НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.272

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Модель потокової системи для виконання багатомісних операцій»

Виконав:
студент VI курсу, групи КМ-61м
Капура Артем Сергійович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н.,
Жабіна В.В. _____

Рецензент:
Доцент кафедри ОТ, к.т.н., доц.,
Верба О.А. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

ЗМІСТ

СПИСОК ТЕРМІНІВ

Примечание [АК1]: сделать

Система реального часу – це система, яка повинна реагувати на події у зовнішньому по відношенню до системи середовищі або впливати на середовище в межах необхідних тимчасових обмежень.

Потокові обчислювальні системи – системи, що використовують механізм управління обчисленнями, при якому команди виконуються, коли стають доступними їх операнди.

МПД – машина потоку даних.

Дрібнозернистий паралелізм – паралелізм на рівні операцій і мікрооперацій.

.

ВСТУП

Обчислювальні системи проникли в усі сфери людського життя від космічних програм до приготування їжі. Задачі обчислювальних систем постійно ускладнюються та потребують більшої потужності. За довгий період часу обчислювальні системи були достатньо стабільними, їх зміни носили еволюційний характер, а покращення не призводили до корінних змін в архітектурі та методах програмування, постійно зростала потужність обчислювальних модулів, за декілька десятків років їх потужність зросла в сотні разів. Однак в наш час просте нарощування потужностей стає дедалі складніше, тому проводиться пошук інших шляхів підвищення швидкодії обчислювальних систем.

В існуючих високонавантажених мультипроцесорних системах існують проблеми організації зв’язку між асинхронно працюючими частинами, що використовують загальні ресурси. На організацію спільної роботи окремих частин витрачається занадто багато ресурсів, тому актуальним стає пошук нових підходів, які використовують більш прості способи роботи.

Одним з таких підходів є застосування обчислювальних систем, що реалізують потокові моделі обчислень. Дані системи дозволяють ефективно реалізовувати наявний в алгоритмах паралелізм. Тому дослідження таких систем та пошук шляхів підвищення їх швидкодії є досить актуальною задачею.

Для дослідження потокових систем є зручним використання імітаційних моделей, адже таким чином можна зручно та без значних витрат моделювати поведінку заданих алгоритмів та обирати найбільш ефективні параметри та способи їх рішення.

Виходячи з цього є доцільною розробка такої імітаційної моделі. Для цього необхідно розробити архітектуру програмного забезпечення, описати вимоги та обрати засоби для розробки.

Створивши дану імітаційну модуль, та застосувавши її при проведені досліджень можна знайти найбільш ефективні шляхи для рішення конкретних задач та запропонувати підходи, що зможуть покращити швидкодію потокових систем.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ПОТОКОВИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

1.1. Класифікація машин потоку даних

Можна виділити два підходи до побудови МПД: машини пошукового типу і машини з безпосередніми зв'язками. В машинах пошукового типу кожен обчислювальний пристрій, закінчивши обробку чергової команди, відправляє результат виконання в пристрій управління, яке поміщає його в осередок за наявним у пакета-результату адресою і вибирає для обчислювального пристрою нову готову до виконання команду [1]. У машинах з безпосередніми зв'язками відбувається реконфігурація з’єднань між обчислювальними пристроями за допомогою комутаторів таким чином, що виходи одних обчислювальних пристроїв з'єднуються зі входами інших обчислювальних пристроїв відповідно до структури інформаційних зв'язків алгоритму.

Відомо поділ потокових машин на машини, керовані рухом даних (data driven), і машини, керовані рухом запитів (demand driven) [2]. Машини, керовані рухом даних, розділяються на машини, керовані потоком самих даних (data-flow), і машини, керовані потоком сигналів, які вказують на готовність даних. Як машин, керованих рухом запитів, розглядаються машини з попередньою обробкою. За рівнем активації відомі машини з активацією на рівні команд, процедур, модулів програм.

У потокових машинах використовуються команди, що зчитуються за один такт операції звернення до пам'яті (горизонтальні), і горизонтальні пакети обміну, що передаються паралельним кодом. Типовим прикладом машини з горизонтальною системою команд є Манчестерська машина [3, 4], що використовує 96-розрядні пакети фіксованої довжини. Для зменшення кількості ліній зв'язку використовуються вертикальні команди, які зчитуються з пам'яті по-байтно.

При роботі МПД можливі ситуації, коли значення одного з операндів, необхідних для виконання операції, надходять в більш швидкому темпі, ніж значення іншого операнда, тоді можливе некоректне обчислення значення або організація черги значень одного з операндів в комутаційної мережі, яка може привести до блокування обчислень. Відомий ряд способів запобігання подібних ситуацій:

- Апаратна перевірка готовності команд-приймачів результатів прийняти результат обчислення попередньої команди;
- Використання механізму неявних підтверджень, який полягає у видачі підтверджують сигналів кожною командою для всіх команд-джерел;
- Використання механізму явних підтверджень, що полягає у видачі сигналів підтверджень тільки в окремі команди;
- Копіювання програм завдання, що дозволяє незалежно обробляти кілька наборів даних;
- Заборона повторного запуску програми завдання до закінчення попереднього завдання;
- Організація для кожної команди нескінченних вхідних черг для операндів;
- Використання розфарбованих міток, що містять додаткову інформацію про номер завдання і номері ітерації. Ця додаткова інформація дозволяє виконати команду, якщо все вхідні операції мають однакову додаткову інформацію "колір", "мітка".

Використання механізму неявних підтверджень гарантує коректність виконання програм, дозволяє виконувати програми в конвеєрному режимі, використовувати команди, в яких операнди можуть надходити від двох або більше джерел, але при цьому завантажує МПД великою кількістю додаткових пакетів і збільшує формат пакета. При використанні явних підтверджень з'являється можливість мінімізації кількості сигналів

підтверджень, але при цьому не завжди гарантується коректність виконання програм і зменшується глибина конвеєра. Копіювання програм завдання вимагає великих витрат пам'яті. Використання для кожної команди нескінченних вхідних операндів вимагає значних витрат апаратури для організації буферів. При використанні розфарбованих міток зростає обсяг інформації, що міститься в пакетах обміну.

1.2. Принципи роботи потокових обчислювальних систем

Примечание [АК2]: 1.1.Опис роботи потокових систем 1.2.

Потокові обчислювальні системи - системи, що використовують механізм управління обчисленнями, при якому команди виконуються, коли стають доступними їх операнди. У таких системах процес обчислення описується так званим графом потоків даних (dataflow graph). Він має вершини, що містять інформацію про операції, та дуг, які відображають потоки обміну операндами між потрібними вершинами [5].

Розвиток паралельних обчислень був викликаний через обмеженість максимальної швидкодії стандартних послідовних обчислювальних машин, а також постійною наявністю задач для яких їх потужностей замало [6]. Взагалі паралельне програмування з’явилось в 1960 роках через появу апаратних модулів, каналів, що взаємодіють з процесором через переривання. Завдяки ним можна виконувати різні послідовності інструкцій паралельно [7]. Ідеологія обчислень, керованих потоком даних (потокової обробки), була розроблена в кінці 60-х роках. На початку 70-х років Денніс, а пізніше і інші почали розробляти комп'ютерні архітектури, засновані на обчислювальної моделі з керуванням від потоку даних на противагу програмному управлінню (Control Flow). Графічну модель керованих даними обчислень запропонував в дисертаційній роботі співробітник Стенфордського університету Дуайн Адамс [8].

Основним в потокових обчислення є те що операція виконується за умови готовності всіх необхідних для виконання цієї операції операндів. Операнд вважається «готовим», якщо відповідним цьому операнду

осередків пам'яті присвоєно значення (обчислено раніше або константи-присвоєно).

У потокової архітектурі відсутнє поняття «послідовність інструкцій» і немає лічильника команд. Програма в потокової системі - це не набір команд, а обчислювальний граф. Кожен вузол графа представляє собою оператор або набір операторів, а гілки відображають залежності вузлів за даними. Черговий вузол починає виконується як тільки доступні всі його вхідні дані. У цьому полягає один з основних принципів архітектури: виконання інструкцій по готовності даних [9].

Потоковий граф, який визначає програму обчислень, зберігається в пам'яті системи у вигляді таблиці. Кожен запис в таблиці представляє одну вершину потокового графа. Порядок розміщення записів в пам'яті є несуттєвим, оскільки момент активації вершини визначається лише наявністю даних на всіх її входах.

Одним з основних переваг потокової архітектури є її масштабованість. Пристрої можуть об'єднуватися найпростішим комутатором. Весь діапазон номерів вузлів просто розподіляється рівномірно між пристроями. Ніяких додаткових заходів для синхронізації обчислювального процесу, на відміну від багатопроцесорної Control Flow архітектури, не потрібно [10].

Примечание [WU3]: Окей вставвка гергель 10

У звичайній машині фон-неймановской архітектури команда підлягає виконанню, коли на неї вказує лічильник команд машини. Порядок настання такої події, як правило, визначається безпосередньо програмістом. У той час, як програма зі стрижневою логікою - це послідовний список команд, потокова програма може бути представлена у вигляді графа, вершинами якого є команди (актори), які взаємодіють з іншими вершинами через дуги (рис. 1.1). Команда підлягає виконанню, коли на її входи надійдуть всі операнди. У прийнятому нами графовому поданні програми це означає, що перш, ніж даний актор буде виконаний, на всіх його вхідних дугах повинні з'явитися значення необхідних даних.

Процес виконання полягає в тому, що спочатку приймаються вхідні дані, потім дані що в них знаходяться обробляються згідно з кодом операції конкретного актора, а потім на вихідні дуги видаються результати операції.

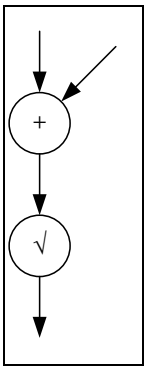


Рис. 1.1 - Простий граф потоку даних

Отже, можна стверджувати, що спосіб організації обчислень в потокових системах характеризується наступними двома фундаментальними принципами:

- асинхронність операцій. Рішення про виконання команди приймається тільки на основі локального критерію. Кожна команда може "локально" виявити наявність своїх операндів. Це властивість дуже приваблива в умовах розподіленої обчислювального середовища, де не повинно бути центрального пристрою управління, що виконує глобальне планування операцій;
- функціональність операцій. Дія кожної операції обмежується тільки формуванням результатів, які будуть використані заданим числом інших акторів. Цим забезпечується відсутність "побічних ефектів". Такого роду ефекти в звичайній системі можуть бути "довгостроковими" тобто виконання команди може вплинути на

стан тієї чи іншої комірки пам'яті, яка буде використана лише значно пізніше іншою командою, яка не пов'язана з першою.

Якщо виконується декілька ітерацій, неминуче доводиться порушувати основний принцип потокової обробки (одноразове присвоєння значень змінним). Справді, при багаторазовому повторенні деякого актора, як це відбувається при ітераціях, за його вхідним дуг повинно надходити кілька різних даних (що відповідають різним ітераціям). Для забезпечення контрольованого порушення цих правил без шкоди для правильного виконання програми було запропоновано декілька рішень. Серед них найбільш детально розглядалися схема квітирування і U-інтерпретатор.

- Схема квітирування. Належне узгодження ознак можна забезпечити шляхом упорядкування їх вироблення. Це можна зробити ретельною побудовою графа програми так, щоб порядок проходження даних, породжуваних на двох різних ітераціях, не міг бути порушений. Крім того, необхідно гарантувати, що ні на якій дузі дані не будуть накопичуватися. Виконати цю умову можна, якщо включати актора тільки тоді, коли дані присутні на всіх його вхідних дугах, і в той же час немає жодного операнду на вихідних дугах. Для цього вершини-наступники, отримавши операнд від вершини-попередника, повинні видати йому квитанцію (рис. 1.2). Отже, актор може бути виконаний, якщо він отримав всі вхідні аргументи і всі квитанції. Паралелізм, який може забезпечити ця схема, зводиться в основному до конвеєризації обчислень, виконуваних в межах однієї ітерації, шляхом їх розподілу між різними акторами. Таким чином, якщо число команд в "тілі ітерації" збігається з числом наявних процесорів, досягається максимальний виграш в продуктивності, що забезпечується таким механізмом роботи. Однак для коротких ітерацій (в порівнянні з числом процесорів в машині)

реалізований паралелізм виявляється нижче потенційно можливого. Тому може виникнути необхідність передбачити в компіляторі розширення коду для векторних операцій. Однак для коротких ітерацій (в порівнянні з числом процесорів в машині) реалізований паралелізм виявляється нижче потенційно можливого. Тому може виникнути необхідність передбачити в компіляторі розширення коду для векторних операцій. Однак для коротких ітерацій (в порівнянні з числом процесорів в машині) реалізований паралелізм виявляється нижче потенційно можливого. Тому може виникнути необхідність передбачити в компіляторі розширення коду для векторних операцій.

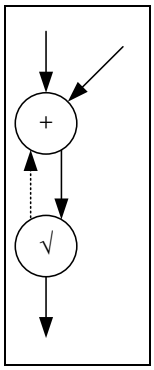


Рис. 1.2 - Схема квітирування

Важлива особливість розглянутої статичної моделі виконання програми полягає в тому, що в ній допускається існування тільки одного примірника кожної команди в будь-який заданий момент часу. Іншими словами, розпаралелювання обчислень в ітераціях засноване в такій схемі насамперед на конвеєризації. При цьому вихідна схема квітирування не допускає реалізації декількох одночасних звернень до однієї і тієї ж функції. Було розроблено кілька машин, побудованих на цьому принципі організації обчислень: архітектура Массачусетського технологічного

інституту, потокова машина фірми Hughes, потоковий процесор сигналів (DFSP), машина TX16 Південно-Каліфорнійського університету та ін.

- Розгортуючий інтерпретатор (U-інтерпретатор). U-інтерпретатор забезпечує максимально можливу асинхронність роботи. Для того щоб домогтися правильного виконання акторів в ітеративній конструкції, операнди забезпечуються додатковою інформацією, що вказує на контекст їх породження. Виконання актора дозволяється тільки в тому випадку, якщо на вході можна знайти пару операндів з співпадаючими мітками. Мітка містить номер ітерації. Основні принципи роботи U-інтепретатора зводяться до наступного [11]. Кожен операнд порції даних забезпечується міткою, що має вигляд u.p.s.i, де p – це ім’я процедури того актора, якому направляються дані, s - адреса цього актора в процедурі p, поле i - номер ітерації, на якій був породжений даний операнд, а поле u - контекст його створення. Поле i використовується для розрізнення операндів, які направляються різним ітераціями одного і того ж актора, а поле u - в ситуаціях, коли наявно декілька звернень до однієї і тієї ж функції, при рекурсивних викликах функції та при вкладених ітераціях.

Для аналізу полів контексту і номера ітерації, що містяться в мітках ярликів, вводяться спеціальні актори. На відміну від схеми квітирування ця динамічна схема потокової обробки забезпечує повністю асинхронне виконання процесів, що відображаються графом програми. Справді, завдяки наявності системи команд можуть одночасно існувати кілька примірників однієї команди. Для паралельного виконання векторних операцій немає необхідності створювати з допомогою компілятора кілька копій графа програми. Точно так же допускаються багаторазові звернення до однієї і тієї ж функції і, зокрема, рекурсивні обчислення, оскільки кожне нове звернення до актору маркується своєю влучною. Це означає, що в тих

випадках, коли необхідно забезпечити швидкі рекурсивні звернення до функцій, U-інтерпретатор слід віддати перевагу статичної моделі. Однак така гнучкість досягається ціною ускладнення апаратури. Було розроблено кілька машин, побудованих за цими принципами: потокова машина з міченими ярликами MTI, машина DDSP фірми ESL, машина Манчестерського університету, Sigma-1 фірми ETL і ін.

Крім описаних вище механізмів організації обчислень нижнього рівня був розроблений і ряд спеціальних потокових мов високого рівня, призначених для полегшення перетворення програми в потоковий граф. Звичайно, без цих мов можна обійтися: так, наприклад, в проекті фірми Texas Instruments використовується програмування на Фортрані із застосуванням модифікованої версії векторного компілятора, розробленого спочатку для ЕОМ ASC. Проте для прототипів машин, керованих потоком даних, запропоновано і багато мов високого рівня. Тут слід зазначити мови VAL (Value Algorithmic Languages) для статичної потокової машини MTI, Id (Irvine Dataflow) для потокової архітектури з міченими ярликами MTI, LUCID і ін. В роботі [12] запропонований мову SISAL (Streams and Iterations in Single Assignment Language).

Розроблено також потокові мови, призначені спеціально для програмування прикладних задач обробки сигналів. До них, зокрема, відноситься мова SIGNAL [13], призначення якої - дати засіб формального опису завдань обробки сигналів і полегшити програмування таких завдань для спеціалізованих і універсальних мультипроцесорних ЕОМ. Одна з основних її особливостей пов'язана з тим, що в ній передбачено поняття часу, за допомогою якого описується взаємодія різних завдань обробки. Таким чином, дана мова є синхронною на відміну від таких асинхронних мов, як CSP або Occam [14]. Ще один засіб формального опису алгоритмів обробки сигналів, заснованої на принципах управління потоком даних - це мова SDF (Synchronous Data Flow) [15].

1.3. Аналіз переваг і недоліків потокових обчислювальних систем

Звичайні методи, засновані на використанні явно вираженого паралелізму, в ряді випадків можуть забезпечити високу продуктивність, вони вимагають від програміста великих зусиль, спрямованих на виявлення і опис паралелізму, заданої прикладної задачі. У той же час функціональний підхід до програмування дозволяє виявляти прихований паралелізм на етапі виконання програми. В цьому випадку кількість зусиль на розробку програми значно знижуються. Таким чином, одним з головних переваг архітектури, керованої потоком даних, є її програмованість, яка в свою чергу призводить до підвищення продуктивності при заданому обсязі витрат на програмування. Крім цього, даний підхід має хорошу масштабованість, що дозволяє вибирати конфігурацію мультипроцесорної системи відповідно до обсягу розв'язуваних прикладних завдань. З іншого боку, планування операцій на етапі виконання програми тягне за собою непродуктивні витрати обчислювальних ресурсів при виконанні операцій регулярного характеру, що знижує досяжну продуктивність.

Отже можна виділити наступні переваги, властиві потоковим системам:

- досягнення продуктивності спеціалізованих систем при збереженні можливостей універсальних систем за рахунок організації обчислень відповідно до структури алгоритму розв'язуваної задачі;
- можливість одночасного виконання декількох завдань в машинах пошукового типу, що значно збільшує продуктивність;
- спрощення конструювання та верифікації паралельних програм;
- можливість широкого застосування функціональних мов програмування, використання яких дозволяє підвищити продуктивність праці програмістів;

- модульність і гнучкість схем зв'язку дозволяють ефективно нарощувати обчислювальні модулі для збільшення продуктивності;
- підвищена надійність, що дозволяє відключати несправні блоки з продовженням обчислювального процесу без зміни алгоритму;
- орієнтація на роботу в реальному масштабі часу, так як використання потокового принципу управління забезпечує швидку реакцію на появу нових даних.

До основних же недоліків потокових систем можна віднести:

- велика довжина команд;
- великий обсяг пересилань - мається на увазі те, що при використанні одних і тих же даних безліччю команд існує необхідність в розмноженні цих даних для кожної команди, що тягне за собою введення нових операндів і знижує продуктивність;
- неефективність функціонування на послідовних ділянках.

1.4. Існуючі рішення для проектування потокових обчислювальних систем

Xilinx Inc. - американський розробник і виробник інтегральних мікросхем програмованої логіки (ПЛІС, FPGA) [16].

Заснована в Силіконовій долині в 1984 році, штаб-квартира компанії розташована в Сан Хосе, США, з додатковими офісами в Лонгмонт, США; Дублін, Ірландія; Сінгапур; Гайдерабад, Індія; Пекін, Китай; Шанхай, Китай; Брісбен, Австралія та Токіо, Японія [17].

Основними продуктами сімейства FPGA є серії Zynq, Virtex (висока продуктивність), Kintex (середній діапазон) і Artix (недорогі), а також Spartan (недорогі) серії. [18] Основними програмами для розробки є Xilinx ISE та Vivado Design Suite [19].

Xilinx розробляє та реалізує програмовані логічні продукти, включаючи інтегральні схеми (ICs), засоби розробки програмного забезпечення, попередньо визначені системні функції, що постачаються як інтелектуальна власність (IP), сервіси дизайну, навчання клієнтів, розробку на місцях та технічну підтримку.

ПЛІС Xilinx були використані для експерименту ALICE (Великий іонний коллайдер експерименту) в європейській лабораторії CERN на кордоні Франції та Швейцарії, щоб відобразити і роз'єднати траєкторії тисяч субатомних частинок [20].

Vivado Design Suite являє собою набір програм, розроблений компанією Xilinx для синтезу та аналізу конструкцій HDL, що замінює Xilinx ISE з додатковими функціями для розробки систем на кристалі [21,22]. Vivado являє собою переосмислення всього дизайнерського досвіду (у порівнянні з ISE), критики описали його як "добре продуманий, легко інтегрований, швидкий та інтуїтивно зрозумілий" продукт [23,24].

Vivado дозволяє розробникам синтезувати свої проекти, виконувати аналіз використаного часу, вивчати діаграми RTL, моделювати дизайну в залежності від параметрів та налаштовувати цільовий пристрій. Vivado є дизайнерським середовищем для продуктів FPGA від Xilinx і тісно пов'язане з архітектурою даних чіпів і не може використовуватися з продуктами від інших постачальників.

Vivado включає в себе інструменти проектування електронного рівня системи (ESL) для синтезу та перевірки алгоритмічного IP на базі C; системної інтеграції всіх типів системних блоків; перевірки блоків і систем [25]. Безкоштовна версія WebPACK Edition Vivado надає дизайнерам обмежену версію середовища розробки [26].

Вигляд середовища Vivado зображено на рис. 1.3.

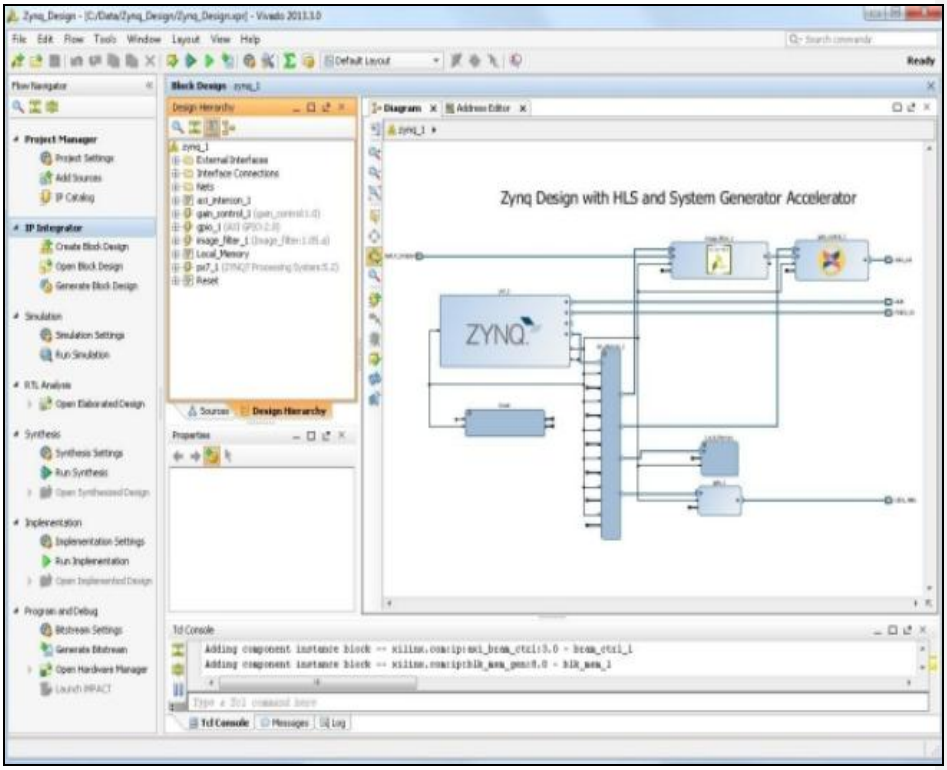


Рис. 1.3. Середовища Vivado

Корпорація Altera - американський виробник програмованих логічних пристроїв (PLDs), реконфігурованих цифрових схем. Альтера випустила свій перший PLD в 1984 році. Основні продукти компанії Altera - це FPGA серії Stratix, Arria та Cyclone, CPLDs серії MAX, [27] програмне забезпечення Quartus II [28]. Intel придбала Altera 1 червня 2015 року.

Altera Quartus II - програмне забезпечення для розробки логічних пристроїв, розроблене компанією Altera, до того як Altera була придбана компанією Intel, а інструмент був перейменований в Intel Quartus Prime. Quartus II дозволяє аналізувати та синтезувати проекти HDL, що дає змогу розробнику втілити свій дизайн, виконати заміри продуктивності, вивчити діаграми RTL, імітувати поведінки при різних параметрах системи та налаштовувати цільовий пристрій. Quartus включає в себе реалізацію VHDL та Verilog для опису апаратного забезпечення, візуального

редагування логічних схем та моделювання векторних сигналів [29]. Вигляд середовища Altera Quartus II зображено на рис. 1.4.

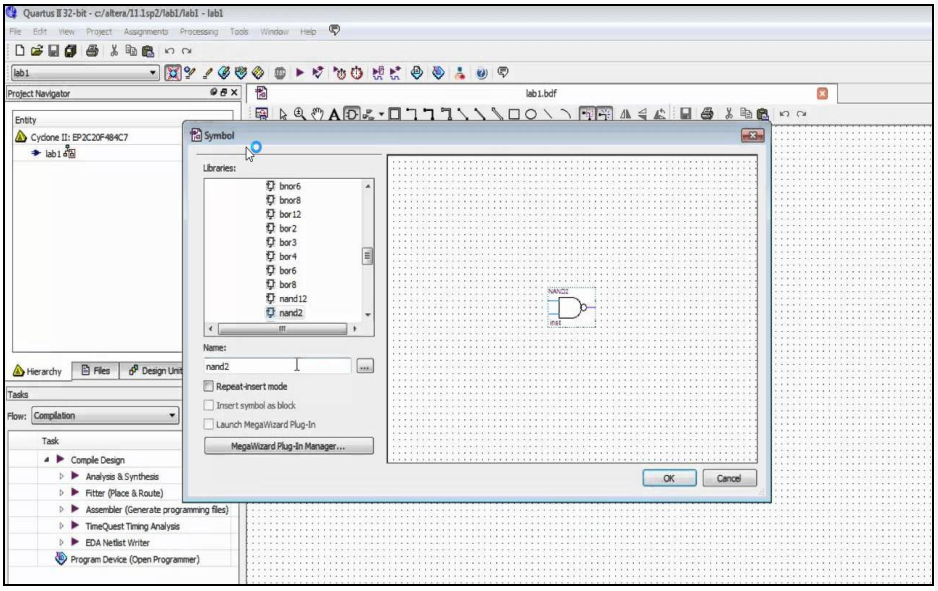


Рис. 1.4. Середовища Vivado

Продукти згаданих вище компаній є потужними універсальними та сучасними інструментами. Проте вони потребують коштів та часу для придбання та опанування, а це не завжди є виправданим особливо при розв’язанні вузько спеціалізованих задач.

1.5. Висновок до розділу 1

В даному розділі був проведений аналіз потокових систем, наведена класифікація машин потоку даних, розглянуті основні принципи виконання обчислень в таких системах. Розглянуто проблеми, що виникають при виконанні обчислень та способи їх вирішення. Наведено мови високого рівня що застосовуються в даній сфері. Це такі мови як VAL, Фортран, SISAL, SIGNAL, Occam та інші.

Було наведено опис переваг потокових обчислювальних систем, до яких відносяться: спрощення конструювання та верифікація програм, підвищення продуктивності праці програміста, можливість широкого

застосування функціональних мов програмування, модульність, гнучкість, підвищена надійність, орієнтація на роботу в реальному масштабі часу.

Також були розглянуті продукти від компаній Xilinx та Altera. Дані компанії мають потужні програмні комплекси такі як Vivado Design Suite та Altera Quartus.

2. АРХІТЕКТУРА ПОТОКОВОЇ СИСТЕМИ ДЛЯ ВИКОНАННЯ БАГАТОМІСНИХ ОПЕРАЦІЙ

2.1. Концепція виконання обчислень в потокових системах

В наш час найбільш розповсюдженим підходом до паралельного програмування є спосіб статичного розпаралелювання процесів [30]. І у такому випадку на етапі розробки програм іноді не вдається виявити прихований паралелізм, через те що повна інформацією про динаміку процесів відсутня. Також, існують певні складності, пов'язані з формуванням і розподілом завдань між обчислювальними модулями системи в режимі ручного керування, синхронізацією паралельних процесів і кількістю пересилань даних.

Для усунення цих недоліків використовують засоби динамічного розпаралелювання обчислень. Одним з підходів для реалізації динамічного розпаралелювання обчислень є використання моделі обчислень, що керуються потоком даних. Розподіл операцій між обчислювальними модулями в такому випадку реалізовується автоматично в процесі обчислень [31].

До перших таких систем, що були реалізовані, відносяться манчестерський проект, який отримав розвиток у багатьох роботах, наприклад [34-36]. Для потокових систем відсутнє поняття програми, як зв’язаного списку команд, які необхідно виконувати у порядку, що задає лічильник команд. Для них характерні наступні особливості:

- спочатку складається список акторів та даних, які можуть приходити в систему в будь-якому порядку;
- для складання списку акторів та даних користуються потоковим графом, поданим в ярусно-паралельній формі (ЯПФ).
- вершинам графа відповідають актори, а дугам – дані;

- граф будується без врахування числа обчислювальних модулів що наявні в системі;
- команди активізуються даними. При наявності всіх необхідних операндів і актора формується команда. В деяких системах команда формується з використанням асоціативної пам'яті, в якій здійснюється пошук акторів і даних за ознаками [31]. У цьому випадку умова готовності i-ї команди визначається як $\lambda_i = \&_{j=1}^{n+1} c_i^j$, де c_i^j – ознаки наявності в асоціативній пам’яті актора и всіх n операндів для i-ї команди;
- коли команда сформована її починає виконувати один з обчислювальних модулів. При чому спочатку йому необхідно завантажити значення операндів в свою пам’ять згідно адрес, що наявні в команді, а потім виконати обчислення над ними;
- після виконання обчислень результат записується в пам’ять.

2.2. Прискорення виконання обчислень

В потокових системах час виконання обчислень залежить в тому числі і від кількості даних, що пересилаються, адже нам потрібно не лише витратити час на сам процес обчислення в ОМ, а й спочатку надати в обчислювальний модуль операнди, над якими обчислення будуть виконуватись. Це займає досить велику кількість часу, особливо при реалізації дрібнозернистих алгоритмів, адже кількість пересилань там значно вище [32].

Кількість пересилань можна зменшити шляхом використання операцій, що можуть виконувати обчислення не лише над двома операндами, а над довільною їх кількістю. Розглянемо простий приклад, що ілюструє алгоритм для додавання трьох чисел з використанням багатомісних та двомісних операцій. Графи для даних алгоритмів показано на рис. 2.1.

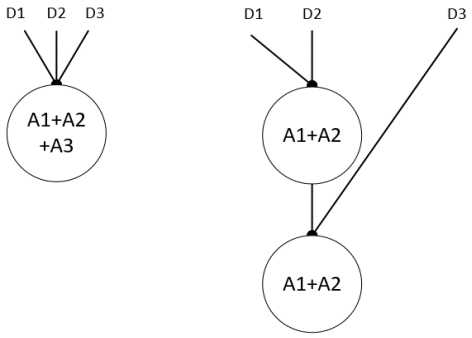


Рис. 2.1. Графи додавання трьох чисел

Тобто навіть для такого простого обчислення вдалось отримати виграш, адже при використанні багатомісної операції додавання ми позбавляємось необхідності пересилати результат додавання перших двох чисел для додавання третього.

Наведемо більш складний алгоритм. А саме знаходження коренів квадратного рівняння. На рис. 2.2 наведено граф з використанням двомісних операцій. А на рис.2.3 відповідно граф з використанням багатомісних операцій. Для графу з багатомісними операціями порядок входження операнду відповідає порядку літери в латинському алфавіті, якою він позначається всередині вершини. Як бачимо кількість пересилань для графа з двомісними функціями 23, а для графу з багатомісними 17, тобто вдалося кількість пересилань на 26%.

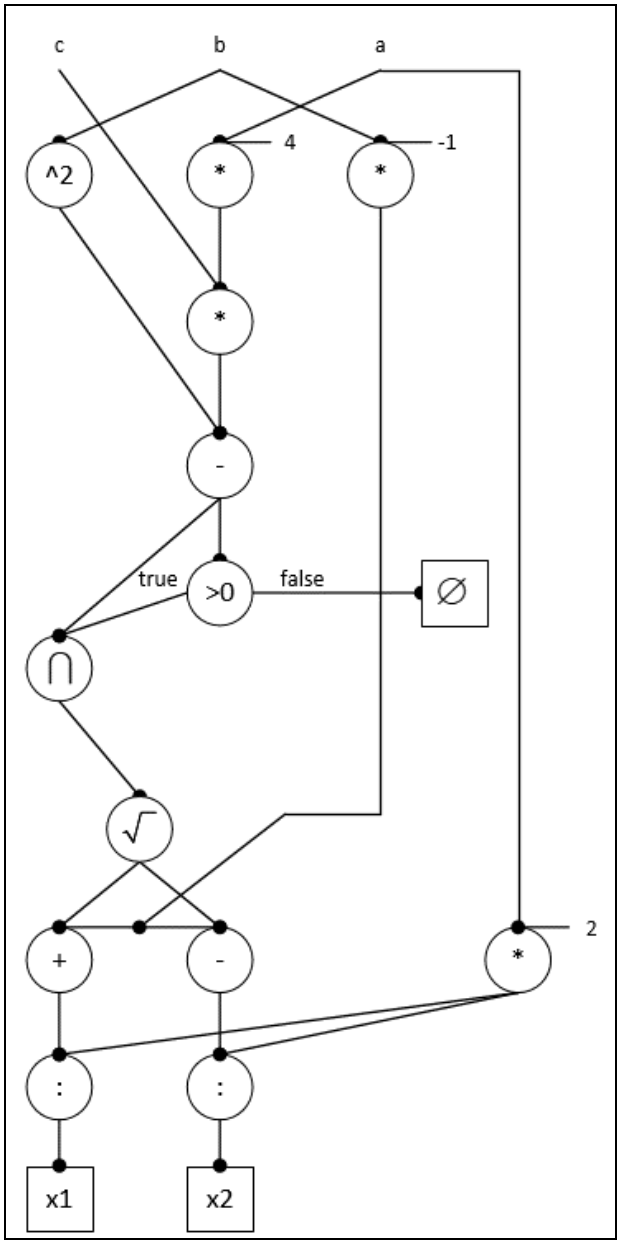


Рис. 2.2. Граф знаходження коренів квадратного рівняння

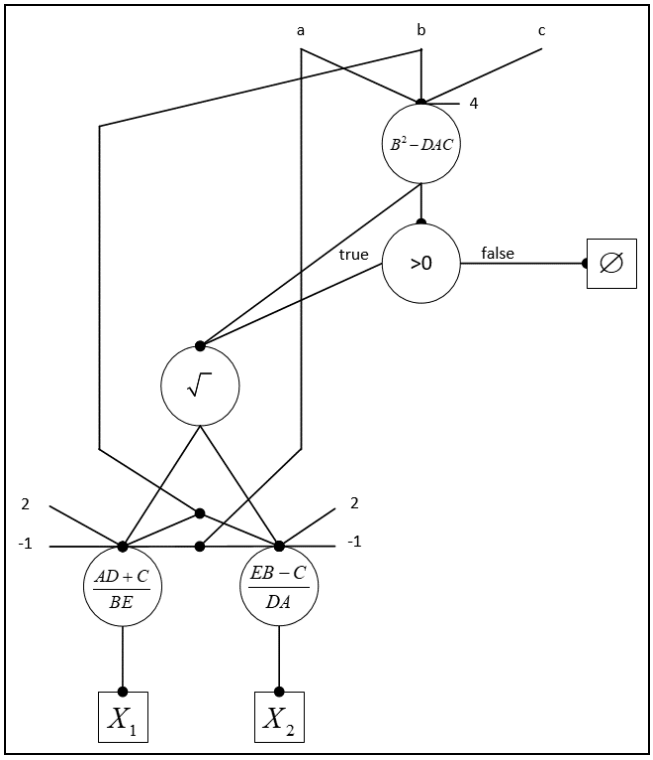


Рис. 2.3. Граф знаходження коренів квадратного рівняння з використанням багатомісних операцій

Концепція пропонованої потокової моделі обчислень базується на наведених нижче принципах.

Обчислювальна система (рис. 2.4) містить апаратні обчислювальні модулі (ОМ), в якості яких можуть використовуватися пристрої з мікропрограмним керуванням, які не вимагають оперативної пам'яті для зберігання програми (для кожної команди зашита відповідна мікропрограма в пам'яті мікропрограм ОМ) [33]. Кожен ОМ виконує конкретну кінцеву систему команд. Команди можуть мати один або декілька операндів. Наприклад, для операції $F = AB$ необхідно два операнди, а для $F = AB/C - D$ потрібно чотири.

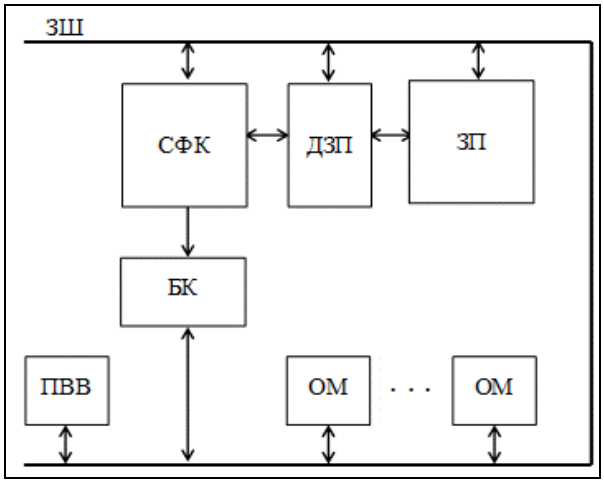


Рис. 2.4. Обчислювальна система

Принципи роботи даної системи полягають в наступному:

1. Середовище формування команд (СФК) формує команди при наявності необхідних даних і відсилає їх в буфер команд (БК), де вони накопичуються.
2. Якщо присутній вільний модуль в системі, він виконує одну з команд, що знаходиться в буфері, при цьому команда з буферу видаляється.
3. За необхідними операндами обчислювальні модулі звертаються до загальної пам'яті (ЗП).
4. Виконавши обчислення результат записується до ЗП.
5. Обмін інформацією в рамках системи між її компонентами здійснюється через загальну шину (ЗШ).
6. Формуванням адрес для звернення до ЗП займається диспетчер запам’ятовуючого пристрою (ДЗП).
7. Задача пристроїв введення-виведення даних (ПВВ) полягає в можливості обміну інформацією з зовнішнім середовищем.
8. Процес розв’язання задачі представляється у вигляді потокового графа, кожній i-й вершині якого ($i = 1,\ldots,m$) відповідає операція,

а кожній дузі - операнд. Операції, що відповідають вершинам графа, взаємодіють між собою тільки через дані.

Операція, i-ї вершини графа описується наступним актором (дескриптором)

$$W_i = \langle N_i, I_i, F_i, T_i, Q_i \rangle,$$

- N_i – ім’я вершини графа (усі імена унікальні);
- I_i – код операції, різні вершини можуть виконувати однакові операції;
- F_i – множина імен входів i-ї вершини графа ;
- T_i – множина імен виходів для i-ї вершини графа;
- Q_i – сумарне число входів для i-ї вершини графа;

Операнд, що відповідає дузі графа між j-ю та i-ю вершинами, визначається наступним набором даних:

$$D_{ji} = \langle N_i, d_{ji}, A_{ji} \rangle,$$

- N_i – ім’я вершини графа (усі імена унікальні);
- d_{ji} – ім’я (номер) входу в i-у вершину;
- A_{ji} – елемент адресації операнду.

При наявності відповідних операндів та актора формується команда у наступному вигляді:

$$Z_i = \langle I_i, T_i, A_i \rangle,$$

- I_i – код операції, різні вершини можуть виконувати однакові операції;
- T_i – множина імен виходів для i-ї вершини графа;
- A_i – множина елементів адресації усіх даних для i-ї команди.

2.3. Моделювання роботи імітаційної системи

Для імітації роботи пропонованої системи був розроблений набір програмних засобів, що дозволяють вводити дані, виконувати моделювання ходу обчислень і надавати інформацію для аналізу ходу обчислення.

Імітаційна ситема моделює роботу кожного елемента архітектури (рис. 2.4). У СФК кількість акторів та операндів, що відносяться до однієї команди, підраховуються на відповідному лічильнику і порівнюються з максимально можливою їх кількістю, а саме $Q+1$. При наявності необхідним операндів та актора, команда записується в буфер, який є пам'яттю типу FIFO. Кожен обчислювальний модуль отримавши команду, завантажує операнди з запам’ятовуючого пристроюй та виконує необхідну операцію. Результат обчислень в стандартному вигляді по загальній шині пересилається в СФК.

2.4. Висновок до розділу 2

В даному розділі була запропонована архітектура потокової системи, що має покращену швидкодію за рахунок виконання обчислень з використанням операцій зі змінною кількістю операндів, що дозволяє скоротити кількість звернень до пам’яті, що в свою чергу веде до значного зменшення часу що відводиться на очікування завантаження операндів до обчислювальних модулів. Особливостями даної архітектури є те, що вона здатна використовувати багатомісні операції, пристосована до відповідних форматів акторів, операндів та команд, а сформовані команди зберігаються в буфері та чекають на виконання.

Примечание [АК4]: выделение

Імітаційна модель, дозволяє отримати результати моделювання для різних завдань, визначати параметри системи, які забезпечують ефективну реалізацію цільової функції. Може бути визначено час вирішення завдання при різній кількості ОМ та різному наборі команд, що дає можливість дослідити та обрати найбільш ефективну конфігурацію системи в кожному

конкретному випадку.

Примечание [WU5]: Написать норм

3. РОЗРОБКА ІМІТАЦІЙНОЇ СИСТЕМИ

3.1. Вимоги до системи

3.1.1. Вимоги користувача

При розробці даної системи передбачалося, що кінцевий користувач має такі вимоги до розроблюваної системи:

- зберігання графів в окремих файлах та можливість їх завантаження при початку роботи з системою
- розділення інформації на різні частини такі як граф, порядок операндів тощо
- задання кількості обчислювальних модулів перед початком роботи
- моніторингу завантаженості обчислювальних модулів
- моніторингу стану СФК, списку операндів, буфера
- покрокового виконання програми
- виконання програми одразу до кінця
- відображення кінцевого результату
- відображення кількості тактів, за які виконались обчислення
- відображення інформації про завантаженість обчислювальних модулів та СФК по завершенні програми

3.1.2. Функціональні вимоги

1. Система повинна імітувати обчислення потокової системи з використанням багатомісних операцій, використовуючи задані компоненти та дані.
   1.1. Система повинна імітувати певний формат операндів, даних та акторів.
   1.2. Система повинна імітувати елементи потокової системи такі як: середовище формування команд, буфер команд,

запам’ятовуючий пристрій, диспетчер запам’ятовуючого пристрою, обчислювальні модулі.

1.3. Система повинна імітувати послідовність роботи потокової системи.

1.4. Система повинна бути здатна використовувати багатомісні та двомісні операції.

1.5. Система має показувати результат обчислень у відповідному полі.

2. Система повинна надавати можливість вводити необхідні дані за допомогою текстових файлів.

2.1. Користувач повинен мати можливість вибрати необхідні файли на початку роботи за допомогою стандартного провідника операційної системи.

2.2. Матриця графа повинна зчитуватися з файлу типу txt.

2.3. Матриця графа задається в наступному вигляді: номери рядків та стовпців відповідають номерам вершин графа, число в матриці вказує на те що вершина, що відповідає номеру стовпця пересилає дані вершині, номер якої відповідає номеру стовпцю, саме число вказує на номер операції.

2.4. Дані для перших вершин повинні зчитуватися з файлу типу json.

2.5. Дані для перших вершин складаються зі значення операнду та списку вершин в які це значення посилається. Також вказується порядок входження операнду до вершини та код операції.

2.6. Інформація про порядок операндів повинна зчитуватися з файлу типу json.

2.7. Інформація про порядок операндів містить в собі таку інформацію: номер вершини з якою пересилаються дані,

номер вершини в яку пересилаються дані, порядок входження операнду в вершину.

3. Система повинна надавати можливість змінювати кількість обчислювальних модулів.
   3.1. Кількість обчислювальних модулів повинна змінюватися на початку роботи програми за допомогою 2 кнопок, що відповідно дозволяють збільшити чи зменшити кількість обчислювальних модулів.
   3.2. Задана кількість обчислювальних модулів повинна показуватись біля кнопок зміни їх кількості у вигляді числа.
   3.3. Мінімальна кількість модулів повинна бути обмежена 1 а максимальна 50.
4. Система повинна відображати інформацію про елементи імітаційної системи в процесі виконання.
   4.1. Оновлення інформації відбувається за кожним тіком.
   4.2. Система повинна показувати інформацію про стан середовища формування команд:
      - список команд що мають бути сформовані;
      - скільки операндів необхідно для початку формування команди;
      - скільки операндів вже є у наявності;
      - номери вершин входів для команди;
      - номери вершин виходів для команди;
      - номер вершини якій відповідає команда.
   4.3. Система повинна показувати інформація про стан буферу, а саме інформацію про команди які очікують виконання. Ця інформація має містити адресу першого операнду та кількість яку необхідно зчитати, номер вершини, що відповідає команді.

4.4. Система повинна показувати інформацію про обчислювальні модулі. Ця інформація має містити:
   - номер обчислювального модуля;
   - номер вершини, обчислення якої виконує обчислювальний модуль;
   - стан, тобто чи вільний модуль, чи виконую обчислення, чи очікує на завантаження операндів;
   - кожний обчислювальний модуль має бути замальований в колір стану. Якщо модуль вільний – червоний, якщо завантажений – зелений, якщо очікує на дані – жовтий;
   - скільки тактів займає даний процес обчислення;
   - скільки тактів вже було виконано.

4.5. Система повинна відображати дані що зберігаються в запам’ятовуючому пристрої, а саме значення та адресу комірки в пам’яті.

5. Система повинна надавати користувачу змогу виконати обчислення такт за тактом чи одразу до кінця.
6. Система повинна показувати зв’язки між елементами системи.
7. Система повинна показувати кількість тактів, що виконала система в процесі обчислень.
8. Після завершення обчислень повинна бути відображена наступна інформація в окремому вікні:
   - загальна кількість тактів;
   - звернень до пам’яті;
   - результат обчислень.

3.1.3. Опис можливостей системи

Для того щоб більш докладно розповісти про можливості що надає система були використані діаграма прецедентів та діаграми послідовності.

Діаграма прецедентів - це граф, що складається з множини акторів, прецедентів (варіантів використання), відношеннями між акторами та прецедентами [37].

Діаграма послідовності має на меті показати взаємодію об'єктів впродовж деякого часу. Діаграма відображає об'єкти, що беруть участь та послідовність обміну повідомленнями між ними [38].

Система є вузькоспеціалізованою тому актор лише один – «Користувач».

Розглянемо діаграму прецедентів наведену на рис. 3.1. Користувач може виконувати наступні дії:

- задавати вхідні дані для роботи програми, що в свою чергу включає в себе такі прецеденти:
    - задання файлу, що містить в собі інформацію про порядок операндів;
    - задання файлу, що містить в собі інформацію про початкові дані;
    - задання файлу, що містить в собі матрицю графа;
    - задання кількості обчислювальних модулів;
- керувати процесом виконання програми, що в свою чергу може включати в себе такі прецеденти:
    - виконати програму одразу до кінця;
    - виконувати програму покроково;
- слідкувати за ходом виконання програми.

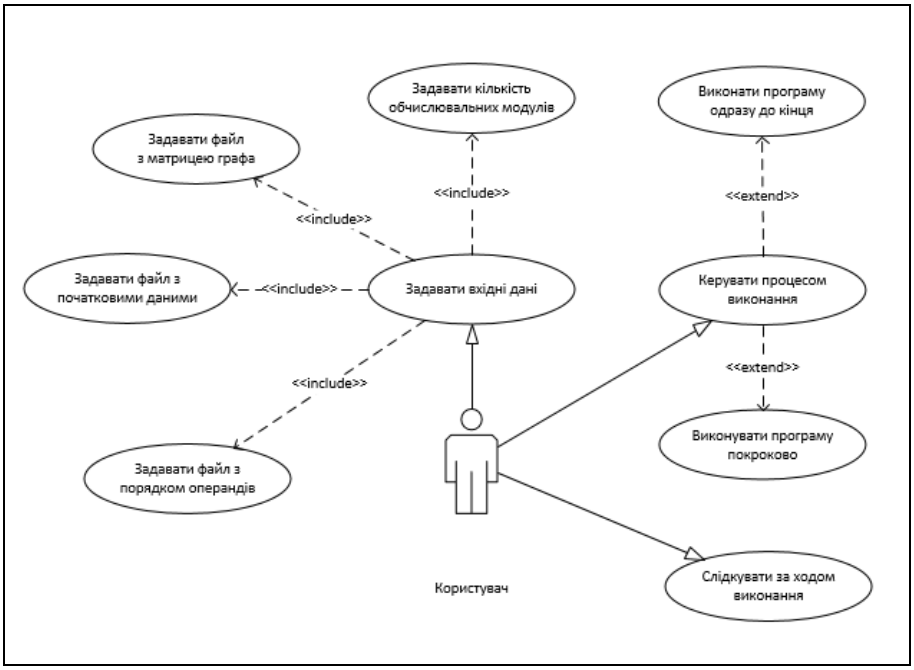


Рис. 3.1. Діаграма прецедентів

Далі наведено дві діаграми послідовності, що докладніше розповідають про такі процеси як:

- ініціалізація системи;
- керування ходом виконання програми.

Діаграма послідовності ініціалізації системи показана на рис. 3.2.

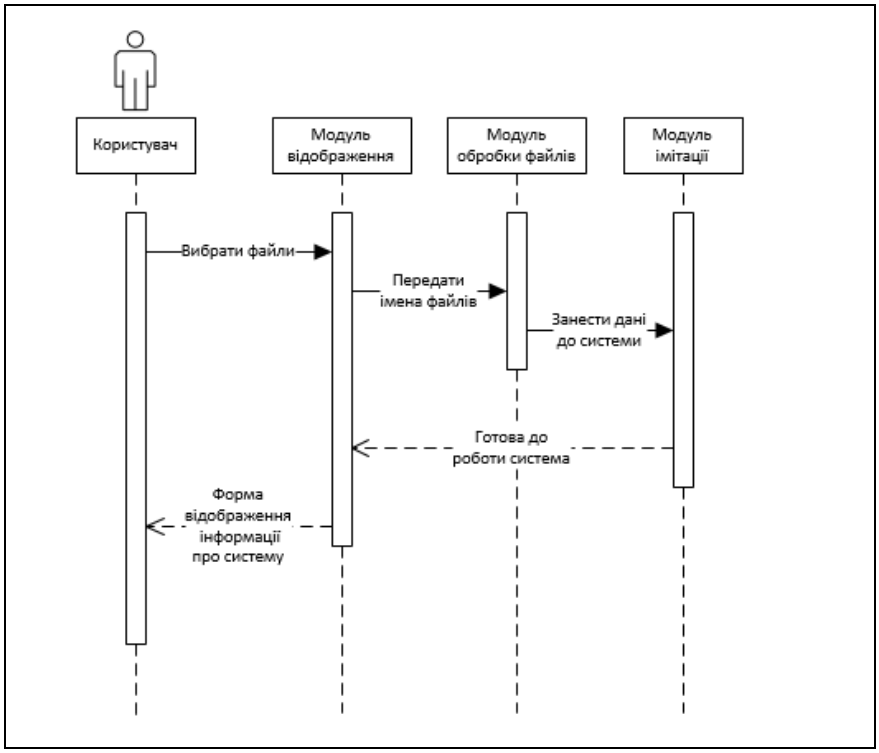


Рис. 3.2. Діаграма послідовності ініціалізації системи

Дана діаграма показує процес того як відбувається процес ініціалізації системи. Для того аби це зробити користувачу необхідно вибрати файли звідки будуть братися дані. Виконує він це в модулі відображення. Далі модуль відображення пересилає цю інформацію, а саме шляхи до потрібних файлів в модуль обробки файлів. модуль обробки файлів виконує потрібні дії та пересилає інформацію далі до модуля, що безпосередньо імітує роботи потокової системи. В модулі імітації інформація оброблюється та сигналізує модулю відображення про те що система пройшла ініціалізацію та готова до роботи. Далі модуль відображення повертає користувачу форму відображення інформації про систему.

Діаграма послідовності керування ходом виконання програми показана на рис. 3.3.

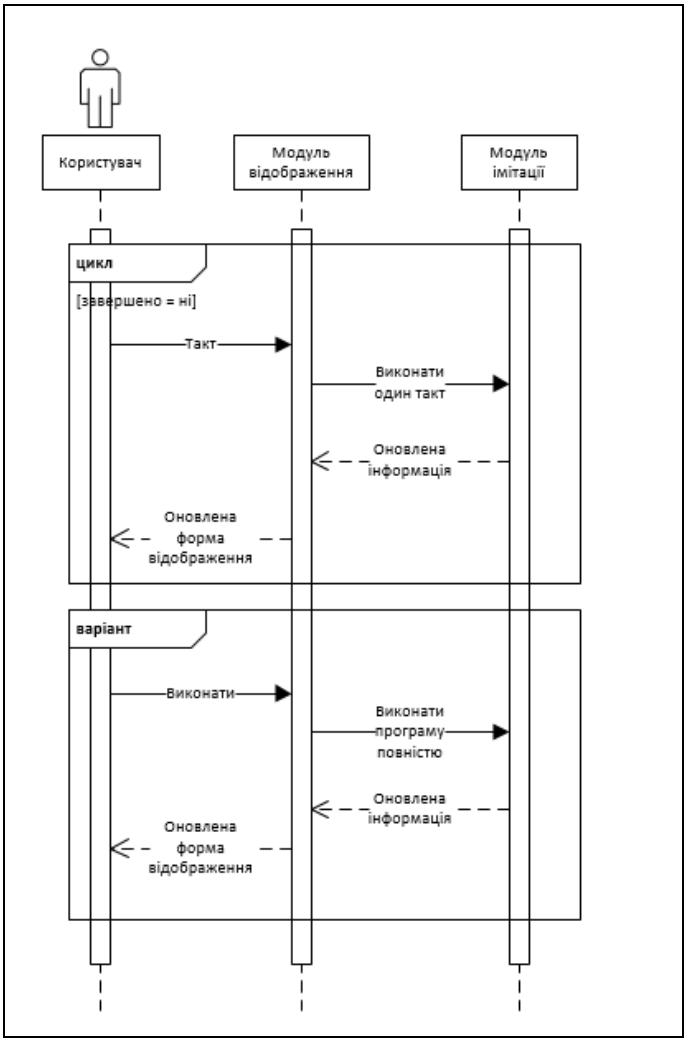


Рис. 3.3. Діаграма послідовності керування ходом виконання програми

Дана діаграма показує процес того як відбувається процес керування ходом виконання програми. В даному процесі участь приймають користувач модуль відображення та модуль імітації роботи потокової системи. Як видно з діаграми ми маємо один цикл та один варіант. Цикл є доступним доки флаг сигналізуючий про завершення є хибою. В даному циклі користувач вказує на те, шо необхідно зробити один такт модулю відображення, той у свою чергу віддає наказ модулю імітації виконати обчислення, що відповідають одному такту. Після завершення обчислень

модуль відображення отримує оновлену інформації про стан системи та виводить її на форму відображення, де вона стає доступно користувачу. Як варіант користувач може вказати на те, що необхідно виконати програму до кінця. Тоді модуль імітації просто виконає необхідну кількість тактів, поверне результат модулю відображення, а той виведе інформацію користувачу.

3.2. Архітектура системи

Архітектура системи наведена на рис. 3.4. В архітектурі імітаційної системи можна виокремити наступні компоненти:

1. Імітатор потокової системи. Це компонент який моделює процес обчислення потокової системи для багатомісних операцій. Він імітує роботу середовища формування команд, буферу, обчислювальних модулів, містить у собі набір акторів, операндів та команд необхідних для роботи. В середині нього проходить процес обчислень, цей компонент є ядром системи. Він використовує бібліотеку функцій та елементів, звідки отримує інформацію про те які саме операції необхідно виконувати над операндами, інтерпретатор графа, який надає інформацію про взаємозв’язок між вершинами, обробник файлів, що надає дані для роботи. Виконуючи обчислення він надає інформацію інтерфейсу користувача.
2. Інтерфейс користувача. Це компонент яким користується користувач для завантаження фалів, керуванням ходом обчислень та перегляду стану системи. Він складається з декількох частин. Інформація про стан системи надходить з імітатора потокової системи, а сам компонент лише оновлює відображення через кожний такт. Також цей компонент надає інформацію обробнику файлів, які саме файли необхідно завантажити.

3. Обробник файлів. Метою цього компоненту є отримання інформації з файлів та представлення її у вигляді об’єктів. Саме за допомогою цього компоненту імітатор отримує необхідну інформацію для того щоб розпочати роботу.
4. Інтерпретатор графа. Цей компонент аналізуючи матрицю графа отримує інформацію про зв’язки між його вершинами, а саме яка вершина в яку пересилає дані та про номер операції, що буде виконана.
5. Бібліотека функцій та елементів. В цьому компоненті містяться багатомісні та двомісні функції, за допомогою яких відбуваються обчислення та структури даних для актора, операнду та команди.

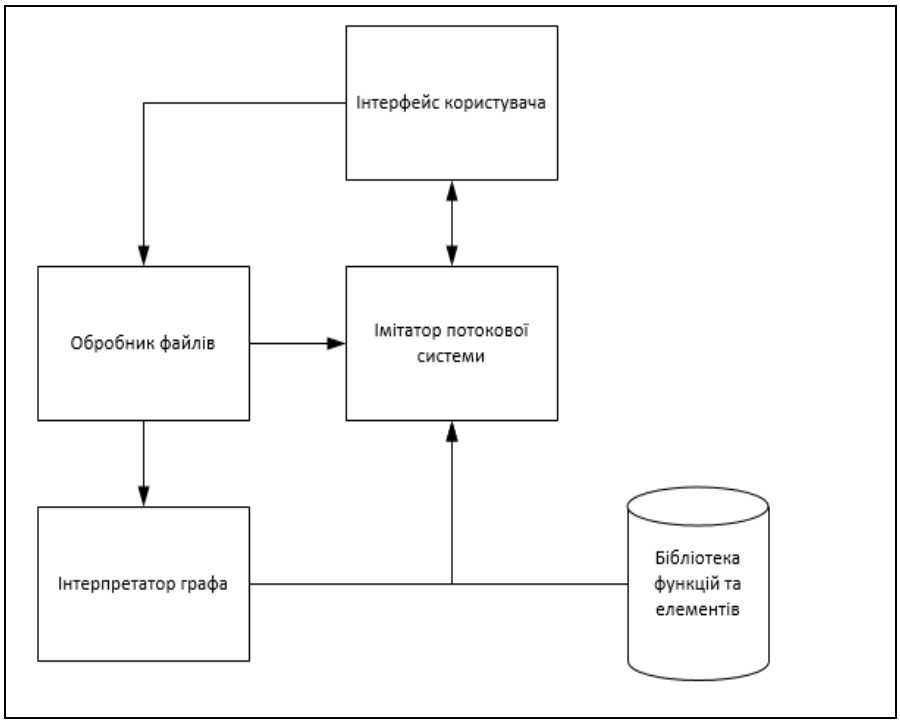


Рис. 3.4. Архітектура системи

3.3. Класи імітаційної системи

Клас *CommandBuffer* показаний на рис. 3.5. Він містить чергу команд які надходять з СФК та за допомогою методів *Enqueue* та *Dequeue* керує нею.

```
public class CommandBuffer
{
    public Queue<Command> commands = new Queue<Command>();
}
```

Рис. 3.5. Клас *CommandBuffer*

Клас *ComputingModule*, що імітує роботу обчислювального модуля наведено на рис. 3.6. Він має такі поля:

- *CMState state* – стан обчислювального модуля;
- *double result* – поле що містить у собі результат обчислень;
- *int ticks* – кількість тіків яку оброблюється операція;
- *int loading* – загальна кількість тіків яку обчислювальний модуль перебував в роботі;
- *Command executingCommand* – поточна виконувана команда;
- *void GetCommandIfFree()* – функція що забирає з буферу команду;
- *int DoTick()* – функція що виконую один тік.

```
public class ComputingModule
{
    public CMState state;
    public double result = 0;
    public int ticks = 0;
    public int loading = 0;

    public Command executingCommand = null;
    2 references | artyomGH, 24 days ago | 1 author, 1 change
    public void GetCommandIfFree(CommandBuffer buffer)...
    2 references | 0 changes | 0 authors, 0 changes
    public int DoTick(List<Operand> operands)...
}
```

Рис. 3.6. Клас *ComputingModule*

Клас *SFK*, що імітує роботу середовища формування команд наведено на рис. 3.7. Він має такі поля:

- *List<Actor> actors* – актори на основі яких формуються команди;
- *List<Operand> availableOperands* – містить у собі доступні операнди яку необхідні для того щоб формувати команди;
- *int loading* – загальна кількість тіків яку СФК перебувало в роботі;
- *void SendToBuffer()* – функція яка відправляє сформовану команду до буфера.

```
public class SFK
{
    public List<Actor> actors;
    public List<Operand> availableOperands;
    public int loading =0 ;
    1 reference | artyomGH, 24 days ago | 1 author, 2 changes
    public SFK()...
    2 references | artyomGH, 24 days ago | 1 author, 1 change
    public void SendToBuffer(CommandBuffer buffer)...
}
```

Рис. 3.7. Клас *SFK*

Клас *StorageDevice*, що імітує роботу запам’ятовуючого пристрою наведено на рис. 3.8. Він має такі поля:

- *Dictionary<int, double> values* – словник шо зберігає пару ключ-значення. Ключем є адреса комірки в пам’яті а значенням – саме значення операнду;
- *void ReInit()* – функція що очищує пам’ять;
- *static StorageDevice Instance()* – властивість, що реалізовує паттерн одинак.

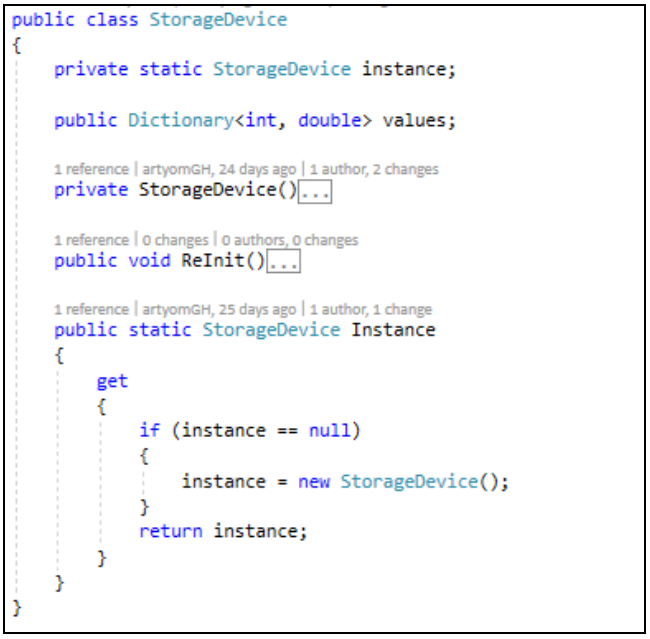


Рис. 3.8. Клас *StorageDevice*

Клас *StorageDeviceDispatcher*, що імітує роботу диспетчера запам’ятовуючого пристрою наведено на рис. 9. Він має такі поля:

- *static StorageDevice storageDevice* – поле, що надає доступ до запам’ятовуючого пристрою;
- *static void Initialise()* – функція що ініціалізує та очищує пам’ять;
- *static void SetOperand ()* – функція, що заносить в пам'ять операнд;
- *static double GetOperand()* – функція, що отримує з пам’яті значення за адресою.

```
public static class StorageDeviceDispatcher
{
    static StorageDevice storageDevice;

    1 reference | 0 changes | 0 authors, 0 changes
    public static void Initialise()...

    2 references | artyomGH, 24 days ago | 1 author, 1 change
    public static void SetOperand(int key, double value)...

    1 reference | artyomGH, 24 days ago | 1 author, 1 change
    public static double GetOperand(int key)...

}
```

Рис. 3.9. Клас *StorageDeviceDispatcher*

Далі наводяться ключові елементи які описують актора, команду та операнд.

Клас актора зображено на рис. 3. 10. Він описується наступними полями:

- *int N* – поле, що вказує на номер вершини графа;
- *int I* – поле, що вказує на номер операції. Під операцією розуміється наприклад піднесення до квадрату;
- *List<int> F* – список входів для даної вершини;
- *public int Q* – кількість входів;
- *public List<int> T* – список виходів для даної вершини.

```
public class Actor
{
    1 reference | artyomGH, 25 days ago | 1 author, 1 change
    public Actor()...
    public int N;//vertex code
    public int I;//operation name(sin cos)
    public List<int> F;//inputs
    public int Q;//sum of inputs
    public List<int> T;//outputs
}
```

Рис. 3.10. Клас *Actor*

Клас команди зображено на рис. 3.11. Він описується наступними полями:

- *int I* – поле, що вказує на номер операції;
- *List<int> T*– список виходів;
- *List<int> A* – список елементів адресації.

```
public class Command
{
    public int I;//operation name
    public List<int> T;//outputs
    public List<int> A;//addresses for the needed elements
}
```

Рис. 3.11. Клас *Command*

Клас операнда зображено на рис. 3.12. Він описується наступними полями:

- *int d* – поле, що вказує на порядок входження операнду до вершини;
- *int A* – елемент адресації операнду;
- *int N* – поле, що вказує на номер операції.

```
public class Operand
{
    public int d;//name(number) of input
    public int A;//element of addresation(to get value)
    public int N;//operation code
}
```

Рис. 3.12. Клас *Operand*

Клас *DataFlowSystem* зберігає в собі всі компоненти імітаційної системи та керує процесом обчислень. Його вигляд зображено на рис. 3.13. Він описується наступними полями:

- *int lastVertexNumber* – поле, що зберігає номер останньої вершини;
- *SFK sfk* – поле, що зберігає екземпляр класу середовища виконання команд;
- *List<ComputingModule> computingModules* – поле, що зберігає список обчислювальних модулів;
- *double? result* – полу що зберігає результат обчислень;
- *void Initialise()* – функція, що ініціалізує систему. Тут відбувається зчитування даних з файлів та занесення їх у відповідні елементи, ініціалізації функцій, створення обчислювальних модулів;

- *void Run()* – функція що виконує обчислення одразу до кінця;
- *bool NextStep()* – функція, що виконує один такт.

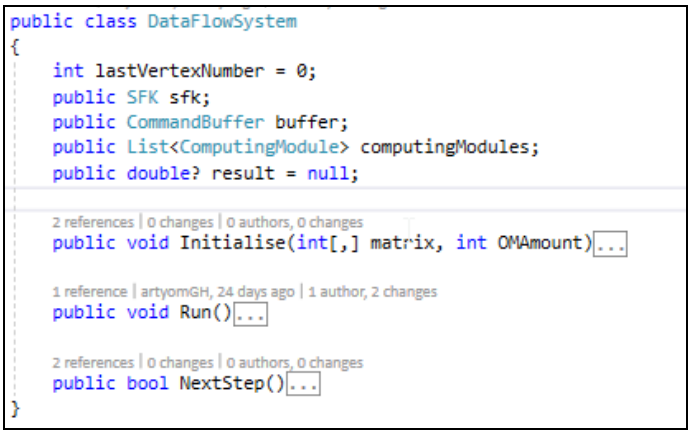


Рис. 3.13. Клас *DataFlowSystem*

Клас *Util* містить у собі різні допоміжні функції та є статичним. Він зображений на рис. 3.14. Містить у собі наступні поля:

- *static List<DataDto> GetDataForStartActors()* – функціія що зчитує інформацію про перші операнди з json файлу;
- *static void GetOperandsOrder()* – функція що зчитує інформацію про порядок операндів з json файлу;
- *static List<Actor> InitialiseActors()* – функція що повертає список акторів з двовимірного масиву, який містить матрицю графа;
- *static Color choseOMStateColor(CMState state)* – функція, що повертає колір обчислювального модуля в залежності від його стану;
- *static List<Operand> InitialiseOperands()* – функція ініціалізує та повертає список операндів. Використовує функцію *GetDataForStartActors()*;
- *static int[,] ReadMatrixFromFile()* – функція що зчитує матрицю графа з txt файлу;

- *static int GenerateRandom()* – функція що генерує число для імітації адреси в запам’ятовуючому пристрої.

```
public static class  Util
{
    2 references | 0 changes | 0 authors, 0 changes
    public static  List<DataDto> GetDataForStartActors()...
    1 reference | artyomGH, 24 days ago | 1 author, 1 change
    public static void GetOperandsOrder()...
    1 reference | artyomGH, 25 days ago | 1 author, 1 change
    public static List<Actor> InitialiseActors(int[,] matrix)...
    1 reference | 0 changes | 0 authors, 0 changes
    public static Color choseOMStateColor(CMState state)...
    1 reference | 0 changes | 0 authors, 0 changes
    public static List<Operand> InitialiseOperands()...
    2 references | 0 changes | 0 authors, 0 changes
    public static int[,] ReadMatrixFromFile()...
    2 references | artyomGH, 25 days ago | 1 author, 1 change
    public static int GenerateRandom()...
}
```

Рис. 3.14. Клас *Util*

Далі наведено так звані DTO класи. Вони слугують для гнучкого способу обміну даними між процесами [39]. В нашому випадку вони використовуються для зручного зчитування даних з json файлів. Всі вони представлені на рис. 3.15.

```
//for json reading
0 references | artyomGH, 25 days ago | 1 author, 1 change
public DataDto()
{
    actors = new List<ActorDto>();
}

public int value;
public int actorsAmount;
public List<ActorDto> actors;
```

```
2 references | artyomGH, 25 days ago | 1 author, 1 change
public class ActorDto {
    public int n;
    public int op;
    public int d;

}
```

```
public class OrderDto
{
    public int from;
    public int to;
    public int d;
}
```

Рис. 3.15. DTO класи

Клас DataDto використовує клас ActorDto, та слугує для зчитування необхідної інформації для перших вершин графу, а саме значень операндів, порядок їх входження, номер операції. Клас OrderDto слугує для зчитування інформації про порядок входження операндів до вершин.

Якщо розглядати процес роботи програми то вона працює наступним чином. Коли користувач натискає кнопку наступного такту викликається функція, що наведена на рис. 3.16.

```
private void button1_Click(object sender, EventArgs e)
{
    if (!isFinished)
    {
        isFinished = dataFlowSystem.NextStep();
        RefreshOnTick();
        if (isFinished == true)
        {
            End();
        }
    }
}
```

Рис. 3.16. Функція обробки команди наступного такту

Всередині функціє перевіряється чи завершено обчислення, якщо так то викликається метод *End()*, якщо ні, то віддається команда імітуючій системі виконати наступний такт, після цього викликаються функція *RefreshOnTick()*, що оновлює інформацію на формі. Функція *RefreshOnTick()* оновлює інформацію про кожний модуль за допомогою декількох функцій, одна з них, що оновлює інформацію про обчислювальний модуль показана на рис. 3.17.

```
private void ReShowOMs()
{
    listView4.Items.Clear();
    foreach (var comm in dataFlowSystem.computingModules)
    {
        var backColor = Util.choseOMStateColor(comm.state);

        if (comm.executingCommand != null)
        {
            var l1 = new ListViewItem(comm.executingCommand._N.ToString());
            l1.BackColor = backColor;
            l1.SubItems.Add(comm.state.ToString());
            l1.SubItems.Add(comm.ticks.ToString());
            l1.SubItems.Add(comm.neededTime.ToString());
            listView4.Items.Add(l1);
        }
        else
        {
            var l1 = new ListViewItem(" ");
            l1.BackColor = backColor;
            l1.SubItems.Add(comm.state.ToString());
            listView4.Items.Add(l1);
        }
    }
}
```

Рис. 3.17. Функція оновлення інформації про обчислювальний модуль

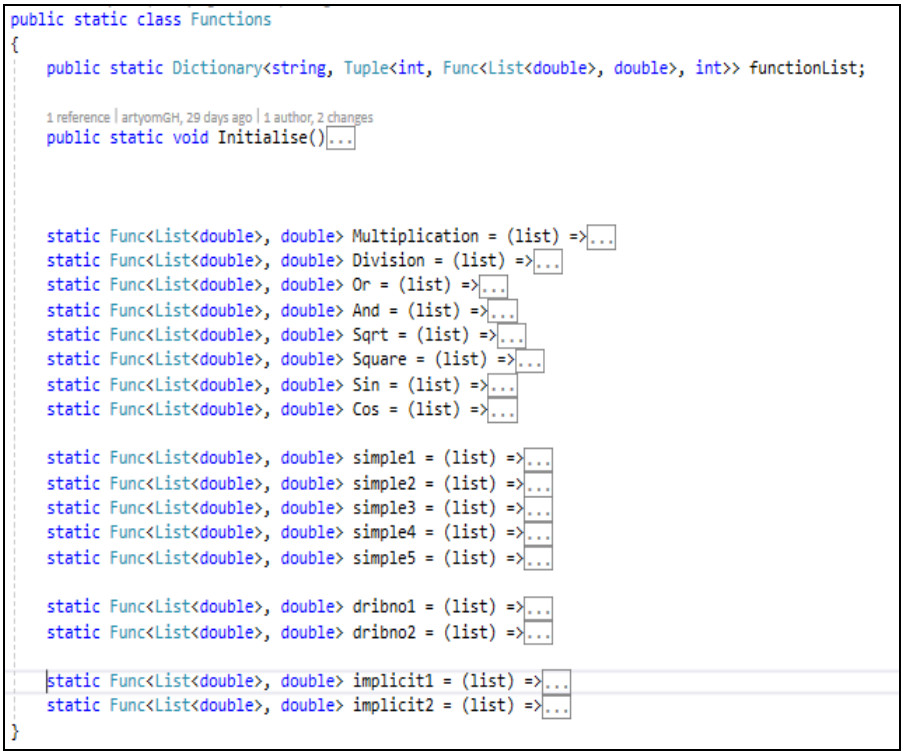


Рис. 3.18. Клас *Functions*

Клас *Functions*, що представлений на рис. 3.18 містить у собі функції які виконують обчислення. Цей клас є статичним. Як видно з рисунку тут описані як загальні функції, такі як додавання множення, взяття кореня тощо, так і спеціальні функція для конкретного графу.

Формат задання операції показано на рис. 3.19. Задаються такі параметри:

- ідентифікатор за яким можна викликати функцію;
- кількість вхідних операндів;
- тип вхідних та вихідного значення;
- час виконання даної операції.

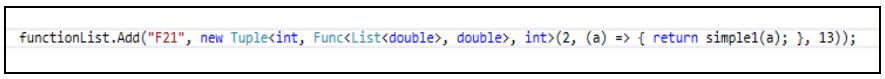


Рис. 3.19. Формат задання операції

3.4. Інтерфейс користувача

3.4.1. Опис форми задання параметрів

Форма наведена на рис. 3.20.

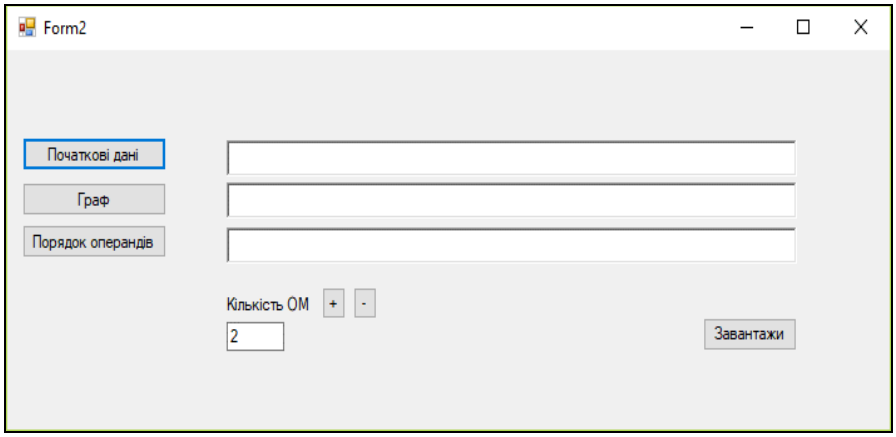


Рис. 3.20. Форма задання параметрів

На даній формі користувач задає таку інформацію:

1. Шлях до файлу з матрицею графа.
2. Шлях до файлу з інформацією про початкові операнди.
3. Шлях до файлу з інформацією про порядок входження операндів.
4. Кількість обчислювальних модулів.

При натисканні на кнопки «Початкові дані», «Граф», «Порядок операндів», відкриваються Провідник, де користувач може зручно вказати шлях до файлу, як це показано на рис. 3.21.

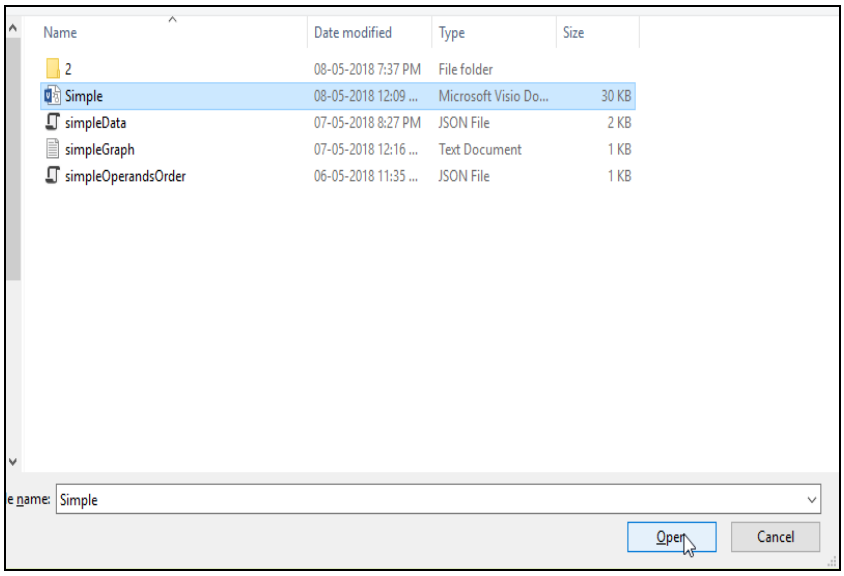


Рис. 3.21. Приклад вибору файлу

Якщо користувач захоче, він може повернутися до цієї форми та переобравши файли запустити процес обчислення заново.

При натисканні на кнопки з іконками плюса та мінуса відбувається відповідно додавання чи віднімання модуля. Їх кількість відображається поруч. При натисканні кнопки «Завантажити» відбувається зчитування даних з файлів та запуск процесу обчислення.

3.4.2. Опис головної форми

Головна форма на якій користувач може слідкувати за перебігом виконання програми зображено на рис. 3.22. Тут показується:

- стан середовища формування команд;
- стан запам’ятовуючого пристрою;
- стан буферу;
- стан обчислювальних модулів;
- зв’язки між компонентами системи. СФК зв’язана з буфером, буфер з обчислювальними модулями. Всі модулі зв’язані з запам’ятовуючим пристроєм по загальній шині, через диспетчер запам’ятовуючого пристрою;

- кнопки для виконання наступного такту та виконання одразу всього обчислення до кінця;
- поле результату;
- полу, що показує кількість тактів.

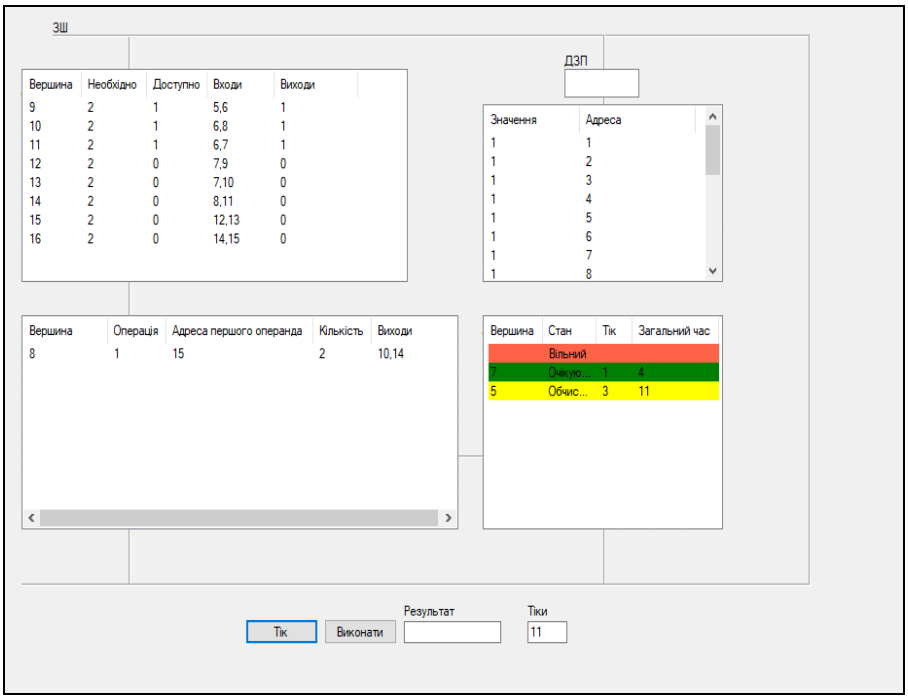


Рис. 3.22. Вигляд головної форми

При натисканні кнопки «Тік» виконується один такт системи, потім всі компоненти що містяться на даній формі оновлюють свій стан. Якщо користувач натисне на кнопку «Виконати» , програма виконається до кінця та виведеться повідомлення, що зображене на рис. 3.23. Повідомлення також виведеться і при завершенні обчислень потактово. Дане повідомлення містить інформацію про загальну кількість тактів, кількість звернень до пам’яті, результат обчислень.

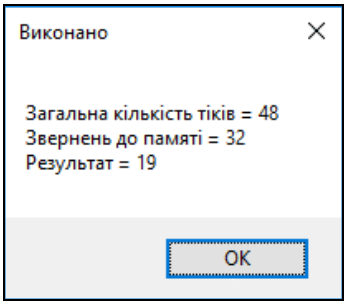


Рис. 3.23. Повідомлення про завершення обчислень

На рис. 3.24 зображено інформацію що показує стан СФК. Виводяться такі дані:

- «Вершина» – номер вершини якій відповідає команда;
- «Необхідно» – скільки операндів необхідно для початку формування команди;
- «Доступно» – скільки операндів доступно на даний такт;
- «Входи» – номера вершин входів;
- «Виходи» – номера вершин виходів.

Вершина	Необхідно	Доступно	Входи	Виходи
1	2	2	-1,-1	2
2	2	2	-1,-1	2
3	2	2	-1,-1	2
4	2	2	-1,-1	2
5	2	1	1,-1	1
6	2	1	2,-1	1
7	2	1	3,-1	1
8	2	1	4,-1	1
9	2	0	5,6	0
10	2	0	6,8	0

Рис. 3.24. Відображення стану середовища формування команд

Для відслідковування стану обчислювальних модулів використовувалась форма відображення, що показана на рис. 3.25. Тут показується наступна інформація:

- «Вершина» – номер вершини якій відповідає команда;

- «Стан» – стан обчислювального модуля. Якщо модуль вільний він замальовується червоним, якщо виконує обчислення – зеленим, якщо чекає на завантаження операндів – жовтим;
- «Тік» – номер такту для поточної операції;
- «Загальний час виконання» – скільки тактів займає виконання поточної операції.

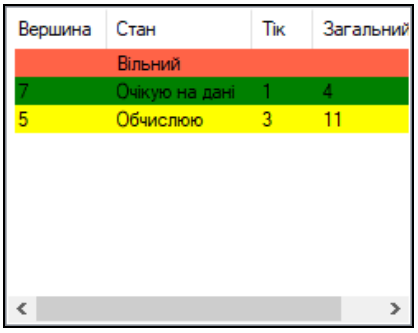


Рис. 3.25. Відображення стану обчислювальних модулів

3.5. Висновок до розділу 3

В даному розділі був описаний процес розробки імітаційної системи для виконання обчислень з багатомісними операціями.

Були описані вимоги користувача, функціональні вимоги та можливості розробленої системи. Наведено архітектура імітаційної моделі на основі якої виконувалась її розробка, інтерфейс користувача, а також описані класи розробленого програмного забезпечення.

4. АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

У даному розділі представлена реалізація декількох алгоритмів. За допомогою розробленої імітаційної системи були отримані дані, які дозволяють показати залежність збільшення швидкодії від кількості обчислювальних модулів, вплив прихованого паралелізму, порівняти швидкодії алгоритмів при використанні двомісних та багатомісних операцій.

4.1. Дослідження графів

Спочатку був використаний звичайний граф, представлений на рис. 4.1.

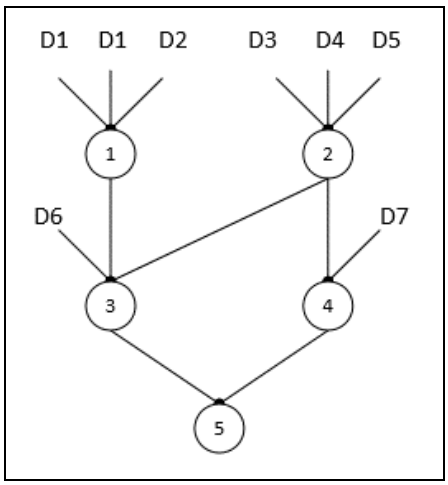


Рис. 4.1. Звичайний граф

На графі наявні наступні позначення:

- число у кружечку вказує на номер вершини;
- D_i - вхідне дане.

Граф використовує багатомісні функції та містить 7 вхідних даних.

Для програми даний граф необхідно подати в вигляді наступної матриці:

$$\begin{bmatrix} 0 & 0 & 23 & 0 & 0 \\ 0 & 0 & 23 & 24 & 0 \\ 0 & 0 & 0 & 0 & 25 \\ 0 & 0 & 0 & 0 & 25 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (1)$$

Числа 23, 24, 25 – це відповідні номери операції для вершин 3, 4 та 5.

Покажемо інформацію про даний алгоритм більш детально в таблиці 4.1.

Таблиця 4.1. Дані алгоритму для звичайного графу

Вершина	Вхідні дані	Операція	Код операції	Час
1	D1, D2, D3	D1*D2*D3+D3/D2	21	13
2	D3, D4, D5	D1*D2*D5	22	8
3	D6, 1, 2	A*D6+D6*B	23	6
4	2, D7	D7*A+25	24	6
5	3, 4	A+B	25	2

Тож для виконання алгоритму ми маємо такі дані:

- номер вершини;
- вхідні дані. Це можуть бути операнди, що надходять з інших вершин, або ж задані заздалегідь, такі дані позначаються D_i;
- визначення операції;
- код операції в бібліотеці функцій;
- час виконання даної операції.

Для порівняння швидкодії алгоритмів з двомісними та багатомісними операціями даний граф був перебудований з використанням двомісних операцій. Перебудований граф показаний на рис. 4.2.

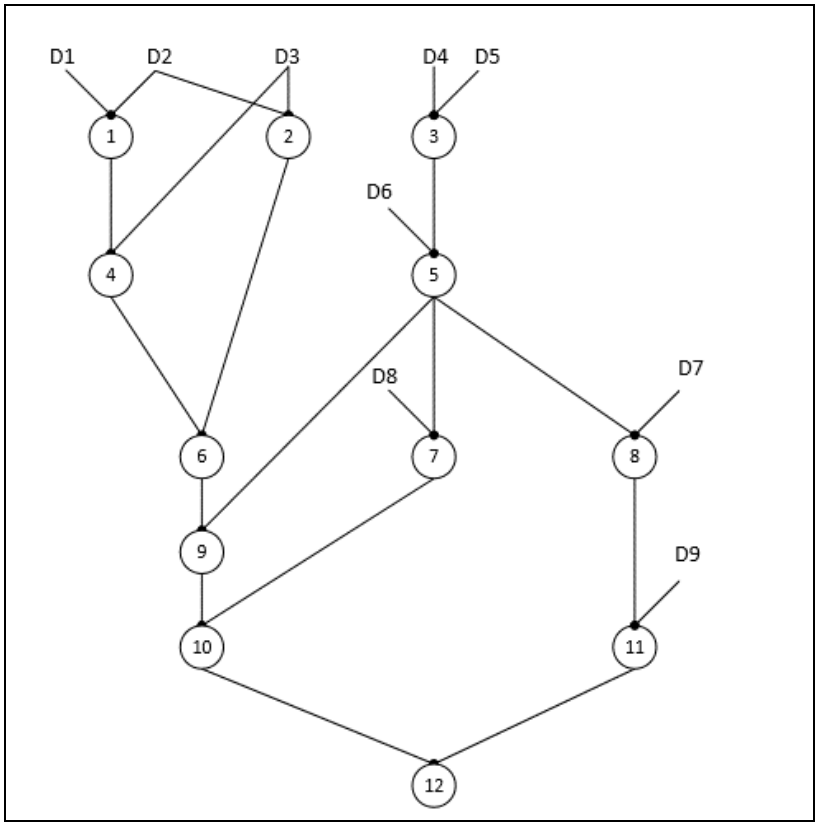


Рис. 4.2. Звичайний граф з використанням двомісних операцій

Використовуючи розроблену систему імітації було проведено дослідження обох алгоритмів з використанням різної кількості обчислювальних модулів. Результати дослідження наведено у таблиці 4.2.

Таблиця 4.2. Результати дослідження для звичайного графу

Тип графу	Кількість ОМ					
	1	2	3	4	5	6
Двомісний	71	43	36	36	36	36
Багатомісний	51	32	32	32	32	32

В таблиці наведено інформацію про кількість тактів для алгоритмів з двомісними та багатомісними операціями в залежності від кількості обчислювальних модулів. Також варто відмітити, що звернень до пам’яті було 24 та 13 відповідно до порядку в таблиці. Для більш наочного представлення поглянемо на рис. 4.3.

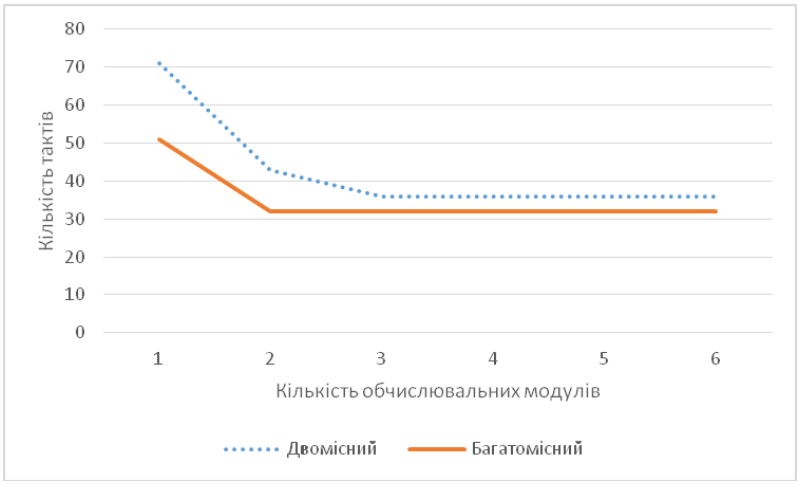


Рис. 4.3. Залежність кількості тактів від кількості ОМ

Максимальну швидкодії ми отримуємо при наявності 3 обчислювальних модулів, це є наслідком того що ми маємо три потоки. Більшої швидкодії шляхом збільшення кількості ОМ досягти не вдасться, адже не можна розпаралелити алгоритм на більшу кількість потоків. Як бачимо чим менша кількість обчислювальних модулів тим більший виграш ми отримуємо для багатомісного графа в швидкодії порівняно з двомісним. Це пояснюється тим що наявна більша кількість команд, і при кожному виконанні команди необхідно чекати завантаження операндів в ОМ. При збільшенні обчислювальних модулів для двомісного графа стає можливим виконуватися на більшій кількості ОМ, команди менше часу знаходяться в буфері. За рахунок того що паралельних гілок набагато більше і операції не такі довготривалі як у багатомісного графа, вони починають зрівнюватися в швидкодії при досягненні деякої кількості обчислювальних модулів. Мінімальний виграш у швидкодії для цього багатомісного графу становить близько 10%, а максимальний, тобто при наявності лише одного обчислювального модуля – близько 40%.

Наступним був досліджений граф з великою кількістю обмінів даними, що представлений на рис. 4.4.

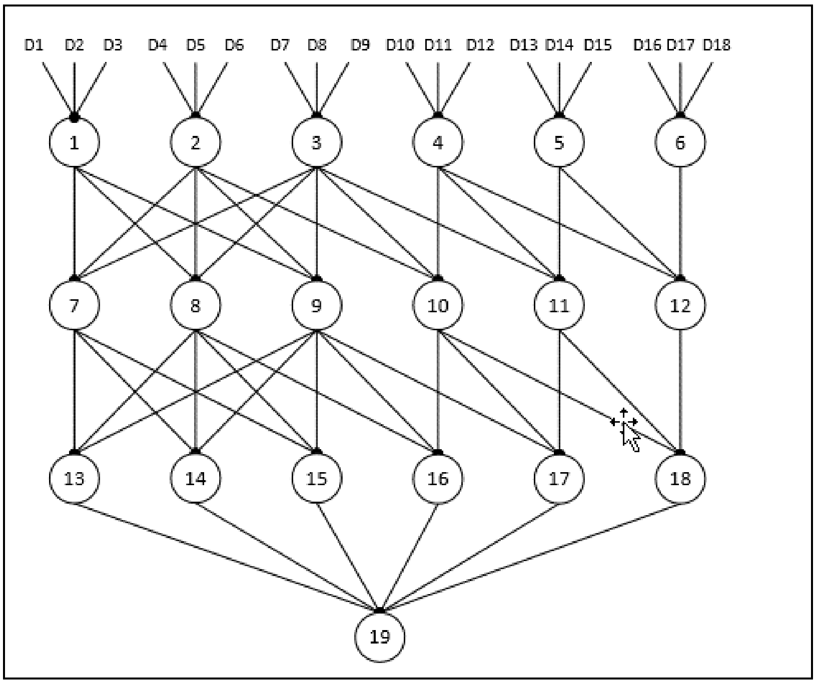


Рис. 4.4. Граф з великою кількістю обмінів даними

Кожна вершина графа має мінімум три вхідних операнди, вершини графа виконують операцію додавання трьох чисел, а дев’ятнадцята вершина шести. Матриця цього графа буде мати вигляд:

$$\begin{bmatrix}
0&0&0&0&0&0&31&31&31&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&31&31&31&31&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&31&31&31&31&31&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&31&31&31&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&31&31&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&31&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&31&31&31&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&31&31&31&31&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&31&31&31&31&31&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&31&31&31&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&31&31&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&31&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&32\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix} \quad (2)$$

Цей граф також був перебудований з використанням двомісних функцій. Перебудований граф показаний на рис. 4.5. Як бачимо кожна вершина, що виконувала додавання трьох чисел тепер була розбита на дві операції, як показано на рис. 4.6. Кожна з таких операцій виконує додавання двох чисел.

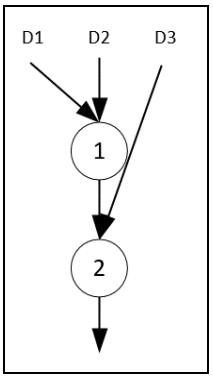


Рис. 4.6. Операція додавання трьох чисел з використанням двомісних функцій

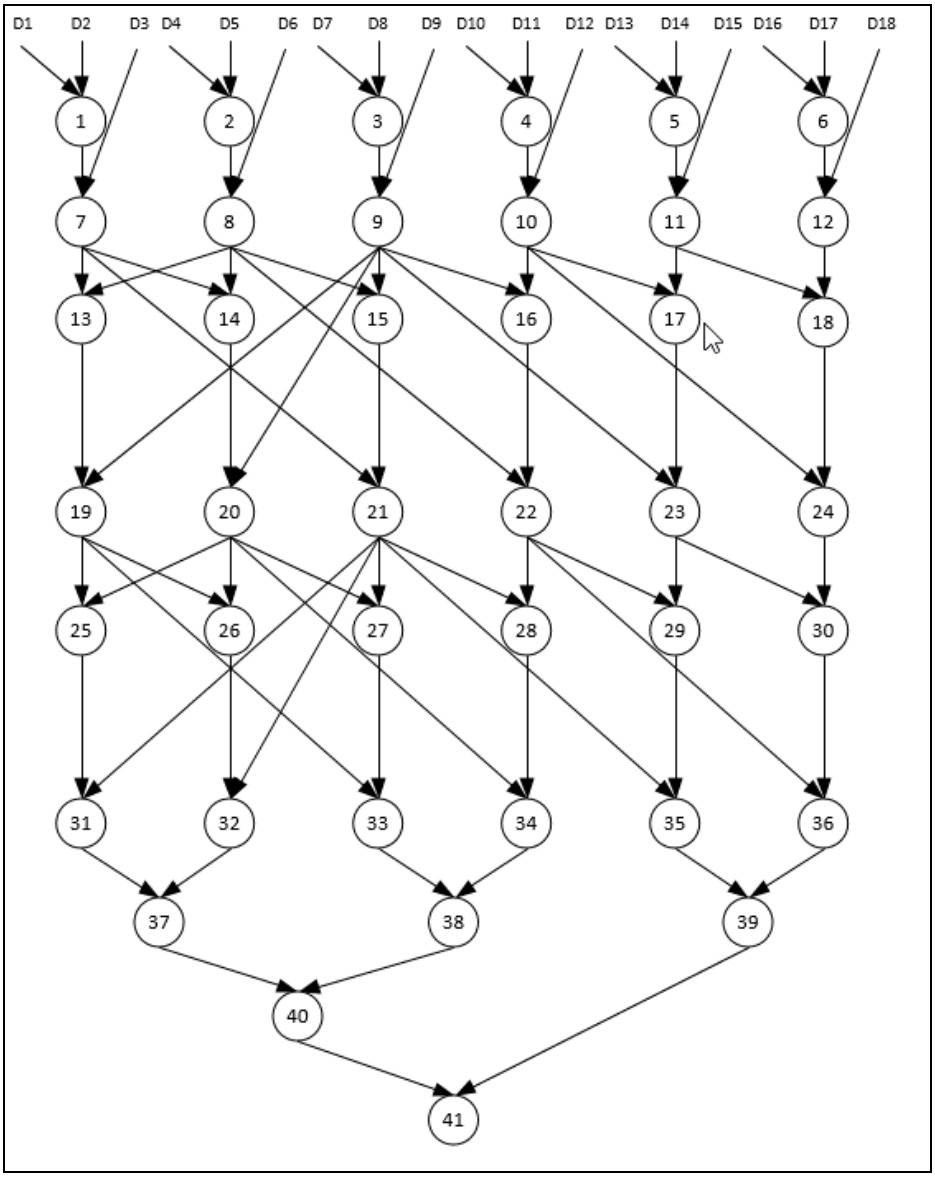


Рис. 4.5. Перебудований граф з великою кількістю обмінів даними

Провівши дослідження даних алгоритмів на розробленій імітаційній системі були отримані результати що показані в таблиці 4.3.

Таблиця 4.3. Результати дослідження для графу з великою кількістю обмінів даними

Тип графу	Кількість ОМ						
	1	2	3	4	5	6	7
Двомісний	166	88	64	53	53	53	53
Багатомісний	140	78	58	51	48	46	46

В таблиці наведено кількість тактів для обох алгоритмів при різній кількості обчислювальних модулів. Кількість звернень до пам'яті для багатомісного та двомісного алгоритму становить відповідно 60 та 82. Графік залежності кількості тактів від кількості обчислювальних модулів наведено на рис. 4.7.

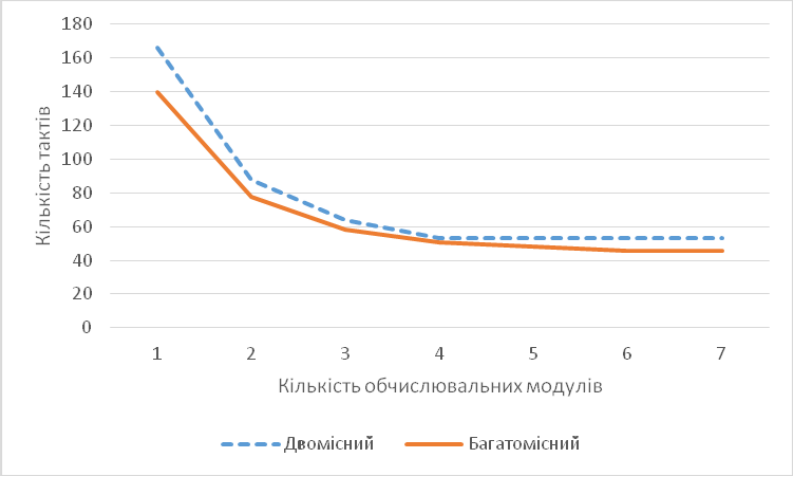


Рис. 4.7. Залежність кількості тактів від кількості ОМ

Для двомісного графа максимальна швидкодія досягається вже при 4 обчислювальних модулях, тоді як для багатомісного необхідно 6. Це пояснюється тим що двомісні функції набагато швидше виконуються. Для багатомісного графа перші 6 операцій додавання будуть виконуватися на 6 обчислювальних модулях тому що формування шостої команди в середовищі формування команд буде завершено до того, як перша операція додавання виконається, і звільниться перший обчислювальний модуль. Тому виконання шостої операції додавання буде виконано на шостому

обчислювальному модулі. Для двомісного графа ми маємо іншу ситуацію. Перший обчислювальний модуль звільниться до того як буде сформовано п’яту команду, тому ця команда буде виконуватися не на п’ятому ОМ, а на першому. Таким чином п’ятий та шостий обчислювальні модулі не будуть задіяні в обчисленнях, а максимальна швидкодія досягається при використанні чотирьом ОМ.

Для даного графу можна спостерігати ту ж залежність, що і для попереднього, а саме що чим менша кількість обчислювальних модулів тим більший виграш ми отримуємо для багатомісного графа в швидкодії порівняно з двомісним. Максимальний виграш у швидкодії для цього багатомісного графу становить близько 19%. При п’яти обчислювальних модулях і більше виграш становить 15%.

Відомо що максимальна швидкодія в потокових системах досягається тоді коли кількість потоків співпадає з кількістю обчислювальних модулів. Але в наших попередніх дослідженнях видно, що інколи це може бути не так. Наприклад для графу з великою кількістю обмінів даними алгоритм з використанням двомісних операцій виконувався найшвидше вже при чотирьох ОМ. Це відбувається тому що поки виконується одна вершина на одному обчислювальному модулі на іншому встигають виконатися декілька інших..

Розглянемо наступний граф, що зображений на рис. 4.8.

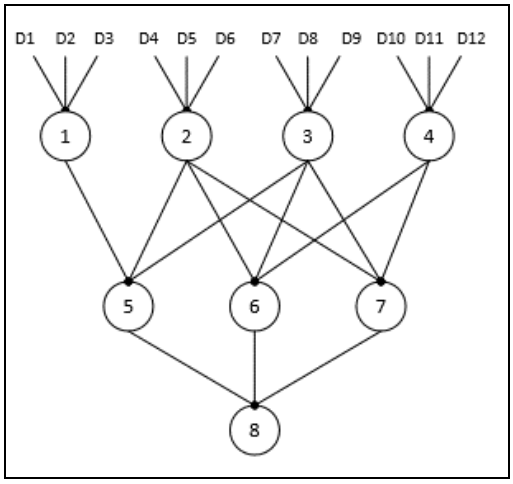


Рис. 4.8. Граф з сильною різницею в часі виконання для різних вершин

Даний граф має наступний віигляд у формі матриці:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 42 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 42 & 41 & 41 & 0 \\ 0 & 0 & 0 & 0 & 42 & 41 & 41 & 0 \\ 0 & 0 & 0 & 0 & 0 & 41 & 41 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 41 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 41 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 41 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (3)$$

Інформацію про даний граф зображено в таблиці 4.4.

Таблиця 4.4. Дані для алгоритму з сильною різницею в часі виконання для різних вершин

Вершина	Вхідні дані	Операція	Код операції	Час
1	D1, D2, D3	D1*D2/D3	42	13
2	D4, D5, D6	D4+D5+D6	41	4
3	D7, D8, D9	D7+D9+D8	41	4
4	D10, D11, D12	D10+D11+D12	41	4
5	1, 2, 3	A*B/C	42	13
6	2, 3, 4	A+B+C	41	4
7	2, 3, 4	A+B+C	41	4
8	5, 6, 7	A+B+C	41	4

Даний граф також був перебудований в граф з використанням двомісних операцій. Перебудований граф показаний на рис. 4.9.

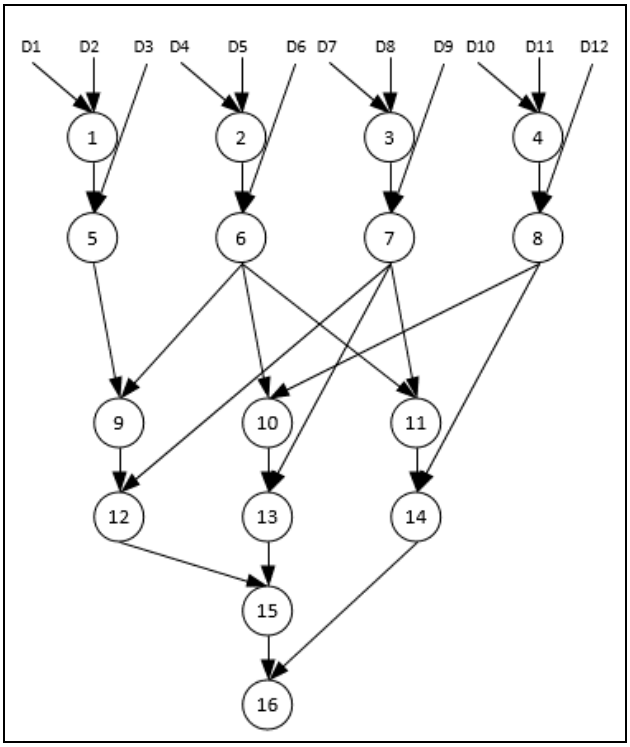


Рис. 4.9. Граф з використанням двомісних операцій

Як видно з графу кожна багатомісна операція була розбита на дві двомісних.

Опис даного графу наведено в таблиці 4.5.

Таблиця 4.5. Дані для алгоритму з сильною різницею в часі виконання для різних вершин з використанням багатомісних операцій

Вершина	Вхідні дані	Операція	Код операції	Час
1	D1, D2	D1*D2	3	4
2	D4, D5	D4+D5	1	2
3	D7, D8	D7+D8	1	2
4	D10, D11	D10+D11	1	2
5	1, D3	A/D3	4	9
6	2, D6	2+D6	1	2
7	3, D9	3+D9	1	2
8	4, D12	4+D12	1	2
9	5, 6	5*6	3	4
10	6, 8	6+8	1	2
11	6, 7	6+7	1	2
12	9, 7	9/7	4	9

13	10, 7	10+7	1	2
14	11, 8	11+8	1	2
15	12, 13	12+13	1	2
16	15, 14	15+14	1	2

Провівши дослідження при якому змінювалась кількість обчислювальних модулів були отримані результати що наведені в таблиці 4.6.

Таблиця 4.6. Результати дослідження для графу з сильною різницею в часі виконання для різних вершин

Тип графу	Кількість ОМ					
	1	2	3	4	5	6
Двомісний	85	51	48	48	48	85
Багатомісний	76	46	42	42	42	76

Графік залежності кількості тактів від кількості обчислювальних модулів наведено на рис. 4.10.

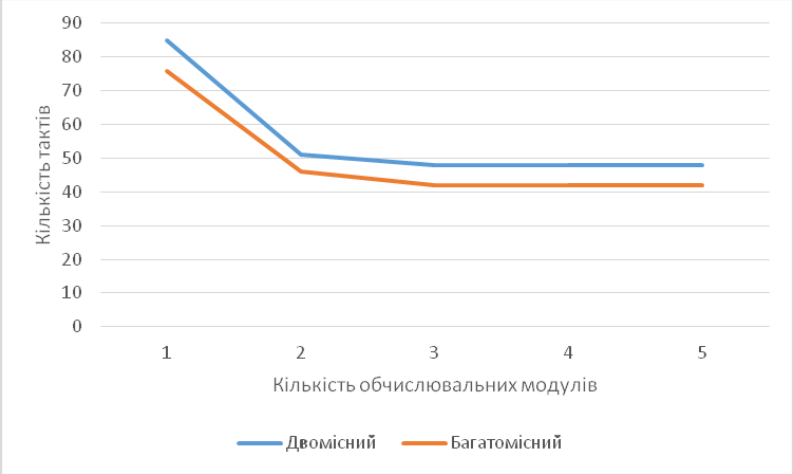


Рис. 4.10. Залежність кількості тактів від кількості ОМ для графу з сильною різницею в часі виконання для різних вершин

З зображення графу видно що наявно 4 потоки, отже найбільша швидкодія повинна досягатися при чотирьох обчислювальних модулях, але

з результатів видно що максимальна швидкодія досягається вже при 3, як для багатомісного так і для двомісного алгоритму. У випадку за багатомісним графом обчислення для вершини 1 займають 13 тактів. За цей час встигнуть виконатися операції для вершин 2 та 3 які займають по 4 такти, а також операція для вершини 4. А далі ми вже маємо вершини 5, 6 та 7 що потребують 3 обчислювальних модулі. Для двомісного графу ми маємо аналогічну ситуацію, поки виконувалися операції для вершин 1 та 5, на інших двох обчислювальних модулях виконались обчислення для вершин 2, 3, 4, 6, 7, 8. Ось і виходить що для максимальної швидкодії нам необхідно лише 3 обчислювальних модуля.

4.2. Дослідження прихованого паралелізму

Прихований паралелізм це таке явище, коли в ході обчислень з'являються додаткові паралельні гілки алгоритму, що можуть бути обчислені окремо.

Для дослідження цього явища використовувався граф, що зображений на рис.4.11.

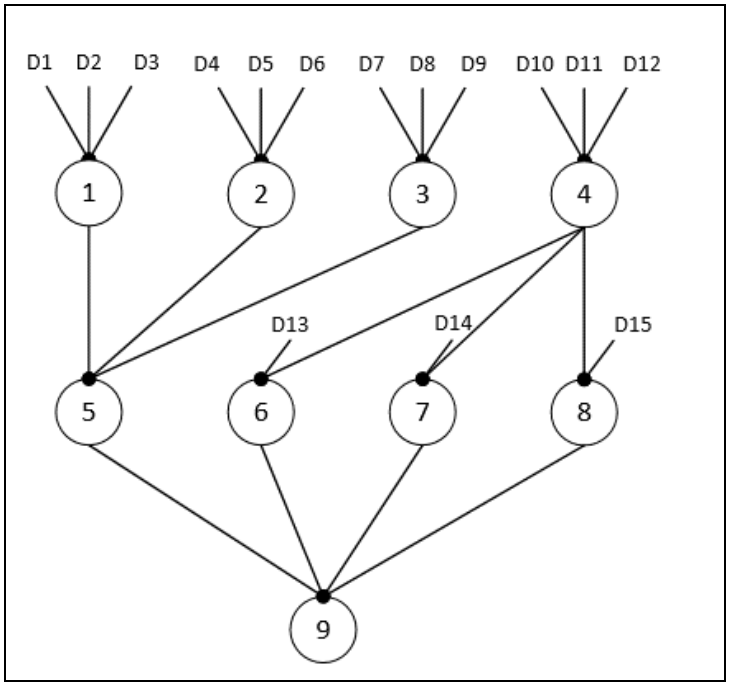


Рис. 4.11. Граф з прихованим паралелізмом

Як видно даний граф має 4 потоки, отже максимальна швидкодія повинна досягатися при чотирьох обчислювальних модулях, проте операція для вершини 4 виконається набагато швидше ніж для перших трьох, вона активізує виконання 6, 7, 8 вершин в той час поки все ще будуть виконуватися перші 3. В такому випаду ми будемо мати 6 завантажених обчислювальних модулі одночасно. Це показано на рис. 4.12, на ньому видно що завантажено 6 обчислювальних модулів, 5 виконують обчислення, а 6 чекає на завантаження значень операндів.

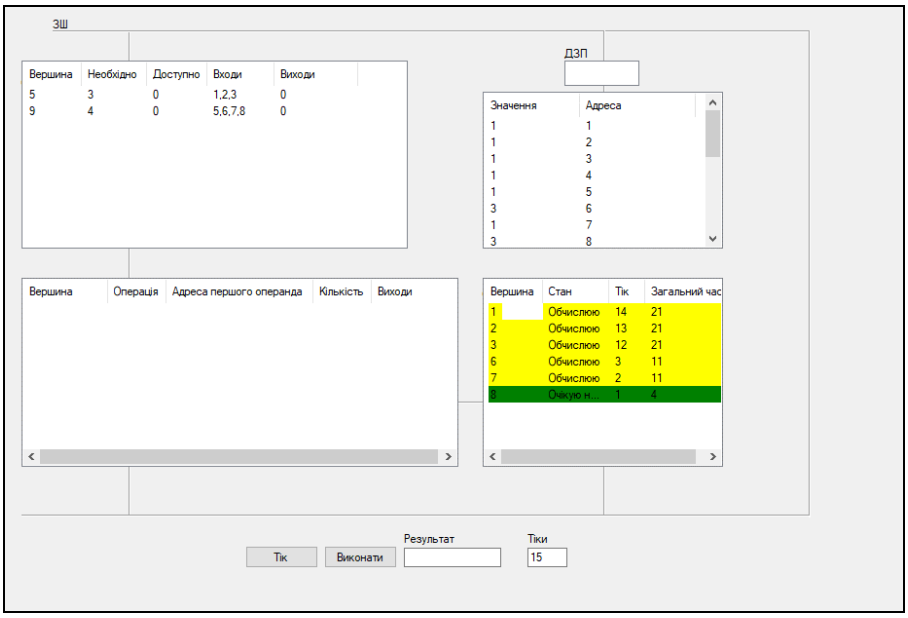


Рис. 4.12. Виконання обчислень на шести ОМ для графу з прихованим паралелізмом

4.3. Висновок до розділу 4

Використовуючи розроблену систему імітації були проведені дослідження трьох графів та графу з прихованим паралелізмом. Були побудовані їх двомісні та багатомісні версії, порівняна їх швидкодія при різній кількості обчислювальних модулів. Було досліджено явище прихованого паралелізму.

Максимальна швидкодія досягається коли кількість обчислювальних модулів дорівнює кількості потоків. Двомісні графи мають меншу швидкодію порівняно з багатомісними через те що мають більшу кількість команд для того ж алгоритму і необхідно чекати на завантаження більшої кількості операндів. Найбільший виграш в швидкодії, порівняно з графом що використовує двомісні операції, досягається при одному обчислювальному модулі, а чим ОМ більше тим менше виграш. Мінімальний виграш в швидкодії для розглянутих графів складав близько 10%. Можна зробити висновок що застосування багатомісних функцій є максимально ефективним коли ці функції є складними і їх розбиття на двомісні призведе то сильного зростання кількості вершин в графі.

Примечание [АК6]: виделение

Для дослідження прихованого паралелізму був використаний граф в якому одна вершина алгоритму виконувалась значно швидше, ніж інші, вона активувала виконання інших вершин, на іншому ярусі, чим підвищувала кількість завантажених ОМ. В ході досліджень було показано, що система здатна виявляти прихований паралелізм. Таким чином в системі могло бути задіяно більше обчислювальних модулів одночасно ніж ширина графу.

Примечание [АК7]: видиление

5. РОЗРОБЛЕННЯ БІЗНЕС МОДЕЛІ

5.1. Аналіз проблеми

На сьогоднішній день без застосування автоматизованої системи управління технологічним процесом (АСК ТП) не обходиться жодна з галузей промисловості. Автоматизація виробничих процесів однозначно призводить до підприємства в цілому, поліпшення якості продукції, а також підвищення рівня безпеки на виробництві [40]. Такі системи дозволяють скоротити час виробництва, зменшити кількість браку, мають більш високу точність позиціонування. Особливо важливим є використання повністю автоматичних систем виробничих процесів на шкідливих для людини виробництвах. Рівень сучасних технологій і швидкість прогресу такі, що вже неможливо вести ефективну трудову діяльність, особливо в області харчової та сільськогосподарської промисловості, без серйозної модернізації технічної бази і впровадження комп'ютерних систем. У поточний момент йде інтенсивний розвиток таких систем управління технологічним обладнанням і розширюється їх номенклатура, постійно охоплюються нові функції управління, удосконалюється програмне забезпечення, адже на автоматичне керування перекладається все більша кількість операцій технологічного процесу і виробництва в цілому. Багато критичних процесів неможливо виконати без використання АСК ТП [41]. Крім техпроцесу автоматизовані системи управління проникають і в інші етапи виробництва, а саме обслуговуючі та допоміжні процеси, які включають в себе технічний контроль якості продукції, транспортне обслуговування, складування, виготовлення інструменту, управління енергоресурсами, обмін даними, обробку, накопичення і зберігання інформації, формування сигналів тривог (alarm), побудова графіків (trend), звітів і т.п [42]. Розглянемо основні проблеми, що виникають при недостатній автоматизації технологічного процесу:

1. Необхідність людського втручання.

Там де людина виконує операції вручну з’являється так званий «людський фактор». І якщо в деяких системах помилка людини може привести лише до невеликої затримки у виробництві та економічних збитків то, наприклад в системах, що керують процесами на електростанціях, це може призвести до катастроф. Людське втручання призводить і до зниження якості продукції. Істотного підвищення якості продукції можна домогтися за рахунок виключення помилок і порушень технологічних режимів, неминучих при ручній праці, на поточних виробництвах, що вимагають високої точності. Ще однією проблемою є те, шо часто на підприємствах людина займається монотонною працею, а що ще гірше, підпадає під вплив шкідливих факторів. Автоматизація дозволить досягти звільн ення людини від малокваліфікованої і монотонної праці, трудомістких і важких операцій, поліпшення умов праці, виключення впливу шкідливих факторів на персонал. Також, якщо на поточному виробництві необхідне втручання людини, то більш частішим є випадок коли, лінія що йде далі, працює не на повну потужність (завантаженість), через затримку людини та коли не має можливості організувати безперервне виробництво.

2. Порушення цілісності лінії автоматизації.

Якщо на лінії виробництва існують розриви автоматизації це створює низку проблем. Ускладнюється документообіг, адже по системі проходить велика кількість даних в режимі реального часу і необхідно подбати, щоб на кожній ланці ці дані не втрачалися. Необхідно документувати і дії людини у таких розривах. Людина може забути щось записати, допустити помилку або ж просто проігнорувати деякі дані. Такі системи складніше модифікувати, адже можуть з’явитися вузькі місця, де ми начебто можемо пришвидшити процес, проте людина фізично не зможе виконувати свої обов’язки. Їх складніше підтримувати і контролювати.

Через людину може не завжди бути зрозумілим, що є причиною тих чи інших обставин, була це провина людини чи машини.

3. Збільшення витрат.

Підвищення рівня автоматизації дозволить значно зменшити витрати на виробництво шляхом зменшення капітальних вкладень, витрат на заробітну плату, скорочення площ і чисельності обслуговуючого персоналу, за рахунок використання технологічного обладнання в три зміни.

4. Збільшення часу реакції та відповіді на подію.

Зазвичай час реакції повністю автоматизованої системи набагато вище. Це особливо важливо коли щось йде не за планом виробництва, наприклад падіння чогось на конвеєрну стрічку або ж з неї, підвищення рівня шкідливих речовин, вихід з ладу якогось модулю. Час реакції у таких випадках може принести не лише економічні, а й більш суттєві збитки.

Всі описані вище проблеми узагальнені у дереві проблем, що зображено на рис. 5.1.

5.2. Зацікавлені сторони

Зацікавлених сторін є одразу декілька: власники підприємтсв, обслуговуючий персонал систем де застосовується АСК ТП, керівники підприємств, служби безпеки виробництва.

Власникам підприємств така система дозволить підвищити якість продукції, зменшити кількість персоналу, зменшити простої обладнання і таким чином знизити витрати на саме виробництво.

Керівникам підприємств стане в нагоді покращена звітність. Стане легше контролювати процес виробництва, виявляти слабкі місці, а у випадку нестандартної ситуації - знайти винних.

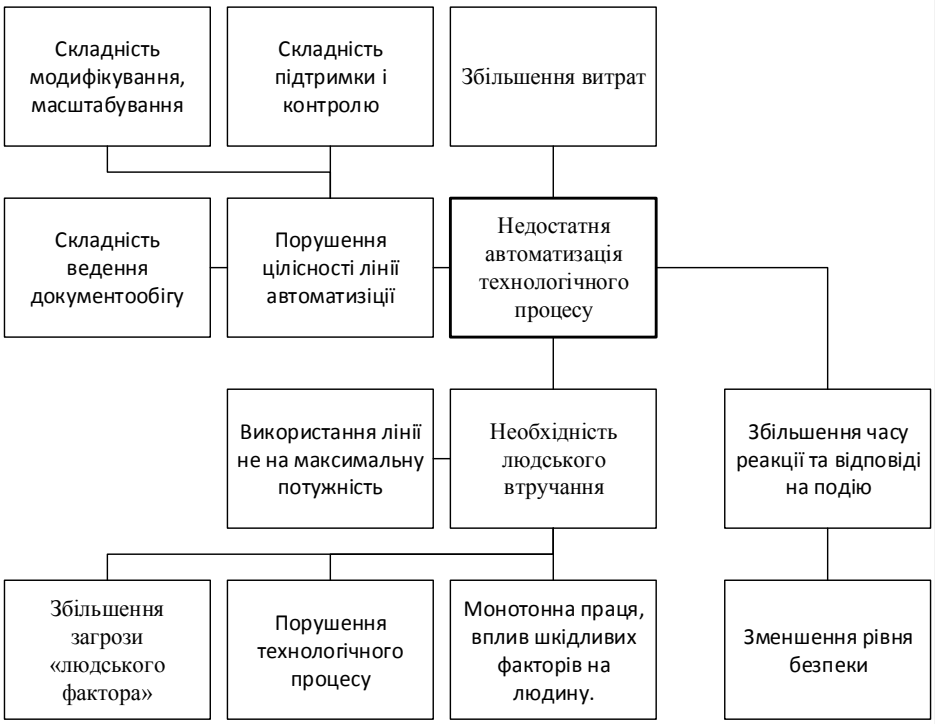


Рис. 5.1. Дерево проблем

Обслуговуючому персоналу буде легше підтримувати, масштабувати систему, вносити зміни, адже система буде їм в цьому допомагати, документуючи свою роботу та повідомляючи про несправності. При внесенні змін не потрібно буде піклуватися про поєднання людини та машини.

Наступною зацікавленою стороною є відповідальні за безпеку виробництва. АСК ТП може мати такі функції, як стеження за граничними показниками, повідомлення операторів про небезпеку, автоматичне відключення агрегатів. Все це дозволить швидко реагувати на небезпеку та приймати рішення швидше.

Опосередковано можна назвати зацікавленими сторонами і простих людей, адже буде зменшена ціна товарів, підвищена надійність небезпечних об’єктів. Там де автоматизація носитиме частковий характер, полегшиться праця людини. Якщо раніше трудомісткі операції необхідно

було виконувати вручну, то тепер, з використанням автоматизації, людина буде застосовувати для цього можливості машини.

Отже, зважаючи на вищесказане, можна коротко відобразити всі зацікавлені сторони у таблиці 5.1.

Таблиця 5.1 Зацікавлені сторони

Зацікавлена сторона	**Інтерес зацікавленої особи**	**Вплив зацікавленої особи**	**Стратегії приваблення зацікавлених сторін**
Власники підприємтсв	Можливість зменшити витрати та збільшити ефективність підприємства	Високий	Презентація продукту. Участь у спеціалізованих конференціях та виставках.
Обслуговуючий персонал	Легкість внесення змін та супроводження системи	Середній	
Керівники підприємств	Краща документованість виробничого процесу	Середній	
Служба безпеки виробництва	Зменшення нещасних випадків, аварійних ситуацій	Середній	

5.3. Комерційне рішення. Основні характеристики

Передбачається, що кінцевий продукт буде представляти собою АСК ТП яка реалізує заданий метод підвищення швидкодії. Він повинен відповідати наступним вимогам, що постають перед сучасними АСК ТП:

- АСК ТП в необхідних обсягах повинна автоматизовано виконувати:

- збір, обробку та аналіз інформації (сигналів, повідомлень, документів і т. п.) про стан об'єкта управління;
- вироблення управляючих впливів (програм, планів і т. п.);
- передачу керуючих впливів (сигналів, вказівок, документів) на виконання і її контроль;
- реалізацію і контроль виконання керуючих впливів;
- обмін інформацією (документами, повідомленнями і т. п.) з взаємопов'язаними автоматизованими системами.

- Склад автоматизованих функцій АСК ТП повинен забезпечувати можливість управління відповідним об'єктом відповідно до будь-якої з цілей, встановлених в ТЗ.
- Склад автоматизованих функцій АСК ТП і ступінь їх автоматизації повинні бути техніко-економічно і (або) соціально обґрунтовані з урахуванням необхідності звільнення персоналу від виконання повторюваних дій і створення умов для використання його творчих здібностей в процесі роботи.

Такі системи є цікавими для власників та керівників підприємств, саме вони будуть приймати рішення про встановлення таких систем. Система допоможе їм підвищити ефективність виробництва та контроль за ним. Отже моделлю бізнесу є B2B – Business-to-Business (бізнес для бізнесу).

Даний продукт реалізовує програмну та апаратну частину. Апаратна частина буде використовувати модель потокової системи. Дана модель є досить специфічною та складною в реалізації, отже і дорогою. Але дозволить збільшити ефективність готового продукту.

5.4. Конкурентні переваги

Сучасний ринок пропонує широкий спектр пристроїв для реалізації систем управління. Різноманітність засобів автоматизації і постійне їх розвиток зумовлений тим, що сучасне виробництво генерує безліч нових

невирішених завдань і постійно підвищує вимоги до обладнання, задоволення яких забезпечується і системами управління. Це призводить до постійного збільшення і вдосконалення компонентів систем управління.

Конкурентами даної системи є класичні АСК ТП. В класичних системах виконання тієї чи іншої програми відбувається за циклічним типом. Зазвичай цикл триває 20, 50, 250 мс, 1, 2, 3, 4, 5 с. Це і є їх головним недоліком - вони обмежені тривалістю циклу.

Головною конкурентною перевагою реалізації АСК ТП як системи реального часу на потоковій системі з використанням досліджуваного метода є підвищення швидкості обробки інформації.

5.5. Клієнти. Сегменти ринку споживання

Клієнтами даного продукту є керівники та власники підприємств.

Сегментацію можна провести за обсягом даних які необхідно обробляти, а також за рівнем критичності відмови системи.

Можна виділити підприємства, де проходить невелика кількість даних. Таким підприємствам скоріше важливо аби система була простою та недорогою в обслуговуванні, ніж щоб вона була надпотужною. Підприємствам з масивними виробничими лініями, де проходить велика кількість інформації , яку необхідно обробити та зберегти, важливо щоб система могла опрацювати ці дані вчасно.

Важливою характеристикою є критичність відмови встановлюваних АСК ТП. Втрата попередніх даних, відмова одного з модулей, задовгий час реакції – все це може призвести до значних матеріальних втрат або аварій. Може на довгий час зупинитися вся лінія, статися вибух або викид шкідливих речовин. На підприємствах, де якісь частини обладнання небезпечно рухаються в безпосередній близькості від людини, може постраждати співробітник. Такі системи потребують окремої уваги. Як правило підприємства готові платити більше за швидкодію та надійність у таких випадках, а також можуть дозволити собі утримувати дорогий обслуговуючий персонал.

5.6. Унікальна ціннісна пропозиція

Даний продукт дозволить вирішити поставлені вище задачі за допомогою використання розглянутого методу підвищення швидкодії. Встановлення такої системи дозволить збільшити продуктивність та надійність виробництва. Унікальність цієї пропозиції полягає в тому, що на ринку не представлено АСК ТП на потоковій архітектурі. Усі представлені АСК ТП працюють за циклічною схемою, що значно обмежує їх швидкодію. В розроблюваній системі такого недоліка немає.

5.7. Доходи і витрати

Передбачається що основний дохід буде надходити від виконання замовлень від підприємств. Виконання замовлення включає в себе повний цикл по устаткуванню підприємства такими системи:

1. Формування вимог до АС
2. Розробка концепції АС
3. Розробка технічного завдання
4. Розробка ескізного проекту
5. Розробка технічного проекту
6. Розробка документації на АСК ТП
7. Введення в дію
8. Супровід АС.

До основних витрат належать витрати на деталі для АСК ТП та витрати на персонал, що буде займатися розробкою та впровадженням систем на підприємствах. Вартість деталей та необхідний їх набір буде варіюватися в залежності від потреб підприємства. Витратним також буде просування нової системи на ринку.

До додаткових витрат можна віднести оренду приміщення, оплату комунальних послуг, залучення окремих експертів з різних предметних областей. Детальніше з додатковими витратами можна ознайомитися з таблиць 5.2.

Витрати на виплату заробітної плати найманому персоналу без податків складає 30000$ на місяць. З урахуванням податків ця сума буде складати 43200$ на місяць. що в рік буде складати 518400$

Таблиця 5.2. Загальні витрати

Список витрат	**Сума в рік , тис. $**	**Періодичність**
Оренда	70	Рік
Комунальні послуги	7	Рік
Оплата не виробничих послуг	5	Рік
Наймання спеціалістів з предметної області	10	
Результат:	**92**	

Прибутки очікуються на четвертому році від продажу продукту. Детальніше ознайомитися зі зведеним планом прибутків та витрат можна з таблиці 5.3.

Таблиця 5.3. Фінансовий результат діяльності

Найменування витрат	**1-й рік, т. $**	**2-й рік, т. $**	**3-й рік, т. $**	**4-й рік, т. $**	**5-й рік, т. $**	**Загальні результати, т. $**
Додаткові витрати	92	92	92	92	92	460
Замовлення деталей	200	200	300	500	700	1900
ЗП	518,4	518,4	518,4	518,4	518,4	25920
Витрати	810.4	810.4	910.4	1110.4	1310.4	4952
Заплановані прибутки	460	500	700	1500	2500	5660
Результат(без оподаткування)	**-350.4**	**-310.4**	**-210.4**	**389.6**	**1189.6**	**708**

5.8. Бізнес модель

Узагальнимо, все написане вище у бізнес-модель у вигляді lean canvas (таблиця 5.3.).

Отже, можна зробити висновок, що даний продукт можна реалізувати та побудувати бізнес, продаючи його зацікавленим в ньому підприємствам. Наведені розрахунки покликані показати, що такий бізнес має сенс лише якщо він розрахований як довгостроковий, адже прибуток можливий лише на 4 році його ведення. Це означає, що на початковому етапі знадобляться великі інвестиції.

Таблиця 5.4. Бізнес-модель lean canvas

<table>
<tr><td rowspan="2">Проблема
Складність ведення документообігу
Необхідність людського втручання
Збільшення часу реакції</td><td>Рішення
Розробка АСК ТП, на потоковій архітектурі</td><td rowspan="2">Унікальна ціннісна пропозиція
Розширення можливостей АСК ТП шляхом її пришвидшення.</td><td>Прихована перевага
Пришвидшення роботи дрібнозернистих алгоритмів</td><td rowspan="2">Споживачі
Підприємства</td></tr>
<tr><td>Ключові метрики
Кількість проданих одиниць продукту кожного виду</td><td>Канали
Керівники підприємств</td></tr>
<tr><td colspan="3">Структура витрат
Закупка деталей
Утримання персоналу
Утримання будівель
Податкові витрати</td><td colspan="2">Потоки доходів
Доходи від продажу та підтримки продукту</td></tr>
</table>

5.9. Висновок до розділу 5

У даному розділі було виділено основні проблеми предметної галузі, переваги запровадження АСУ ТП на виробництві. Були наведені та проаналізовані зацікавлені сторони у вирішенні даних проблем, визначено їх вплив. Проведено дослідження потенційних клієнтів та варіантів сегментування ринку споживання. У якості комерційного рішення було запропоновано продукт, який базується на досліджені, що було проведено

в даній дисертації. Були обґрунтовані конкурентні переваги та унікальна ціннісна пропозиція запропонованого продукту. На основі наведених вище досліджень було проаналізовано потенційні доходи і витрати. Як результат була сформована бізнес-модель, що доводить потенціальну вигідність заснування бізнесу з продажу даного продукту.

ВИСНОВКИ

Дана робота була присвячена дослідженню шляхів підвищення продуктивності потокових системи. Були описані принципи роботи таких систем, їх преваги та недоліки.

Дане дослідження є актуальним, так як дозволяє підвищити продуктивність потокових систем, та дає поштовх для розробки інших покращених методів.

Був запропонований метод підвищення продуктивності потокових систем шляхом використання багатомісних операцій. Розглянувши існуючі системи було прийнято рішення, що вони не завжди підходять для рішення конкретних задач і є досить дорогими, та що необхідно розробити власну імітаційну систему для проведення досліджень. Були розглянуті загальна архітектура системи, формати акторів, операндів та команд для їх використання в побудові імітаційної системи.

Проведені дослідження показали доцільність використання запропонованого підходу для підвищення швидкодії потокових систем. Рівень пришвидшення сильно залежить від кількості обчислювальних модулів та рівня зменшення кількості звернень. Пришвидшення тим більше чим менша кількість ОМ та значніше зменшення кількості звернень. Мінімальний виграш в швидкодії для розглянутих прикладів складав близько 10%.

Примечание [AK8]: виделение

Примечание [AK9]: виделение

Розроблена імітаційна система дозволяє проводити дослідження процесу обчислень в потокових системах, використовуючи двомісні та багатомісні операції, а також варіювати кількість обчислювальних модулів, надає необхідну інформацію для порівняння ефективності різних алгоритмів. При розробці застосовувалась мова програмування C#, графічна бібліотека Windows Forms та середовище розробки Visual Studio 2017.

Була розроблена бізнес модель, що дозволить комерціалізувати дане дослідження.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Міллер Р.М., Кок Д. ЕОМ із змінною конфігурацією. Новий клас універсальних машин. [Текст] / Міллер Р.М., Кок Д : Ч. II. Новосибірськ: ВЦ СО АН СРСР, 1972. С. 121-135.
2. Treleaven P.C. Fifth generation computers [Text] / Treleaven P.C. Gouveia L.I. — 1982. Vol. 26 N. 3-4. P. 277-283.
3. Silva J.G.D. Design of processing subsystems for Manchester data flow computer [Text] / J.G.D.Silva, J.V.Wood — N.Y.: IEEE, 1981. — P. 218-224.
4. Watson I. A practical data flow computer [Text] / Watson I., Guard J. — 1982. Vol. 15. N. 2. P. 51-57.
5. Brewer E. Clustering: Multiply and Conquer [Text] — July, 1997.
6. Гергель В.П. Теория и практика параллельных вычислений [Text] — М.: НОУ "Интуит", 2016. — 4 с.
7. Грегори Р. Эндрюс. Основы многопоточного, параллельного и распределенного программирования [Текст]. — М.: Вильямс, 2015 — С. 2-3.
8. Dennis J.B. A preliminary architecture for basic data flow processor [Text] / J.B.Dennis, D.P.Missunas — N.Y.: IEEE, 1975. — P. 126-132.
9. Dennis J.B. Dataflow Supercomputers [Text] — 1980. — P. 48-56.
10. Цилькер Б.Я. Организация ЭВМ и систем: учебник для вузов [Текст] / С.А. Орлов, Б.Я. Цилькер. — СПб.: Питер, 2011. — 688 с.
11. Mark D. Hill. Readings in Computer Architecture [Text] / D.H. Mark, P.J. Norman, S.S. Gurindar — San Francisco.: Morgan Kaufmann Publishers, 2000 — P. 339-340.

Примечание [WU10]: http://www.williamspublishing.com/Books/5-8459-0388-2.html

Примечание [WU11]: Первій подраздел http://www.nsc.ru/win/elbib/data/show_page.dhtml?77+861

Примечание [WU12]: https://books.google.com.ua/books?id=I7o8teBhz5wC&pg=PA339&lpg=PA339&dq=U-interpreter+dataflow+system&source=bl&ots=Q5vqt9A-EN&sig=bxieh2a4aZcsusG06qc7Pul6i9U&hl=en&sa=X&ved=0ahUKEwiKjJe5mpXbAhUGDiwKHeh7C5IQ6AEIVjAG#v=onepage&q=U-interpreter%20dataflow%20system&f=false

12. McGraw R. SISAL: Streams and iteration in a single assignment language, language reference manual, version 1.2 [Text] / R. McGraw and S.K Skedzielewski — Lawrence Livermore Nat. Lab, 1985. — P. 5-7.
13. Guernic P. Le. SIGNAL-A data flow-oriented language for signal processing [Text] / P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier — 1986. — P. 4-5.
14. Brookes S.D. A theory of communicating sequential processes [Text] / S.D. Brookes, C.A. Hoare, A.W. Roscoe — 1984. — P. 560-599.
15. Lee E.A. Static scheduling of synchronous data flow programs for digital signal processing [Text] / E.A. Lee and D.C. Messerschmitt — 1987. — P. 24-35.
16. Xilinx. [Електронний ресурс]. — Режим доступу: https://ru.wikipedia.org/wiki/Xilinx — Дата доступу : Квітень 2018. — Назва з екрана.
17. Xilinx Inc, Form 8-K, Current Report, Filing Date Jan. [Електронний ресурс]. — Режим доступу: http://edgar.secdatabase.com/2296/74398818000005/filing-main.htm. — Дата доступу : Квітень 2018. — Назва з екрана.https://en.wikipedia.org/wiki/Xilinx - cite_note-Xilinx-Inc-May-2017-10-K-5
18. Xilinx Inc, Form 10-K, Annual Report. [Електронний ресурс]. — Режим доступу: http://secdatabase.com/1004/0000743988-17-000046.htm — Дата доступу : Квітень 2018. — Назва з екрана.
19. Brian Bailey, EE Times. Second generation for FPGA software. [Електронний ресурс]. — Режим доступу: https://www.eetimes.com/document.asp?doc_id=1315621. — Дата доступу : Квітень 2018. — Назва з екрана.
20. CERN Scientists Use Virtex-4 FPGAs for Big Bang Research. [Електронний ресурс]. — Режим доступу:

http://www.xilinx.com/publications/xcellonline/xcell_65/xc_pdf/p28_31_65_F_XiWild.pdf. — Дата доступу : Квітень 2018. — Назва з екрана.

21. Xilinx Inc, Form 8-K, Current Report, Filing Date Apr 25, 2012. [Електронний ресурс]. — Режим доступу: http://edgar.secdatabase.com/846/119312512182086/filing-main.htm. — Дата доступу : Квітень 2018. — Назва з екрана.
22. FPGAs Cool Off the Datacenter, Xilinx Heats Up the Race. [Електронний ресурс]. — Режим доступу: https://www.eejournal.com/article/20141118-datacenter. — Дата доступу : Квітень 2018. — Назва з екрана.
23. Xilinx vs. Altera, Calling the Action in the Greatest Semiconductor Rivalry. [Електронний ресурс]. — Режим доступу: https://www.eejournal.com/article/20140225-rivalry. — Дата доступу : Квітень 2018. — Назва з екрана.
24. Xilinx Vivado [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Xilinx_Vivado. — Дата доступу : Квітень 2018. — Назва з екрана.
25. The Vivado Design Suite accelerates programmable systems integration and implementation by up to 4X [Електронний ресурс]. — Режим доступу: https://www.edn.com/electronics-products/other/4375467/The-Vivado-Design-Suite-accelerates-programmable-systems-integration-by-up-to-4X. — Дата доступу : Квітень 2018. — Назва з екрана.
26. WebPACK edition of Xilinx Vivado Design Suite now available [Електронний ресурс]. — Режим доступу: https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html. — Дата доступу : Квітень 2018. — Назва з екрана.
27. Altera Shipping 28-nm FPGAs [Електронний ресурс]. — Режим доступу: https://www.nasdaq.com/g00/article/altera-shipping-28nm-fpgas-analyst-blog. — Дата доступу : Квітень 2018. — Назва з екрана.

28. Altera's Quartus II design software features Qsys System Integration Tool [Електронний ресурс]. — Режим доступу: https://www.eetimes.com/document.asp?doc_id=1316604. — Дата доступу : Квітень 2018. — Назва з екрана.
29. Latest and greatest Quartus II design software from Altera [Електронний ресурс]. — Режим доступу: https://www.eetimes.com/document.asp?doc_id=1316845. — Дата доступу : Квітень 2018. — Назва з екрана.
30. Воеводин В.В. Параллельные вычисления [Текст] / В.В.Воеводин, Вл.В.Воеводин. — СПб.: БХВ-Петербург, 2002. – 608 с.
31. Соснин В.В. Введение в параллельные вычисления [Текст] / В.В. Соснин, П.В. Балакшин. — СПб.: Университет ИТМО, 2015. – 8 с.
32. Капура, А.С. Дослідження моделі потокової системи для виконання багатомісних операцій [Текст] / А.С. Капура, В.В. Жабіна // Наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 — Київ, 2018. — С. 2-4.
33. Капура, А.С. Підвищення ефективності реалізації операцій з різною кількістю операндів в потокових системах [Текст] / А.С. Капура, В.В. Жабіна // VIII Міжнародній конференції студентів і молодих учених "Сучасні Інформаційні Технології 2018" — Одеса, 2018 — С. 1-2.
34. Sagiv M. Precise interprocedural dataflow analysis with applications to constant propagation [Text] / M.Sagiv, T.Reps, S.Horwitz. — 1996. — P. 131-170.
35. Жабина В.В. Повышение эффективности параллельной обработки данных на уровне операций в потоковых системах [Текст] / В.В.Жабина // Вісник НТУУ „КПІ". Інформатика, управління та обчислювальна техніка: Зб. наук. пр. – К.: „ВЕК+", 2008. — №49. — С. 112-116.

Примечание [WU13]:

Примечание [WU14]: https://books.ifmo.ru/file/pdf/1900.pdf

36. Жабин В.И. Архитектура вычислительных систем реального времени / В.И.Жабин. — К.: ТОО “ВЕК+”, 2003. — 176 с.
37. Діаграма прецедентів [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Діаграма_прецедентів. — Дата доступу : Квітень 2018. — Назва з екрана.
38. Діаграма послідовності [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Діаграма_послідовності. — Дата доступу : Квітень 2018. — Назва з екрана.
39. Data transfer object [Електронний ресурс]. — Режим доступу : https://en.wikipedia.org/wiki/Data_transfer_object. — Дата доступу : Квітень 2018. — Назва з екрана.
40. Федоров Ю.Н. Справочник инженера по АСК ТП: Проектирование и разработка. Учебно-практическое пособие [Текст]. — М.: Инфра-Инженерия, 2008. — 928 с.
41. АСУ ТП Энергоблоков [Електронний ресурс]. — Режим доступу : http://www.westron.kharkov.ua/SB_article_3. — Дата доступу : Квітень 2018. — Назва з екрана.
42. Назначение, цели создания, и функции АСУТП [Електронний ресурс]. — Режим доступу : https://automation-system.ru/spravochnik-inzhenera/35-glava8/316-8-1.html. — Дата доступу : Квітень 2018. — Назва з екрана.

Примечание [WU15]: Сделать нормальніе ссілки на литературу

Примечание [WU16]: 5 раздел

Примечание [WU17]: http://www.westron.kharkov.ua/files/SB_article_3.pdf

ДОДАТКИ

Додаток 1

Копія презентації

Модель потокової системи для виконання багатомісних операцій

ВИКОНАВ СТУДЕНТ:
КАПУРА А.С.
НАУКОВИЙ КЕРІВНИК:
К.Т.Н., ЖАБІНА В.В.

Мета роботи: дослідження можливості прискорення обробки інформації за рахунок динамічного розподілу робіт між обчислювальними вузлами в мультипроцесорних системах. Результати дослідження дадуть змогу розробити програмний емулятор для вивчення характеристик в різних режимах реалізації паралельних алгоритмів, що передбачають виконання багатомісних операцій.

Об’єктом дослідження є процеси обробки інформації в паралельних системах реального часу, що пов’язані з проблемами підвищення швидкодії.

Предметом дослідження є методи і засоби прискорення обчислень в паралельних системах за рахунок динамічного розпаралелювання процесів на рівні обробки завдань та засоби їх моделювання.

Статичне розпаралелювання :

Мови програмування: C#, JavaScript, C++

- Необхідність ручного розпаралелювання
- Складність виявлення прихованого паралелізму
- Необхідність враховувати конфігурацію апаратних засобів

Динамічне розпаралелювання:

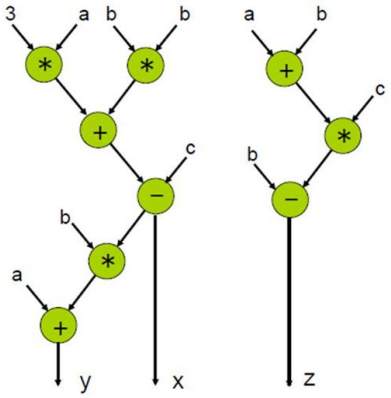
Мови програмування: VAL, LUCID, SISAL

- Необхідність створення бібліотек функцій
- Зі збільшенням зернистості алгоритмів істотно зростають обсяги даних, що пересилаються, що робить таку модель обчислень неефективною.

5

Data flow graph (DFG)

```
x = 3*a + b*b - c;
y = a + b*x;
z = b - c*(a + b);
```



6

Закон Амдала

У разі, коли задача поділяється на кілька частин, сумарний час її виконання на паралельній системі не може бути менше часу виконання найдовшого фрагмента.

$$S_p = \frac{1}{a + \frac{1-a}{p}}$$

де a — величина послідовного фрагменту, p — кількість процесорів

При частці послідовних обчислень *a* загальний приріст продуктивності не може перевищити 1 / *a*.

$\alpha \setminus p$	10	100	1000
0	10	100	1000
10%	5.263	9.174	9.910
25%	3.077	3.883	3.988
40%	2.174	2.463	2.496

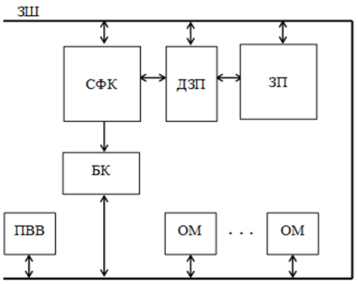
Системи моделювання мультипроцесорних систем

Vivado Design Suite від компанії Xilinx, Altera Quartus від компанії Altera

Недоліки:

- Дорогі
- Потребують багато часу для освоєння
- Не завжди можуть враховувати специфіку конкретних задач

Архітектура обчислювальної системи



Актор

Операція, яка відповідає i-й вершині графа, описується актором

$$W_i = (N_i, I_i, F_i, T_i, Q_i)$$

де N_i– ім'я операції; , I_i– код операції; F_i– множина імен входів i – ї вершини графа ; T_i– множина імен виходів i – ї вершини графа Q_i– сумарне число входів i – ї вершини графа.

11

Операнд

Операнд, що відповідає дузі графа між i-ю та j-ю вершинами, визначається дескриптором даних

$$D_{ji} = (N_i, d_{ji}, A_{ji},)$$

де d_{ji}– ім'я (номер) входу в i-у вершину; A_{ji} – елемент адресації операнда.

12

Команда

Команда, що формується для виконання обчислень для i-й вершини графа, описується наступним чином

$$Z_i = (I_i, T_i, A_i)$$

де A_i– множина елементів адресації усіх даних для i-ї команди.

13

Умова готовності команди

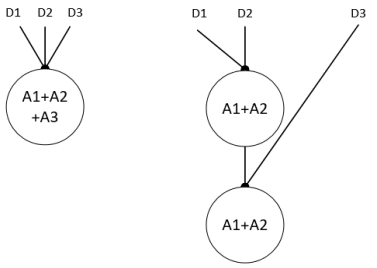
Умова готовності i-ї команди визначається як

$$\lambda_i = \&_{j=1}^{n+1} c_i^j$$

c_i^j - ознаки наявності актора та всіх n операндів для i- ї команди.

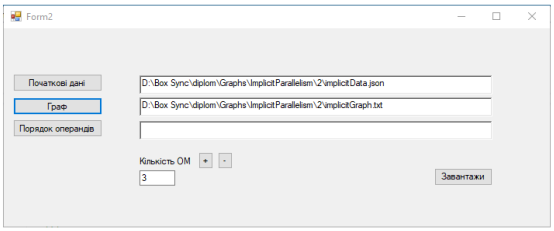
14

Багатомісні операції



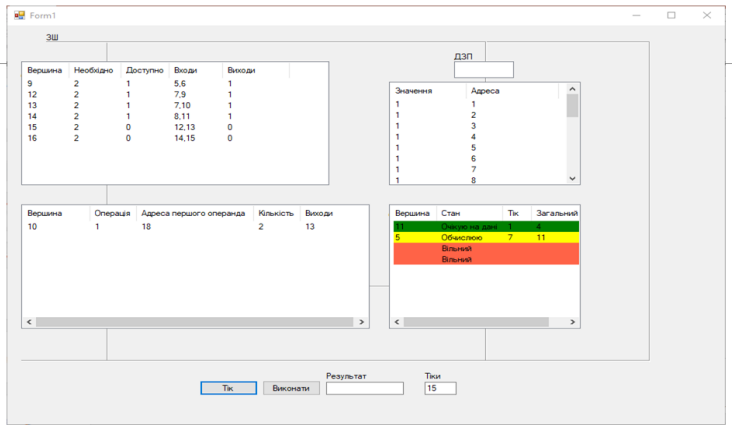
15

Форма задання параметрів



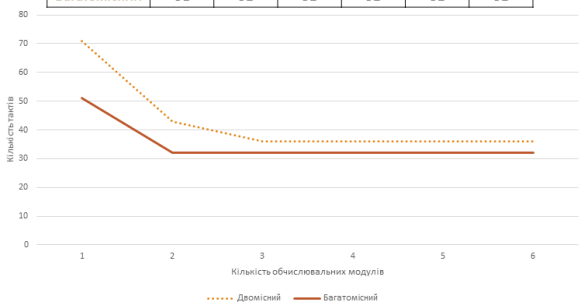
16

Форма відслідковування ходу обчислень



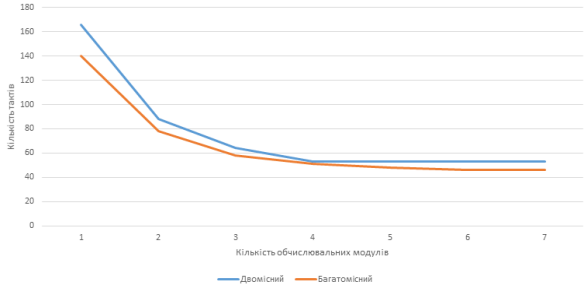
Результати дослідження для першого графу

Тип графу	Кількість ОМ					
	1	2	3	4	5	6
Двомісний	71	43	36	36	36	36
Багатомісний	51	32	32	32	32	32

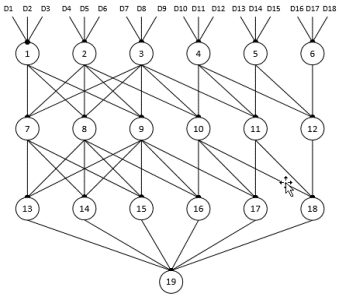


Результати дослідження для другого графу

Тип графу	Кількість ОМ						
	1	2	3	4	5	6	7
Двомісний	166	88	64	53	53	53	53
Багатомісний	140	78	58	51	48	46	46



Приклад графу



Наукова новизна

- Запропоновано метод підвищення швидкодії потокових систем шляхом забезпечення виконання багатомісних операцій.
- Розроблено відповідну імітаційну модель, що дозволяє досліджувати процес обчислення в потокових системах при різних параметрах.

Висновки

Використовуючи розроблену систему імітації були проведені дослідження графів. Були побудовані їх двомісні та багатомісні версії, порівняна їх швидкодія при різній кількості обчислювальних модулів.

Максимальна швидкодія досягається коли кількість обчислювальних модулів дорівнює кількості потоків. Двомісні графи мають меншу швидкодію порівняно з багатомісними через більшу кількість операндів, що пересилаються в системі.

Застосування потокової системи з багатомісними функціями є максимально ефективним коли ці функції є складними і їх розбиття на двомісні призведе то сильного зростання кількості вершин в графі.

Апробація роботи

- Конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 (Київ, 2018 р.)
- VIII Міжнародна конференція студентів і молодих учених «Сучасні Інформаційні Технології 2018» (Одесса, 2018 р.).

Дякую за увагу!

Додаток 2

Лістинги програми

Лістинг 1. Допоміжного класу Util

```
public static class  Util
    {
        public static List<OrderDto> operandsOrders;

        public static string firstDataPath = "";
        public static string graphDataPath = "";
        public static string operandsOrderDataPath = "";

        public static  List<DataDto> GetDataForStartActors()
        {
            var res = new List<DataDto>();

            var jsonData = new List<DataDto>();
            string json = "";
            using (StreamReader r = new StreamReader(firstDataPath))
            {
                json = r.ReadToEnd();
                dynamic array = JsonConvert.DeserializeObject(json);
                jsonData = JsonConvert.DeserializeObject<List<DataDto>>(json);
                res = jsonData;
            }

            return res;
        }

        public static void GetOperandsOrder()
        {
            var res = new List<OrderDto>();

            var jsonData = new List<OrderDto>();
            string json = "";
            using (StreamReader r = new StreamReader(operandsOrderDataPath))
            {
                json = r.ReadToEnd();
                dynamic array = JsonConvert.DeserializeObject(json);
                jsonData = JsonConvert.DeserializeObject<List<OrderDto>>(json);
                res = jsonData;
            }

            operandsOrders =  res;
```

```
}

public static List<Actor> InitialiseActors(int[,] matrix)
{
    var size = matrix.GetLength(1);
    var actors = new List<Actor>();

    for (int i = 0; i < size; i++)
    {
        var actor = new Actor();
        actor.N = i + 1;
        for (int j = 0; j < size; j++)
        {
            if (matrix[i, j] != 0)
            {
                actor.T.Add(j + 1);
            }
        }

        var q = 0;
        for (int j = 0; j < size; j++)
        {
            if (matrix[j, i] != 0)
            {
                q++;
                actor.F.Add(j + 1);
            }
        }
        actor.Q = q;

        actors.Add(actor);
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (matrix[j,i] != 0)
            {
                actors[i].I = matrix[j, i];
            }
        }
    }
```

```
                var dataForStart = Util.GetDataForStartActors();

                foreach (var data in dataForStart)
                {
                    for (int i = 0; i < data.actors.Count; i++)
                    {
                        actors.FirstOrDefault(x => x.N == data.actors[i].n).F.Add(-
1);
                        actors.FirstOrDefault(x => x.N == data.actors[i].n).Q += 1;
                        actors.FirstOrDefault(x  =>  x.N  ==  data.actors[i].n).I  =
data.actors[i].op;
                    }
                }

                return actors;
            }

            public static Color choseOMStateColor(CMState state)
            {
                var color = Color.Tomato;
                if(state ==CMState.Processing)
                {
                    color = Color.Yellow;
                }
                else if(CMState.WaitForData == state)
                {
                    color = Color.Green;
                }
                return color;
            }
            public static List<Operand> InitialiseOperands()
            {
                var dataForStart = GetDataForStartActors();

                var operands = new List<Operand>();

                foreach (var data in dataForStart)
                {
                    for (int i = 0; i < data.actors.Count; i++)
                    {
                        var operand = new Operand();
                        Random rnd = new Random();
```

```
            operand.d = data.actors[i].d;
            operand.N = data.actors[i].n;

            var rand = GenerateRandom();
            operand.A = rand;
            StorageDeviceDispatcher.SetOperand(rand, data.value);

            operands.Add(operand);
        }
    }

    return operands;
}

public static int[,] ReadMatrixFromFile()
{
    StreamReader file = new StreamReader(graphDataPath);

    var lines = file.ReadToEnd().Split('\n');
    int size = lines[0].Trim().Split('\t').Count();
    var matrix = new int[size, size];

    int i = 0, j = 0;
    foreach (var row in lines)
    {
        j = 0;
        foreach (var col in row.Trim().Split('\t'))
        {
            matrix[i, j] = int.Parse(col.Trim());
            j++;
        }
        i++;
    }
    return matrix;
}

static int random= 0;
public static int GenerateRandom()
{
    random+=10;
    return random;
}
```

```
            public static string GetEnumDescription(Enum value)
            {
                FieldInfo fi = value.GetType().GetField(value.ToString());

                DescriptionAttribute[] attributes =
                    (DescriptionAttribute[])fi.GetCustomAttributes(
                    typeof(DescriptionAttribute),
                    false);

                if (attributes != null &&
                    attributes.Length > 0)
                    return attributes[0].Description;
                else
                    return value.ToString();
            }

        }
}
```

Лістинг 2. Списку станів обчислювального модуля

```
    public enum CMState
    {
        [Description("Вільний")]
        Free,
        [Description("Очікую на дані")]
        WaitForData,
        [Description("Обчислюю")]
        Processing
}
```

Лістинг 3. Класу DataFlowSystem

```
    public class DataFlowSystem
    {
            int lastVertexNumber = 0;
            public SFK sfk;
            public CommandBuffer buffer;
            public List<ComputingModule> computingModules;
            public double? result = null;

            public void Initialise(int[,] matrix, int OMAmount)
            {
                buffer = new CommandBuffer();

                computingModules = new List<ComputingModule>();
```

```
            while (OMAmount != 0)
            {
                computingModules.Add(new ComputingModule());
                OMAmount--;
            }

            lastVertexNumber = matrix.GetLength(0);
            var actors = Util.InitialiseActors(matrix);
            StorageDeviceDispatcher.Initialise();
            var operands = Util.InitialiseOperands();
            Util.GetOperandsOrder();

            sfk = new SFK();
            sfk.actors = actors;
            sfk.availableOperands = operands;

            Functions.Initialise();
        }

        public void Run()
        {
            var k = 500;
            while (k!=0 && result==null)
            {
                sfk.SendToBuffer(buffer);
                foreach (var cm in computingModules)
                {
                    cm.GetCommandIfFree(buffer);
                    var                 numberOfExecutedVertex                 =
cm.DoTick(sfk.availableOperands);
                    if(numberOfExecutedVertex == lastVertexNumber)
                    {
                        result = cm.result;
                        break;
                    }
                }
                k--;
            }
        }

        public bool NextStep()
        {
            var isFinish = false;
```

```
                if( result == null)
                {
                    sfk.availableOperands.ForEach(x  =>  x.isAvailableOnlyNextTact  =
false);
                    foreach (var cm in computingModules)
                    {
                        cm.GetCommandIfFree(buffer);
                        var                 numberOfExecutedVertex                =
cm.DoTick(sfk.availableOperands);
                        if (numberOfExecutedVertex == lastVertexNumber)
                        {
                            result = cm.result;
                            isFinish = true;
                            break;
                        }
                    }

                    sfk.SendToBuffer(buffer);

                }
                return isFinish;
            }
    }
```

Лістинг 4. Головної форми програми

```
    public partial class Form1 : Form
    {
            public class MemoryCellViev
            {
                public double value;
                public int key;
                public int viewKey;
            }
            DataFlowSystem dataFlowSystem;
            int ticks = -1;
            bool isFinished = false;
            List<MemoryCellViev> viewStorage = new List<MemoryCellViev>();

            public Form1(int OMAmount)
            {
                InitializeComponent();

                var matrix = Util.ReadMatrixFromFile();
```

```
    dataFlowSystem = new DataFlowSystem();
    dataFlowSystem.Initialise(matrix, OMAmount);
    //dataFlowSystem.Run();
    RefreshOnTick();
}

private void button1_Click(object sender, EventArgs e)
{
    if (!isFinished)
    {
        isFinished = dataFlowSystem.NextStep();
        RefreshOnTick();
        if (isFinished == true)
        {
            End();
        }
    }
}

private void listView1_SelectedIndexChanged(object sender, EventArgs e)
{

}

private void RefreshOnTick()
{
    ReShowStorageValues();
    ReShowTicks();
    ReShowSFK();
    ReShowStorageDevice();
    ReShowOMs();
    ReShowBuffer();
}
private void ReShowTicks()
{
    ticks++;
    textBox2.Text = ticks.ToString();
}

private void ReShowOMs()
{
    listView4.Items.Clear();
```

```
                foreach (var comm in dataFlowSystem.computingModules)
                {
                    var backColor = Util.choseOMStateColor(comm.state);

                    if (comm.executingCommand != null)
                    {
                        var l1 = new
ListViewItem(comm.executingCommand._N.ToString());
                        l1.BackColor = backColor;

l1.SubItems.Add(Util.GetEnumDescription((CMState)comm.state).ToString());
                        l1.SubItems.Add(comm.ticks.ToString());
                        l1.SubItems.Add(comm.neededTime.ToString());
                        listView4.Items.Add(l1);
                    }
                    else
                    {
                        var l1 = new ListViewItem(" ");
                        l1.BackColor = backColor;

l1.SubItems.Add(Util.GetEnumDescription((CMState)comm.state).ToString());
                        listView4.Items.Add(l1);
                    }
                }
            }

            private void ReShowBuffer()
            {
                listView3.Items.Clear();
                foreach (var comm in dataFlowSystem.buffer.commands)
                {
                    var inputs = string.Join(",", comm.A.Select(x =>
x.ToString()).ToList());
                    var outputs = string.Join(",", comm.T.Select(x =>
x.ToString()).ToList());


                    var l1 = new ListViewItem(comm._N.ToString());
                    l1.SubItems.Add(comm.I.ToString());
                    l1.SubItems.Add(viewStorage.FirstOrDefault(x => x.key ==
comm.A.OrderBy(a => a).FirstOrDefault()).viewKey.ToString());
                    l1.SubItems.Add(comm.A.Count.ToString());
                    //l1.SubItems.Add(inputs);
```

```
                l1.SubItems.Add(outputs);
                listView3.Items.Add(l1);

            }
        }

        private void ReShowSFK()
        {
            listView1.Items.Clear();
            foreach (var comm in dataFlowSystem.sfk.actors)
            {
                var l1 = new ListViewItem(comm.N.ToString());
                l1.SubItems.Add(comm.Q.ToString());
                var available = dataFlowSystem.sfk.availableOperands.Where(x =>
x.N == comm.N).Count();
                l1.SubItems.Add(available.ToString());
                var inputs = string.Join(",", comm.F.Select(x =>
x.ToString()).ToList());
                l1.SubItems.Add(inputs);

                l1.SubItems.Add(available.ToString());
                listView1.Items.Add(l1);
            }
        }

        private void ReShowStorageDevice()
        {
            listView2.Items.Clear();


            foreach (var comm in viewStorage)
            {
                var l1 = new ListViewItem(comm.value.ToString());
                l1.SubItems.Add(comm.viewKey.ToString());
                //l1.SubItems.Add(comm.key.ToString());
                listView2.Items.Add(l1);
            }
        }

        private void ReShowStorageValues()
        {
            viewStorage = new List<MemoryCellViev>();
```

```
                var storageList = StorageDeviceDispatcher.GetAllValues().OrderBy(x
=> x.Key).ToList();

                for (var i = 0; i < storageList.Count(); i++)
                {
                    viewStorage.Add(new MemoryCellViev() { value =
storageList[i].Value, key = storageList[i].Key, viewKey = i + 1 });
                }
            }

            private void button2_Click(object sender, EventArgs e)
            {
                //run to end
                while (!isFinished)
                {
                    isFinished = dataFlowSystem.NextStep();
                    RefreshOnTick();
                    if (isFinished == true)
                    {
                        End();
                    }
                }
            }
            private void End()
            {
                textBox1.Text = dataFlowSystem.result.ToString();
                var text = "Загальна кількість тіків = "+ ticks+"\n";
                text += "Звернень до памяті = " +
StorageDeviceDispatcher.invokes.ToString() + "\n";
                text += "Результат = " + dataFlowSystem.result.ToString() + "\n";

                MessageBox.Show(text, "Виконано");
            }
        }
```

Лістинг 5. Класу ComputingModule що імітує роботи обчислювального модуля

```
    public class ComputingModule
    {
            public CMState state;
            public double result = 0;
            public int ticks = 0;
            public int neededTime = 0;
```

```
        public int loading = 0;

        public Command executingCommand = null;
        public void GetCommandIfFree(CommandBuffer buffer)
        {
            if (state == CMState.Free && buffer.commands.Count>0)
            {
                state = CMState.WaitForData;
                executingCommand = buffer.commands.Dequeue();

                neededTime =
                    Functions.functionList["F"    +    executingCommand.I].Item3
+//time of operation executing
                        executingCommand.A.Count;//time of operands loading
            }
        }
        public int DoTick(List<Operand> operands)
        {
            var res = 0;

            if (state == CMState.WaitForData)
            {
                ticks++;
                if(ticks == executingCommand.A.Count)
                {
                    state = CMState.Processing;
                }
            }

            else if (state==CMState.Processing)
            {
                ticks++;
                loading ++;

                if (ticks == neededTime)
                {
                    //command has been executed
                    var operandsValues = new List<double>();
                    foreach (var operand in executingCommand.A)
                    {

operandsValues.Add(StorageDeviceDispatcher.GetOperand(operand));
                    }
```

```
                        result = Functions.functionList["F" +
executingCommand.I].Item2(operandsValues);
                        //create operands from result
                        foreach(var output in executingCommand.T)
                        {
                            var newOperand = new Operand();
                            newOperand.N = output;
                            newOperand.d = Util.operandsOrders.FirstOrDefault(x =>
                                x.from == executingCommand._N &&
                                x.to == output).d;

                            //set operand adres from 10 range in memory
                            var rand = Util.GenerateRandom();

                            var firstOperandForVertex = operands.LastOrDefault(x =>
x.N == newOperand.N);
                            if (firstOperandForVertex == null)
                                newOperand.A = rand;
                            else
                                newOperand.A = firstOperandForVertex.A + 1;

                            operands.Add(newOperand);
                            StorageDeviceDispatcher.SetOperand(newOperand.A,
result);
                        }
                        ticks = 0;
                        state = CMState.Free;
                        res = executingCommand._N;
                        executingCommand = null;
                    }
                }

                return res;
            }
        }
```

Лістинг 6. Класу SFK що імітує роботи середовища формування команд

```
    public class SFK
    {
        //public List<Command> commands;
        public List<Actor> actors;
        public List<Operand> availableOperands;
```

```
        public int loading =0 ;
        public SFK()
        {
            //commands = new List<Command>();
            actors = new List<Actor>();
            availableOperands = new List<Operand>();
        }
        public void SendToBuffer(CommandBuffer buffer)
        {
            Actor sendedActor = null;
            foreach(var actor in actors)
            {
                var availableOperandsForActor = availableOperands.Where(x => x.N ==
actor.N && x.isAvailableOnlyNextTact==false).ToList();
                if (availableOperandsForActor.Count == actor.Q)
                {
                    //send to buffer
                    var command = new Command();
                    command.A      =      availableOperandsForActor.OrderBy(x      =>
x.d).Select(x => x.A).ToList();
                    command.I = actor.I;
                    command.T = actor.T;
                    command._N = actor.N;
                    buffer.commands.Enqueue(command);
                    //delete used operands
                    availableOperands.RemoveAll(x => x.N == actor.N);
                    sendedActor = actor;
                    loading++;
                    break; //only one command per tick can be prepared
                }
            }
            if (sendedActor != null)
            {
                actors.Remove(sendedActor);
            }
        }
    }
```

Лістинг 7. Класу Form2 . Форма на якій задаються вхідні дані.

```
    public partial class Form2 : Form
    {
        public int OMAmount =1;
        public Form2()
```

```
        {
            InitializeComponent();
            textBox1.Text = OMAmount.ToString();
        }

        private        void        fileSystemWatcher1_Changed(object        sender,
System.IO.FileSystemEventArgs e)
        {

        }

        private void Form2_Load(object sender, EventArgs e)
        {

        }

        private void button1_Click(object sender, EventArgs e)
        {
            var frm = new Form1(OMAmount);
            frm.Location = this.Location;
            frm.StartPosition = FormStartPosition.Manual;
            frm.FormClosing += delegate { this.Show(); };
            this.Hide();
            frm.Show();

        }

        private void button2_Click(object sender, EventArgs e)
        {
            //operands order data
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                richTextBox1.Text = openFileDialog1.FileName;
                Util.operandsOrderDataPath = openFileDialog1.FileName;
            }
        }

        private void button3_Click(object sender, EventArgs e)
        {
            //graph file
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                richTextBox2.Text = openFileDialog1.FileName;
```

```
            Util.graphDataPath = openFileDialog1.FileName;
        }
    }

    private void button4_Click(object sender, EventArgs e)
    {
        //file for first data layer
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            richTextBox3.Text = openFileDialog1.FileName;
            Util.firstDataPath = openFileDialog1.FileName;
        }
    }

    private void button5_Click(object sender, EventArgs e)
    {
        OMAmount++;
        textBox1.Text = OMAmount.ToString();
    }

    private void textBox1_TextChanged(object sender, EventArgs e)
    {
    }

    private void button6_Click(object sender, EventArgs e)
    {
        OMAmount--;
        textBox1.Text = OMAmount.ToString();
    }
}
```

Лістинг 8. Класу OrderDto, що служить для читання даних з файлу про порядок входження операндів.

```
public class OrderDto
{
    public int from;
    public int to;
    public int d;
}
```

Лістинг 9. Класу Functions, що описує операції.

```
public static class Functions
{
```

```
        public static Dictionary<string, Tuple<int, Func<List<double>, double>,
int>> functionList;

        public static void Initialise()
        {
            functionList = new Dictionary<string, Tuple<int, Func<List<double>,
double>, int>>();

            functionList.Add("F1", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return Plus(a); }, 2));
            functionList.Add("F2", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return Minus(a); }, 2));
            functionList.Add("F3", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return Multiplication(a); }, 4));
            functionList.Add("F4", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return Division(a); }, 9));

            functionList.Add("F9", new Tuple<int, Func<List<double>, double>,
int>(1, (a) => { return Plus1(a); }, 4));
            //functionList.Add("F10", new Tuple<int, Func<List<double>, double>>(3,
(a) => { return Plus3(a); }));
            functionList.Add("F11", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return Plus2(a); }, 2));


            double s = functionList["F1"].Item2(new List<double> { 10, 10 });

            //simple graph functions
            functionList.Add("F21", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return simple1(a); }, 13));
            functionList.Add("F22", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return simple2(a); }, 8));
            functionList.Add("F23", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return simple3(a); }, 6));
            functionList.Add("F24", new Tuple<int, Func<List<double>, double>,
int>(1, (a) => { return simple4(a); }, 6));
            functionList.Add("F25", new Tuple<int, Func<List<double>, double>,
int>(2, (a) => { return simple5(a); }, 2));

            //dribno1 graph functions
            functionList.Add("F31", new Tuple<int, Func<List<double>, double>,
int>(3, (a) => { return dribno1(a); }, 4));
```

```
            functionList.Add("F32",  new  Tuple<int,  Func<List<double>,  double>,
int>(6, (a) => { return dribno2(a); }, 6));


            //implicit graph functions
            functionList.Add("F41",  new  Tuple<int,  Func<List<double>,  double>,
int>(3, (a) => { return implicit1(a); }, 4));
            functionList.Add("F42",  new  Tuple<int,  Func<List<double>,  double>,
int>(6, (a) => { return implicit2(a); }, 13));

            //paralelism graph functions
            functionList.Add("F51",  new  Tuple<int,  Func<List<double>,  double>,
int>(3, (a) => { return par1(a); }, 18));
            functionList.Add("F52",  new  Tuple<int,  Func<List<double>,  double>,
int>(3, (a) => { return par2(a); }, 4));
            functionList.Add("F53",  new  Tuple<int,  Func<List<double>,  double>,
int>(2, (a) => { return par3(a); }, 9));
            functionList.Add("F54",  new  Tuple<int,  Func<List<double>,  double>,
int>(2, (a) => { return par4(a); }, 2));
            functionList.Add("F55",  new  Tuple<int,  Func<List<double>,  double>,
int>(2, (a) => { return par5(a); }, 4));

        }

        static Func<List<double>, double> Plus1 = (list) =>
        {
            var a1 = list.ElementAt(0);
            return a1;
        };
        static Func<List<double>, double> Plus2 = (list) =>
        {
            var a1 = list.ElementAt(0);
            var a2 = list.ElementAt(1);
            return a1 + a2;
        };
        static Func<List<double>, double> Plus3 = (list) =>
        {
            var a1 = list.ElementAt(0);
            var a2 = list.ElementAt(1);
            var a3 = list.ElementAt(2);
            return a1 + a2 + a3;
        };
```

```
static Func<List<double>, double> Plus = (list) =>
{
    //2
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1 + a2;
};

static Func<List<double>, double> TriplePlus = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    var a3 = list.ElementAt(1);
    return a1 + a2 + a3;
};

static Func<List<double>, double> Minus = (list) =>
{
    //2
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1 - a2;
};

static Func<List<double>, double> Minus3 = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    var a3 = list.ElementAt(2);
    return a1 - a2 - a3;
};

static Func<List<double>, double> Multiplication = (list) =>
{
    //4
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1 * a2;
};

static Func<List<double>, double> Division = (list) =>
{
    //9
```

```
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1 / a2;
};

static Func<List<double>, double> Or = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);

    return (double)((int)a1 | (int)a2);
};

static Func<List<double>, double> And = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return (double)((int)a1 & (int)a2);
};

static Func<List<double>, double> Sqrt = (list) =>
{
    var a1 = list.ElementAt(0);
    return Math.Sqrt(a1);
};

static Func<List<double>, double> Square = (list) =>
{
    var a1 = list.ElementAt(0);
    return a1 * a1;
};

static Func<List<double>, double> Sin = (list) =>
{
    var a1 = list.ElementAt(0);
    return Math.Sin(a1);
};

static Func<List<double>, double> Cos = (list) =>
{
    var a1 = list.ElementAt(0);
    return Math.Sin(a1);
};
```

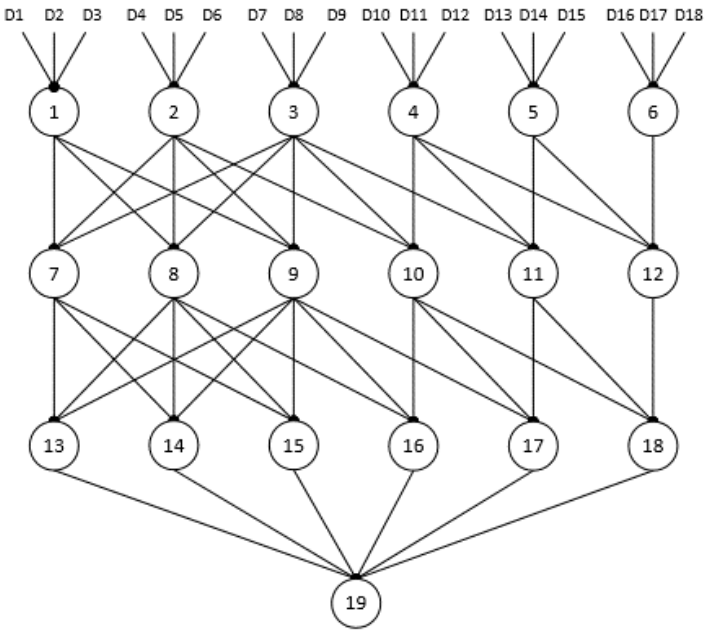
```
//simple graph functions
static Func<List<double>, double> simple1 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a * b * c + c / b;
};
static Func<List<double>, double> simple2 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a*b * c ;
};
static Func<List<double>, double> simple3 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a * b + b * c;
};

static Func<List<double>, double> simple4 = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1* a2 + 25;
};

static Func<List<double>, double> simple5 = (list) =>
{
    var a1 = list.ElementAt(0);
    var a2 = list.ElementAt(1);
    return a1 + a2;
};

//DRIBNOZERNUSTUJ graph functions
static Func<List<double>, double> dribno1 = (list) =>
{
    var a = list.ElementAt(0);
```

```
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a + b + c;
};
static Func<List<double>, double> dribno2 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    var d = list.ElementAt(3);
    var e = list.ElementAt(4);
    var f = list.ElementAt(5);
    return a + b + c +d+e+f;
};

//implicit graph functions
static Func<List<double>, double> implicit1 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a + b + c;
};
static Func<List<double>, double> implicit2 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a * b / c;
};

//PARALLESLIM graph functions
static Func<List<double>, double> par1 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
    var c = list.ElementAt(2);
    return a / b / c;
};
static Func<List<double>, double> par2 = (list) =>
{
    var a = list.ElementAt(0);
    var b = list.ElementAt(1);
```

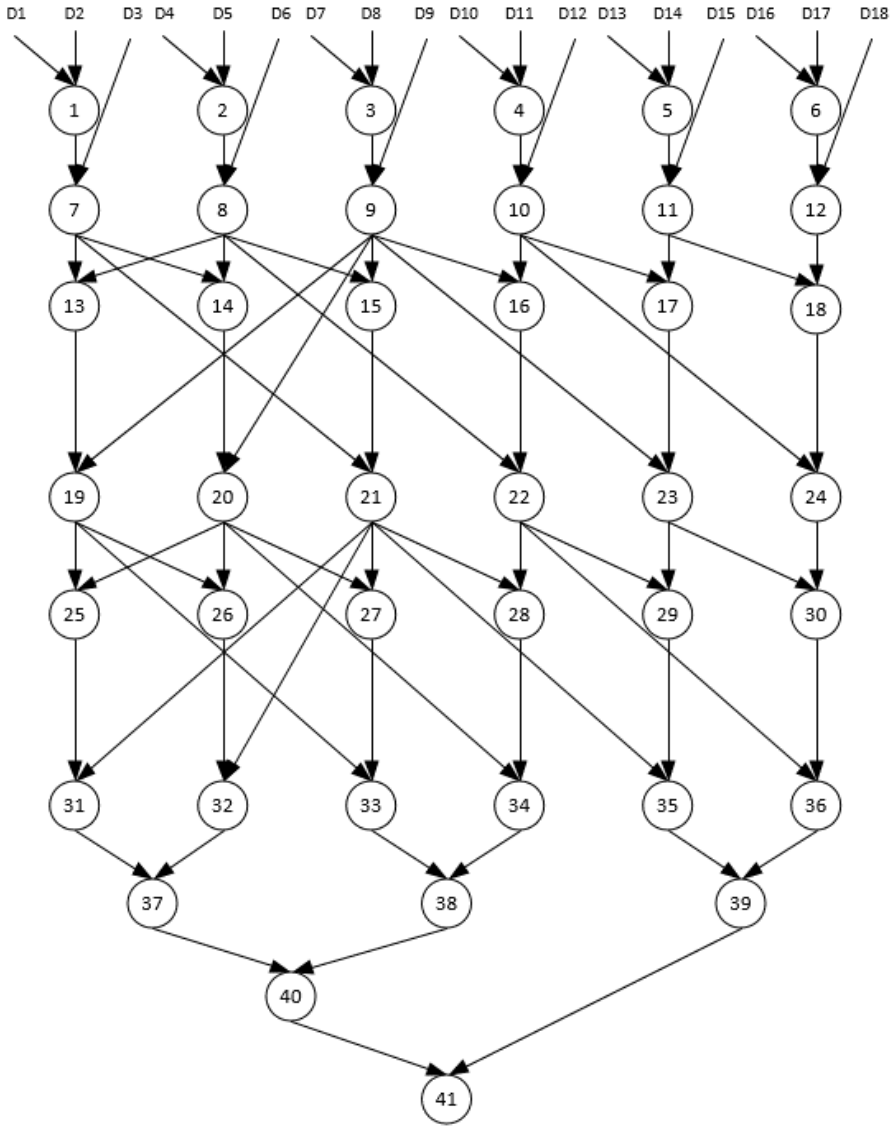
```
            var c = list.ElementAt(2);
            return a + b + c;
        };
        static Func<List<double>, double> par3 = (list) =>
        {
            var a = list.ElementAt(0);
            var b = list.ElementAt(1);
            return a / b;
        };
        static Func<List<double>, double> par4 = (list) =>
        {
            var a = list.ElementAt(0);
            var b = list.ElementAt(1);
            return a + b;
        };
        static Func<List<double>, double> par5 = (list) =>
        {
            var a = list.ElementAt(0);
            var b = list.ElementAt(1);
            var C = list.ElementAt(2);
            var D = list.ElementAt(3);
            return a + b+C+D;
        };
}
```

Додаток 3

Приклади графів



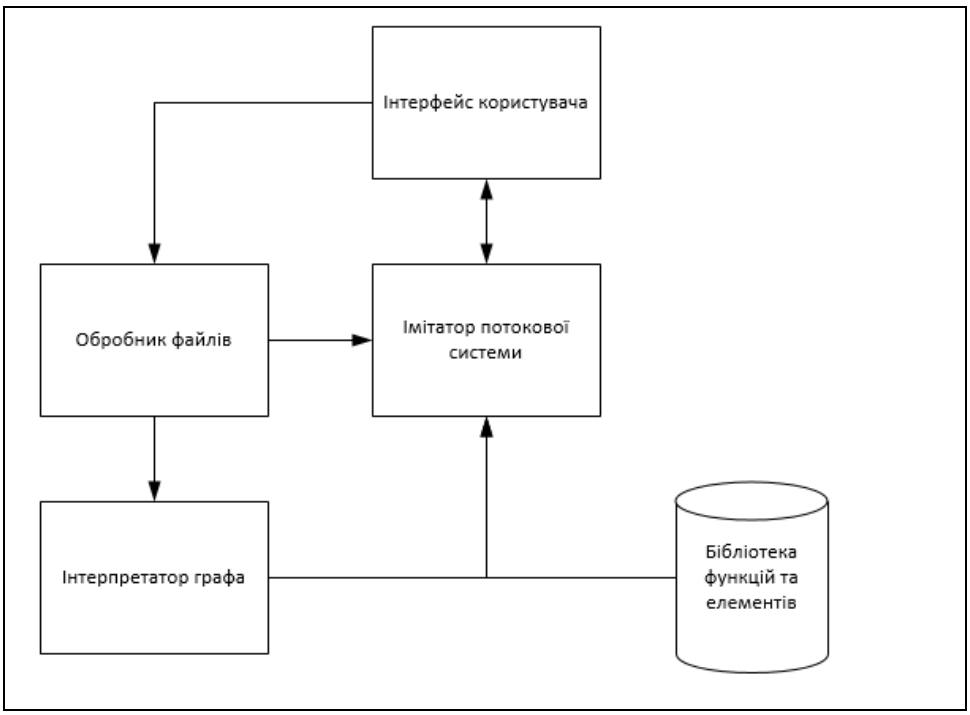
Граф 1. З використанням багатомісних операцій



Граф 2. Версія першого графу з використанням двомісних операцій

Додаток 4

Архітектура імітаційної системи


Інтерфейс користувача
Обробник файлів
Імітатор потокової системи
Інтерпретатор графа
Бібліотека функцій та елементів