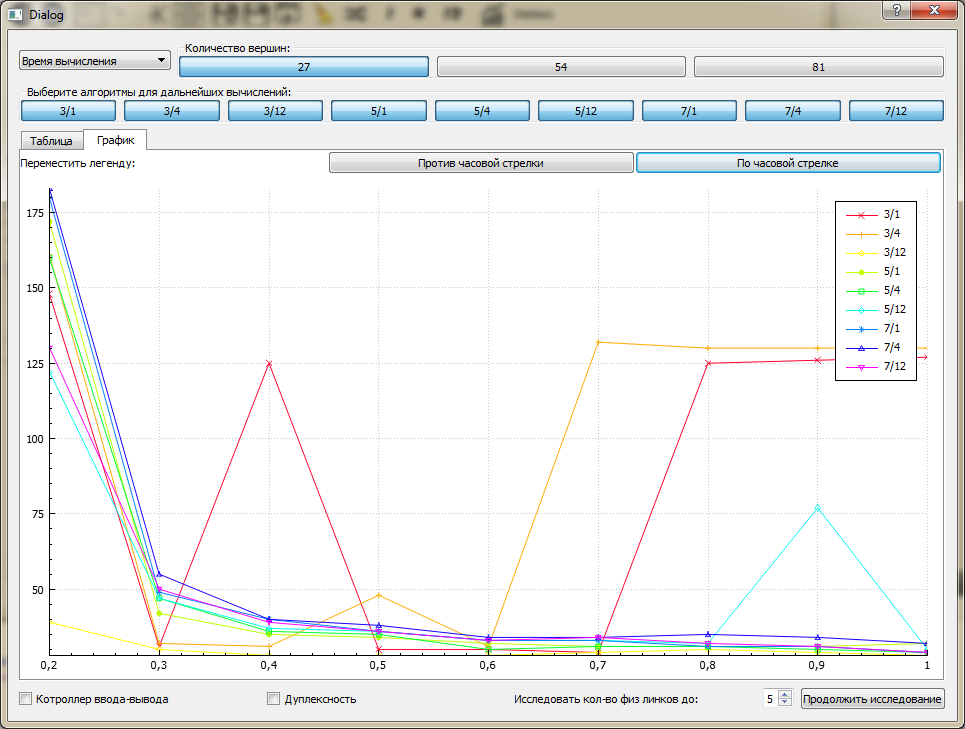


Рис. 3.11. Графік часу обчислення для задач розміром 27 вершин.

Відповідно до рис. 3.11, при 27 вершинах у графі задач, середній час виконання всіх алгоритмів крім 3/1, 3/4, 3/12 та 5/12 відрізняється не сильно. Однак, для алгоритмів 3/1 та 3/12 при певних значеннях зв’язності кількість часу, затрачена на моделювання надмірно висока. А алгоритм 3/12, навпаки, займає значно менше часу на моделювання в порівнянні з іншими алгоритмами.

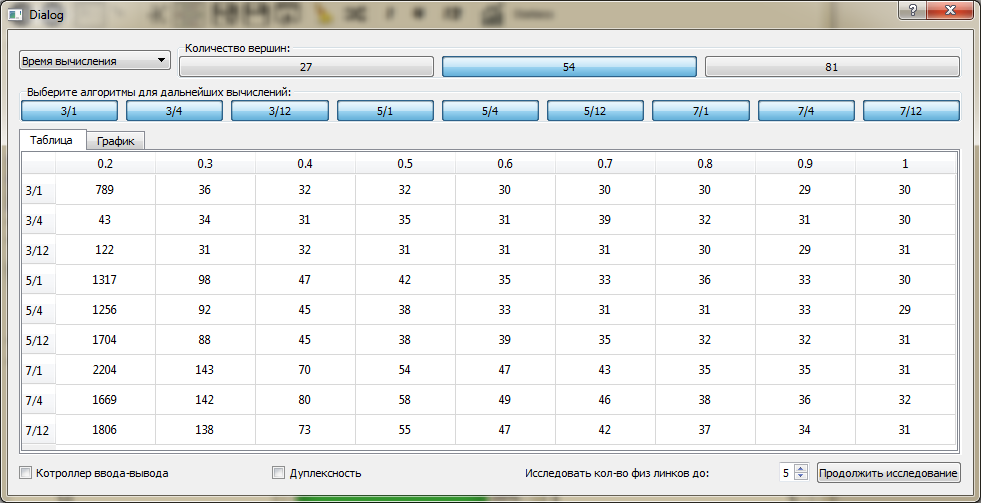


Рис. 3.12. Таблиця часу обчислення для задач розміром 54 вершини.

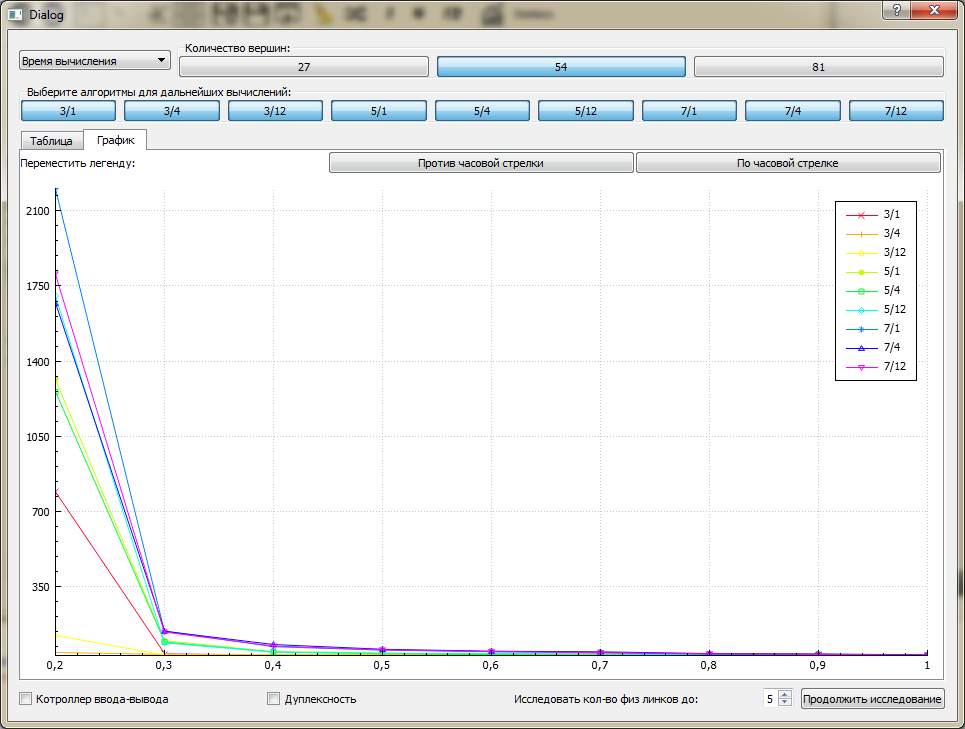
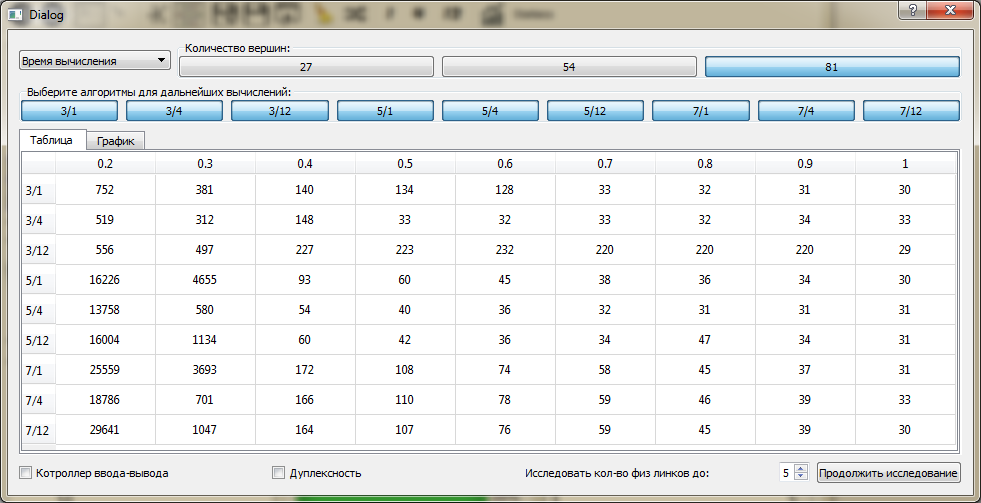


Рис. 3.13. Графік часу обчислення для задач розміром 54 вершини.

При 54 вершинах у графі задач, та при низьких значеннях зв’язності різниця між часом виконання становиться більш чіткою.

 Рис. 3.14. Таблиця часу обчислення для задач розміром 81 вершини.

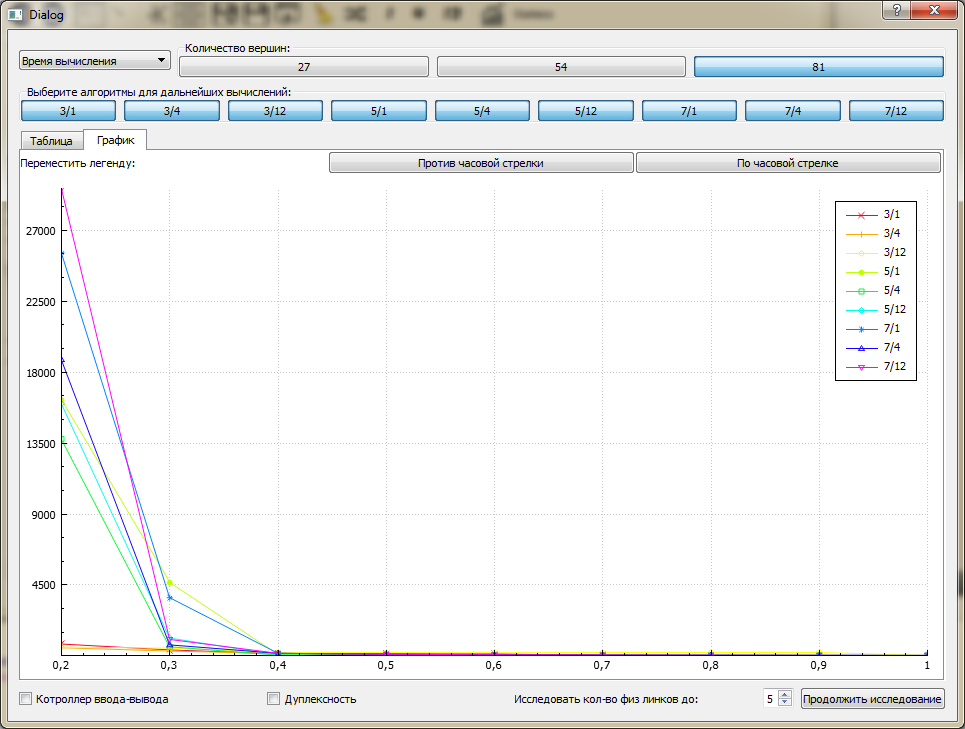


Рис. 3.15. Графік часу обчислення для задач розміром 81 вершин.

Графіки часу виконання алгоритмів планування для графів задачі, що складаються з 54 та з 81 вершин (рис.3.13 та рис. 3.15 відповідно) схожі. Саме на них необхідно проводити порівняння алгоритмів за часом виконання.

Як і прогнозувалось, 3 алгоритм призначення займає значно менше часу за інші, а 7 алгоритм призначення – навпаки, найбільше.

При цьому, алгоритм 3/1 найбільш нестабільний, і при певних значеннях може займати багато часу на виконання. Серед алгоритмів фаворитів по ефективності, алгоритми 7/1 та 7/12 вимагають більше часу на обчислення ніж 7/4, при цьому 7/4 виявляється ефективніше за них.

## Дослідження додаткових параметрів системи

Додатковими параметрами системи, що необхідно дослідити у рамках цієї лабораторної роботи є кількість фізичних зв’язків, наявність контролеру вводу-виводу та наявність дуплексності у каналах зв’язків системи.

Дослідження додаткових параметрів проводяться на найбільш оптимальному алгоритмі планування. Згідно порівнянню, проведеному у попередньому розділі, найбільш оптимальним алгоритмом є алгоритм 7/4. Він виявився найбільш ефективним, при цьому він займає менше часу на виконання обчислень у порівнянні з іншими алгоритмами 7/1 та 7/12.

### Дослідження впливу кількості фізичних зв’язків

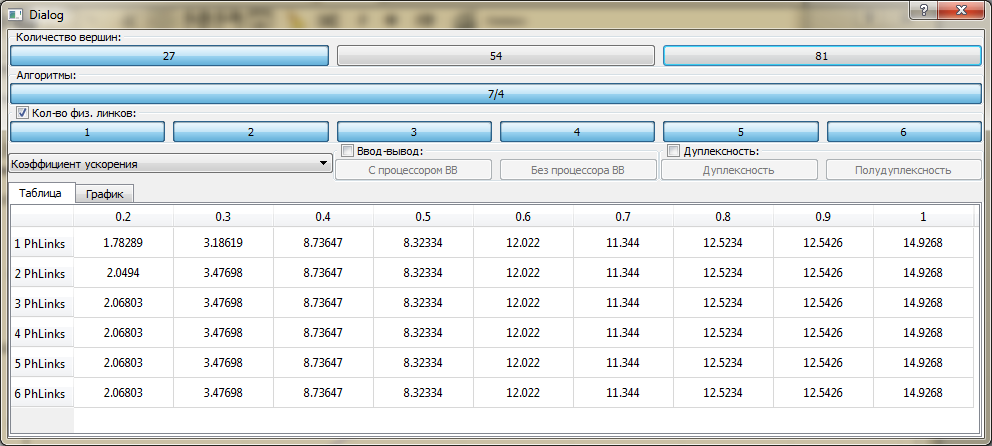


Рис. 3.16. Таблиця коефіцієнтів прискорення при різній кількості фізичних зв’язків для задач розміром 27 вершин.

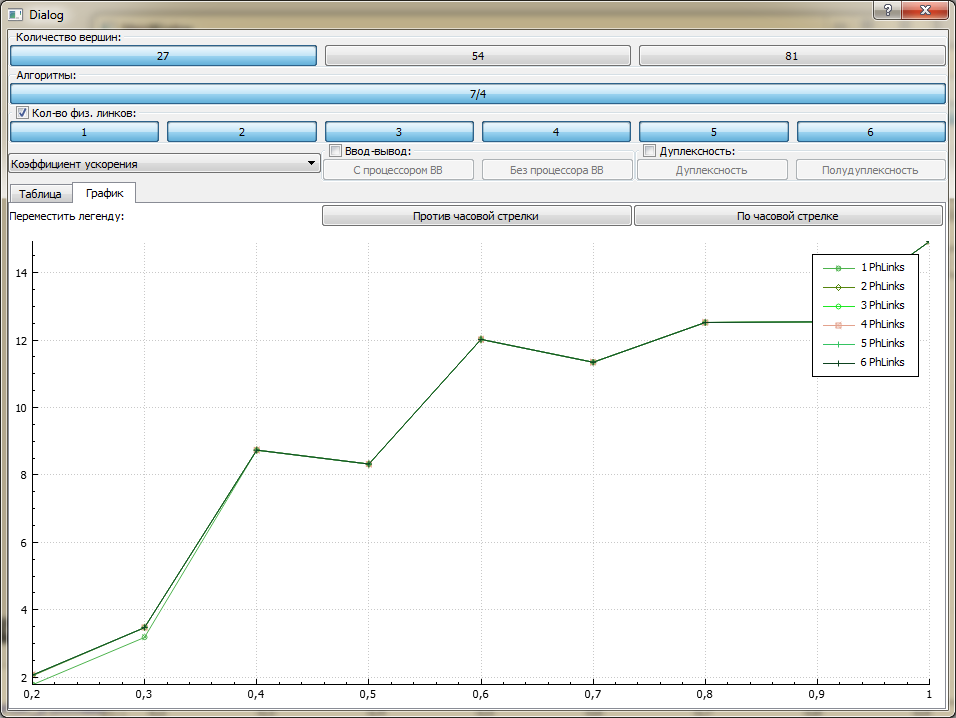


Рис. 3.17. Графік коефіцієнту прискорення при різній кількості фізичних зв’язків для задач розміром 27 вершин.

При 27 вершинах у графі задачі, різниця між ефективністю при використанні додаткових фізичних зв’язків мінімальна. Причому, при використанні більше 2 фізичних зв’язків приросту ефективності немає, отже при низькій зв’язності слід використовувати не більше 2 фізичних зв’язків, а при великих значеннях коефіцієнту зв’язності – не більше одного зв’язку.

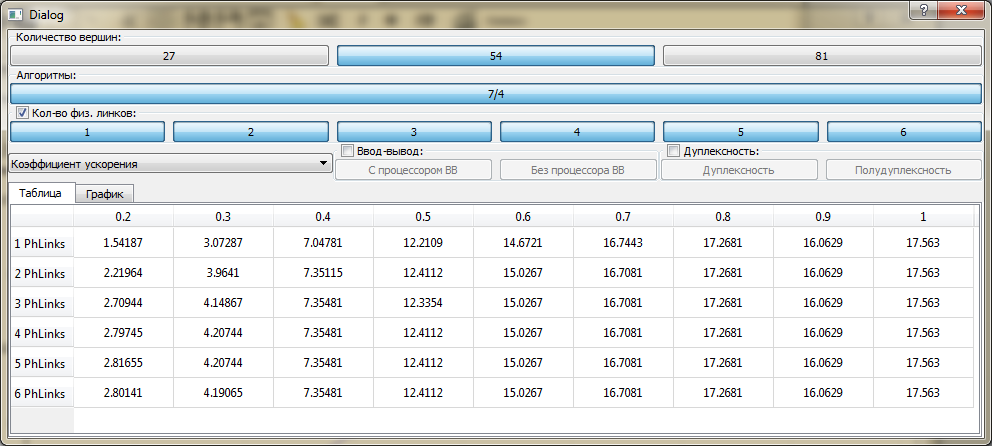


Рис. 3.18. Таблиця коефіцієнту прискорення при різній кількості фізичних зв’язків для задач розміром 54 вершини.

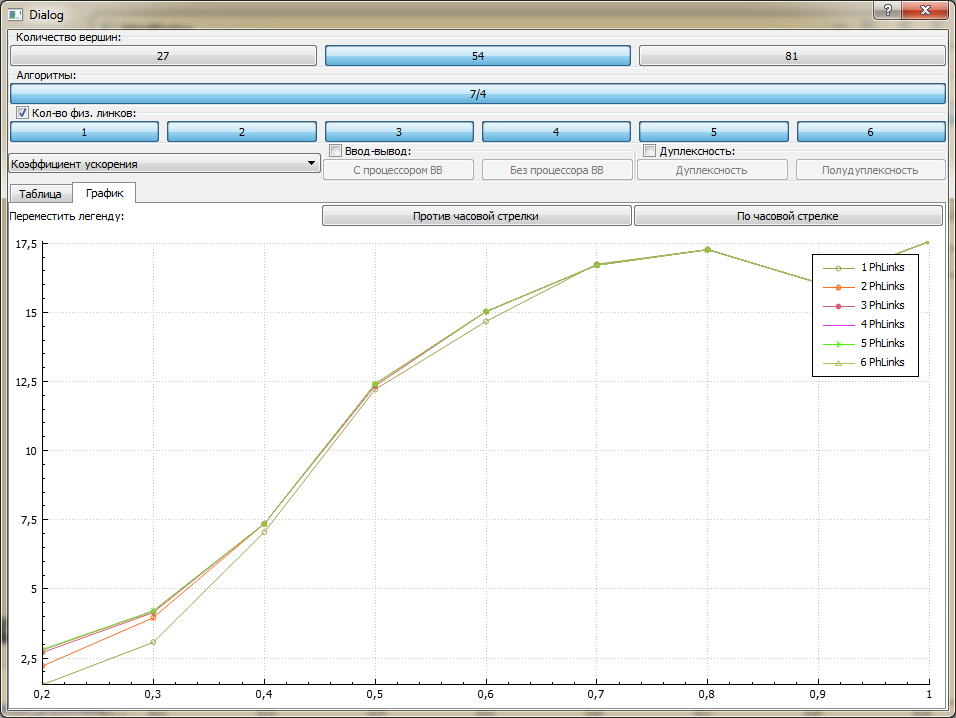


Рис. 3.19. Графік коефіцієнту прискорення при різній кількості фізичних зв’язків для задач розміром 54 вершини.

Для задач розміром 54 вершини, при коефіцієнті зв’язності менше 0.3, має сенс використовувати 3 фізичні канали, при коефіцієнті зв’язності 0.4-0.7 – 2 фізичні канали, а при коефіцієнті зв’язності більше 0.7, взагалі один канал.

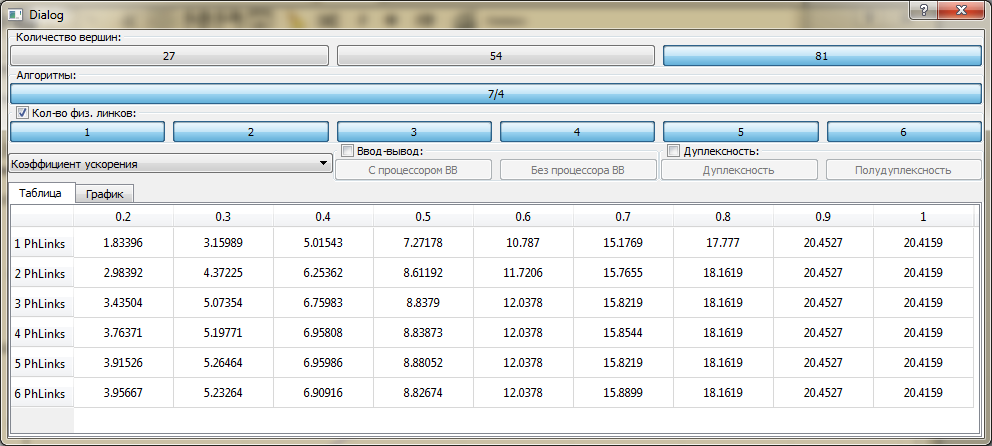


Рис. 3.20. Таблиця коефіцієнту прискорення при різній кількості фізичних зв’язків для задач розміром 81 вершину.

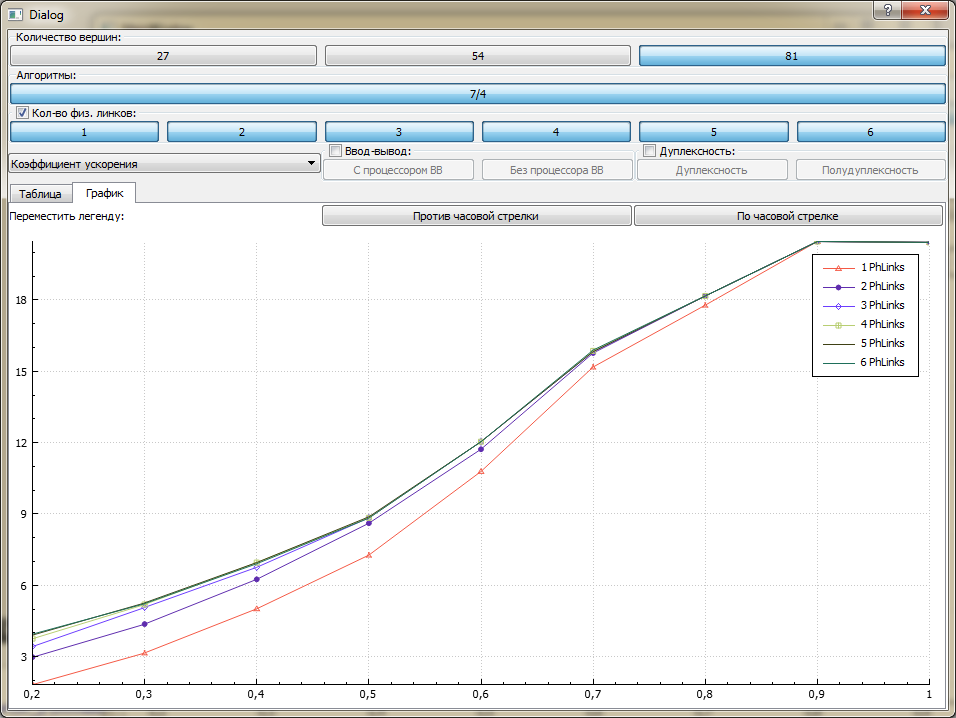


Рис. 3.21. Графік коефіцієнту прискорення при різній кількості фізичних зв’язків для задач розміром 81 вершину.

Для задач розміром 81 вершину, для найбільшої ефективності в системі слід використовувати:

* 4 фізичні зв’язки, при зв’язності задачі менше 0.5;
* 3 фізичні зв’язки, для задач зі зв’язністю 0.5-0.7;
* 2 фізичні зв’язки, для задач зі зв’язністю 0.7-0.8;
* 1 фізичний зв’язок, для задач зі зв’язністю більше 0.8.

Можна використовувати і меншу кількість фізичних зв’язків, ніж зазначену вище, але, у цьому випадку, ефективність системи буде значно гіршою. Так, наприклад, при зв’язності 0.2 ефективність системи з 1 та з чотирма фізичними зв’язками вище у 2 рази.

Зі збільшенням кількості вершин і зі зменшенням коефіцієнту зв’язності у графі задачі, збільшується і необхідність у додаткових фізичних зв’язках.

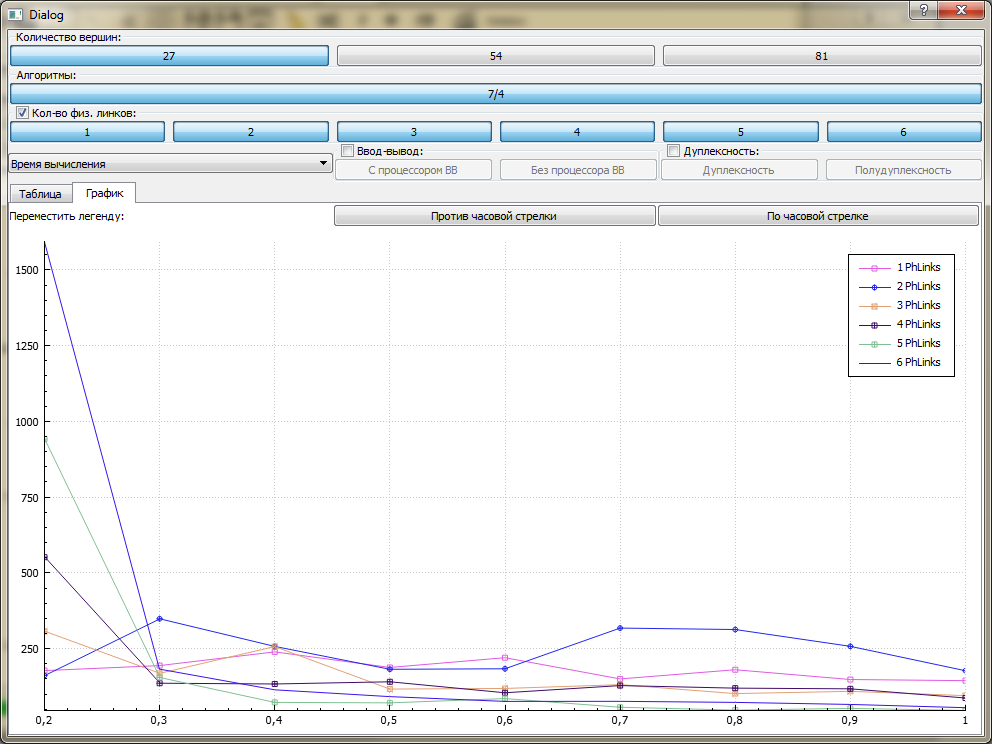


Рис. 3.22. Графік часу виконання при різній кількості фізичних зв’язків для задач розміром 27 вершин.

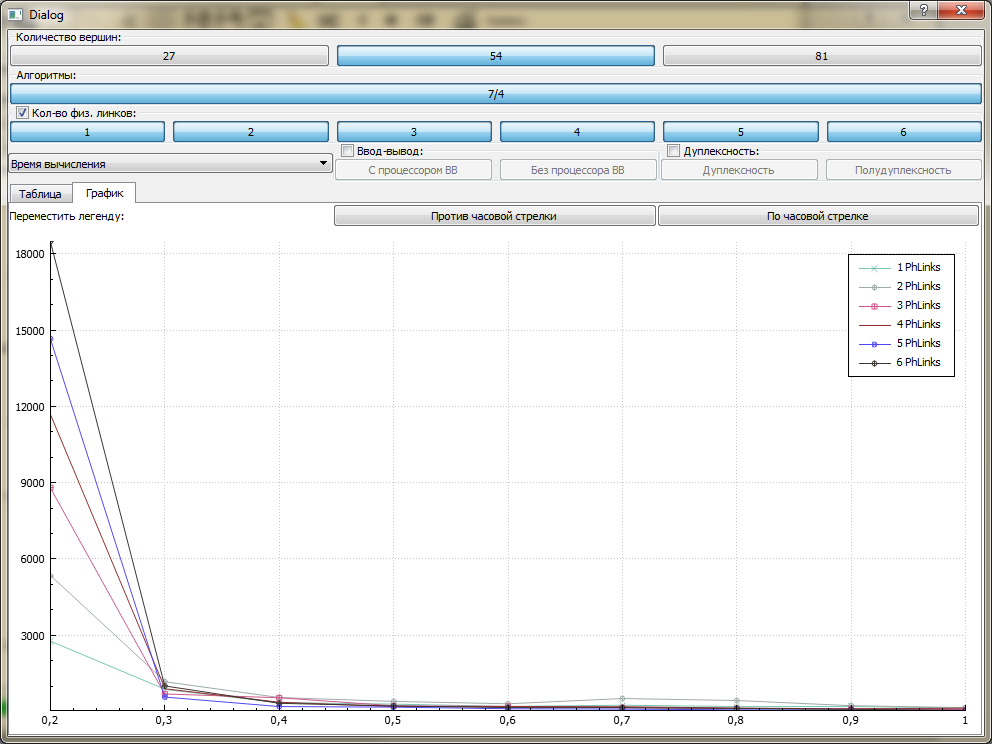


Рис. 3.23. Графік часу виконання при різній кількості фізичних зв’язків для задач розміром 54 вершини.

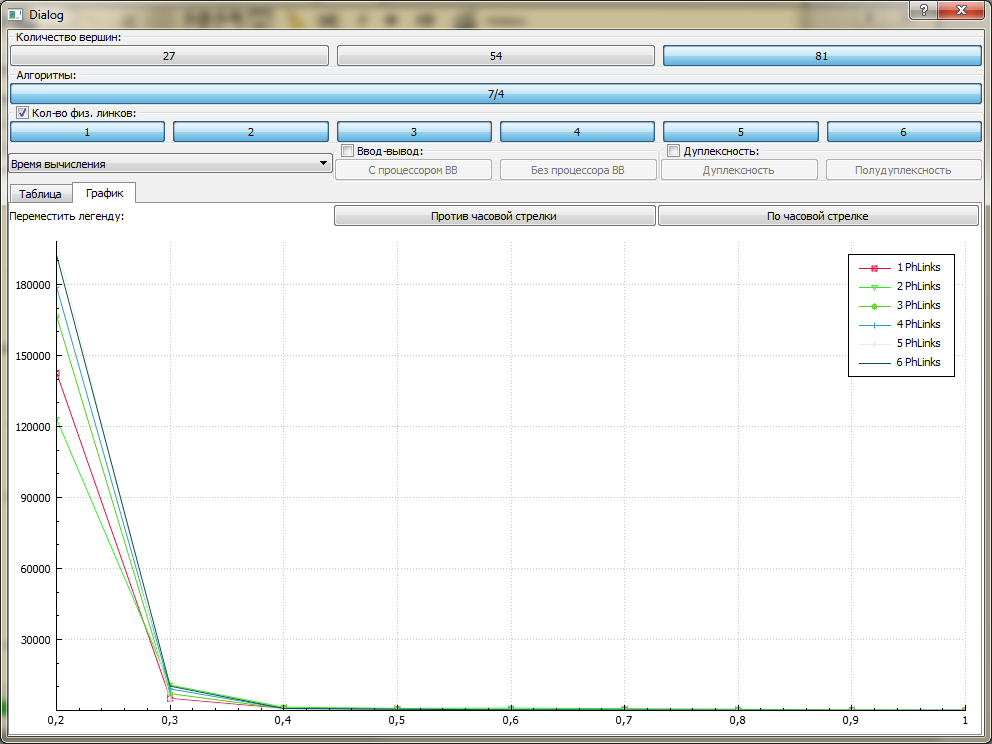


Рис. 3.24. Графік часу виконання при різній кількості фізичних зв’язків для задач розміром 81 вершину.

Якщо порівнювати час обчислення для різної кількості фізичних зв’язків (рисунки 3.22-3.24), то можна помітити деякі закономірності. З ростом кількості фізичних зв’язків збільшується час необхідний для обчислення. Слід зазначити, що для задач розміром 54 вершини, час обчислення прямо пропорційний кількості фізичних зв’язків. Так, час обчислення для 1 фізичного зв’язку дорівнює 3с, для 2 зв’язків – 6с, для 3 – 9с, і т.д. Проте, для задачі розміром 81 вершину такої закономірності не спостерігається. По-перше, час виконання обчислень з використанням 5 та 6 зв’язків співпадають. По-друге, обчислення для 2 фізичних зв’язків займають менше часу, ніж для одного фізичного зв’язку.

### Дослідження впливу контролеру вводу-виводу

Контролер вводу-виводу дозволяє виконувати передачу даних та обчислення на процесорі. При відсутності передач даних, або при їх невеликому обсязі, необхідність у контролері вводу-виводу відпадає.

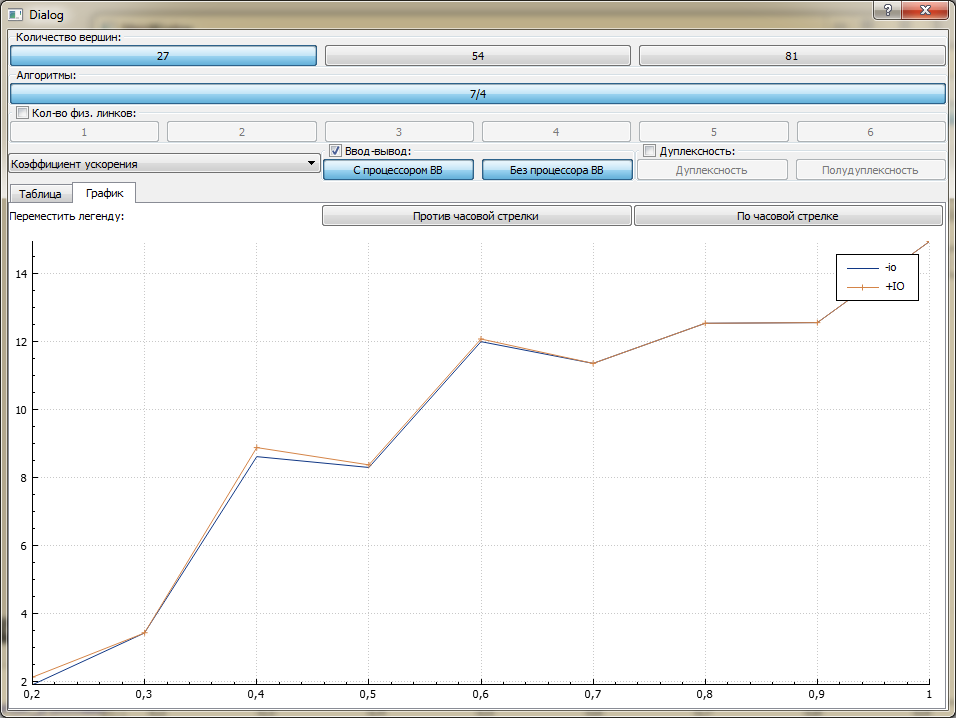


Рис. 3.25. Графік коефіцієнту прискорення з/без контролером(а) вводу-виводу для задач розміром 27 вершин.

Для задач розміром 27 вершин різниця між ефективністю з використанням/без використання контролеру вводу-виводу мінімальна.

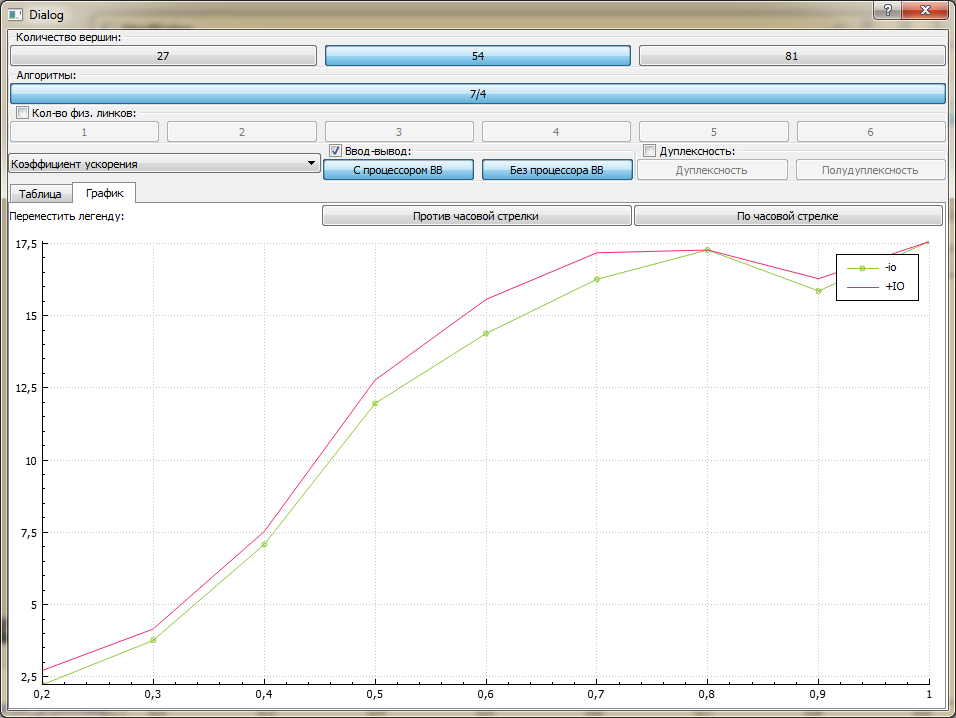


Рис. 3.26. Графік коефіцієнту прискорення з/без контролером(а) вводу-виводу для задач розміром 54 вершини.

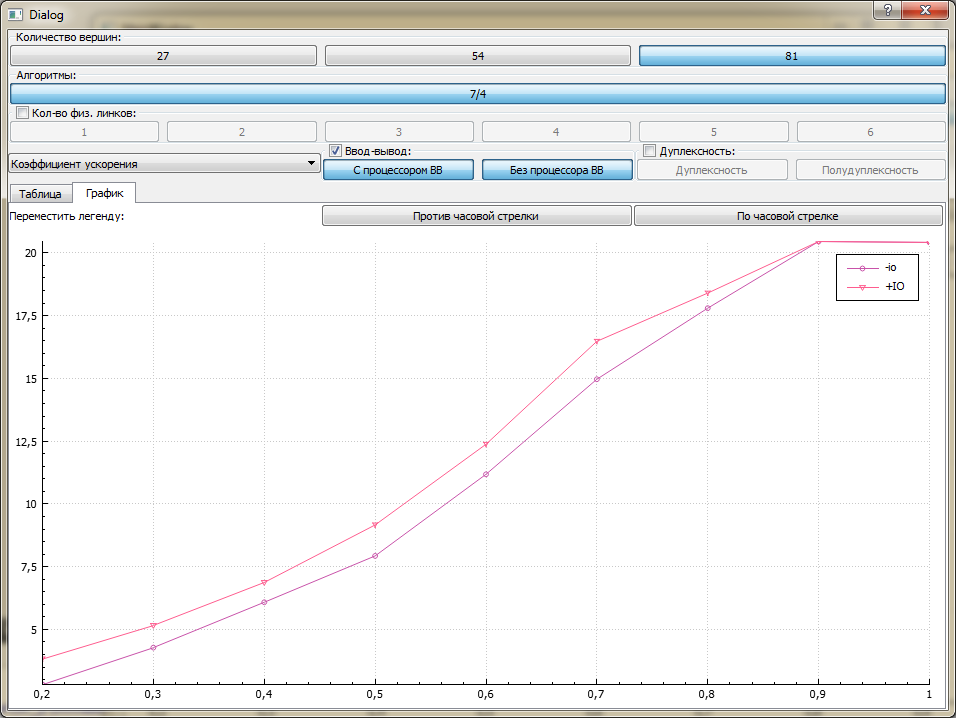


Рис. 3.27. Графік коефіцієнту прискорення з/без контролером(а) вводу-виводу для задач розміром 81 вершину.

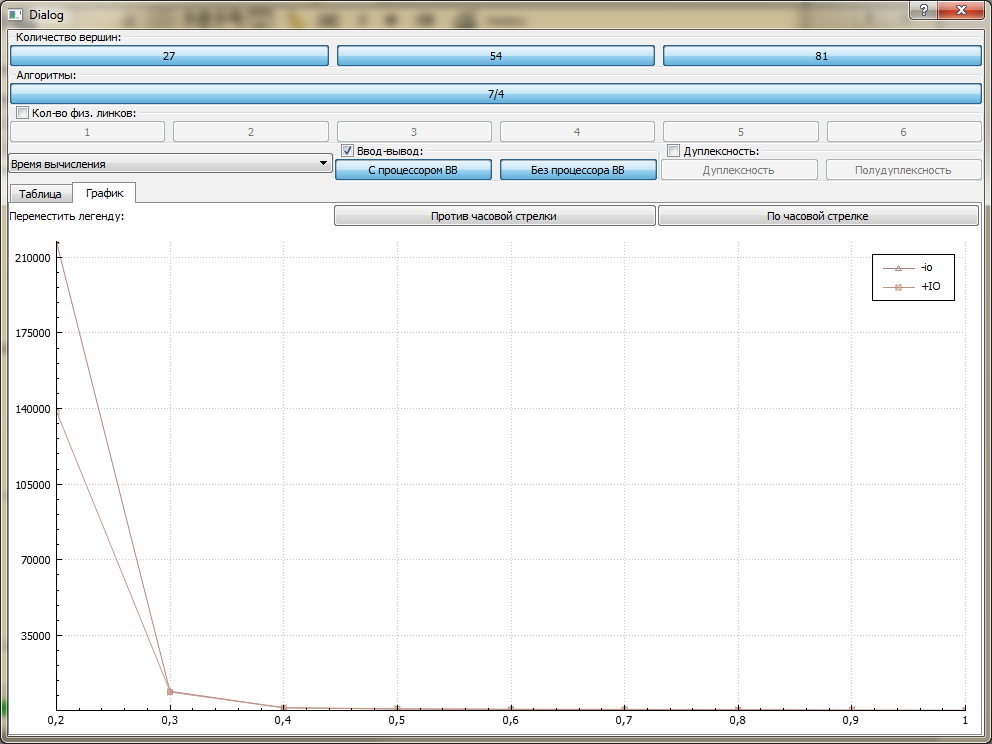


Рис. 3.28. Загальний графік часу виконання з/без контролером(а) вводу-виводу.

Зі збільшенням кількості вершин, а також зі збільшенням зв’язності, кількість передач зростає, як і потреба у контролері вводу-виводу. Для задач розміром 81 вершину, при зв’язності 0.2 та використанні процесору вводу-виводу, коефіцієнт прискорення майже у 3 рази вищій, аніж при його відсутності.

Крім приросту ефективності, використання процесору вводу-виводу дозволяє зменшити час, що витрачається на виконання планування, оскільки, при плануванні пересилок відпадає потреба у перевірках зайнятості самого процесору.

В цілому, приріст ефективності від використання процесору вводу-виводу значно більший ніж використання декількох фізичних зв’язків.

### Дослідження впливу дуплексності зв’язків

Дуплексність у порівнянні з напівдуплексністю, дозволяє виконувати передачу даних в обох напрямках по логічному каналу одночасно.

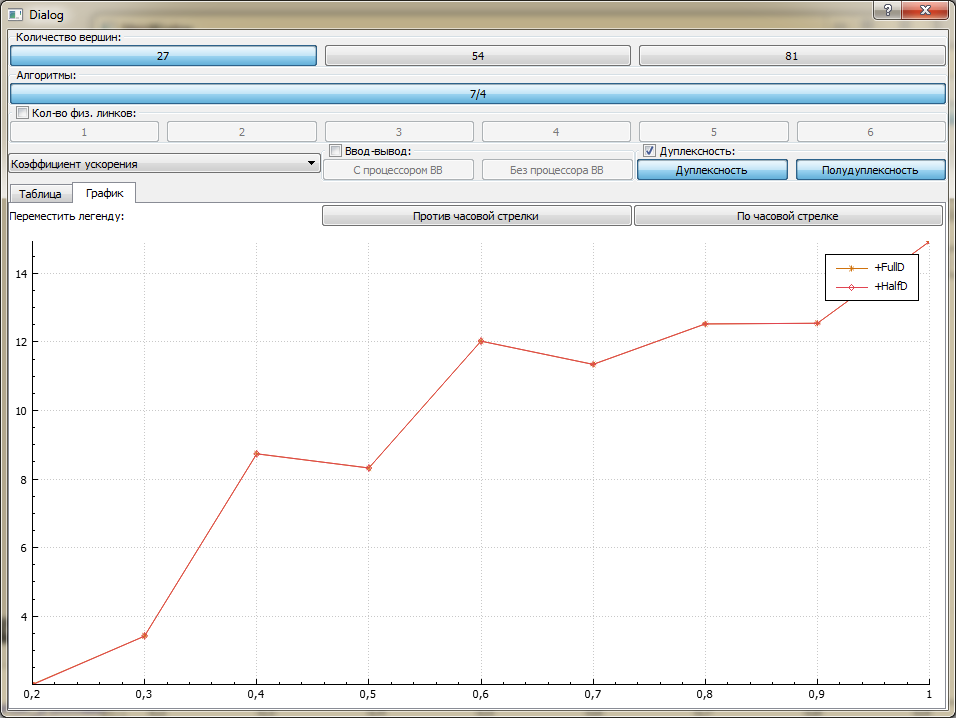


Рис. 3.29. Графік коефіцієнту прискорення при використанні дуплексних/напівдуплексних каналів зв’язку для задач розміром 27 вершин.

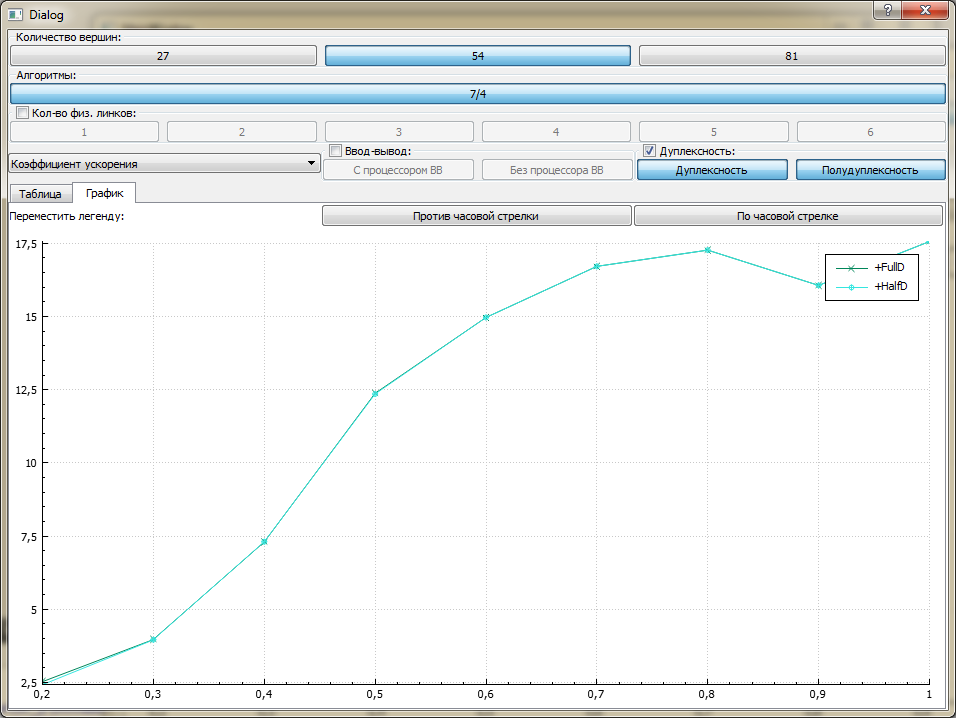


Рис. 3.30. Графік коефіцієнту прискорення при використанні дуплексних/напівдуплексних каналів зв’язку для задач розміром 54 вершини.

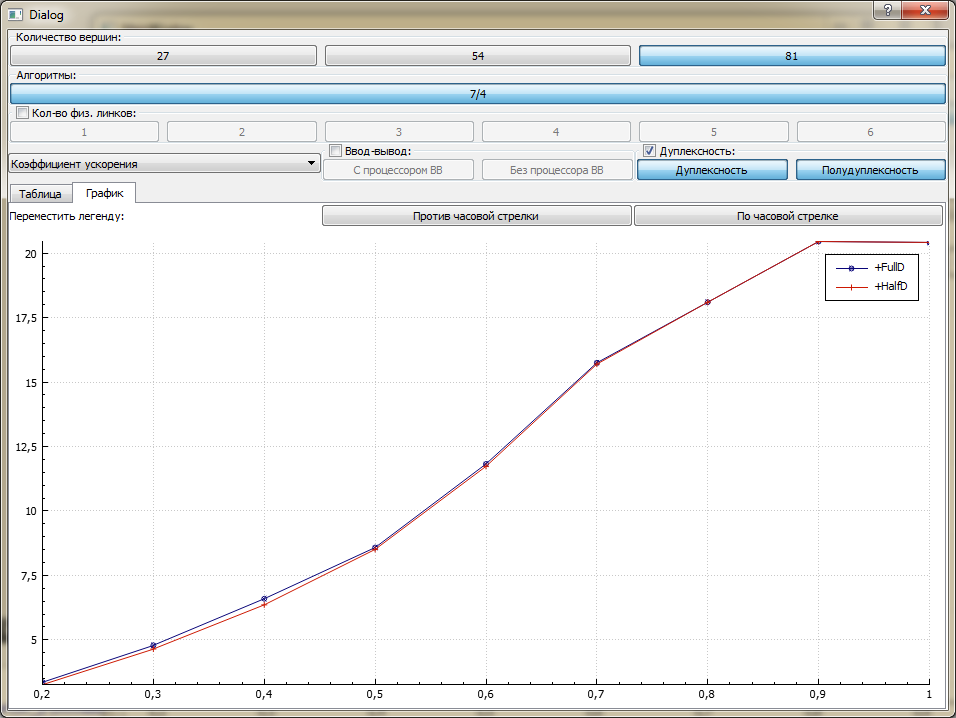


Рис. 3.31. Графік коефіцієнту прискорення при використанні дуплексних/напівдуплексних каналів зв’язку для задач розміром 81 вершини.

Випадки, коли необхідно передати дані між двома сусідніми процесорами в обох напрямах одночасно трапляються дуже рідко, тому приріст ефективності від використання дуплексних каналів зв’язку мінімальний, незалежно від розмірності задачі.

# ВИСНОВКИ

В даній курсовій роботі розглянуто та реалізовано 9 алгоритмів планування, що є комбінацією трьох алгоритмів формування черги та трьох алгоритмів призначення.

В ході виконання практичної частини курсової роботи, порівняно ефективність цих алгоритмів планування. На найбільш ефективному алгоритмі проведено дослідження впливу додаткових показників, таких як кількість фізичних зв’язків, дуплексність/напівдуплексність та наявність/відсутність контролеру вводу-виводу на ефективність роботи системи.

Найбільш ефективними алгоритмами планування виявились алгоритми, які базуються на 7 алгоритмі призначення. Серед них алгоритм, що використовує 4 алгоритм формування черги виявився лідером. Майже при всіх значеннях зв’язності, він надає найбільш ефективні показники ефективності та коефіцієнту прискорення. Алгоритми, які базуються на 5 алгоритмі призначення програють по ефективності 7 алгоритму призначення, а 3 алгоритм призначення, у порівнянні з іншими взагалі виявився неефективним.

Серед додаткових показників системи, найбільший вплив на ефективність має наявність/відсутність контролеру вводу-виводу. Для задач великих розмірів з низькою зв’язністю і великою кількістю залежностей між підзадачами, контролер вводу-виводу може показувати ефективність, яка в рази вище, ніж при відсутності контролеру вводу-виводу.

Наявність декількох фізичних зв’язків також може підвищити ефективність системи, також для задач великих розмірів з низькою зв’язністю і великою кількістю залежностей між підзадачами.

Дуплексність/напівдуплексність спричиняє мінімальний вплив на ефективність системи, без залежності від розмірів задач.

Всі ці додаткові показники не мають сенсу при використанні задач які не перевищують розмірів системи та мають великий коефіцієнт зв’язності. У цьому випадку всі задачі можна розподілити між існуючими процесорами, і ніякого приросту ефективності не відбудеться від зміни параметрів системи.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Русанова О.В., Ярох Ю.Н. Планирование вычислений в гетерогенных кластерных системах. Вісник НТУУ”КПИ”.Сер. Інформатика, управління та обчислювальна техніка. - 2011. Випуск 54. - С.155 –163.
2. Кулаков Ю.А.,Русанова О.В., Шевело А.П. Иерархический способ планирования для GRID Вісник НТУУ”КПИ”.Сер. Інформатика, управління та обчислювальна техніка. - 2009. Випуск 51. - С.57 – 66.
3. Барский А.Б. Параллельные процессы в вычислительных системах. Планирование и организация.-М.:Радио и связь, 1990.- 256 с.
4. Дж.Ортега Введение в параллельные и векторные методы решения линейных систем.-М.:Мир, 1991.-367 с.
5. Корнеев В.В. Параллельные вычислительные системы.-М:Мир, 1999.
6. Трахтенгерц Э.А. Программное обеспечение параллельных процессов.- М.: Наука, 1987.- 356 с.
7. Сайт Qt 5.4. [http://doc.qt.io/qt-5/].

# Додаток А

**Сирцевий код програмної моделі**

**gantt\_chart.h**

#ifndef GANTTCHART\_H

#define GANTTCHART\_H

#include <QMap>

#include <QSet>

#include <QVector>

class UndirectedGraph;

class DirectedGraph;

class Node;

class Task;

class Processor;

class TaskPriorityAlgorithm;

class DestinationAlgorithm;

class GanttChart

{

public:

GanttChart(const UndirectedGraph \*systemGraph, const DirectedGraph \*taskGraph, const QMap<Node\*, int> &linksCount,

TaskPriorityAlgorithm \*taskAlgorithm, DestinationAlgorithm \*destAlgorithm, bool hasIO, bool halfDuplexConnectionType);

~GanttChart();

GanttChart(const GanttChart&) = delete;

GanttChart &operator=(const GanttChart&) = delete;

GanttChart(GanttChart&& old) {

\_maxTEnd = old.\_maxTEnd;

\_maxNEnd = old.\_maxNEnd;

\_criticalDataSet = old.\_criticalDataSet;

\_taskAlgorithm = old.\_taskAlgorithm;

\_destAlgorithm = old.\_destAlgorithm;

old.\_criticalDataSet.clear();

old.\_taskAlgorithm = nullptr;

old.\_destAlgorithm = nullptr;

}

const DirectedGraph \*taskGraph() const;

int processorsCount() const;

int sumTaskWeight() const;

double calcAcceleration() const;

double calcEfficiency() const;

private:

void calcCriticalWay(const QMap<Node \*, Task \*> &tasksByNodes);

Task \*calcCriticalWayBegin(Task \*task);

Task \*calcCriticalWayEnd(Task \*task);

QSet<Task\*> \_criticalDataSet;

int \_sumTaskWeight;

int \_maxTEnd,

\_maxNEnd;

const DirectedGraph \*\_taskGraph;

TaskPriorityAlgorithm \*\_taskAlgorithm;

DestinationAlgorithm \*\_destAlgorithm;

};

class TaskPriorityAlgorithm

{

public:

virtual void *performCalculations*(const QMap<Node\*, Task\*> &tasksByNodes, int maxTEnd, int maxNEnd) const = 0;

virtual ~*TaskPriorityAlgorithm*();

protected:

TaskPriorityAlgorithm();

};

class TaskPriorityAlgorithm1 : public TaskPriorityAlgorithm

{

public:

TaskPriorityAlgorithm1();

virtual ~*TaskPriorityAlgorithm1*();

virtual void *performCalculations*(const QMap<Node\*, Task\*> &tasksByNodes, int maxTEnd, int maxNEnd) const;

};

class TaskPriorityAlgorithm4 : public TaskPriorityAlgorithm

{

public:

TaskPriorityAlgorithm4();

virtual ~*TaskPriorityAlgorithm4*();

virtual void *performCalculations*(const QMap<Node\*, Task\*> &tasksByNodes, int maxTEnd, int maxNEnd) const;

};

class TaskPriorityAlgorithm12 : public TaskPriorityAlgorithm

{

public:

TaskPriorityAlgorithm12();

virtual ~*TaskPriorityAlgorithm12*();

virtual void *performCalculations*(const QMap<Node\*, Task\*> &tasksByNodes, int maxTEnd, int maxNEnd) const;

};

class DestinationAlgorithm {

public:

virtual ~*DestinationAlgorithm*();

virtual void *startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes) = 0;

void addReadyTask(Task \*task);

const QVector<Processor\*> &processors();

protected:

DestinationAlgorithm();

void init(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes);

QMultiMap<int, QPair<Processor\*, Task\*>> \_processorsReleaseTime;

QVector<Processor\*> \_processors;

QMap<int, Task\*> \_readyTasks;

};

class DestinationAlgorithm3 : public DestinationAlgorithm {

public:

DestinationAlgorithm3();

virtual ~*DestinationAlgorithm3*();

virtual void *startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes);

};

class DestinationAlgorithm5 : public DestinationAlgorithm {

public:

DestinationAlgorithm5();

virtual ~*DestinationAlgorithm5*();

virtual void *startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes);

};

class DestinationAlgorithm7 : public DestinationAlgorithm{

public:

DestinationAlgorithm7();

virtual ~*DestinationAlgorithm7*();

virtual void *startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes);

};

#endif // GANTTCHART\_H

**gantt\_chart.cpp**

#include "gantt\_chart.h"

#include "logical\_link.h"

#include "physical\_link.h"

#include "processor.h"

#include "time\_span.h"

#include "connection.h"

#include "graph.h"

#include "node.h"

#include "edge.h"

#include <assert.h>

#include <QDebug>

GanttChart::GanttChart(const UndirectedGraph \*systemGraph, const DirectedGraph \*taskGraph,

const QMap<Node\*, int> &linksCount,

TaskPriorityAlgorithm \*taskAlgorithm, DestinationAlgorithm \*destAlgorithm,

bool hasIO, bool halfDuplexConnectionType)

: \_taskGraph(taskGraph), \_taskAlgorithm(taskAlgorithm), \_destAlgorithm(destAlgorithm)

{

// Create Processors

QMap<Node\*, Processor\*> processorsByNodes;

int proc\_key = 1;

for (auto node : systemGraph->nodes())

{

int lCount = linksCount[node];

Processor \*proc;

if (hasIO)

proc = new IOProcessor(node->id(), proc\_key, lCount, node->relations().count(), halfDuplexConnectionType);

else

proc = new SimpleProcessor(node->id(), proc\_key, lCount, node->relations().count(), halfDuplexConnectionType);

processorsByNodes.insert(node, proc);

proc\_key += lCount + 1;

}

// Create LogicalLinks

QMap<Edge\*, LogicalLink\*> logicalLinksByEdges;

for (auto edge : systemGraph->edges())

{

Processor \*from = processorsByNodes[edge->nodeFrom()];

Processor \*to = processorsByNodes[edge->nodeTo()];

LogicalLink \*logLink;

if (halfDuplexConnectionType)

logLink = new HalfDuplexLink(from, to);

else

logLink = new FullDuplexLink(from, to);

logicalLinksByEdges.insert(edge, logLink);

from->addLogicalLink(logLink, to);

to->addLogicalLink(logLink, from);

}

// Calculate DistanceVectors for Processors

for (auto processor : processorsByNodes)

processor->spreadDistanceVector();

// Create Tasks

QMap<Node\*, Task\*> tasksByNodes;

\_sumTaskWeight = 0;

for (auto node : taskGraph->nodes())

{

tasksByNodes.insert(node, new Task(node->id(), node->weight(), destAlgorithm));

\_sumTaskWeight += node->weight();

}

// Create Connections

for (auto edge : taskGraph->edges())

{

Task \*from = tasksByNodes[edge->nodeFrom()];

Task \*to = tasksByNodes[edge->nodeTo()];

to->addPredecessor(from);

assert(edge->weight() > 0);

from->addSuccessor(to, new Connection(edge->weight(), from, to));

}

calcCriticalWay(tasksByNodes);

taskAlgorithm->*performCalculations*(tasksByNodes, \_maxTEnd, \_maxNEnd);

destAlgorithm->*startModeling*(processorsByNodes, tasksByNodes);

for (auto log\_link : logicalLinksByEdges)

delete log\_link;

}

GanttChart::~GanttChart()

{

for (auto set : \_criticalDataSet)

delete set;

if (\_taskAlgorithm != nullptr)

delete \_taskAlgorithm;

if (\_destAlgorithm != nullptr)

delete \_destAlgorithm;

}

const DirectedGraph \*GanttChart::taskGraph() const

{

return \_taskGraph;

}

int GanttChart::processorsCount() const

{

return \_destAlgorithm->processors().count();

}

int GanttChart::sumTaskWeight() const

{

return \_sumTaskWeight;

}

double GanttChart::calcAcceleration() const

{

int maxReleaseTime = 0;

for (auto proc : \_destAlgorithm->processors())

maxReleaseTime = std::max(proc->releaseTime(), maxReleaseTime);

return maxReleaseTime;

}

double GanttChart::calcEfficiency() const

{

return calcAcceleration() / \_destAlgorithm->processors().count();

}

void GanttChart::calcCriticalWay(const QMap<Node\*, Task\*> &tasksByNodes)

{

\_maxTEnd = 0;

\_maxNEnd = 0;

for (auto task : tasksByNodes)

{

if (!\_criticalDataSet.contains(task))

{

calcCriticalWayEnd(task);

}

}

for (auto task : tasksByNodes)

{

if (\_maxTEnd > task->tEnd)

\_maxTEnd = task->tEnd;

if (\_maxNEnd > task->nEnd)

\_maxNEnd = task->nEnd;

calcCriticalWayBegin(task);

}

}

Task \*GanttChart::calcCriticalWayBegin(Task \*task)

{

if (task->nBegin >= 0)

return task;

task->nBegin = 0;

task->tBegin = 0;

for (auto prev\_task : task->predecessors())

{

calcCriticalWayBegin(prev\_task);

const int prevNBegin = prev\_task->nBegin + 1;

const int prevTBegin = prev\_task->tBegin + prev\_task->weight();

if (task->nBegin < prevNBegin)

task->nBegin = prevNBegin;

if (task->tBegin < prevTBegin)

task->tBegin = prevTBegin;

}

return task;

}

Task \*GanttChart::calcCriticalWayEnd(Task \*task)

{

task->nBegin = -1;

task->tBegin = -1;

task->nEnd = 0;

task->tEnd = 0;

\_criticalDataSet.insert(task);

for (auto map\_iter = task->successors().begin(); map\_iter != task->successors().end(); ++map\_iter)

{

auto vec\_it = \_criticalDataSet.find(map\_iter.key());

Task \*next\_task = (vec\_it != \_criticalDataSet.end() ?

\*vec\_it : calcCriticalWayEnd(map\_iter.key()));

if (task->nEnd < next\_task->nEnd)

task->nEnd = next\_task->nEnd;

if (task->tEnd < next\_task->tEnd)

task->tEnd = next\_task->tEnd;

}

task->nEnd += 1;

task->tEnd += task->weight();

return task;

}

TaskPriorityAlgorithm::TaskPriorityAlgorithm() { }

TaskPriorityAlgorithm::~*TaskPriorityAlgorithm*() { }

// <<<< 1

TaskPriorityAlgorithm1::TaskPriorityAlgorithm1() : TaskPriorityAlgorithm() { }

TaskPriorityAlgorithm1::~*TaskPriorityAlgorithm1*() { }

void TaskPriorityAlgorithm1::*performCalculations*(const QMap<Node \*, Task \*> &tasksByNodes,

int maxTEnd, int maxNEnd) const

{

//QMultiMap<double, Task\*> sortedExecutions;

QMultiMap<QPair<double, int>, Task\*> sortedExecutions;

for (auto task : tasksByNodes)

{

//Чем больше key, тем важнее задача. Самые важные taskи в конце

sortedExecutions.insert(qMakePair((static\_cast<double>(task->tEnd) / maxTEnd +

static\_cast<double>(task->nEnd) / maxNEnd), task->id()),

task);

}

int priority = sortedExecutions.count();

for (auto task : sortedExecutions)

//Чем ниже приоритет, тем важнее task

task->setPriority(--priority);

}

// 1 >>>>

// <<<< 4

TaskPriorityAlgorithm4::TaskPriorityAlgorithm4() : TaskPriorityAlgorithm() { }

TaskPriorityAlgorithm4::~*TaskPriorityAlgorithm4*() { }

void TaskPriorityAlgorithm4::*performCalculations*(const QMap<Node \*, Task \*> &tasksByNodes,

int maxTEnd, int maxNEnd) const

{

Q\_UNUSED(maxTEnd);

Q\_UNUSED(maxNEnd);

QMultiMap<QPair<int, int>, Task\*> sortedTasks;

for (auto task : tasksByNodes)

{

//Чем больше key, тем важнее задача. Самые важные taskи в конце

sortedTasks.insert(qMakePair(task->nEnd,

task->predecessors().count() +

task->successors().count()),

task);

}

int priority = sortedTasks.count();

for (auto task : sortedTasks)

//Чем ниже приоритет, тем важнее task

task->setPriority(--priority);

}

// 4 >>>>

// <<<< 12

TaskPriorityAlgorithm12::TaskPriorityAlgorithm12() : TaskPriorityAlgorithm() { }

TaskPriorityAlgorithm12::~*TaskPriorityAlgorithm12*() { }

void TaskPriorityAlgorithm12::*performCalculations*(const QMap<Node \*, Task \*> &tasksByNodes,

int maxTEnd, int maxNEnd) const

{

Q\_UNUSED(maxTEnd);

Q\_UNUSED(maxNEnd);

QMultiMap<int, Task\*> sortedTasks;

for (auto task : tasksByNodes)

//Чем больше key, тем важнее задача. Самые важные taskи в конце

sortedTasks.insert(task->successors().count(), task);

int priority = sortedTasks.count();

for (auto task : sortedTasks)

//Чем ниже приоритет, тем важнее task

task->setPriority(--priority);

}

// 12 >>>>

DestinationAlgorithm::DestinationAlgorithm() { }

DestinationAlgorithm::~*DestinationAlgorithm*() { }

void DestinationAlgorithm::addReadyTask(Task \*task)

{

\_readyTasks.insert(task->priority(), task);

}

void DestinationAlgorithm::init(const QMap<Node \*, Processor \*> &processorsByNodes, const QMap<Node \*, Task \*> &tasksByNodes)

{

QMultiMap<std::tuple<int, int, int>, Processor\*> sortedProcessors;

for (auto processor : processorsByNodes)

sortedProcessors.insert(std::make\_tuple(- processor->logicalList()->count(), processor->calcWeight(), processor->id()), processor);

\_processors.reserve(sortedProcessors.count());

for (auto processor : sortedProcessors)

//Чем ниже приоритет, тем важнее processor

\_processors.push\_back(processor);

for (auto ex : tasksByNodes)

if (ex->predecessors().count() == 0)

\_readyTasks.insert(ex->priority(), ex);

auto proc\_iter = \_processors.begin();

auto exec\_iter = \_readyTasks.begin();

while (proc\_iter != \_processors.end() && !\_readyTasks.empty())

{

(\*proc\_iter)->addTask(exec\_iter.value(), 0);

\_processorsReleaseTime.insert(exec\_iter.value()->toTime(),

qMakePair(\*proc\_iter, exec\_iter.value()));

exec\_iter = \_readyTasks.erase(exec\_iter);

++proc\_iter;

}

}

const QVector<Processor\*> &DestinationAlgorithm::processors()

{

return \_processors;

}

// <<<< 3

DestinationAlgorithm3::DestinationAlgorithm3() { }

DestinationAlgorithm3::~*DestinationAlgorithm3*() { }

void DestinationAlgorithm3::*startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes)

{

init(processorsByNodes, tasksByNodes);

while (!\_processorsReleaseTime.empty())

{

int cur\_time = \_processorsReleaseTime.firstKey();

do

{

QPair<Processor\*, Task\*> pair = \_processorsReleaseTime.take(cur\_time);

pair.second->finish();

} while (!\_processorsReleaseTime.empty() && cur\_time == \_processorsReleaseTime.firstKey());

while (!\_readyTasks.empty() && \_processorsReleaseTime.count() < \_processors.count())

{

Task\* task = \_readyTasks.take(\_readyTasks.firstKey());

//qDebug() << "CurrentTask:" << task->id();

QVector<Connection\*> activeConnections;

for (auto prev\_task : task->predecessors())

activeConnections.push\_back(prev\_task->findConnectionTo(task));

int time;

int bestTime = std::numeric\_limits<int>::max();

Processor \*bestProc = nullptr;

for (auto cur\_proc : \_processors)

{

if (cur\_proc->releaseTime() <= cur\_time)

{

if (activeConnections.empty())

time = cur\_proc->simulateReception(task);

else

time = cur\_proc->simulateReception(activeConnections, task);

if (time < bestTime)

{

bestTime = time;

bestProc = cur\_proc;

}

break;

}

}

assert(bestTime != std::numeric\_limits<int>::max());

if (!activeConnections.empty())

bestProc->performReception(activeConnections, bestTime, task);

bestProc->addTask(task, bestTime);

\_processorsReleaseTime.insert(task->toTime(), qMakePair(bestProc, task));

}

}

}

// 3 >>>>

// <<<< 5

DestinationAlgorithm5::DestinationAlgorithm5() { }

DestinationAlgorithm5::~*DestinationAlgorithm5*() { }

void DestinationAlgorithm5::*startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes)

{

init(processorsByNodes, tasksByNodes);

while (!\_processorsReleaseTime.empty())

{

int cur\_time = \_processorsReleaseTime.firstKey();

do

{

QPair<Processor\*, Task\*> pair = \_processorsReleaseTime.take(cur\_time);

pair.second->finish();

} while (!\_processorsReleaseTime.empty() && cur\_time == \_processorsReleaseTime.firstKey());

while (!\_readyTasks.empty() && \_processorsReleaseTime.count() < \_processors.count())

{

Task\* task = \_readyTasks.take(\_readyTasks.firstKey());

//qDebug() << "CurrentTask:" << task->id();

QVector<Connection\*> activeConnections;

for (auto prev\_task : task->predecessors())

activeConnections.push\_back(prev\_task->findConnectionTo(task));

int time;

int bestTime = std::numeric\_limits<int>::max();

Processor \*bestProc = nullptr;

for (auto cur\_proc : \_processors)

{

if (cur\_proc->releaseTime() <= cur\_time)

{

if (activeConnections.empty())

time = cur\_proc->simulateReception(task);

else

time = cur\_proc->simulateReception(activeConnections, task);

if (time < bestTime)

{

bestTime = time;

bestProc = cur\_proc;

}

}

}

assert(bestTime != std::numeric\_limits<int>::max());

if (!activeConnections.empty())

bestProc->performReception(activeConnections, bestTime, task);

bestProc->addTask(task, bestTime);

\_processorsReleaseTime.insert(task->toTime(), qMakePair(bestProc, task));

}

}

}

// 5 >>>>

// <<<< 7

DestinationAlgorithm7::DestinationAlgorithm7() { }

DestinationAlgorithm7::~*DestinationAlgorithm7*() { }

void DestinationAlgorithm7::*startModeling*(const QMap<Node\*, Processor\*> &processorsByNodes, const QMap<Node\*, Task\*> &tasksByNodes)

{

init(processorsByNodes, tasksByNodes);

while (!\_processorsReleaseTime.empty())

{

int cur\_time = \_processorsReleaseTime.firstKey();

do

{

QPair<Processor\*, Task\*> pair = \_processorsReleaseTime.take(cur\_time);

pair.second->finish();

} while (!\_processorsReleaseTime.empty() && cur\_time == \_processorsReleaseTime.firstKey());

while (!\_readyTasks.empty() && \_processorsReleaseTime.count() < \_processors.count())

{

Task\* task = \_readyTasks.take(\_readyTasks.firstKey());

//qDebug() << "CurrentTask:" << task->id();

QVector<Connection\*> activeConnections;

for (auto prev\_task : task->predecessors())

activeConnections.push\_back(prev\_task->findConnectionTo(task));

int time;

int bestTime = std::numeric\_limits<int>::max();

Processor \*bestProc = nullptr;

for (auto cur\_proc : \_processors)

{

if (activeConnections.empty())

time = cur\_proc->simulateReception(task);

else

time = cur\_proc->simulateReception(activeConnections, task);

if (time < bestTime)

{

bestTime = time;

bestProc = cur\_proc;

}

}

assert(bestTime != std::numeric\_limits<int>::max());

if (!activeConnections.empty())

bestProc->performReception(activeConnections, bestTime, task);

bestProc->addTask(task, bestTime);

\_processorsReleaseTime.insert(task->toTime(), qMakePair(bestProc, task));

}

}

}

// 7 >>>>