МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук,
управління та адміністрування
Кафедра інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка методів та програмних засобів пошуку тупиків у паралельних Java-програмах»

Виконав студент 2 курсу
групи МІС-21зф
спеціальності 122 Комп'ютерні науки
Гуляк Роман Миколайович

Керівник к.ф.н., доцент
Цира Олександра Василівна

Консультант д.т.н., професор
Зайцев Дмитро Анатолійович

Рецензент к.т.н., доцент
Гнатовська Ганна Арнольдівна

Одеса 2022

АНОТАЦІЯ

Розробка методів та програмних засобів пошуку тупиків у паралельних Java-програмах. Гуляк Роман Миколайович.

Мета роботи – перевірка на коректність паралельних Java-програм.

Об’єкт дослідження – паралельні Java-програми, що використовують реалізацію стандарту MPI (Message-Passing Interface). Предмет дослідження – наявність тупиків в паралельних Java-програми. Метод дослідження – створення моделі програми у вигляді сітки Петрі.

У першому розділі виконано огляд специфікації MPI і його реалізацій. Другий розділ містить опис розробки та реалізації аналізатора. У третьому розділі показано кілька прикладів побудованих сіток Петрі та виконано їхній аналіз. Також наведено рекомендації щодо влаштування тупиків у програмах MPI.

У результаті роботи розроблено сітку Петрі та метод її побудови за заданою програмою MPI. Розроблена сітка Петрі націлена на перевірку, що у вихідній програмі MPI відсутні тупики. Розроблено статичний аналізатор коду, який автоматично виконує побудову сітки Петрі. У реалізації аналізатора використовуються як традиційні підходи з області розробки компіляторів, так і нові підходи, спеціалізовані під поточну задачу. Розроблені моделі показали свою застосовність як на простому прикладі, у якому тупик відбувається під час кожного запуску програми, так і на складнішому прикладі, у якому тупик відбувається недетерміновано.

Магістерська кваліфікаційна робота: 89 стор., 30 рис., 8 табл., 24 джерела.

Ключові слова: ГРАФ ПОТОКУ УПРАВЛІННЯ, ДОМІНАТОР, ПАРАЛЕЛЬНА СИСТЕМА, СІТКА ПЕТРІ, СТАТИЧНИЙ АНАЛІЗАТОР КОДУ, ТУПИК, JAVA, MPI, MPJ EXPRESS, SCALA, SSA, TINA.

SUMMARY

Development of methods and software tools for finding deadlocks in parallel Java programs. Roman N. Guliak.

The goal of this thesis is to verify the correctness of parallel Java programs.

The object of research are parallel Java programs that are using an implementation of MPI (Message-Passing Interface) standard. The subject of reasearch is presence of deadlocks in parallel Java-programs. Research method is creation of program model in the form of Petri net.

Section 1 contains a review of MPI (Message-Passing Interface) specification and its implementation. In Section 2 a development and an implementation of analyzer are described. In Section 3, several examples of Petri nets for code fragments are presented and analyzed. Also, deadlock prevention recommendations are provided for MPI programs.

As a result of this work, a Petri net and a method of its construction for a given MPI program have been developed. The developed Petri net is aimed at checking that there are no deadlocks in the original MPI program. A static code analyzer has been developed. The analyzer automatically performs Petri net construction. The implementation of the analyzer uses both traditional approaches from the field of compiler development and new approaches specialized for the current task. The developed models showed their applicability both on a simple example, in which deadlock occurs every time the program is run, and on a more complex example, in which deadlock occurs nondeterministically.

Master’s thesis: 89 pages, 30 pictures, 8 tables, 24 sources.

CONTROL FLOW GRAPH, DEADLOCK, JAVA DOMINATOR, MPI, MPJ EXPRESS, PARALLEL SYSTEM, PETRI NET, SCALA, SSA, STATIC CODE ANALYZER, TINA.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

Аналіз потоку даних – це різновид аналізу програм, що дає змогу отримати інформацію про те, яким чином дані переміщуватимуться під час виконання програми.

Граф потоку управління – орієнтований граф, що моделює потік управління між базовими блоками в програмі.

Тупикова ситуація (тупик) – ситуація, при якій процеси не можуть продовжити роботу через взаємну залежність.

Часткове обчислення – це техніка оптимізації програм, яка полягає у створенні спеціалізованої версії програми на підставі частини наданих даних.

ОС	– операційна система.
ЦП	– центральний процесор.
AST	– abstract syntax tree.
HIL	– higher level intermediate language.
LIL	– lower level intermediate language.
MIMD	– multiple instruction, multiple data.
MPI	– Message Passing Interface.
SSA	– static single assignment.

ВСТУП

На сьогодні розподілені та паралельні програмні системи відіграють величезну роль у найрізноманітніших галузях застосування, і в науці, і в промисловості. Паралельні програми вже протягом десятиліть розробляються для суперкомп'ютерів. Однак розробка паралельних програм набуває дедалі більшої важливості і для персональних комп'ютерів. Це пов'язано з тим, що темп зростання продуктивності ядер процесора знизився. Тепер для отримання зростання продуктивності програм необхідне задіяння багатьох ядер процесора.

Одним з інструментів розробки паралельних програмних систем є MPI. MPI дозволяє розбивати програму на процеси, які обмінюються між собою повідомленнями. Під час такого обміну повідомленнями можуть виникати небажані ситуації, які відомі як тупикові ситуації або просто тупики. Ці ситуації відбуваються, коли програма не може продовжити свою роботу, оскільки процеси перебувають у взаємній залежності. Тупики нелегко виявити під час виконання, оскільки не завжди зрозуміло, які саме процеси заблоковано.

Звідси виникає необхідність в аналізаторах коду, які б виконували автоматичний пошук тупиків у програмах MPI. Такі інструменти бувають динамічними, які запускають програму, і статичними, які тільки аналізують її вихідний код. Динамічні аналізатори мають більшу точність, а перевага статичних аналізаторів полягає в тому, щоб вони не запускають програму.

Тому метою цієї роботи є знаходження альтернативного методу статичного аналізу програм MPI, який би мав кращу точність. Можливою сферою застосування є серйозні програми MPI в галузі високопродуктивних обчислень.

Розроблено методику моделювання програм MPI сітками Петрі, які мають можливість адекватно представляти поведінку паралельних систем.

Також розроблено статичний аналізатор коду, який створює сітку Петрі за заданою програмою Java, що використовує реалізацію MPI. У реалізації аналізатора використовуються як традиційні підходи з області розробки компіляторів, так і нові підходи, спеціалізовані під поточне завдання.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис MPI

1.1.1 Опис стандарту та реалізацій

MPI (Message Passing Interface, інтерфейс передавання повідомлень) являє собою специфікацію інтерфейсу бібліотек інтерфейсу передавання повідомлень. Програми MPI складаються з процесів, які взаємодіють між собою за допомогою обміну повідомлень. Кожен процес має свій адресний простір, процеси працюють за принципом MIMD (multiple instruction, multiple data – множинний потік команд, множинний потік даних). Головним змістом стандарту MPI є процедури або функції, які можуть бути викликані з мов програмування FORTRAN і C.

При створенні стандарту MPI, його автори переслідували такі цілі:

– досягнення ефективної комунікації між процесами;
– зручний інтерфейс для мов C і FORTRAN;
– семантика інтерфейсу має бути незалежною від мови;
– надійний інтерфейс комунікації: проблеми з помилками комунікації мають вирішуватися на рівні підсистеми комунікації, а не користувачем.

Творці MPI намагалися використати найпривабливіші особливості систем передавання повідомлень, що існували на той момент. На MPI вплинули такі системи як NX/2, p4, PARMACS, Zipcode, Chimp, PVM. У стандартизації MPI брали участь представники великих постачальників паралельних комп'ютерів, наукові дослідники, урядові лабораторії, а також представники промисловості. Перша версія стандарту MPI вийшла 1994 року під назвою «A Message-Passing Interface Standard Version 1.0». У 1995 році було опубліковано стандарт MPI 1.1. MPI 1.1 описував такий функціонал як передача та отримання повідомлень між окремими процесами, колективна взаємодія процесів, робота з групами процесів і топології процесів.

Стандарт MPI 2.0 з'явився в 1997 році. Він містив у собі також стандарт MPI 1.2, який мав невеликі відмінності від версії 1.1. MPI 2.0 мав такі нововведення розширена модель обчислень, динамічне створення процесів, односторонння комунікація, паралельне введення/виведення.

Після мінорних версій 2.1 (2008 рік) і 2.2 (2009 рік) був випущений стандарт MPI 3.0. Він містив неблокуючі версії колективних операцій і розширення односторонніх комунікацій. Стандарт MPI 3.1 (2015 рік) містить правки та уточнення.

І, нарешті, останнім на сьогоднішній день є стандарт MPI 4.0, який був випущений у 2021 році. Найбільшими змінами є використання великого цілочисельного типу в багатьох процедурах для подолання обмежень простого цілочисельного типу, розділені комунікації та поліпшення для обробки помилок.

Нині широко використовуються кілька реалізацій MPI. MPICH від Argonne National Laboratories та OpenMPI є двома найбільш поширеними. Постачальники обладнання часто мають спеціалізовані версії однієї з цих двох реалізацій для своїх платформ.

У цій роботі буде використовуватися MPJ Express [1]. MPJ Express – це бібліотека передавання повідомлень Java з відкритим вихідним кодом, яка дає змогу розробникам додатків писати та виконувати паралельні додатки для багатоядерних процесорів і обчислювальних кластерів/хмар. Далі ми будемо використовувати терміни MPI і MPJ Express як синоніми.

1.1.2 Актуальність використання MPI

Протягом більше двох десятиліть мікропроцесори, засновані на одиничних центральних процесорах, проявляли особливі поліпшення в продуктивності та зниження вартості в комп'ютерних додатках [2]. Ці поліпшення дали можливість додавати більше функціональності в програмне забезпечен-

ня, створювати кращі призначені для користувача інтерфейси і виробляти більше корисних результатів. Однак, з 2003 року ця тенденція почала сповільнюватися через споживання енергії і проблем з відведенням тепла, що обмежило збільшення частоти процесора і рівень продуктивної діяльності, яка може виконуватися в кожен такт в рамках одного ЦП. Відтоді практично всі постачальники мікропроцесорів перейшли на моделі, в яких для збільшення обчислювальної потужності в кожному чіпі використовується кілька процесорних блоків, званих процесорними ядрами. Цей перехід справив величезний вплив на спільноту розробників програмного забезпечення. Тепер для отримання приростів продуктивності в додатках, недостатньо створювати традиційні послідовні програми, що працюють тільки на одному процесорі. Натомість, для задіяння поліпшень у комп'ютерному обладнанні, програми мають складатися з кількох потоків виконання, які взаємодіють між собою для того, щоб виконати роботу швидше.

Практика паралельного програмування не є новою. Спільнота високопродуктивних обчислень протягом десятиліть розробляє паралельні програми. Ці програми зазвичай працюють на великих і дорогих комп'ютерах. Більшість додатків, що вимагають продуктивності, традиційно знаходяться в науковій сфері. Але прогнозується, що додатки штучного інтелекту і машинного навчання стануть основними користувачами великомасштабних обчислень [3]. Деякі приклади таких додатків включають:

– моделювання пожеж для допомоги пожежним командам;
– моделювання цунамі та штормових нагонів від ураганів;
– розпізнавання голосу для комп'ютерних інтерфейсів;
– моделювання поширення вірусу і розробка вакцини;
– моделювання кліматичних умов протягом десятиліть і століть;
– розпізнавання зображень для безпілотних автомобілів.

MPI успішно застосовується у високопродуктивних обчисленнях. Додатки, написані в MPI, успішно працюють на кластерних обчислювальних

системах з більш ніж 100 тис. вузлів [2].

1.2 Огляд програмного інтерфейсу MPJ Express

Далі розглянуті базові поняття MPI. Як було згадано, програма MPI складається з процесів, що взаємодіють. Процеси можуть належати до груп. Групи – це впорядкована множина процесів. Кожен процес у групі асоціюється з цілочисельним рангом. Нумерація рангів починається з нуля. На базі груп створюються комунікатори. Кожен комунікатор містить групу учасників; ця група завжди включає локальний процес. Відправник і одержувач повідомлення ідентифікуються рангом процесу всередині цієї групи.

Є особливий комунікатор – MPI_COMM_WORLD. Це початковий комунікатор, що складається з усіх локальних процесів, з якими процес може взаємодіяти після ініціалізації, включно з самим собою. У пакеті MPJ Express аналогічним комунікатором є статичне поле COMM_WORLD у класі mpi.MPI.

1.2.1 Операції загального призначення

В стандарті MPI є операції загального призначення, не пов'язані з передачею повідомлень.

Однією з таких операцій є Init. У MPJ Express цей метод оголошений у класі mpi.MPI і має таке визначення:

```
public static java.lang.String[] Init(java.lang.String[] argv) throws
MPIException
```

Аналогом у MPI є операція MPI_Init. Цей метод проводить ініціалізацію оточення MPI. Програми, що використовують MPJ Express, повинні починатися з виклику Init. Виклик більшості інших операцій має сенс тільки

після виклику Init. Також ми бачимо, що Init викидає виняток MPIException. Це загальне виключення, яке викидають більша частина методів бібліотеки MPJ Express.

Наступною важливою операцією є Finilize, яка також визначена в mpi.MPI:

```
public static void Finalize() throws MPIException
```

Аналогом у MPI є операція MPI_Finilize. Ця операція очищує стан MPI. Якщо програма завершується нормально, а не через виклик Abort чи іншу непередбачувану ситуацію, то кожен процес має викликати Finilize перед виходом.

Операція Abort належить до класу mpi.Comm і має таке оголошення:

```
public void Abort(int errorcode) throws MPIException
```

Аналогом у MPI є функція MPI_Abort. Ця операція намагається перервати всі завдання в групі комунікатора. Код помилки errorrcode буде передано в середовище виконання.

Операція Size і Rank належать класу mpi.Comm і оголошені таким чином:

```
public int Size() throws MPIException
public int Rank() throws MPIException
```

Аналогами в MPI є відповідно функції MPI_Comm_size і MPI_Comm_rank. Size повертає кількість процесів у групі комунікатора. Rank повертає ранг процесу в групі комунікатора.

1.2.2 Операції типу «точка-точка»

Надсилання та отримання повідомлень процесами є базовим механізмом комунікації в MPI. Базовими операціями є Send і Recv. Наведемо простий приклад використання цих операцій.

```
import mpi.MPI;

public class MpiSendRecv {
  public static void main(String[] args) {
    int rank = MPI.COMM_WORLD.Rank();
    int[] buffer = {0, 2, 3, 4};
    if (rank == 0) {
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
    } else if (rank == 1) {
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
    }
  }
}
```

У цьому прикладі ми бачимо як процес із рангом 0 (rank = 0) передає повідомлення процесу з рангом 1; взаємодія відбувається всередині комунікатора COMM_WORLD. Перші чотири аргументи для Send – це буфер із даними, зміщення від початку буфера, кількість елементів і тип даних. Крім самих даних передається ще й так званий *конверт повідомлення*. Цей конверт вказує одержувача повідомлення, а також містить додаткову інформацію, яка допоможе операції Recv вибрати певне повідомлення. Останні два аргументи для Send – це ранг одержувача і *тег* повідомлення. Тег являє собою ідентифікатор, який допомагає відрізнити повідомлення між собою і він є частиною конверта повідомлення.

В свою чергу, процес із рангом 1 викликає операцію отримання – Recv. Повідомлення вибирається відповідно до його конверта, потім дані повідомлення зберігаються в буфер отримання. І знову, перші чотири аргументи для Recv вказують інформацію про буфер: сам буфер, зміщення, кількість елементів і тип. Останні два аргументи вказують ранг відправника і тег.

Операції типу «точка-точка» можуть бути:

– операції, що блокуються – виклик не завершується, поки не стане безпечно використовувати буфер;
– операції, що не блокуються – виклик завершується відразу.

Аналізатор буде обробляти тільки операції, що блокуються.

Далі детально розглядаються методи Send і Recv. Метод Send належить до класу mpi.Comm і має таке визначення:

```
public void Send(java.lang.Object buf, int offset, int count, Datatype
datatype, int dest, int tag) throws MPIException
```

Аналогом у MPI є функція MPI_Send. У табл. 1 надані параметри методу Send.

Таблиця 1 – Параметри методу Send

Ім’я параметру	Опис параметру
buf	буферний масив, який підлягає відправленню
offset	зміщення від початку масиву
count	кількість елементів для відправлення
datatype	тип даних кожного елементу в масиві
dest	ранг одержувача
tag	тег повідомлення

Операція Send – це операція, що блокується. Фактичний аргумент, пов'язаний з buf, має бути одновимірним масивом. Зсув значення є нижнім індексом у цьому масиві, що визначає позицію першого елемента повідомлення. Довжина повідомлення визначається параметром count і вказує кількість елементів заданого типу, а не кількість байт.

Метод Recv також належить до класу mpi.Comm і має таке визначення:

```
public Status Recv(java.lang.Object buf, int offset, int count, Datatype
datatype, int source, int tag) throws MPIException
```

Аналогом у MPI є функція MPI_Recv. Це також операція, що блокується У табл. 2 подано параметри методу Recv. Фактичний аргумент, пов'язаний з buf, також має бути одновимірним масивом. Метод Recv повертає об'єкт класу mpi.Status, який описано нижче.

Таблиця 2 – Параметри методу Recv

Ім’я параметру	Опис параметру
buf	буферний масив, у який будуть записані дані
offset	зміщення від початку масиву
count	кількість елементів для отримання
datatype	тип даних кожного елементу в масиві
source	ранг відправника
tag	тег повідомлення

Клас mpi.Status містить інформацію про статус отримання повідомлення. Розглянемо найважливіші поля цього класу:

```
public int source;
public int tag;
```

Поле source містить ранг відправника, а tag – тег отриманого повідомлення. Причина, з якої ця інформація може бути корисною, така. При виклику Recv користувач може вказати константу mpi.MPI.ANY_SOURCE як ранг відправника. У цьому випадку, повідомлення буде вибиратися від будь-якого відправника. Для того, щоб дізнатися, від якого процесу було отримано

повідомлення, користувач може скористатися полем source. Аналогічно, користувач може вказати mpi.MPI.ANY_TAG як тег повідомлення, і пізніше скористатися полем tag в об'єкті класу mpi.Status.

Операції Send і Recv маються такі відмінності:

– для виконання операції Send обов'язково потрібно вказувати одержувача;
– для виконання операції Recv вказувати конкретного відправника необов'язково.

1.2.3 Колективні операції

Колективні операції – це операції, у яких приймають участь група або групи процесів [4]. Одним із ключових аргументів для колективних операцій є комунікатор, який визначає групу процесів. Синтаксис і семантика колективних операцій узгоджується з операціями типу «точка-точка». Тому в колективних операціях застосовні ті самі типи даних, що й в операціях типу «точка-точка». Кілька колективних операцій мають один вихідний процес або процес, що отримує, який має назву *кореневий процес*. Деякі аргументи операцій мають значення тільки для кореневого процесу. Колективні операції не мають тег як аргумент.

Операції Bcast і Barrier є одними з найбільш часто використовуваних колективних операцій, згідно з [5].

Метод Bcast визначено в класі mpi.Intracomm:

```
public void Bcast(java.lang.Object buf, int offset, int count, Datatype type,
int root) throws MPIException
```

Аналогом у MPI є функція MPI_Bcast. Цей метод поширює дані з кореневого процесу, який позначений параметром root, іншим процесам у групі комунікатора. Опис параметрів методу Bcast наданий у табл. 3.

Таблиця 3 – Параметри методу Bcast

Ім’я параметру	Опис параметру
buf	буферний масив
offset	початкове зміщення в буфері
count	кількість елементів масиву
datatype	тип даних кожного елементу в масиві
root	ранг кореневого процесу

На рис. 1 показано, як кореневий процес із рангом 0 відправляє масив {11, 22, 33, 44, 55} процесам із рангами 1, 2 і 3.

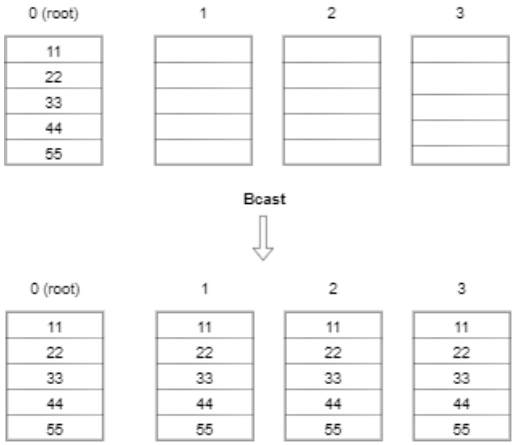

Рисунок 1 – Результат роботи операції Bcast

Метод Barrier належить класу mpi.Intracomm і визначений таким чином:

```
public void Barrier() throws MPIException
```

Аналогом у MPI є функція MPI_Barrier. Метод Barrier блокує процес,

що викликає, доти, доки всі процеси в групі комунікатора також не викличуть його. Цей метод буває корисний, коли потрібно синхронізувати всі процеси групи в одній точці.

1.2.4 Компіляція та запуск програм MPJ Express

Нижче розглянута програма, що використовує MPJ Express.

```
import mpi.MPI;
public class HelloMPJ {
  public static void main(String[] args) {
    MPI.Init(args);
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    System.out.printf("Process %d of %d\n", rank, size);
    MPI.Finalize();
  }
}
```

Ця проста програма здійснює ініціалізацію MPI, отримує кількість процесів у групі та ранг поточного процесу і виводить цю інформацію в консоль.

Для роботи з MPJ Express необхідно встановити цей пакет, який можна завантажити з офіційного сайту [6].

Якщо пакет установлено за шляхом, зазначеним у змінній оточення MPJ_HOME, то в ОС Windows компілювати програму, що використовує MPJ Express, можна таким чином:

```
javac -cp %MPJ_HOME%/lib/mpj.jar HelloMPJ.java
```

У результаті створюється файл HelloMPJ.class, який потім можна запустити таким чином:

```
%MPJ_HOME%/bin/mpjrun.bat -np 4 HelloMPJ
```

Скрипт mpjrun.bat необхідний для того, щоб розподілити програму за

процесами. Аргумент командного рядка -np 4 вказує, що кількість процесів має бути чотири. Результат запуску може бути таким:

```
MPJ Express (0.44) is started in the multicore configuration
Process 2 of 4
Process 0 of 4
Process 3 of 4
Process 1 of 4
```

1.3 Проблема тупиків у програмах MPI

У цьому підрозділі описується проблема виникнення тупиків під час використання блокувальних викликів операцій типу «точка-точка» – Send і Recv. *Тупикова ситуація* або *тупик* – це ситуація, при якій процеси не можуть продовжити роботу через взаємну залежність.

Під час відправлення з блокуванням, реалізація MPI часто здійснює запис вихідного повідомлення в буфер [5]. У цьому випадку операція відправлення може завершитися і до того, як буде викликана відповідна операція отримання. З іншого боку, буферного простору може виявитися недостатньо. Тоді відправлення не буде завершено доти, доки не буде викликано відповідну операцію отримання і дані не будуть передані одержувачу.

При отриманні з блокуванням, блокування виклику операції відбувається в будь-якому разі – операція завершується тільки після того, як переданий буфер містить отримане повідомлення. Отримання може завершитися і до того, як завершиться відповідне відправлення.

Стандарт MPI описує різні гарантії виконання операцій Send і Recv. Далі розглянуті найважливіші для цієї роботи.

Припустимо, що відправник викликає Send двічі для одержувача. Одержувач викликає операцію Recv, яка відповідає обом повідомленням. У цьому випадку одержувач гарантовано отримає перше повідомлення.

Якщо процес надсилає два повідомлення послідовно і вони обидва відповідають виклику Recv у процесі одержувача, то друга операція Send не мо-

же початися, поки не завершиться перша.

Якщо два процеси відсилають повідомлення третьому процесу, а операція отримання в третьому процесі відповідає повідомленням з обох процесів, то MPI не гарантує, яке саме повідомлення буде отримано третім процесом.

Далі представлені три приклади використання операцій відправлення та отримання.

Приклад 1. У цьому прикладі на стороні процесу 0 відбувається послідовно відправлення та отримання. На стороні процесу 1 – отримання та надсилання.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

Цей фрагмент виконається успішно в будь-якому разі, навіть якщо буферного простору недостатньо.

Приклад 2. Цей приклад відрізняється від попереднього прикладу тим, що процес 0 спочатку викликає Recv, потім Send.

```
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

У цьому фрагменті завжди буде відбуватися тупик. Операція Recv у процесі 0 може завершитися, тільки якщо буде викликано Send від процесу 1. Recv у процесі 1 має завершитися перед своїм викликом Send, але це може статися, тільки якщо буде викликано Send від процесу 1. Таким чином відбувається взаємне очікування і блокування обох процесів.

Приклад 3. У цьому прикладі обидва процеси викликають спочатку Send і потім Recv.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
}
```

Цей фрагмент може виконатися успішно, тільки якщо буде застосовуватися збереження повідомлення в буфер під час надсилання. У цьому випадку виклик Send завершиться і буде викликано операцію Recv, для якої буде знайдено відповідне повідомлення. Однак, якщо збереження в буфер не відбудеться, то трапиться тупик, як і в попередньому прикладі.

1.4 Огляд наявних аналізаторів MPI

Аналізатори, що перевіряють коректність програм MPI, можна розділити на дві групи:

– статичні;

– аналіз часу запуску.

Статичні аналізатори працюють тільки з вихідним кодом програми, витягуючи з нього потрібні властивості. Потім відбувається перевірка цих властивостей на коректність. У цьому разі запуску програми не відбувається.

Під час використання аналізаторів часу запуску, навпаки, відбувається запуск самої програми. Для знаходження помилкових станів виконується перехоплення викликів деяких операцій. Інформація про виклики обробляється і таким чином аналізатор може робити висновки про коректність цільової програми.

Одним зі статичних аналізаторів для програм MPI є MPI-Checker [7]. Він заснований на інфраструктурі Static Analyzer компілятора Clang. MPI-

Checker витягує інформацію про програму, використовуючи тільки абстрактне синтаксичне дерево програми. Одна частина перевірок спрямована на виклики, що блокуються. Друга частина перевірок переконується в коректності типів для викликів MPI. Наприклад, MPI-Checker повідомляє помилку, якщо для виклику MPI_Send переданий буфер має тип double*, а тип даних вказано MPI_INT.

MPI-SV [8] також можна віднести до статичних аналізаторів. MPI-SV здійснює верифікацію операцій MPI, що не блокуються. Він використовує символьне виконання (symbolic execution) і перевірку моделей (model checking). MPI-SV заснований на KLEE – двигуні символьного виконання. Символьне виконання [9] – ця техніка аналізу, за якої дійсні вхідні дані програми замінюються довільними символьними значеннями. Потім програма інтерпретується відповідно до семантики мови програмування. При цьому дані в програмі можуть бути представлені формулами від вхідних символьних даних. Зауважимо, що цей підхід дає змогу не виконувати програму, а лише інтерпретувати її. Перевірка моделей – це техніка автоматичного доведення, що дана модель будь-якої системи відповідає заданим властивостям. Перевірка моделей працює шляхом систематичного дослідження всіх станів моделі.

У [10] автори показують аналізатор часу запуску. Цей інструмент запускає програму MPI і відстежує послідовність викликів функцій MPI. З цієї послідовності викликів вибудовується формула при використанні особливої логічної схеми кодування. Потім відбувається перевірка на здійсненність цієї формули. Цей аналізатор націлений на пошук тупиків.

1.5 Огляд сіток Петрі

1.5.1 Огляд структури сітки Петрі

Сітки Петрі складаються з 4 елементів: позиції, переходів, дуг і фішок (див. рис. 2). Позиції зображуються колами, переходи – прямокутниками. Дуги можуть з'єднувати позиції з переходами або переходи з позиціями, проте не можуть з'єднувати позиції з позиціями і переходи з переходами. Фішки розташовуються всередині позицій і зображуються жирними крапками.

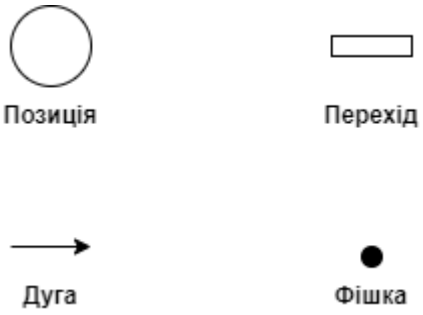

Рисунок 2 – Елементи сітки Петрі

Якщо дуга спрямована від позиції до переходу, то така позиція називається вхідною для цього переходу. Якщо дуга спрямована від переходу до позиції, то позиція називається вихідною.

Під час спрацьовування переходу фішки видаляються із вхідних позицій, а у вихідні позиції додаються. Для спрацьовування переходу необхідно, щоб у всіх вхідних позиціях переходу були присутні фішки. Якщо ця умова виконується, то перехід називається дозволеним.

На рис. 3 зображено приклад спрацьовування переходу. До спрацьовування перехід є дозволеним, а після спрацьовування не є таким, тому що вхідні позиції не містять фішок.

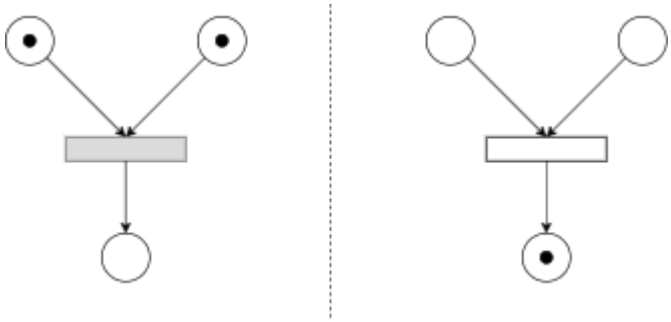

Рисунок 3 – Спрацювання переходу

Може виникнути ситуація, за якої сітка Петрі має більше одного збудженого переходу. У цьому випадку спрацьовує перехід, обраний випадковим чином. З цієї причини в сітці Петрі виникає недетерменізм. Наявність недетерменізму в сітках Петрі дає змогу природно представляти паралельний характер модельованих систем.

Крім одиничних дуг у сітках Петрі допускаються кратні дуги [11]. При цьому умови спрацьовування повинні виконуватися для кожного екземпляра дуги. Як правило, кратні дуги зображують шляхом написання кількістю дуг над одиничною дугою. На рис. 4 показано приклад спрацьовування переходу на сітці Петрі з кратними дугами.

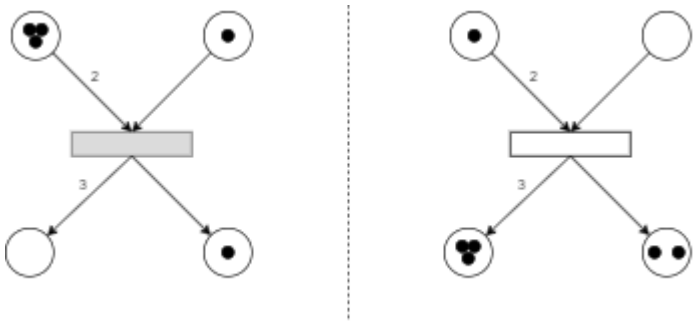

Рисунок 4 – Спрацювання переходу з кратними дугами

1.5.2 Огляд властивостей сіток Петрі

Для викладу важливих для цієї роботи властивостей сіток Петрі, далі формально визначається сітка Петрі, згідно з роботою [12]. Умовимося, що NN – це множина натуральних чисел, включно з 0. Сітка Петрі – це набір $N=(P, T, F, M_0)$, де:

P – скінченна множина всіх позицій;

T – скінченна множина всіх переходів;

$F: P\times T\cup T\times P\to \mathbb{N}$ – функція інцидентності, яка задає дуги та їхню кратність;

$M_0: P\to \mathbb{N}$ – початкове маркування сітки.

Уведемо спеціальні позначення для вхідних і вихідних дуг позицій:

$$^\bullet p=\{t|F(t, p)>0\},\ p^\bullet=\{t|F(p, t)>0\}.$$

Аналогічно, введемо спеціальні позначення для вхідних і вихідних дуг переходів:

$$^\bullet t=\{p|F(p, t)>0\},\ t^\bullet=\{p|F(t, p)>0\}.$$

Функціонування сітки Петрі описується формально за допомогою множини послідовностей спрацьовувань і множини досяжних у сітці маркувань.

Стан сітки Петрі визначається її маркуванням. *Маркування* сітки N – це функція $M: P\to \mathbb{N}$. Маркування задає розподіл фішок за позиціями. *Простір станів* сітки Петрі є множина всіх маркувань.

Якщо припустити, що всі позиції сітки N строго впорядковані якимось чином, тобто $P=(p_1, \ldots, p_n)$, то маркування M сітки (зокрема й початкове

маркування) можна задати як вектор чисел $M=(m_1,\dots,m_n)$ такий, що

$$\forall\, i,\ 1\le i\le n: m_i=M(p_i).$$

Якщо позиції впорядковані, то можна кожному переходу t зіставити два цілочисельних вектори ${}^\bullet F(t)$ і $F^\bullet(t)$ довжиною n, де $n=|P|$:

$${}^\bullet F(t)=(b_1,\dots,b_n),\ \text{де}\ b_i=F(p_i,t),$$
$$F^\bullet(t)=(b_1,\dots,b_n),\ \text{де}\ b_i=F(t,p_i).$$

Перехід t може спрацювати (перехід t дозволений) за деякої розмітки M сітки N, якщо

$$\forall\, p\in{}^\bullet t: M(p)\ge F(p,t),$$

тобто кожна вхідна позиція p переходу t має кількість фішок, не меншу, ніж кратність дуги, що з'єднує p і t. Цю умову можна переписати у векторній формі таким чином:

$$M\ge {}^\bullet F(t).$$

Спрацьовування переходу t при маркуванні M породжує нове маркування M' за таким правилом:

$$\forall\, p\in P: M'=M(p)-F(p,t)+F(t,p),\ \text{або}$$
$$M'=M-{}^\bullet F(t)+F^\bullet(t).$$

Таким чином, спрацьовування переходу t змінює маркування так, що

маркування кожної його вхідної позиції p зменшується на $F(p,t)$, тобто на кратність дуги, що з'єднує p і t, а маркування кожної його вихідної позиції p збільшується на $F(t,p)$, тобто на кратність дуги, що з'єднує t і p.

Уведемо на множині маркувань відношення безпосереднього слідування маркувань:

$$M \to M' \Leftrightarrow \exists t \in T : (M \geq {}^{\bullet}F(t)) \wedge (M' = M - {}^{\bullet}F(t) + F^{\bullet}(t)).$$

Будемо використовувати уточнювальне позначення $M \to_t M'$, якщо M' безпосередньо слідує після M у результаті спрацьовування переходу t.

Кажуть, що маркування M' досяжне від маркування M, якщо існує послідовність маркувань $M, M_1, M_2, \ldots, M'$ і послідовність переходів $t_1 t_2 \ldots t_k$ з множини T такі, що

$$M \to_{t_1} M_1 \to_{t_2} M_2 \ldots \to_{t_k} M'.$$

Множину маркувань, досяжних у сітки N від маркування M, позначимо через $R(N,M)$. Множиною досяжних маркувань сітки N називатимемо множину всіх маркувань, досяжних у N від початкової розмітки M_0, і позначатимемо $R(N) = R(N, M_0)$.

Далі розглянуті властивість живості та інші пов'язані з нею властивості.

Перехід t у сітки Петрі $N = (P, T, F, W, M_0)$ називається потенційно живим при маркуванні $M \in R(N)$, якщо

$$\exists M' \in R(N, M) : M' \geq {}^{\bullet}F(t),$$

тобто існує досяжне від M маркування, що досяжне від M M'за якого

перехід t дозволений. Якщо $M = M_0$, то t називається потенційно живим у сітки N. Перехід t називається мертвим при M, якщо він не є потенційно живим при M. Перехід t називається мертвим, якщо він мертвий за будь-якого досяжного в сітки маркування.

Перехід t називається живим у сітки Петрі N, якщо

$$\forall M \in R(N), \exists M' \in R(N, M): M' \geq^{\bullet} F(t),$$

тобто перехід t є потенційно живим за будь-якого досяжного в сітки маркування. Перехід t – потенційно мертвий, якщо існує досяжне маркування M таке, що за будь-якого маркування $M' \in R(N, M)$ перехід t не може спрацювати. Маркування M у цьому випадку називається t-тупиковим. Якщо воно t-тупикове для всіх $t \in T$, то воно є тупиковим. Якщо M – тупикова розмітка, то $R(N, M) = \{M\}$. Сітка називається живою, якщо всі її переходи живі.

1.5.3 Огляд інструментів пакету Tina

Tina (Time Petri Net Analyzer) – це набір інструментів для редагування та аналізу сіток Петрі. Пакет Tina було розроблено в OLC, потім VerTICS – дослідницьких групах дослідницької лабораторії LAAS/CNRS.

Tina містить графічний редактор сіток Петрі. На ОС Windows редактор можна знайти з директорії встановлення Tina за таким шляхом: bin\nd.exe. За допомогою редактора можна створити таку сітку, яку показано на рис. 5.

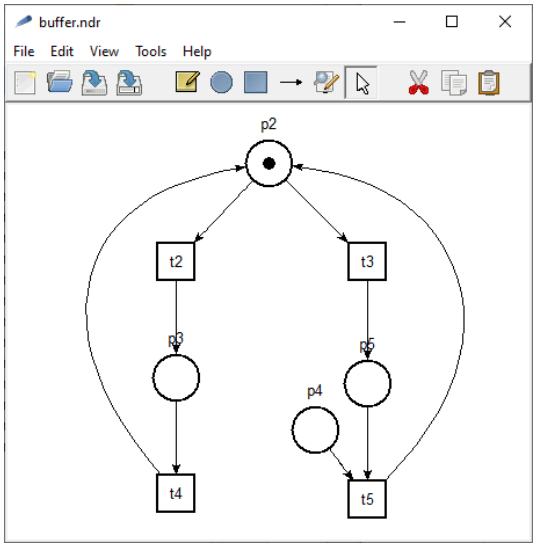

Рисунок 5 – Редактор сіток Петрі Tina

Графічний редактор nd має можливість симуляції запуску сітки Петрі. Для переходу в режим симуляції потрібно перейти в пункт меню Tools \ stepper simulator, зображений на рис. 6.

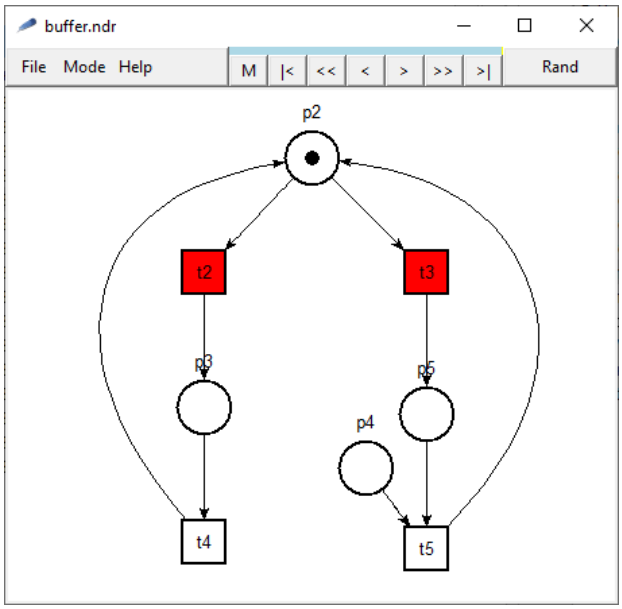

Рисунок 6 – Режим симуляції сітки Петрі

Дозволені переходи сітки виділені симулятором. При натисканні на перехід відбувається його спрацьовування. За допомогою симуляції ми можемо переконатися, що сітка заходить у тупик при виборі переходу t3. У цьому разі фішка буде тільки в позиції p5, але не p4, і тому ні перехід t5, ні будь-який інший перехід, не буде дозволено.

Ще одним інструментом із пакета Tina, який ми будемо використовувати, є sift. sift будує різні абстракції простору станів для сіток Петрі [13]. Він приймає як вхідні дані описи в текстовій формі (формати .net, .pnml, .tpn) або графічній формі (формат .ndr файлів, створених nd, .pnml з графікою). sift дає змогу ефективно обробляти великі простори станів.

Розглянемо роботу sift на прикладі сітки Петрі, яка була збережена у файлі simpleNet.ndr:

```
> C:\programs\tina-3.7.0\bin\sift.exe .\simpleNet.ndr -dead

some state violates condition -f:
  p5
  firable:
3 marking(s), 3 transitions(s)
```

Опція -dead вказує на знаходження тупиків і зупиняє sift у разі їх знаходження. Результат роботи sift показує, що при маркуванні p5, тобто коли фішка міститься тільки в позиції p5, відсутні дозволені переходи (firable).

2 РОЗРОБКА АНАЛІЗАТОРУ

2.1 Сітка Петрі як інструмент моделювання розподілених систем

Програми MPI складаються з процесів, що взаємодіють між собою. Характер цих взаємодій має паралельний характер. Для моделювання цієї поведінки для цієї роботи було обрано сітки Петрі. Як було зазначено в підпункті 1.5.1, сітки Петрі дають змогу природно представляти паралельний характер модельованих систем. Крім того, сітки Петрі легко представляти в графічному вигляді, що сприяє їхньому розумінню та побудові. Також, теорія сіток Петрі чітко формулює властивості сіток, які стосуються тупиків.

В якості прикладу успішного застосування сіток Петрі для моделювання паралельних систем можна вказати роботу [14]. У цій роботі було побудовано модель багатовимірної решітки взаємодії для комутатора, яка реалізує недетерміністичне рішення пересилання пакетів.

Для сіток Петрі було розроблено ефективні програмні пакети, такі як Tina. Tina є переможцем Конкурсу перевірки моделей (Model Checking Contest) [15].

Одним з альтернативних підходів для перевірки коректності паралельних систем є алгебра процесів (process algebra). В алгебрі процесів поведінка системи описується у вигляді алгебраїчних виразів [16].

Як зазначено в роботі [17] алгебра процесів належить до моделей, які виявляють мінливий характер (interleaving). У таких моделях паралельне виконання завдань імітується як вибір їх упорядкування. Навпаки, сітки Петрі належать до моделей, які виявляють дійсний паралелізм (true concurrency). Поведінку систем у таких моделях уявляють у вигляді причинних відносин між подіями в різних локаціях системи [17].

2.2 Побудова сітки Петрі для програми MPI

Основний внесок цієї роботи полягає в розробленні методу побудови моделі у вигляді сітки Петрі за заданою програмою MPI з метою подальшої перевірки на відсутність тупиків. На даний момент у цього методу існують такі обмеження:

– програма має бути представлена в одному методі, виклики інших методів, визначених користувачем, не обробляються;
– можливо обробляти тільки цикли з умовами, які можна обчислити на етапі компіляції;
– ранг процесу можливо визначити на етапі компіляції;
– кількість процесів у програмі має бути обмежена.

Більша частина програм MPI використовує таку схему. На самому початку програми відбувається отримання рангу поточного процесу. Далі відбувається перевірка на ранг процесу і кожен процес виконує своє завдання.

Нижче наведено приклад програми, у якій відбувається тупик.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

Для такого прикладу буде побудована сітка Петрі, показана на рис. 7. Отримана сітка є циклічною. Зворотні ребра завжди присутні, тому що позиції, які відповідають початку і кінцю програми, пов'язані проміжним переходом.

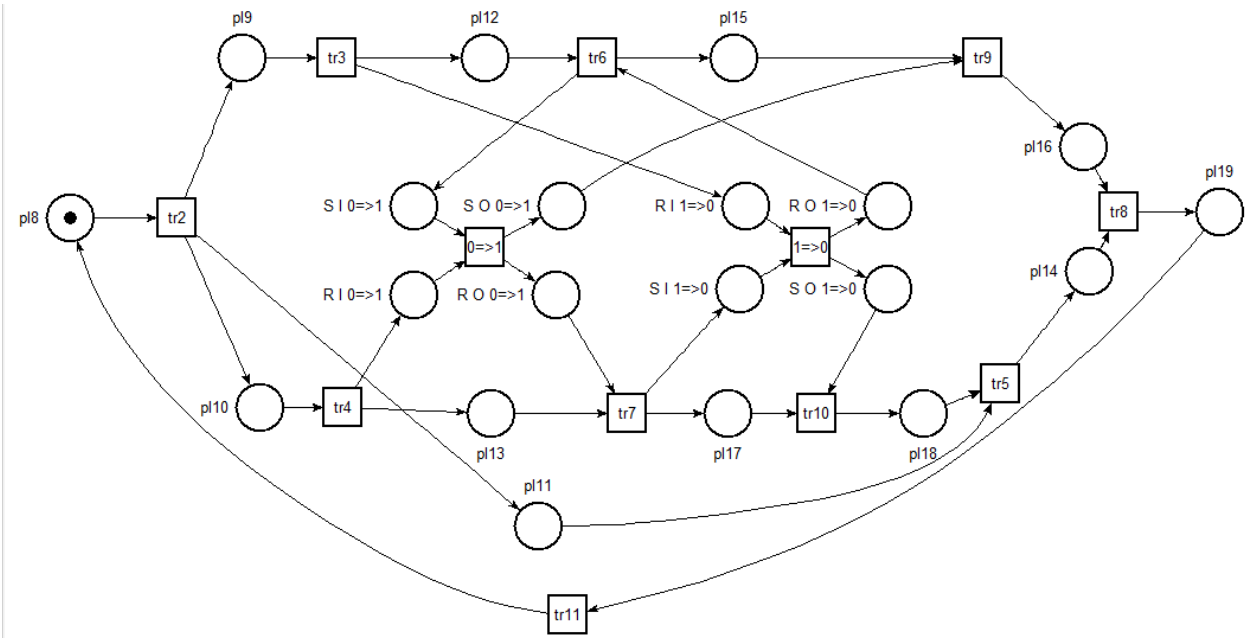

Рисунок 7 – Побудована сітка Петрі

Програма є коректною з погляду аналізатора, якщо в ній відсутні тупики, спричинені операціями Send і Recv. Тоді в сітки Петрі, що відповідає коректній програмі, не повинно бути маркувань, які можна досягти з початкового маркування, і які є тупиковими. Інакше кажучи, сітка має бути безтупиковою.

У початковому маркуванні побудованої сітки Петрі фішка знаходиться в позиції, яка відповідає початку програми. На рис. 7 такою позицією є pl8. На рис. 8 фішки знаходяться в таких позиціях: pl11, pl12, pl13, «R I 0=>1» і «R I 1=>0». Фішка в позиції «R I 0=>1» говорить про те, що процес 0 викликав операцію отримання повідомлення від процесу 1. Аналогічно, фішка в позиції «R I 1=>0» вказує, що процес 1 викликав операцію отримання повідомлення від процесу 0.

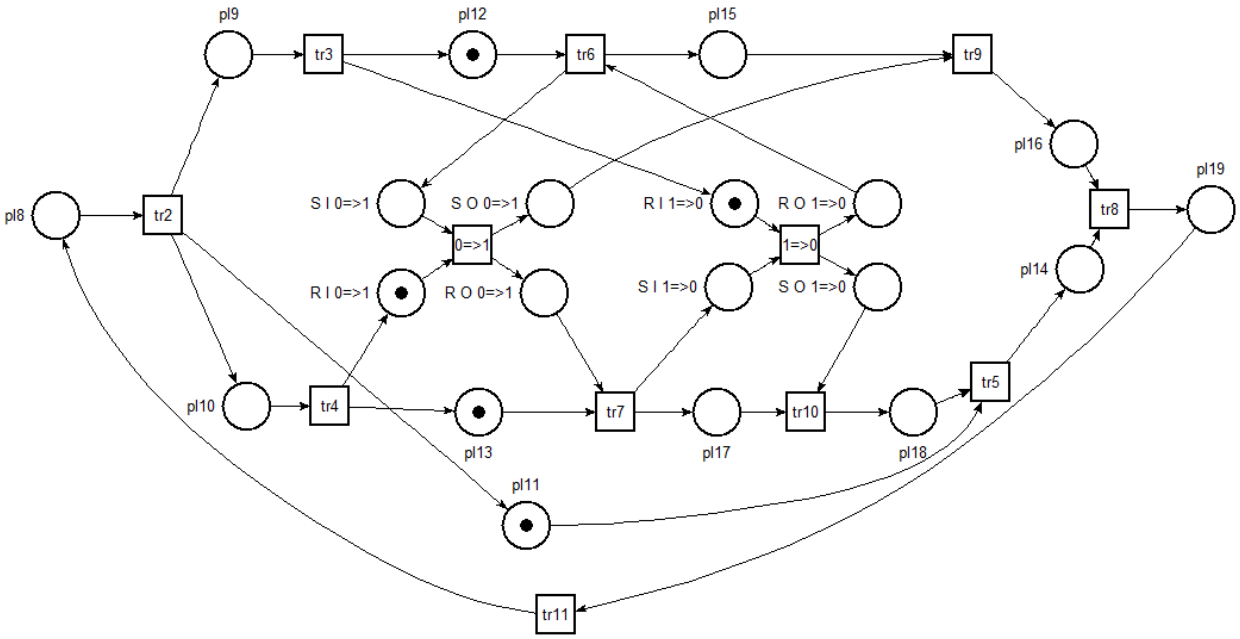

Рисунок 8 – Тупикова маркування сітки Петрі

Показане на рис. 8 маркування є тупиковим, тому що при ньому не дозволено жодного переходу. Наприклад, перехід tr6 вимагає, щоб процес 0 завершив отримання повідомлення від процесу 1, що позначає позиція «R O 1=>0». Але це неможливо, тому що в цей час процес 1 у цей час навіть не почав операцію відправлення. Подібним чином, перехід tr7 є недозволеним, тому що він вимагає завершення отримання повідомлення процесом 1, тобто вимагає наявність фішки в позиції «R O 0=>1».

Під час побудови сітки Петрі потрібно враховувати такі важливі моменти:

– моделювання блокування під час виконання операцій MPI;

– послідовність, у якій операції викликаються;

– точки поділу процесів;

– точки синхронізації процесів.

Для моделювання блокування при операціях MPI використовуються групи з позицій і переходу. Такі групи ми назвемо шлюзами. Шлюзи повинні блокувати обидва процеси, які обмінюються повідомленнями, поки не буде

викликано і відправлення, і отримання. Якщо в програмі присутні повторні виклики операцій MPI для тієї самої пари відправника й одержувача, то шлюз для них не дублюються.

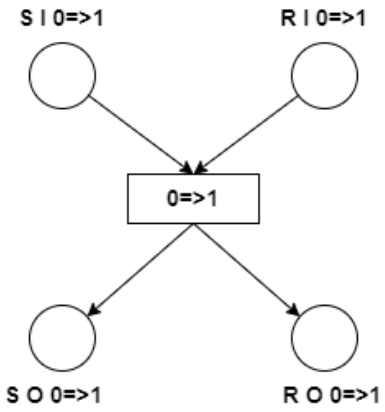

Рисунок 9 – Шлюз для надсилання повідомлення від процесу 0 до процесу 1

На рис. 9 зображено приклад шлюзу для надсилання повідомлення від процесу 0 до процесу 1. У середині розташовується перехід, який позначений символами «0=>1». Зверху зображено позиції, які представляють входи в операції: «S I 0=>1» – вхід в операцію відправлення, «R I 0=>1» – вхід в операцію отримання. Внизу зображено аналогічні позиції, які відповідають виходам з операцій: «S O 0=>1» і «R O 0=>1». Перехід «0=>1» не дозволено, доки не будуть викликані Send і Recv для відповідних процесів. Після спрацювання цього переходу подальша робота сітки Петрі стає можливою.

Для представлення послідовності виконання операцій використовується проста комбінація з позицій і переходів. Надалі такі послідовності можна використовувати для асоціації з конкретними точками у вихідній програмі.

Нижче представлено фрагмент коду, у якому процес 0 послідовно надсилає два повідомлення процесу 1.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
}
```

На рис 10. зображено фрагмент відповідної сітки Петрі. На рис. 10 праворуч показано шлюз, який було описано вище. Позиціями pl1, pl2, pl3, pl4 і переходами tr1, tr2, tr3 ліворуч на рис. 10 виражається послідовність операцій у програмі. Позиція pl1 відповідає точці програми до першого виклику Send. Позиція pl2 говорить про те, що Send було викликано вперше, але не закінчено. Позиція pl3 говорить про те, що перший виклик Send був успішно завершений, а другий виклик розпочато. Позиція pl4 відповідає точці програми після успішного завершення другого виклику Send. Шлюз використовується один, оскільки обидва повідомлення надсилаються від процесу 0 до процесу 1.

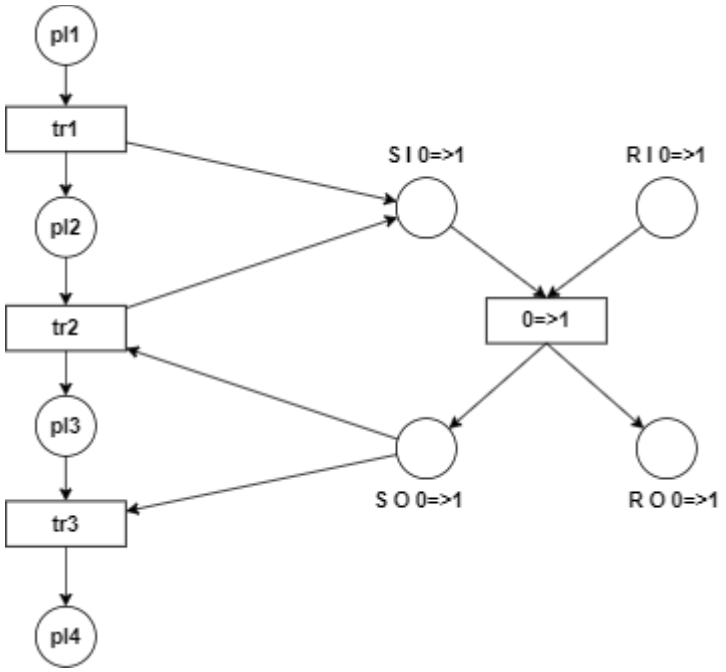

Рисунок 10 – Фрагмент сітки Петрі, що показує послідовність операцій

Насправді, одна й та сама програма MPI виконується кількома про-

цесами паралельно. У цій програмі є фрагменти, які виконуються всіма процесами, а є фрагменти, що виконуються тільки певними процесами. На побудованій сітки Петрі частини програми, які виконуються всіма процесами, не дублюються. Це дає змогу скоротити кількість переходів і позицій у сітці. Тому виникає необхідність представлення в сітці точок поділу процесів і точок синхронізації.

Для точок поділу використовується набір елементів сітки Петрі, показаний на рис. 11. Він також відомий як AND-split (І-розділення) [18]. AND-split розділяє потік виконання на два паралельні процеси.

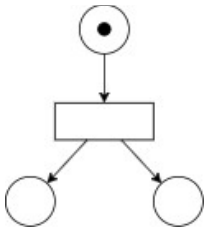

Рисунок 11 – AND-split

Для представлення точок синхронізації використовується набір елементів AND-join (І-з'єднання) [18], показаний на рис. 12. AND-join моделює з'єднання двох паралельних потоків.

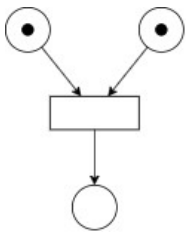

Рисунок 12 – AND-join

2.3 Загальний опис алгоритму побудови сітки Петрі

Для автоматичної побудови сітки Петрі було розроблено алгоритм. Для аналізу програми використовувалися як традиційні підходи з області розробки компіляторів (домінатори, форма SSA, аналіз потоку даних, та інші), так і нові підходи, спеціалізовані під поточне завдання, такі як визначення рангів процесів і граф операцій.

Нижче описується алгоритм побудови сітки Петрі за заданою програмою. Відповідну блок-схему алгоритму показано на рис. 13.

1) Текст вхідної програми розбирається за допомогою бібліотеки Eclipse JDT. У результаті створюється абстрактне синтаксичне дерево (abstract syntax tree, AST).
2) AST перетворюється на форму HIL – високорівневу проміжну мову.
3) Форма HIL перетворюється на форму LIL – низькорівневу проміжну мову.
4) На основі форми LIL виконується побудова графа потоку управління.
5) За допомогою форми LIL і графа потоку управління виконується часткове виконання програми. Здебільшого, це виконується з метою розгортання циклів.
6) Виконується аналіз живості змінних.
7) Виконується обчислення інформації про домінаторів.
8) На основі аналізу живості та інформації про домінаторів виконується побудова форми SSA.
9) Виконується просування предикатів, яке дає змогу визначати, які блоки коду виконується яким процесом.
10) Виконується побудова графа операцій.
11) Граф операцій перетворюється на сітку Петрі.
12) сітка Петрі записується у файл із розширенням .net формату Tina.

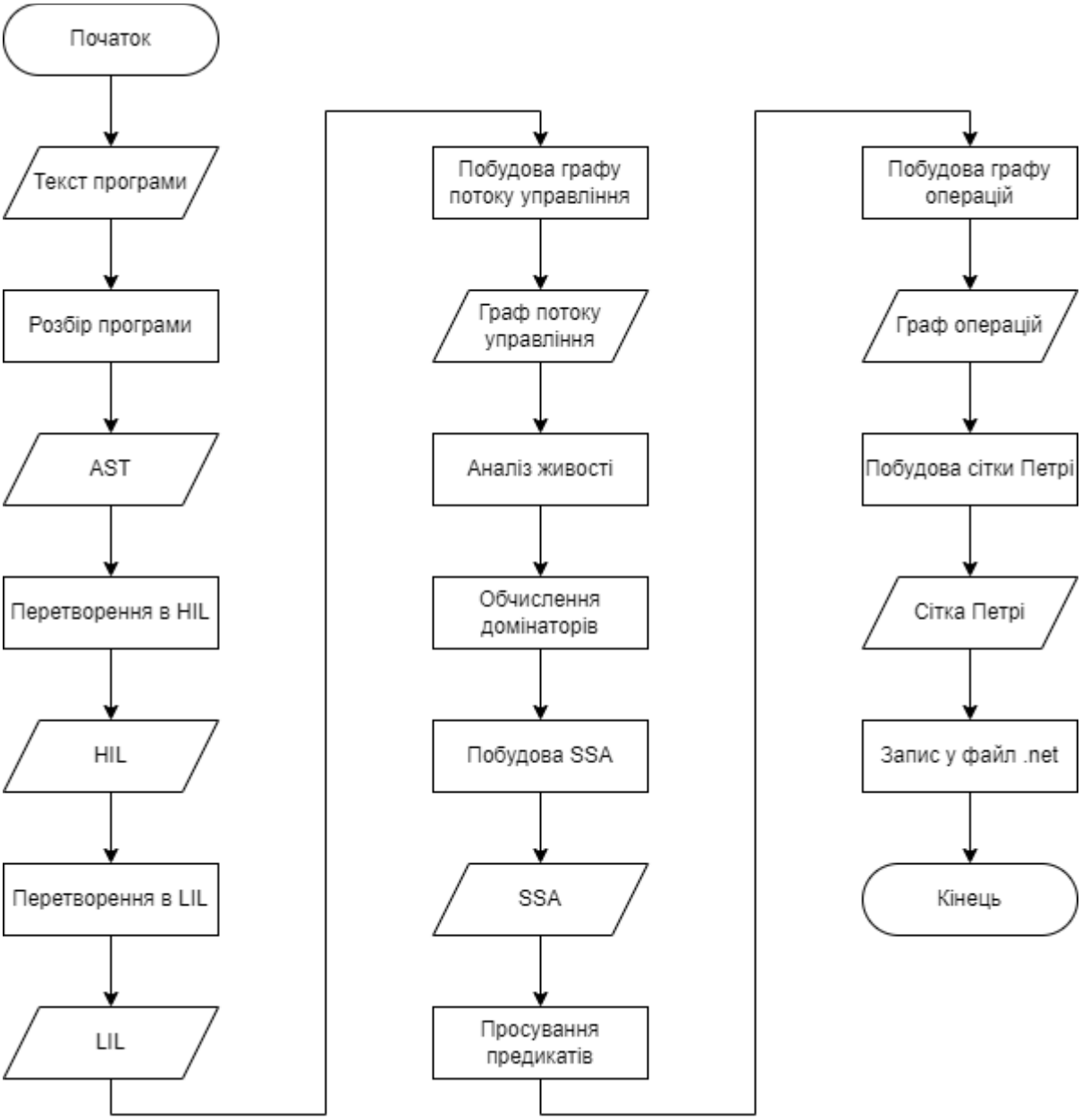

Рисунок 13 – Блок-схема алгоритму побудови сітки Петрі

На рис. 14 показано залежність проміжних подань і допоміжних структур даних. Текст програми, Eclipse AST, HIL і LIL перетворюються один в одного послідовно. Фундаментальним проміжним поданням є граф потоку управління, який використовується для побудови майже всіх наступних проміжних подань. Важливими даними є інформація про домінаторів,

яка використовується під час побудови форми SSA та інформації про ранги процесів. Під час побудови кінцевої сітки Петрі використовується тільки граф операцій, тому вони досить схожі між собою структурно та відрізняються рівнем деталізації.

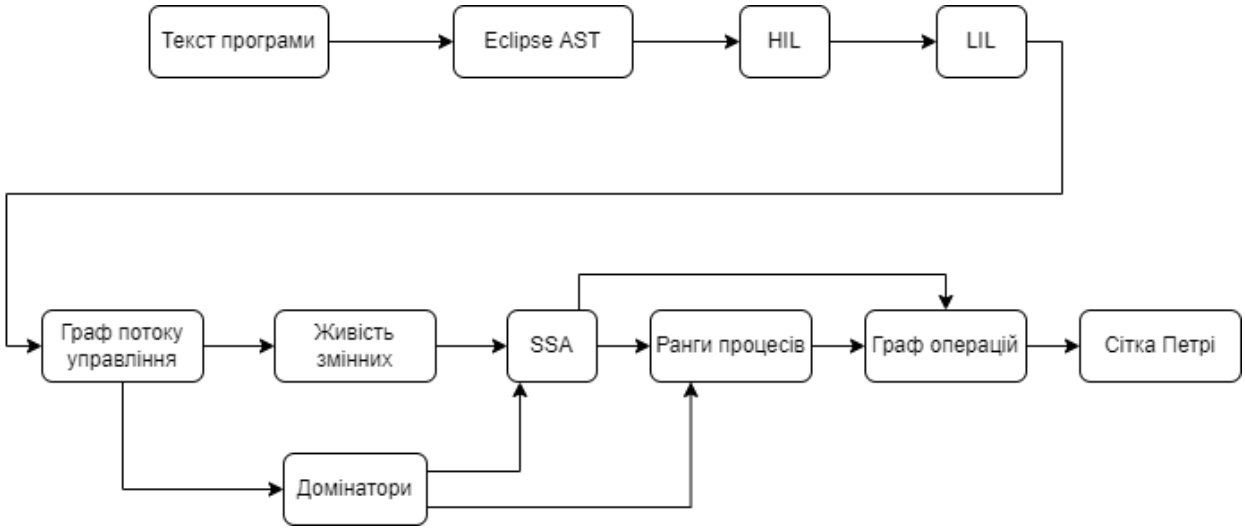

Рисунок 14 – Залежність проміжних форм та структур даних

Аналізатор перетворює програму Java, що використовує MPJ Express, на сітку Петрі у форматі Tina. Отриману сітку можна досліджувати на наявність тупиків за допомогою інструментів пакета Tina.

Аналізатор написаний мовою Scala, яка працює на JVM (Java Virtual Machine). В аналізаторі використовується бібліотека Eclipse JDT для розбору програмних текстів мовою Java.

2.4 Розбір Java за допомогою Eclipse JDT

Eclipse – це інтегроване середовище розробки програм. Розвивається та підтримується організацією Eclipse Foundation. Спочатку Eclipse розроблялася як наступник середовища розробки IBM VisualAge. Eclipse є програмним

забезпеченням із відкритим кодом.

На основі Eclipse створено середовища розробки для різних мов програмування: Java, C/C++, Python, PHP.

Бібліотека Java Development Tools (JDT) – це набір плагінів, які разом із платформою Eclipse утворюють середовище розробки для Java. Це середовище розробки має такі можливості: створення та управління проектами Java, редагування коду, автоматичні підказки, інструменти рефакторингу коду, інструменти для дослідження моделі Java проекту (наприклад, ієрархія викликів, ієрархія спадкування, пошук посилань), складання проекту, відлагодження програм Java, інтеграція з різними системами збірок (такими як Maven) та інші.

JDT складається з наступних компонентів:

– JDT APT – обробка анотацій;
– JDT Core – інфраструктура для роботи з кодом: збирач, модель мови, підтримка написання коду, рефакторинг, пошук символів;
– JDT Debug – відладник Java додатків;
– JDT Text – підсвічування синтаксису, форматування коду;
– JDT UI – реалізує графічний інтерфейс користувача до роботи з Java проектами.

Наш аналізатор використовує компонент JDT Core, зокрема його парсер. Парсер – це компонент, який перетворює послідовність символів вихідного коду в абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (англ. Abstract syntax tree, AST) в – це скінченна множина, позначене і орієнтоване дерево, в якому внутрішні вершини співставлені з відповідними операторами мови програмування, а листя з відповідними операндами.

Розглянемо, як відбувається робота з парсером JDT.

```
def parse(source: String): CompilationUnit =
  val parser = ASTParser.newParser(AST.JLS16)
  val mpjJar = Path.of(System.getenv("MPJ_HOME"))
```

```
  .resolve("lib").resolve("mpj.jar")
parser.setSource(source.toCharArray)
parser.setKind(ASTParser.K_COMPILATION_UNIT)
parser.setResolveBindings(true)
parser.setEnvironment(Array(mpjJar.toString),
  Array.empty, Array.empty, true)
parser.setUnitName("Unit name")
parser.createAST(null).asInstanceOf[CompilationUnit]
```

Створення парсера відбувається за допомогою ASTParser.newParser(). У цей метод передається версія мови Java, з якою буде працювати парсер; в даному випадку вказується версія 16. Посилання на створений парсер зберігається в змінній parser.

Далі проводиться пошук шляху до бібліотеки MPJ за допомогою змінної оточення MPJ_HOME.

Виклик parser.setSource(…) вказує парсеру текст програми, для якої потрібно побудувати синтаксичне дерево.

Виклик parser.setKind(…) вказує, що має бути результатом аналізу. В даному випадку – це одиниця компіляції (compilation unit), в інших випадках може бути окрема команда (statement) або вираз (expression).

Виклик parser.setResolveBindings(true) вказує, що крім аналізу програми потрібно ще зробити аналіз зв'язувань. Зв'язування з'єднують місця використання символів, таких як змінних або методів, з місцями їх визначення. Це допомагає, наприклад, визначити тип змінної у кожній точці її використання.

За допомогою виклику parser.setEnvironment(…) парсер отримує інформацію про залежності програми, тут такою залежністю є бібліотека MPJ.

Далі ми виставляємо ім'я для одиниці компіляції parser.setUnitName() і отримуємо результат parser.createAST().

Отже, бібліотека JDT визначає парсер та синтаксичне дерево Java з усіма його вузлами. Окрім цього, JDT надає клас ASTVisitor. Відвідувач (visitor) – це шаблон проектування, що виконує обхід деревоподібних структур даних. У JDT такою структурою даних є синтаксичне дерево із вузлами,

базовим класом яких є ASTNode. Для кожного типу вузла в ASTVisitor визначено два методи – visit() та endVisit(). Перший викликається перед відвідуванням дочірніх вузлів, другий викликається після відвідування дочірніх вузлів. На рис. 15 зображена діаграма класів для шаблону проектування “відвідувач”. Зображені лиш тільки деякі спадкоємці ASTNode – CompilationUnit (одиниця компіляції) та IfStatement (умовний оператор).

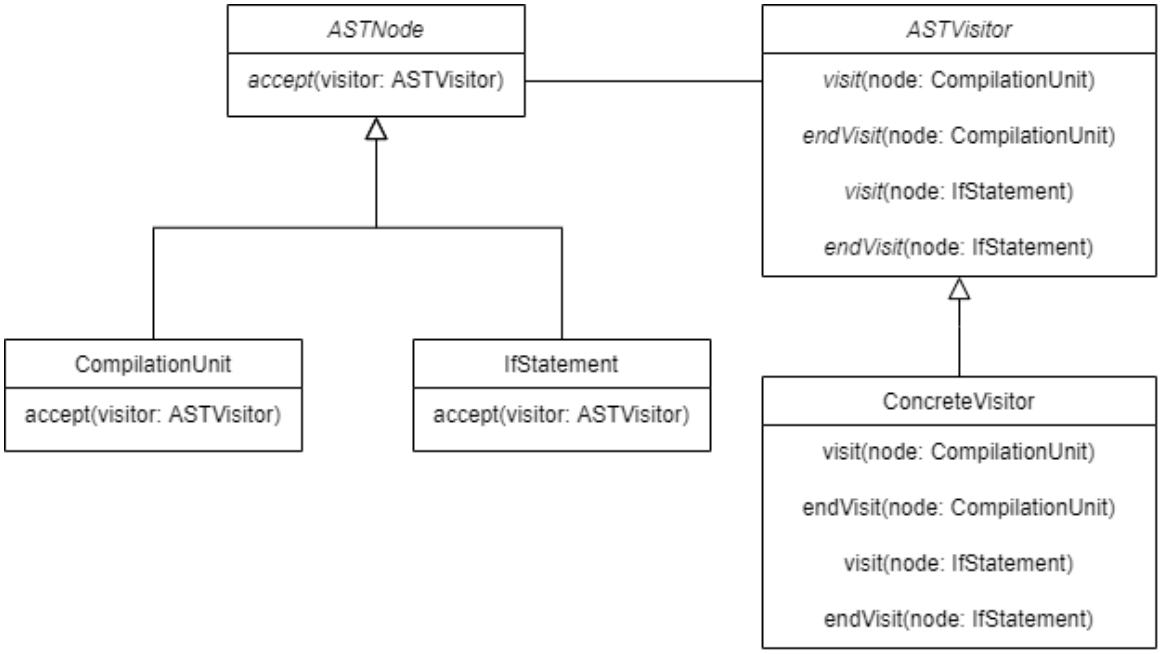

Рисунок 15 – Шаблон Відвідувач

Для використання класу ASTVisitor потрібно створити його клас-спадкоємець та перевизначити методи visit() та endVisit() для вузлів, які нас цікавлять. Потім потрібно зробити виклик методу accept() і передати туди створений об'єкт спадкоємця. Створення та використання відвідувача показано на прикладі нижче.

```
def apply(cu: CompilationUnit): Program =
  val visitor = new Visitor
  cu.accept(visitor)
  Program(visitor.getFuncs)
```

2.5 Проміжна мова високого рівня

HIL (higher level intermediate language) – це проміжна мова високого рівня. HIL створюється шляхом перетворення синтаксичного дерева, отриманого з парсеру JDT.

Найвищим типом у HIL є програма (Program). Вона складається із списку визначень функцій (FuncDecl). Кожна функція складається зі списку параметрів та блоку коду (Block). Блок коду містить у собі послідовність інструкцій (Stmt – скор. від statement).

Інструкції можуть бути такі:

– переривання циклу (Break);
– перехід до наступної ітерації циклу (Continue);
– повернення (Return);
– присвоєння (Assignment);
– оголошення змінної (VarDecl);
– інструкція виклику функції (CallStmt).
– умовний оператор (IfThenElse);
– цикл (WhileLoop);
– блок.

Деякі інструкції, такі як присвоєння та повернення, містять вирази. Вирази можуть бути простими та складними. До простих виразів належать:

– доступ до змінної (Variable);
– константа (IntLiteral);
– звернення до статичного поля (StaticFieldAccess).

Складні вирази, на відміну простих, можуть містити інші виразі, проте лише прості. Таким чином досягається обмеження вкладеності виразів. Розглянемо види складних виразів:

– унарний вираз (UnaryExpr);
– виклик функції (CallExpr). Відрізняється від інструкції CallStmt;

– бінарний вираз (BinaryExpr).

Нижче представлена граматика для HIL.

```
<Program> ::= <FuncDecl>+
<FuncDecl> ::= "func" <Name> "(" [<Param> ["," <Param>]+] ")" ":" <Type>
<Block>
<Param> ::= <Name> : <Type>
<Stmt> ::= <Break> | <Continue> | <Return> | <Assignment> | <VarDecl> |
           <CallStmt> | <IfThenElse> | <WhileLoop> | <Block>
<Break> ::= "break"
<Continue> ::= "continue"
<Return> ::= "return" [<Expr>]
<Assignment> ::= <Name> "=" <Expr>
<VarDecl> ::= <Name> ":" <Type> ["=" <Expr>]
<CallStmt> ::= <CallExpr>
<IfThenElse> ::= "if" <Condition> <Stmt> ["else" <Stmt>]
<Condition> ::= "(" <SimpleExpr> ")"
<WhileLoop> ::= "while" <Block> <Condition> <Block>
<Block> ::= "{" <Stmt>+ "}"
```

Символ <Name> означає будь-який допустимий ідентифікатор Java. Квадратні дужки […] означають, що вміст елемента є необов'язковим. + означає повторення один і більше разів; * означає повторення нуль і більше разів.

Тепер розглянемо приклад перетворення Java коду в HIL.

```
public class Example14 {
    static int func1(int a, int b, int c, int count) {
        for (int i = 0; i < count; i++) {
            a = a + b * c;
        }
        return a;
    }
}
```

Тут бачимо статичний метод, у якому є цикл for, що виконує прості арифметичні обчислення. HIL буде виглядати так:

```
func Example14.func1(a: int, b: int, c: int, count: int): int {
  {
    var i: int = 0
    while {
      var t~1: boolean = i<count
    } (t~1) {
      var t~2: int = b*c
      a = a+t~2
      i = i+1
```

```
    }
  }
  return a
}
```

HIL є більш простою мовою, ніж Java, тому деякі конструкції Java транслюються в декілька більш простих конструкцій HIL. Менша кількість конструкцій у мові значно спрощують подальшу роботу з ним, тому що для проведення аналізу потрібно буде враховувати меншу кількість випадків.

У HIL вкладеність виразів обмежена. Для того, щоб представити вирази з довільною вкладеністю з Java до HIL ми використовуємо додавання тимчасових змінних, які містять проміжні результати виразу. У фрагменті HIL вище такими змінними є t~1 і t~2. Прикладом перетворення виразів є вираз a+b*c. У HIL ми спочатку створюємо тимчасову змінну t~2, яка зберігає результат b*c, а потім виконуємо додавання a+t~2.

У HIL немає циклу for, тому використовується цикл while, тому що цикл for завжди можна перетворити на цикл while. У прикладі вище бачимо блок перед умовою циклу. Цей блок містить тимчасові змінні, які використовуються у виразі умови. В іншому цикл while має звичні частини: умова циклу та тіло циклу.

Ще одним прикладом перетворень є зведення інкременту i++ до присвоєння i = i + 1.

Над формою HIL відбувається одне важливе перетворення – це додавання інструкцій assert. Ці інструкції додаються на початку гілок умовних операцій. Вони захоплюють вираз умови інструкції if-then-else. Для блоку then додається сама умова, для блоку else додається заперечення умови.

Це перетворення є важливим, оскільки на основі інструкції assert виводиться інформація про те, який процес робить виклики функцій MPJ. Наприклад, фрагмент

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
```

```
} else {
  MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

перетворюється на

```
var t~1: boolean = rank==0
if (t~1) {
  assert t~1
  mpi.Comm.Send(mpi.MPI.COMM_WORLD, buffer, 0, 4, mpi.MPI.INT, 1, 0)
} else {
  assert !t~1
  mpi.Comm.Recv(mpi.MPI.COMM_WORLD, buffer, 0, 4, mpi.MPI.INT, 0, 0)
}
```

2.6 Проміжна мова низького рівня

LIL (lower level intermediate language) – це проміжна мова низького рівня. Форми HIL та LIL мають багато спільного, проте HIL є формою більш вищого рівня.

У програмі LIL функція розбивається на набір базових блоків. Базовий блок (basic block) [19] – це послідовність інструкцій, що завжди виконуються разом. Базовий блок починається з мітки (label) та закінчується інструкцією переходу. На відміну від HIL, базові блоки LIL не можуть бути вкладеними. Кожен блок отримує своє ім'я – мітку. Умовні операції та цикли з HIL перетворюються на інструкції переходів, також відомі як інструкції goto.

У LIL є такі інструкції переходів:

– безумовний стрибок (Jump);

– умовний стрибок (CondJump);

– повернення із функції (Return).

Нижче представлена частина граматики LIL, яка відрізняється від HIL.

```
<FuncDecl> ::= "func" <Name> "(" [<Param> ["," <Param>]+] ")" ":" <Type>
               <Block>+
<Block> ::= <Stmt>* <Transfer>
<Stmt> ::= <Assignment> | <CallStmt>
<Transfer> ::= <Jump> | <CondJump> | <Return>
```

```
<Jump> ::= "jump" <Name>
<CondJump> ::= "condJump" <SimpleExpr> <Name> <Name>
<Return> ::= "return" <SimpleExpr>
```

Приведемо приклад перетворення фрагменту з підрозділу 2.6 в LIL.

```
entry:
rank = mpi.Comm.Rank
buffer = new int[3]
t~1 = rank==0
condJump t~1 bb2 bb4
bb2:
mpi.Comm.Send
jump bb3
bb4:
t~2 = rank==1
condJump t~2 bb5 bb6
bb5:
mpi.Comm.Recv
jump bb6
bb6:
jump bb3
bb3:
jump end
end:
return
```

На цьому прикладі ми бачимо, що виконання починається з базового блоку entry. В ньому проходить перевірка на ранг процесу і виконується стрибок на блок bb2, якщо ранг дорівнює нулю, або інакше на блок bb4. У блоку bb2 викликається Send і потім виконання переходить до блоку bb3, і потім до end, де відбувається повернення з функції (return). Подібним чином відбувається виконання блоку bb4.

2.7 Граф потоку управління

На основі базових блоків LIL будується граф потоку управління. *Граф потоку управління* (control flow graph) [19] моделює потік управління між базовими блоками в програмі. Він являє собою спрямований граф $G=(N, E)$. Кожен вузол $n \in N$ відповідає базовому блоку. Кожне ребро $e=(n_i, n_j) \in E$ відповідає можливому переходу управління від блоку n_i до блоку n_j.

Побудова графа потоку виконання з LIL відбувається так. Формується список усіх ребер у графі. Для цього з кожного блоку отримується інформація про наступні блоки завдяки його інструкції переходу. Наприклад, якщо інструкція переходу – звичайний стрибок, то до списку додається ребро від

поточного блоку до наступного. Потім зі списку ребер створюється сам об'єкт графа.

Оскільки структура даних спрямованого графа використовується не тільки для графа потоку управління, а й для інших даних, в аналізаторі використовується базовий клас для спрямованих графів. Наведемо його визначення.

```
class Graph[N](val edges: List[(N, N)]):
  protected val map: Map[N, List[N]] = edges.groupMap(_._1)(_._2)
  protected val reverseMap: Map[N, List[N]] = edges.groupMap(_._2)(_._1)

  val nodes: List[N] =
    edges.flatMap((n, m) => List(n, m)).distinct

  def successors(node: N): List[N] =
    map.getOrElse(node, List())

  def predecessors(node: N): List[N] =
    reverseMap.getOrElse(node, List())
```

Тут використовується типовий параметр N, який є типом вузлів. У графі потоку управління вузли представлені типом String. Граф створюється з набору дуг. Кожна дуга – це пара початковий/кінцевий вузол. З цього набору обчислюється асоціативний масив map, ключем якого є вузол, а значенням – список наступних вузлів даного вузла. Також будується обернений асоціативний масив – від вузла до його вузлів-попередників. Цим двом асоціативним масивам відповідають методи successors() і predecessors().

На рис. 16 показано граф потоку управління, побудований для фрагмента програми:

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
} else {
  MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

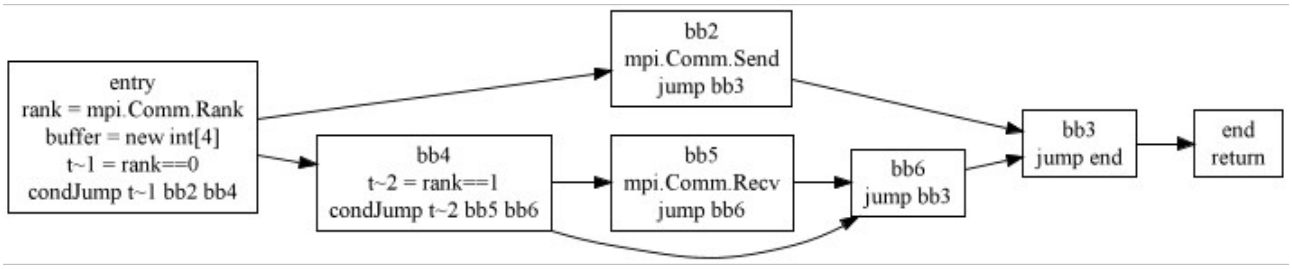

Рисунок 16 – Граф потоку управління для простої програми

2.8 Часткове обчислення програми

Часткове обчислення (partial evaluation) [20] – це техніка оптимізації програм, яка полягає у створенні спеціалізованої версії програми на підставі частини наданих даних. Під час запуску програми з наданням іншої частини даних результат буде такий самий, як і під час запуску оригінальної програми за надання повного набору даних (див. рис. 17).

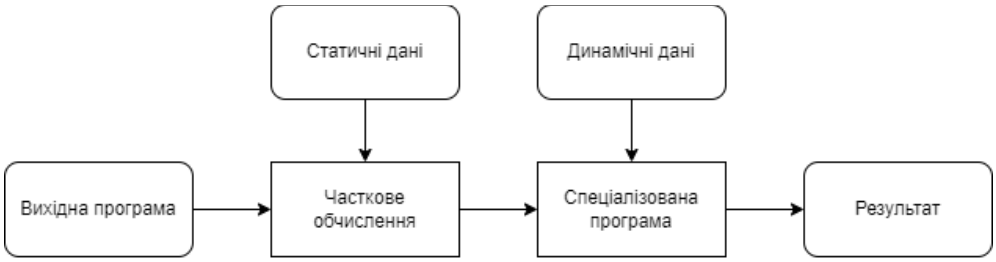

Рисунок 17 – Схема роботи часткового обчислення

Іншими словами, часткове обчислення дає змогу обчислити частину програми під час компіляції, використовуючи дані, які були відомі на цей час. Основна перевага цієї оптимізації полягає в тому, що спеціалізована версія програми не буде заново проводити обчислення, які вже були виконані

під час компіляції. Скорочення кількості обчислень може значно збільшити швидкість виконання програми.

Над формою LIL проводиться часткове обчислення. Метою цього перетворення в аналізаторі є обробка циклів з відомою кількістю ітерацій, тобто з кількістю ітерацій, яку можна вивести під час аналізу коду.

2.8.1 Поділ змінних

Під час аналізу неможливо повністю обчислити програму, оскільки частина даних невідома. Тому необхідно визначити, яка частина даних невідома. Цей процес називається *поділ змінних* (division) [20] на:

– статичні – змінні, які можна визначити під час часткового обчислення, тобто під час аналізу;
– динамічні – змінні, які не є статичними.

Іншими словами, динамічною змінною можна назвати змінну, яка залежить від інших динамічних змінних.

Процес поділу змінних називається *аналізом часу зв'язування*, тому що він визначає, в який час може бути обчислено значення змінної, тобто час, у який значення зв'язується зі змінною.

Позначимо всі змінні в програмі $x_1, \ldots, x_N$ і припустимо, що вхідні змінні подано набором $x_1, \ldots, x_n$, де $0 \le n \le N$. Припустимо, що дано значення часу зв'язування для вхідних змінних $b'_1, \ldots, b'_n$, де b'_j дорівнює або S (статична змінна), або D (динамічна змінна). Завдання полягає в обчисленні поділу для *всіх* змінних у програмі: $B = (b_1, \ldots, b_N)$ таким, що якщо $b'_i = D$, то $b_i = D$. Це завдання досягається таким алгоритмом:

1) Задати початкове значення для B таким, що дорівнює $(b'_1, \ldots, b'_n, S, \ldots, S)$, тобто всі змінні, крім вхідних, вважаються статичними.

2) Якщо програма містить присвоювання $x_k = expr$, при чому змінна x_j використовується у виразі *expr* і $b_j = D$ у B, то виставити $b_k = D$ у B.
3) Повторювати крок 2 доти, доки B буде змінюватися. Потім алгоритм завершується з поділом B.

2.8.2 Перетворення програми

Після обчислення поділу змінних можна приступити до перетворення програми. Завдання цього перетворення полягає в тому, щоб зробити всі обчислення, які можливі за наявності відомих статичних даних і змінити програму відповідним чином. Для цих обчислень використовується сховище (store), завдяки якому за ім'ям змінної можна отримати її значення. Сховище містить інформацію тільки про статичні змінні.

Під час перетворення програми необхідно знати, які частини програми треба перетворювати. Для цього можна використовувати список базових блоків. При цьому один і той самий блок може бути виконаний з двома різними сховищами. Прикладом можуть послужити дві різні ітерації циклу, під час яких значення змінних можуть бути різними. Тому в алгоритмі перетворення використовується робочий список пар (label, store), де label – це мітка базового блоку, store – це сховище, з яким треба буде обчислювати цей базовий блок.

1) Задаємо початкове значення робочому списку worklist = {(entry, store0)}, де entry – мітка початкового базового блоку, store0 = порожнє сховище.
2) Якщо worklist порожній – завершити алгоритм.
3) Узяти пару (label, store) зі списку.
4) Якщо пара позначена – перейти на крок 2.
5) Позначити пару.
6) Згенерувати код для блоку з міткою label.

7) Додати такі пари в worklist.

8) Перейти на крок 2.

Під час генерації нового коду послідовно переглядається кожна інструкція в базовому блоці, за потреби створюється нова інструкція і модифікується сховище. Детально генерацію коду для інструкцій відображено в табл. 4.

Таблиця 4 – Генерація коду для інструкцій

Інструкція	Дія під час аналізу	Генерована інструкція
`x = exp` (якщо x динамічна)	`reducedExp = reduce(exp, store)`	`x = reducedExp`
`x = exp` (якщо x статична)	`val = evaluate(exp, store)` `store = store[x → val]`	`---`
`jump label`	`newLabel = generateLabel(label, store)`	`jump newLabel`
`condJump exp l1 l2` (якщо вираз `exp` статичний)	`reducedExp = reduce(exp, store)`	`condJump reducedExp l1 l2`
`condJump exp l1 l2` (якщо вираз exp статичний і val = true)	`val = evaluate(exp, store)`	`jump l1`
`condJump exp l1 l2` (якщо вираз exp статичний і val = false)	`val = evaluate(exp, store)`	`jump l2`

Якщо інструкцію не показано в табл. 4 – значить вона додається в згенерований код без змін. Тут використовуються такі допоміжні функції:

– reduce(exp, store) – виконує згортання констант (constant folding) статичних частин динамічного виразу exp, наприклад, якщо b = 2, то вираз b * b + a може бути замінено на 4 + a;

– evaluate(exp, store) – обчислює значення статичного виразу exp;

– generateLabel(label, store) – створює нову мітку або повертає вже наявну для пари (label, store).

Після генерації коду для базового блоку сховище може бути змінено, у разі якщо зустрічалися присвоювання до статичних змінних. Потім потрібно визначити, які наступні блоки потрібно обробляти. Детально цей вибір описано в табл. 5. Наступні блоки додаються в робочий список разом з оновленим сховищем (позначається як store').

Таблиця 5 – Вибір наступних блоків

Інструкція переходу	Умова	Результат
`return`	---	`{}`
`jump label`	---	`{(label, store')}`
`condJump exp l1 l2`	exp – динамічний вираз	`{(l1, store'), (l2, store')}`
`condJump exp l1 l2`	exp обчислюється як true	`{(l1, store')}`
`condJump exp l1 l2`	exp обчислюється як false	`{(l2, store')}`

2.9 Аналіз потоку даних

Аналіз потоку даних (dataflow analysis) – це різновид аналізу програм, що дає змогу отримати інформацію про те, яким чином дані переміщуватимуться під час виконання програми. Задачі потоку даних формулюються на основі набору рівнянь над множинами, які асоціюються з вузлами та ребрами в графі потоку управління.

Аналізи потоку даних бувають двох видів:

– прямого напрямку – якщо інформація поширюється у напрямку ребер у графі потоку управління;
– зворотного напрямку – якщо інформація обернена до напрямку ребер у графі.

В аналізаторі реалізовано аналіз живості змінних, який належить до сімейства аналізів потоку даних. Іншими прикладами аналізу потоку даних є [20]:

– просування констант (constant propagation);
– усунення загальних підвиразів (common subexpression elimination);
– усунення зайвих операцій завантаження регістрів (elimination of redundant register load operation);
– аналіз досяжних визначень (reaching definition analysis).

2.10 Аналіз живості змінних

Над формою LIL відбувається аналіз, який називається *аналіз живості змінних* (liveness analysis). Цю інформацію буде використано під час перетворення LIL на SSA. Аналіз живості належить до сімейства алгоритмів аналізу потоку даних.

Аналіз живості має зворотний напрямок. Далі наведено формальне визначення живості для змінної і потім виконаний опис алгоритму.

Змінна v є живою в точці програми p, якщо існує такий шлях від p до її використання, на якому змінній v не присвоюється нове значення. Позначимо $LiveOut(n)$ як множину змінних, які є живими під час виходу з базового блоку n. Для обчислення цієї множини будемо використовувати таке рівняння

$$\bigcup_{\forall m \in successors(n)} UEVar(m) \cup (LiveOut(m) \setminus VarKill(m)) \qquad (1)$$

Тут використовуються такі додаткові визначення:
– $successors(n)$ – множина всіх наступних блоків для n;
– $UEVar(m)$ – множина всіх змінних, які використовуються в наступному блоці m до перевизначення в m;
– $VarKill(m)$ – безліч змінних, які перевизначаються в блоці m.

Розглянемо формулу (1) детальніше. $LiveOut(n)$ – це об'єднання всіх

змінних, які є живими на початку деякого блоку *m*, що послідує безпосередньо за блоком *n*. Інакше кажучи, змінна *v* належить $LiveOut(n)$, якщо вона є живою на початку хоча б одного наступного блоку, необов'язково для всіх наступних блоків.

Далі, внесок кожного наступного блоку m визначається цим виразом:

$$UEVar(m) \cup (LiveOut(m) \setminus VarKill(m))$$

Для того, щоб змінна *v* була живою на початку блоку *m*, потрібно щоб

- вона або використовувалася в блоці *m* до перевизначення в *m*, тобто *v* належить множині $UEVar(m)$;
- або вона є живою на виході з блоку *m* і не перевизначається в *m*, тобто $v \in LiveOut(m) \setminus VarKill(m)$.

Комбінуючи ці умови за допомогою оператору об'єднання, ми отримуємо внесок для одного з наступних блоків *n*. Інформація поширюється від наступних блоків до попередніх; саме тому аналіз живості має зворотний напрямок.

Для обчислення множин *LiveOut* можна скористатися таким алгоритмом.

1) Ініціалізувати множини $UEVar(n)$і $VarKill(n)$для кожного блока *n* у графі потоку управління.
2) Ініціалізувати робочий список worklist вихідним вузлом exit у графі потоку управління, присвоїти $LiveOut(exit) = \varnothing$.
3) Якщо worklist порожній – завершити алгоритм.
4) Отримати базовий блок n із worklist.
5) Перерахувати $LiveOut(n)$ на основі інформації для наступних блоків.
6) Якщо значення $LiveOut(n)$ змінилося – додати всі попередні блоки в worklist.
7) Перейти на крок 3.

Ініціалізація $UEVar(n)$ і $VarKill(n)$ відбувається один раз, оскільки ця інформація не змінюється протягом роботи алгоритму. Для обчислення цих множин обробляються визначення і використання змінних.

Алгоритм завершиться, тому що множини *LiveOut* скінченні, оскільки кількість змінних у функції скінченна. Повторне обчислення *LiveOut* для блоку може тільки збільшити цю множину. Звичайно, у формулі (1) є виключення множини *VarKill*, але її розмір не змінюється під час роботи алгоритму. Тому, з часом множини *LiveOut* перестануть змінюватися і алгоритм завершиться.

2.11 Обчислення домінаторів

Один блок *домінує* над іншим, якщо будь-який шлях від вхідного блоку до другого блоку проходить через перший блок. *Домінатор* (dominator) – це блок, що домінує над даним блоком.

Домінатори обчислюються на основі графу потоку управління. Інформацію про домінатори використовують у подальших стадіях аналізатора під час побудови форми SSA і під час просування предикатів, які буде описано в наступних підрозділах.

На рис. 18 зображено приклад графа потоку управління з базовими блоками A, B, C, D, E, F і G. У табл. 6 відображено домінатори для кожного блоку в цьому графі.

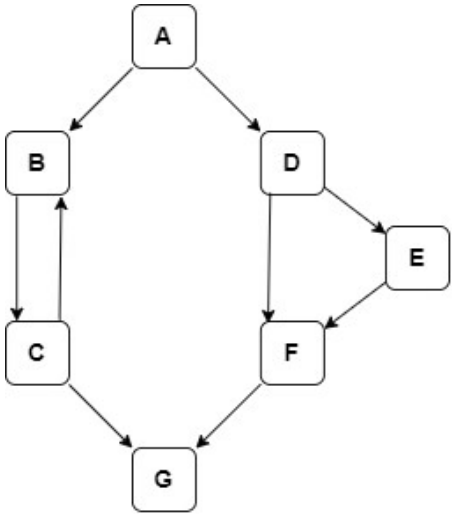

Рисунок 18 – Приклад графу потоку управління

Для обчислення домінаторів було запропоновано різні алгоритми. Далі буде розглянуто алгоритм, який належить до сімейства аналізів потоку даних. Перед наданням самого алгоритму потрібно визначити рівняння множин домінаторів.

Таблиця 6 – Блоки та їх домінатори

Блок	Домінатори
A	{A}
B	{A, B}
C	{A, B, C}
D	{A, D}
E	{A, D, E}
F	{A, D, F}
G	{A, G}

Для позначення множини блоків, які домінують над даним блоком b, будемо використовувати позначення $Dom(b)$. $Dom(b)$ має такі властивості:

– блок n у графі потоку управління домінує над блоком b, якщо n лежить на кожному шляху від вхідного блоку до блоку b;
– за визначенням, $b \in Dom(b)$;
– $Dom(b)$ містить кожен блок n, який домінує над b;
– якщо $x, y \in Dom(b)$, то або $x \in Dom(y)$, або $y \in Dom(x)$.

Спираючись на перелічені властивості, сформуємо рівняння потоків даних для *Dom*. У рівняннях нижче ми припускаємо, що n_0 – це вхідний блок графа.

$$Dom(n_0) = \{n_0\}$$
$$Dom(n) = \{n\} \cup \bigcup_{\forall m \in predecessors(n)} Dom(p) \quad (2)$$

Множина $Dom(n_0)$ для вхідного вузла складається з єдиного елементу – самого вхідного вузла.

Для решти вузлів $Dom(n)$ обчислюється на основі попередників. Вузол $x \in Dom(n)$ тільки тоді, коли він є домінуючим для всіх попередників блоку n. Це стверджує права частина об'єднання в рівнянні (2). Ліва частина додає сам вузол n до множини $Dom(n)$.

Блоки в графі управління можна пронумерувати кількома способами. Зворотний порядок (post-order) – це порядок, у якому всі наступні блоки відвідуються, наскільки це можливо, перед відвідуванням конкретного блоку. Порядок, протилежний зворотному (reverse post-order), який ми позначимо RPO – це порядок, за якого попередні блоки відвідуються, наскільки це можливо, перед поточним блоком. Більшість графів мають більше однієї нумерації RPO, однак основна зазначена властивість, яка нас цікавить, при цьому зберігається.

Таблиця 7 – Порядки нумерації графу потоку управління

	A	B	C	D	E	F	G
Зворотний порядок	7	3	2	6	5	4	1
RPO	1	5	6	2	3	4	7

Обидва розглянуті порядки можна обчислити, скориставшись обходом у глибину (depth-first search) графа. Приклад нумерація блоків у графі на рис. 18 показано в табл. 7.

На основі рівнянь визначимо алгоритм обчислення Dom(n). Це алгоритм аналізу потоку даних прямого напрямку, оскільки інформація поширюється від попередників до наступних блоків. Тут N – множина всіх блоків у графі потоку управління.

1) Пронумерувати всі блоки в порядку RPO.
2) Для кожного блоку *n*, задати початкове значення $Dom(n)=\{1,\ldots,|N|\}$.
3) Обчислюємо $Dom(n)$ для кожного блоку *n* у порядку RPO згідно з формулою (2).
4) Якщо якась із множин $Dom(n)$ змінилася – перейти на крок 3.
5) Інакше – завершити алгоритм.

Причина, через яку блоки відвідуються в порядку RPO, полягає в тому, що інформація повинна поширюватися від попередніх блоків до наступних. Це дає змогу зменшити кількість ітерацій алгоритму і, отже, поліпшити його продуктивність. Приклад роботи алгоритму відображено в табл. 8.

Для обчислення домінаторів у цьому прикладі достатньо двох ітерацій. Після другої ітерації жодних змін із множинами $Dom(n)$ не відбувається і алгоритм завершується.

Таблиця 8 – Приклад виконання алгоритму пошуку домінаторів

	A	B	C	D	E	F	G
RPO	1	5	6	2	3	4	7
Після ініціалізації	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}
Ітерація 1	{1}	{1,5}	{1,5,6}	{1,2}	{1,2,3}	{1,2,4}	{1,7}
Ітерація 2	{1}	{1,5}	{1,5,6}	{1,2}	{1,2,3}	{1,2,4}	{1,7}

Покажемо, що алгоритм завершується в загальному випадку. На початку алгоритму множини $Dom(n)$ ініціалізуються значенням $\{1,\ldots,|N|\}$, де N – множина всіх блоків. На кожній ітерації для кожного блоку n, $Dom(n)$ може або зменшуватися, тому що зменшується $Dom(p)$ для одного з попередників p блоку n, або залишатися з таким самим розміром. Довжина множин Dom обмежена кількістю блоків у графі, а змінюватися множини Dom можуть тільки в бік зменшення, тому на якійсь ітерації множини перестануть змінюватися і алгоритм завершиться.

2.12 Форма SSA

Static single assignment (єдине статичне присвоювання, SSA) – це форма, у якій кожній змінній у тексті програми присвоюється значення не більше одного разу. В аналізаторі SSA застосовується в подальшому для просування предикатів.

Форма SSA була розроблена наприкінці 1980-х років [21] як проміжне представлення, що давало змогу ефективно і точно виконати аналіз потоку даних. Відтоді форма SSA швидко набула популярності через її інтуїтивну природу та простий алгоритм побудови. На сьогоднішній день, форма SSA використовуються в більшості комерційних компіляторах і компіляторах з

відкритим вихідним кодом, таких як GCC, LLVM, HotSpot (віртуальна машина Java) і V8 (движок Javascript).

Переваги форми SSA полягає в тому, що значно скорочує витрати на побудову та використання ланцюгів «визначення-використання» (def-use chains). Причина полягає в тому, що у формі SSA кожна змінна має єдине місце визначення. Тому в ланцюгах «визначення-використання» потрібно набагато менше зв'язків між визначеннями і використаннями змінних, а також скорочується час на знаходження потрібної інформації про змінні.

Ще однією перевагою SSA є те, що інформація про визначення та використання може бути легко оновлена. Це важливо, тому що під час застосування оптимізацій, таких як просування констант (constant propagation), код програми змінюється, деякі блоки можуть бути видалені, і ця інформація має бути оновлена.

Для досягнення унікальності визначень змінних у SSA використовуються:

– вставка φ-функцій;

– перейменування змінних.

φ-функція – це особлива конструкція форми SSA. Вона дозволяє з'єднувати потік значень із різних блоків в одну змінну. φ-функції вставляються в так званих точках сходження (join points). Точки сходження – це блоки в графі потоку управління, які мають більше одного попереднього блоку; у них сходяться потоки значень із різних блоків.

На рис. 19 зображено приклад ситуації, коли необхідна вставка φ-функції в блоці B3. Це необхідно з тієї причини, що в блоках B1 і B2 змінній x присвоюється значення, яке потім використовується в блоці B3. Під час виконання програми потік управління може потрапити як у блок B1, так і в блок B2. Для того, щоб подальший аналіз враховував ці дві можливості і додається φ-функція на початку блоку B3.

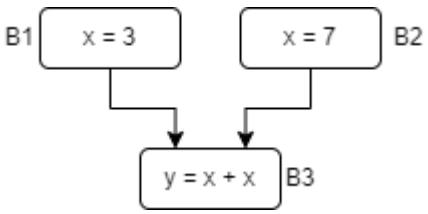

Рисунок 19 – Фрагмент графу потоку управління, на якому необхідно додати φ-функцію

Приклад перетворення на форму SSA показано на рис. 20: додано φ-функцію в блоці B3, змінну x перейменовано в місцях оголошення та використання.

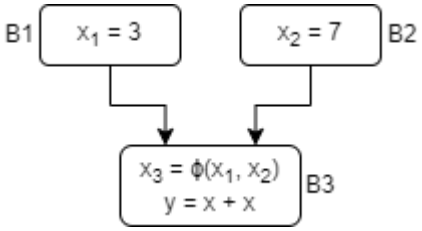

Рисунок 20 – Додавання φ-функції

Побудову SSA можна розділити на дві частини, які розглядатимуться нижче:

– розстановка φ-функцій;

– перейменування змінних.

2.12.1 Розстановка φ-функцій

Простим рішенням для розстановки φ-функцій було б вставка їх у всіх точках сходження. Однак, більшість φ-функцій у цьому випадку були б зайвими. Велика кількість φ-функцій тільки ускладнить і сповільнить подальший аналіз, оскільки він має враховувати всі зайві φ-функції. Для ефе-

ктивного аналізу необхідно визначити, в яких місцях φ-функції дійсно потрібні.

Далі розглянуті поняття, які будуть використовуватися під час розстановки φ-функцій. Нагадаємо, що базовий блок *n* домінує над базовим блоком *m*, якщо кожен шлях від вхідного блоку до блоку m проходить через блок *n*. Блок *n строго домінує* над блоком *m*, якщо до того ж *n* не дорівнює *m*.

Блок *n* називається *безпосереднім домінатором* блоку *m*, якщо він є найближчим блоком до *m*, який строго домінує над *m*. Вхідний блок не має безпосереднього домінатора. Усі інші блоки в графі потоку управління мають рівно один безпосередній домінатор.

Межа домінування для блока *n* – це множина таких блоків *m*, що у блока *m* є попередній блок *k* такий, що *n* строго домінує над *k*, але *n* не домінує строго над *m*. Межа домінування для блока *n* позначається $DF(n)$. Для обчислення меж домінування використовується множини $Dom(n)$, які були описані раніше.

Припустимо, що змінна *x* визначена в блоці *n*. Також припустимо, що *m* – це один із блоків, над якими суворо домінує блок *n*; у *m* є використання змінної *x*. Якщо існує блок *p* між блоками *n* і *m*, у якому є перевизначення змінної *x*, то тоді потрібно додавати φ-функцію для *x*. Якщо такого блоку *p* немає, то і φ-функцію додавати не потрібно, тому що в блок *m* завжди буде потрапляти значення змінної *x*, яке було задано в блоці *n*. Крім того, вставка необхідна в місцях, де закінчується область домінування даного блоку, тому що до цих місць можна прокласти шлях, який не включатиме блок *n*.

Отже, якщо в блоці *n* відбувається присвоєння змінній *x*, то значить потрібно поставити φ-функцію на початку кожного блоку з множини $DF(n)$. Однак під час додавання φ-функції відбувається нове визначення для змінної, тому процес пошуку межі домінування потрібно продовжити для блоку, в якому було додано φ-функцію [22].

Для розстановки φ-функцій можна використовувати такий алгоритм, який виконується для кожної використаної змінної в програмі.

1) Нехай x – змінна, для якої будуть розставлені φ-функції.
2) Ініціалізувати робочий список блоків worklist. Для кожного блоку n, у якому є присвоювання до змінної x, додати в worklist множину $DF(n)$.
3) Якщо список worklist порожній – закінчити алгоритм.
4) Взяти блок із worklist, назвемо його m.
5) Якщо блок m позначений – перейти на крок 3.
6) Якщо в блоці не відбувається присвоювання до x – додати в worklist $DF(m)$.
7) Якщо змінна є живою на початку блоку m – додати φ-функцію для змінної x на початку блоку m.
8) Позначити блок m.
9) Перейти на крок 3.

Зауважимо, що в наведеному алгоритмі використовується інформація про живість змінних. Починаючи від блоків, у яких визначено змінну, алгоритм переходить від блоків до їхніх меж домінування і додає в потрібних місцях φ-функції. Оброблені блоки позначаються, щоб не обробляти їх заново.

У реалізації аналізатора використовується модифікований варіант φ-функцій, якій зберігає їхній зміст, описаний вище. Кожен блок, який повинен мати φ-функції, спочатку отримує замість цього набір змінних – набір параметрів. У місцях, де відбуваються переходи на такі блоки, вказується, які змінні будуть передані в наступний блок, тобто формується набір аргументів. У цьому випадку кожен блок робиться схожим на визначення функції, а перехід на блок – перехід у функцію без повернення.

2.12.2 Перейменування змінних

Під час перейменування змінних ім'я змінної у вихідній програмі береться за основу, але при цьому до цього імені додається унікальний номер. Таким чином нове ім'я складається зі старого імені та унікального номера.

Під час перейменування змінних використовується структура даних, яка називається *дерево домінаторів* (dominator tree) [22]. Множина вузлів у дереві домінаторів збігається з множиною блоків у графі потоку управління. Якщо є блоки *n* і m, що з'єднані ребром у дереві домінаторів, то блок *n* є безпосереднім домінатором блока *m*. *Коренем* у дереві домінування є вхідний блок. На рис. 21 зображено приклад графа управління та відповідного дерева домінаторів.

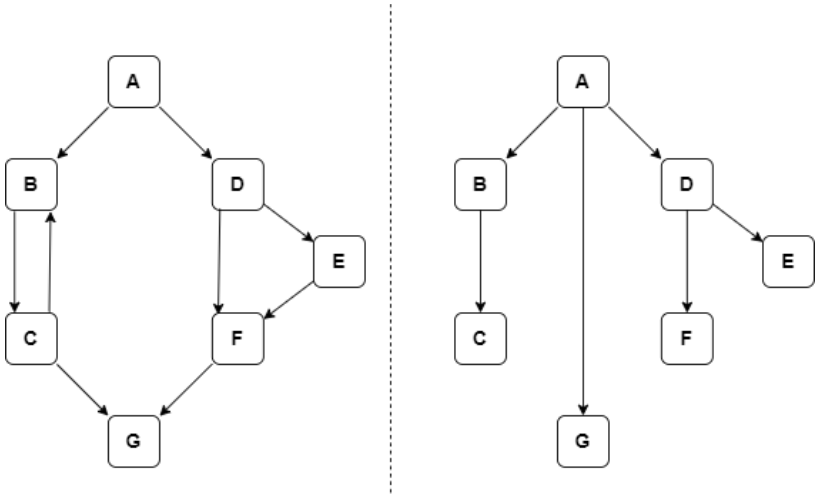

Рисунок 21 – Граф потоку управління (ліворуч) та відповідно дерево домінаторів

Під час перейменування змінних використовується стек і лічильник для кожної перейнятої у вихідній програмі. Блоки відвідуються в такому порядку, в якому вони розташовані в дереві домінаторів. Під час обробки інструкцій присвоювання створюється нове ім'я для змінної, лічильник

збільшується і на стек додається нове ім'я. Усі використання змінної в даному блоці замінюються новим ім'ям зі стека. Після обробки всіх інструкцій алгоритм переходить до спадкоємців у дереві домінаторів. При обробці всіх спадкоємців ім'я зі стека видаляється.

На рис. 22 показано приклад перетворення LIL в SSA. Усі змінні у формі SSA отримують крім імені ще й унікальний номер, цей номер записується через. Блоки bb2, bb6 отримують параметри для змінних x, y. Попередні блоки передають ці змінні при переході в них.

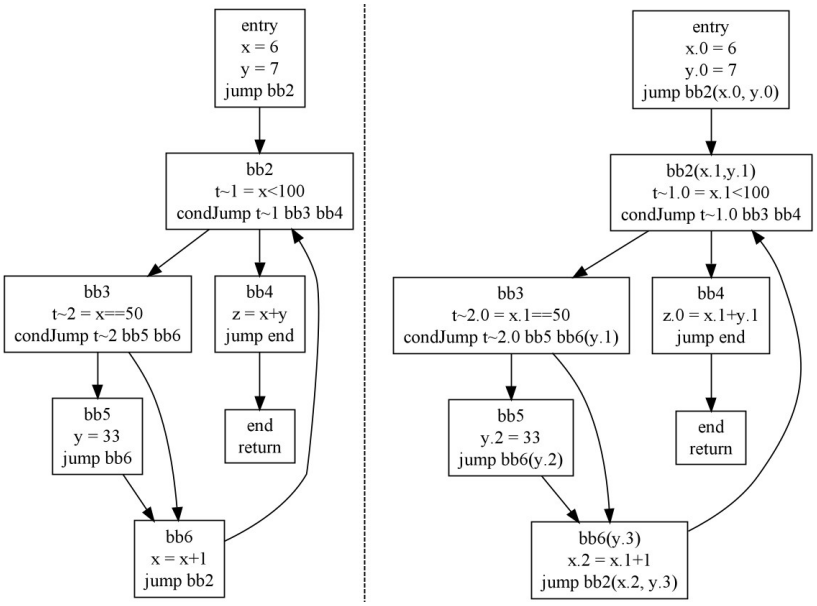

Рисунок 22 – Перетворення LIL (ліворуч) в SSA (праворуч)

2.13 Просування предикатів

Наступна фаза після перетворення у форму SSA – *просування предикатів*. Просування предикатів була розроблена у ході виконання цієї роботи Завдання цієї фази полягає в тому, щоб знати, які базові блоки будуть виконуватися яким процесом. Ця інформація потім використовується під час побудови графа операцій.

Розглянемо приклад коду, який часто використовується під час написання програм MPI.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
} else if (rank == 1) {
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

Тут викликається метод Rank і ранг процесу зберігається у змінній rank. Потім відбувається перевірка на ранг поточного процесу. Завдяки умовам if-інструкцій можливо зрозуміти, що Send виконуватиме процес 0, а Recv – процес 1. На рис. 23 зображено відповідний граф потоку управління.

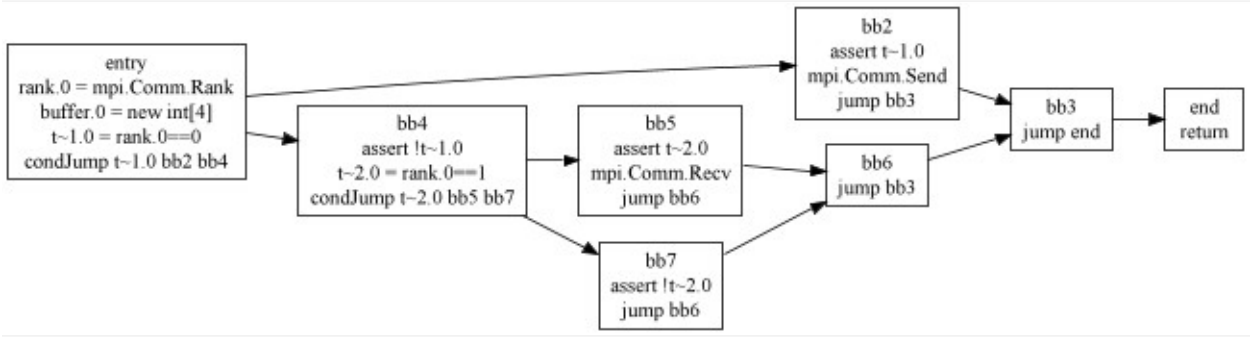

Рисунок 23 – Граф потоку управління для фрагмента коду з операціями MPI

Під час просування предикатів використовуються інструкції assert, які були додані раніше. Інструкції assert вказують умову, за якої буде виконано той чи інший блок. Ми можемо обчислити цю умову і спробувати отримати з неї інформацію про ранг процесу. Наприклад, блок bb2 починається з інструкції assert t~1.0. Змінна t~1.0 містить значення true якщо змінна rank. 0 дорівнює 0. Змінній rank.0 присвоюється результат виклику методу Rank. Звідси можна зробити висновок, що блок bb2 виконується процесом із

рангом 0. Так само з інструкцій assert отримується інформація про ранг інших процесів.

Граф потоку управління на рис. 23 містить блоки у формі SSA. Це дає змогу з легкістю знаходити, які значення присвоюються кожній змінній, оскільки кожна змінна має лише одне місце визначення.

Наведемо опис алгоритму просування предикатів.

1) Для кожного блоку розглядається його assert інструкції.
2) За допомогою інформації про визначення обчислюється ранг процесу.
3) Якщо інформацію про ранг процесу неможливо отримати – перейти на крок 1 до наступного блоку.
4) Асоціювати поточний блок з отриманим рангом.
5) Асоціювати кожен блок, над яким домінує поточний блок, з отриманим ранг.
6) Перейти на крок 1 до наступного блоку.

У цьому алгоритмі використовується інформація про домінаторів, яка описувалася раніше. Причина, з якої асоціюються всі блоки, над якими домінує поточний блок, така, що процес не змінюється протягом виконання програми. Якщо для блоку встановлено, який саме процес його виконує, то цей самий процес виконуватиме і всі блоки, над якими він домінує.

У результаті роботи алгоритму для графу потоку управління на рис. 23 буде обчислено, що блок bb2 буде виконуватися процесом 0, а блок bb5 буде виконуватися процесом 1.

2.14 Граф операцій

Граф операцій являє собою орієнтований граф. Цей граф відображає послідовність, у якій запускаються операція Send і Recv, а також які саме процеси запускають ці операції. Граф операцій – це останнє проміжне пред-

ставлення програми перед створенням сітки Петрі для програми. Граф операцій також був розроблений у ході виконання цієї роботи. Побудова графа операцій відбувається на основі графа потоку управління.

Граф операцій є поданням досить високого рівня, оскільки він ігнорує багато деталей програми. Наприклад, граф операцій не залишає інформації про змінні, присвоювання, умови виконання базових блоків. Виділяється найважливіша інформація – яким чином процеси обмінюються між собою повідомленнями.

Граф операцій складається з таких основних вузлів:

– вузол Send – відповідає виклику операції Send;
– вузол Recv – відповідає виклику операції Recv;
– проміжний вузол – пов'язує вузли Send і Recv та інші проміжні вузли;
– вузли поділу і з'єднання – використовуються в разі, коли різні операції можуть бути викликані одним процесом за різних умов.

Наведемо фрагмент коду, для якого буде побудовано граф операцій.

```
if (rank == 0) {
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 1, 0);
} else if (rank == 1) {
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 0, 0, 0);
}
```

На рис. 24 ліворуч показано граф потоку управління для фрагмента, праворуч показано граф операцій.

Зауважимо, що проміжні вузли (Intermediate) створюються на основі базових блоків. Наприклад, вхідний вузол у графі операцій «Intermediate entry_3» пов'язаний із вхідним вузлом у графі потоку управління.

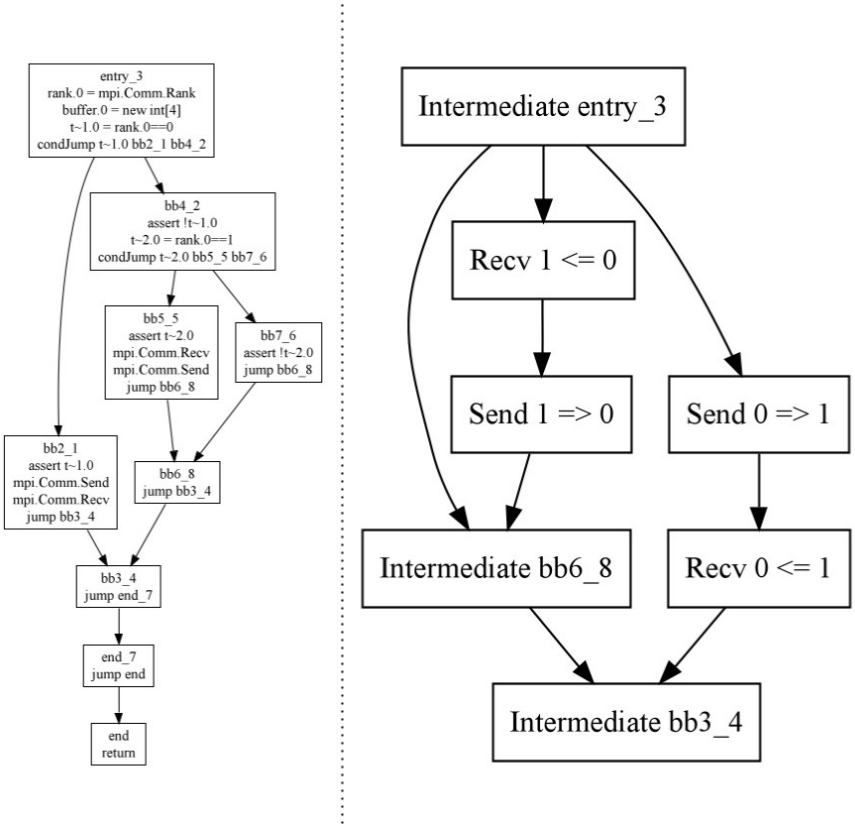

Рисунок 24 – Графу потоку управління (ліворуч) та відповідний граф операцій (праворуч)

Якщо з проміжного вузла виходить більше одного вузла, то це означає, що відбувається поділ на процеси. Наприклад, зі згаданого вхідного вузла виходять вузли «Recv 1 <= 0» і «Send 0 => 1». Тому тут відбувається поділ на процеси 1 і 0; кожна гілка буде виконуватися окремим процесом.

Якщо в проміжний вузол сходиться більше ніж один вузол, то це означає, що починається спільна ділянка для кількох процесів. Наприклад, у вузол «Intermediate bb6_8» переходить вузол «Send 1 => 0», який виконується процесом 1. Крім того, в «Intermediate bb6_8» переходить і вхідний вузол. У цьому випадку це означає, що якщо ранг процесу не дорівнює 0 і 1, то процес переходить із вхідного вузла безпосередньо у вузол «Intermediate bb6_8».

Далі розглянутий алгоритм перетворення графа потоку управління на граф операцій. У ньому використовується асоціація між базовими блоками і

рангами процесів, яку було обчислено під час просування предикатів. Як і в багатьох алгоритмах, описаних у цій роботі, тут використовується робочий список – worklist, кожен елемент якого є парою (label, node), де label – це мітка базового блоку, який буде оброблятися, а node – вузол графа операцій, з якого буде виходити ребро до нового вузла.

1) Ініціалізувати worklist парою (entry, node0), де entry – це мітка вхідного базового блоку, node0 – проміжний вузол, який відповідає вхідному блоку. node0 – це початковий вузол для графа операцій.
2) Якщо worklist порожній – алгоритм завершується.
3) Інакше взяти з worklist таку пару (label, node). Нехай block – базовий із міткою label.
4) Якщо block має більше одного попереднього блоку – створити проміжний вузол new, додати ребро від node до new і присвоїти node = new.
5) Якщо для блоку з міткою label відсутній ранг процесу, то виконати addNextBlocks(label, node) і перейти на крок 2.
6) Переглянути кожну інструкцію в block, додавати вузли Send і Recv і додавати ребра для створених вузлів. Нехай lastNode – останній створений вузол або node, якщо жоден вузол не був створений. Виконати addNextBlocks(label, lastNode).
7) Перейти на крок 2.

В алгоритмі використовується допоміжна функція addNextBlocks(label, node), яка додає в worklist пари (next, node) для кожного наступного блоку з міткою next.

2.15 Побудова сітки Петрі з графу операцій

Завершальним проміжним поданням в аналізаторі є сітка Петрі, яка зрештою записується у файл . net формату Tina. сітка Петрі будується з графу

операцій.

На рис. 25 показано граф операцій (ліворуч) і побудовану відповідну сітку Петрі (праворуч).

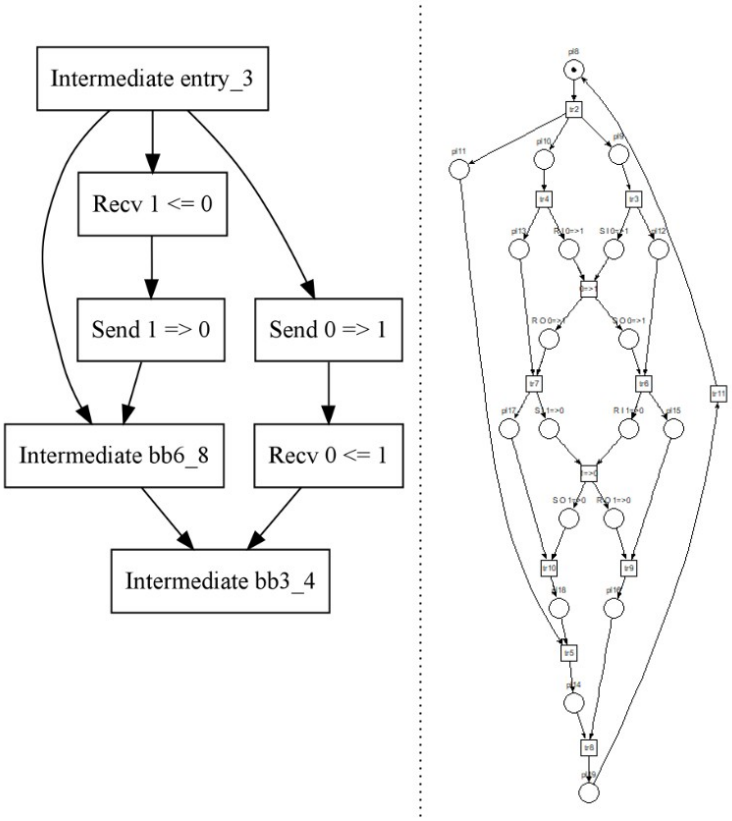

Рисунок 25 – Граф операцій (ліворуч) та побудована для нього сітка Петрі (праворуч)

Якщо порівняти граф операцій із сіткою Петрі, то можна помітити, що сітка Петрі містить набагато більше елементів. сітка Петрі є більш низькорівневим представленням і вимагає того, щоб семантика програми, а точніше семантика викликів операцій MPI, була показана більш явно. Однак, сама структура двох форм є досить схожою.

Під час побудови сітки Петрі використовується алгоритм, схожий з алгоритмом побудови графа операцій. Тут також використовується робочий список пар (ogNode, transition), де ogNode – це вузол із графа операцій, transition – наступний перехід у сітки Петрі.

3 АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Простий приклад програми без тупиків

У цьому підрозділі описується порядок запуску аналізатора і розбір результатів його запуску для простого прикладу коректної програми MPI, що не має тупиків. Програма показана нижче.

```
import mpi.MPI;
public class MpiSendRecvSimple {
  public static void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();
    int[] buffer = {0, 2, 3, 4};
    int tag = 1;
    if (rank == 0) {
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, tag);
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 1, tag);
    } else if (rank == 1) {
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, tag);
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 0, tag);
    }
    MPI.Finalize();
  }
}
```

У цій програмі тупиків не повинно виникнути ніколи, оскільки процеси викликають операції MPI у правильній послідовності: спочатку процес 0 відсилає повідомлення процесу 1, потім процес 1 відсилає повідомлення процесу 0.

Аналізатор було реалізовано мовою Scala, тому для його збирання та запуску використовується інструмент під назвою sbt (simple build tool, простий інструмент збирання). sbt також використовується для управління залежностями проєкту, тобто закачуванням необхідних для проєкту бібліотек. Опис бібліотек відбувається у файлі build.sbt.

Передусім, потрібно запустити оболонку sbt з кореневої директорії проекту:

```
deadlock-finder> sbt
[info] welcome to sbt 1.6.2 (Azul Systems, Inc. Java 17.0.2)
```

```
[info] loading global plugins from …
….
[info] sbt server started at local:sbt-server-8374cb68f19317221c80
[info] started sbt server
```

Далі виконується запуск аналізатора за допомогою команди run із зазначенням шляху до вхідного файлу Java:

```
sbt:deadlock-finder> run examples/parallel/MpiSendRecvSimple.java
[info] running deadlockFinder.Main examples/parallel/MpiSendRecvSimple.java
[success] Total time: 2 s, completed 26 нояб. 2022 г., 08:25:43
```

Аналізатор успішно завершує свою роботу. Побудована сітка Петрі зберігається за шляхом target\net.net. Ця сітка зображена на рис. 26. На цьому простому прикладі коректність програми можна побачити навіть візуально, адже тут відсутні взаємні одночасні залежності, які спричиняють тупики.

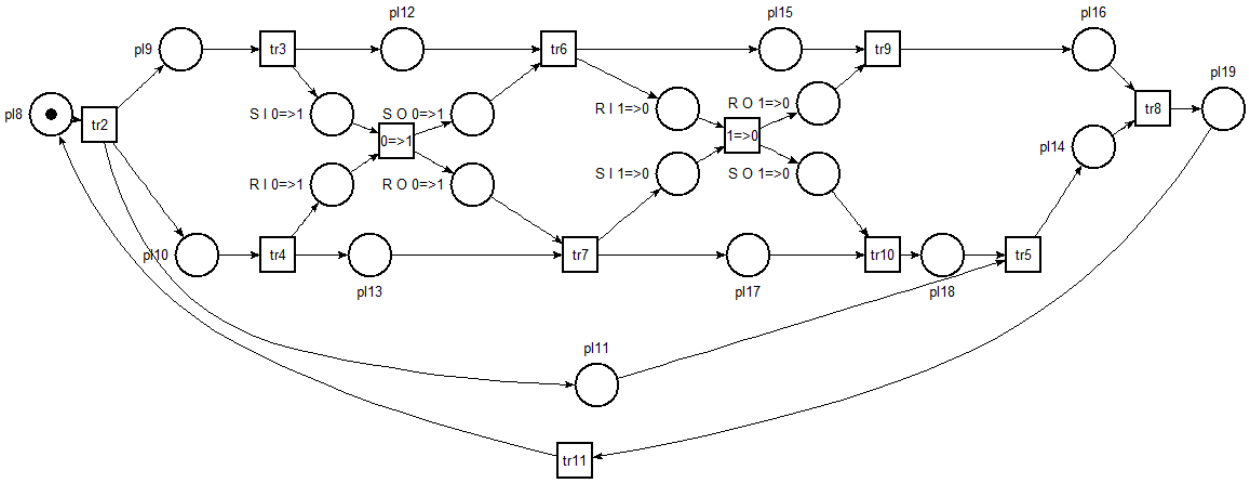

Рисунок 26 – Приклад сітки Петрі для коректної програми

Для перевірки на відсутність тупиків буде використовуватися утиліта sift із пакета Tina. sift – це високопродуктивний дослідник і верифікатор простору станів.

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -dead
```

```
16 marking(s), 20 transitions(s)
0.000s
```

Для утиліти sift вказується сітка Петрі й аргумент -dead, який вказує, що sift має шукати тупики в заданій сітці. У висновку утиліти не вказано наявність тупиків; вказано лише кількість маркувань (16) і кількість переходів (20) у просторі станів. Це означає, що ця програма MPI є коректною.

3.2 Простий приклад програми з тупиком

Далі наведений фрагмент програми з тупиком.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

Для нього виконується побудова сітки Петрі (див. рис. 27) аналізатором у такий самий спосіб, як показано в попередньому підрозділі.

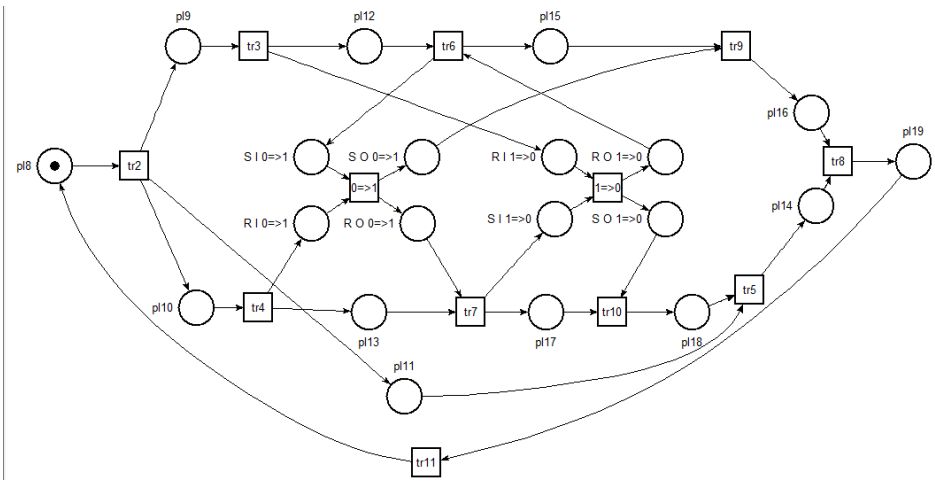

Рисунок 27 – Сітка Петрі для програми з тупиком

У результаті запуску sift показує такий результат:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -dead
some state violates condition -f:
  {R I 0=>1} {R I 1=>0} pl11 pl12 pl13
  firable:
5 marking(s), 5 transitions(s)
0.000s
```

Результати згодні з описом у підрозділі 2.2. sift вказує маркування, за якого немає дозволених (firable) переходів. При маркуванні «{R I 0=>1} {R I 1=>0} pl11 pl12 pl13» у сітці відбувається тупик. Фішки в позиціях «R I 0=>1» і «R I 1=>0» говорять про те, що були викликані операції Recv обома процесами 0 і 1.

3.3 Приклад із простим циклом

У цьому підрозділі показано приклад програми з циклом. Аналізатор може обробляти цикли, умови яких можна обчислити під час компіляції завдяки частковому обчисленню програми (див. підрозділ 2.8). Нижче наведено фрагмент програми.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  int counter = 0;
  for (int i = 0; i < 2; i++) {
    MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 1, tag);
    counter++;
  }
  for (int j = 0; j < counter; j++) {
    MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 1, tag);
  }
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, tag);
}
```

У цьому фрагменті процес 0 виконує два цикли. Перший цикл виконується двічі, у ньому відбувається виклик Send до процесу 1. Також у першому

циклі збільшується лічильник (counter), який буде використовуватися як умова для другого циклу. Таким чином, умова другого циклу залежить від першого циклу. У другому циклі відбувається виклик Recv від процесу 1. Процес 1 виконує відповідні виклики Recv і Send, тільки без використання циклів.

Для підтвердження того, що цикли для процесу 0 обробляються правильно, на рис. 28 показано граф операцій.

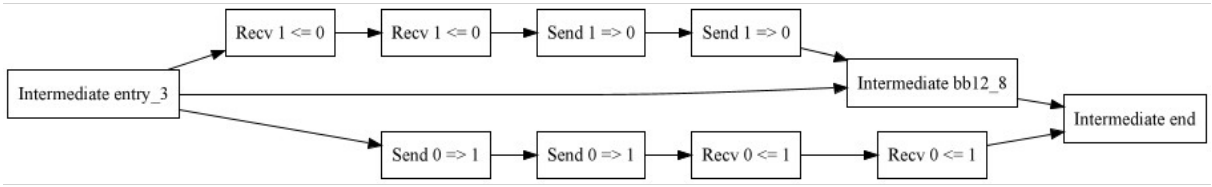

Рисунок 28 – Граф операцій для програми з циклом

Сітка Петрі, побудована за допомогою аналізатора, показана на рис. 29.

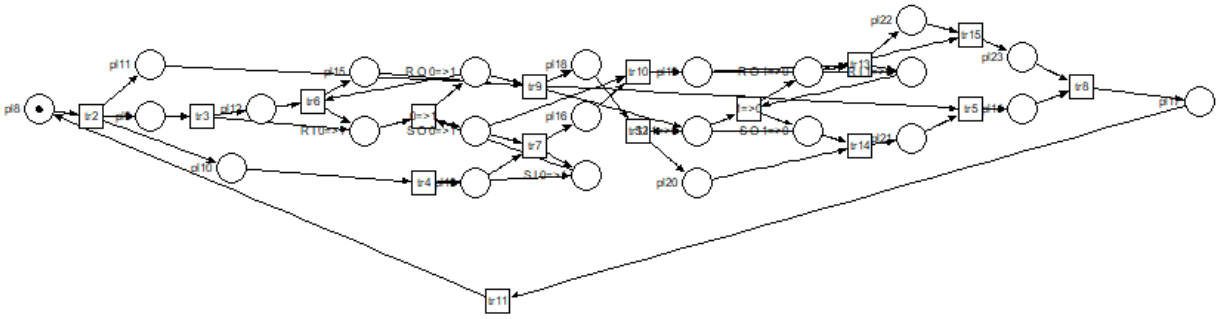

Рисунок 29 – Сітка Петрі для програми з циклом

Утиліта sift дає такий результат:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -dead
24 marking(s), 30 transitions(s)
0.000s
```

Результат свідчить про те, що в цій програмі відсутні тупики. Дійсно, процеси викликають операції MPI у правильній послідовності: спочатку процес 0 відсилає два повідомлення процесу 1, потім процес 1 відсилає два повідомлення процесу 0.

3.4 Складніший приклад із тупиком

Як було згадано раніше, під час отримання повідомлення можна вказати довільного одержувача. У MPJ Express для цього використовується константа MPI.ANY_SOURCE. У цьому разі під час отримання повідомлення не відбуватиметься перевірка на ранг відправника. Нижче показано складніший фрагмент програми, в якій застосовується ця константа. Цей приклад узято з набору тестів MBI (MBI Bugs Initiative) [23] і адаптовано під Java.

```
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 3, 0);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
} else if (rank == 1) {
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 3, 0);
} else if (rank == 2) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, 0);
} else if (rank == 3) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 1, 0);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, 0);
} else if (rank == 4) {
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 2, 0);
}
```

Аналізатор побудує сітку Петрі, яка показана на рис. 30.

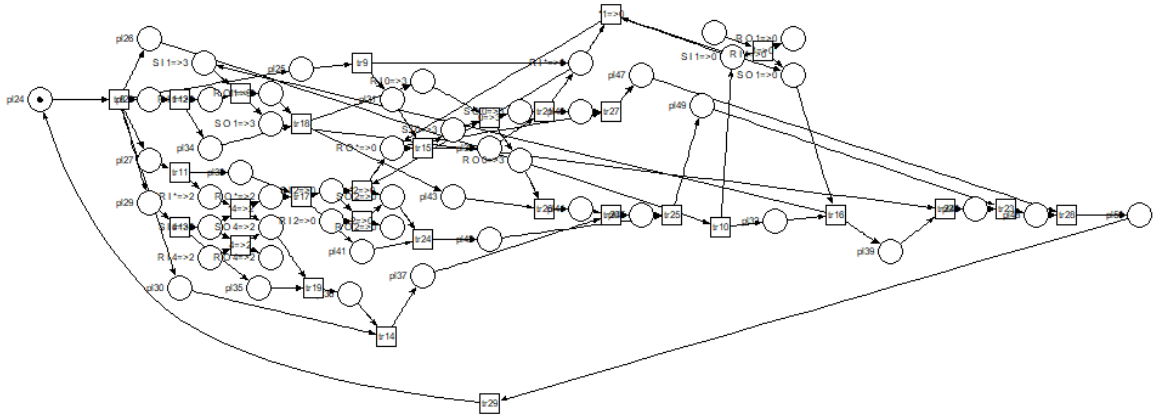

Рисунок 30 – Сітка Петрі для складного прикладу програми з тупиком

Результат запуску sift буде таким:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -dead
some state violates condition -f:
  {R I 1=>3} {S I 0=>3} {S I 1=>0} pl32 pl34 pl37 pl38 pl42
  firable:
302 marking(s), 833 transitions(s)
0.000s
```

sift знаходить таке тупикове маркування: {R I 1=>3} {S I 0=>3} {S I 1=>0} pl32 pl34 pl37 pl38 pl42»

. Із цього можна визначити, що тупик відбувається, коли:

– процес 3 отримує повідомлення від процесу 1 (R I 1=>3);

– процес 0 надсилає повідомлення процесу 3 (S I 0=>3);

– процес 1 надсилає повідомлення процесу 0 (S I 1=>0).

Цей тупик не може вирішитися, оскільки для того, щоб процес 3 отримав повідомлення від процесу 1, необхідно, щоб процес 1 успішно надіслав повідомлення процесу 0. Але це неможливо, оскільки процес 0 заблокований надсиланням повідомлення процесу 3.

Така ситуація сталася тому, що перший виклик Recv у процесі 0 отримав повідомлення від процесу 2. Якби процес 0 отримав у цьому місці повідомлення від процесу 1, то тупика б не сталося. Тупикова ситуація в цьому

прикладі не відбуватиметься під час кожного запуску програми, але відбуватиметься недетерміновано. Причиною є те, що під час виклику Recv з константою MPI.ANY_SOURCE повідомлення від різних процесів обирається довільним чином.

3.5 Рекомендації щодо усунення тупиків

Після знаходження тупиків за допомогою аналізатора, потрібно детально розглянути, яким чином процеси залежать один від одного. За необхідності, потрібно змінити черговість викликів Send і Recv. Наприклад, у підрозділі 3.2 обидва процеси викликали спочатку Recv і тому нескінченно очікували один від одного Send. Цю ситуацію можна вирішити просто поставивши в процесі 0 спочатку Send і потім Recv.

Якщо тупик виникає за наявності операцій отримання від довільного відправника (MPI.ANY_SOURCE) і є більш ніж один підходящий відправник, то можна використовувати теги для того, щоб упорядкувати одержувані повідомлення.

Крім того, для розв’язання тупиків під час використання блокувальних операцій можна використовувати аналоги, що не виконують блокування [24]. Наприклад, в одному з процесів можна використовувати замість операції Recv операцію Irecv. У цьому разі операція Irecv не блокуватиме процес, але одразу здійснить повернення. Потім після виконання необхідних дій, коли отримані дані знадобляться, можна викликати блокувальний виклик Wait, який очікуватиме до завершення отримання даних.

ВИСНОВКИ

У результаті роботи було виконано розробку моделей, методів та алгоритму аналізу коректності розподілених програм.

Було реалізовано статичний аналізатор коду, який виконує автоматичну побудову розробленої моделі у вигляді сітки Петрі за даною програмою MPI. Для аналізу програми використовувалися як традиційні підходи з області розробки компіляторів (домінатори, форма SSA, аналіз потоку даних, та інші), так і нові підходи, спеціалізовані під поточне завдання, такі як визначення рангів процесів і граф операцій.

Незважаючи на те, що на вхідні програми накладаються обмеження, розроблені моделі можуть слугувати фундаментом для подальших досліджень.

Застосовність аналізатора було показано на складному прикладі програми, у якій тупик виникає недетерміновано. Далі подані можливі напрямки подальших досліджень.

На даний момент аналізатор працює тільки з реалізацією MPI для Java – MPJ Express. У майбутньому можлива підтримка мови C. Для цього необхідно лише підключити до аналізатора парсер мови C та зробити перетворення синтаксичного дерева в HIL. Наступні фази та проміжні уявлення не доведеться змінювати. Підтримка мови C дасть можливість порівняти аналізатор зі схожими рішеннями та виміряти його продуктивність для складніших прикладів.

Аналізатор не обробляє виклики функцій, визначених у коді користувача. Однак, це можливо вирішити це завдяки вбудовуванню функцій, на кшталт inline функцій C++. Тіло функції, що викликається, буде вбудовуватися в місці її виклику, при заміні параметрів аргументами, які були передані у виклик. Це перетворення буде виконано над формою HIL.

Як було зазначено, результатом аналізатора є сітка Петрі, яка перевіря-

ється на наявність тупиків запуском симуляції в пакеті Tina. Можливо зробити запуск Tina або sift автоматичним, щоб аналізатор викликав цю програму та опрацьовував результати її роботи. Завдяки цьому використовувати аналізатор буде набагато легше.

Наступним кроком буде витяг з результатів Tina інформації про те, за якого маркування стався тупик. Ця інформація використовуватиметься для того, щоб повідомити користувача, в якому саме місці його програми (рядок, файл) можливий тупик.

На даний момент, можлива коректна обробка циклів лише з умовою, яку можна обчислити на етапі компіляції. В подальшому, потрібно знайти спосіб обробки циклів з довільними умовами. Це дозволить аналізувати програми MPI більш великого масштабу і складності, чим приклади у цій роботі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Javed A., Qamar B., Jameel M., Shafi A., Carpenter B. Towards Scalable Java HPC with Hybrid and Native Communication Devices in MPJ Express. International Journal of Parallel Programming (IJPP) 2015. Springer.
2. Kirk D. B., Hwu W. W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2016. 576 p.
3. Robey R., Zamora Y. Parallel and High Performance Computing. Manning Publications, 2021. 704 p.
4. MPI: A Message-Passing Interface Standard Version 3.1. Knoxville, Tennessee: Message Passing Interface Forum, June 4, 2015. 836 p.
5. Laguna I., Marshall R., Mohror K., Ruefenacht M., Skjellum A., Sultana N. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC ’19), November 17–22, 2019, Denver, CO, USA. ACM, New York, NY, USA. 12 p.
6. MPJ Express Project. URL : http://www.mpjexpress.org/index.html. (дата звернення 10.11.2022).
7. Droste, A., Kuhn, M., and Ludwig, T. MPI-checker: Static Analysis for MPI. 2nd Workshop on the LLVM Compiler Infrastructure in HPC, 2015. P. 1–10.
8. Yu H., Chen Z., Fu X., Wang J., Su Z., Sun J., Huang C., Dong W. Symbolic Verification of Message Passing Interface Programs. 42nd International Conference on Software Engineering, May 23-29, 2020, Seoul, South Korea. ACM, New York, NY, USA. 13 p.
9. King J.C. Symbolic execution and program testing. Commun. ACM, 19. 1976. P. 385-394.
10. Forejt V., Joshi S., Kroening D., Narayanaswamy G., Sharma S. Precise

Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. ACM Transactions on Programming Languages and Systems. 39, 4, December 2017. 27 p.

11. Зайцев Д.А. Математичні моделі дискретних систем: Навч. посібник. Одеса: ОНАЗ ім. О.С. Попова, 2004. 40 с.
12. Котов В.Е. Сети Петри. М. : Наука. Главная редакция физико-математической литературы, 1984. 160 с.
13. Manual Page – sift(n). (загол. з екрану). URL : https://projects.laas.fr/tina/manuals/sift.html. (дата звернення 10.11.2022).
14. Zaitsev D.A., Shmeleva T.R., Guliak R.N. Analyzing Multidimensional Communication Lattice with Combined Cut-Through and Store-and-Forward Switching Node. Kumar R., Mishra B.K., Pattnaik P.K. (eds) Next Generation of Internet of Things. Lecture Notes in Networks and Systems, vol 201. Singapore : Springer, 2021. P. 705-715
15. MCC'2022 – Results. (загол. з екрану). URL : https://mcc.lip6.fr/2022/results.php. (дата звернення 11.11.2022).
16. Fokkink W. Introduction to Process Algebra. Berlin, Heidelberg : Springer, 1999. 168 p. (Texts in Theoretical Computer Science. An EATCS Series)
17. Smolka S. Strategic Directions in Concurrency Research. ACM Computing Surveys. 1996. V. 28. P. 607-625.
18. Aalst W. Three Good Reasons for Using A Petri-Net-Based Workflow Management System. Engineering and Computer Science. 1998. V. 11. P. 161-182.
19. Cooper K., Torczon L. (2011). Engineering a compiler: Second edition. Morgan Kaufmann, 2011. 824 p.
20. Jones N. D., Gomard C. K., Sestoft, P. Partial Evaluation and Automatic Program Generation. Prentice Hall, 1993. 415 p. (Prentice-hall International Series in Computer Science)
21. Rosen B., Wegman M., Zadeck K. Global value numbers and redundant

computations. 15th Annual ACM Symposium on Principles of Programming Languages. 1988. P. 12-27.

22. Cytron R., Ferrante J., Rosen B., Wegman M., Zadeck K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems. October 1991. V. 13. P. 451-490.
23. Laurent M., Saillard E., Quinson M. The MPI BUGS INITIATIVE: a Framework for MPI Verification Tools Evaluation. Correctness 2021: Fifth International Workshop on Software Correctness for HPC Applications, November 2021, St. Louis, United States. P. 1-9.
24. Рольщиков В.Б. Застосування засобів інтерфейсу передачі повідомлень при програмуванні розподілених систем мовою Java: навчальний посібник. Одеса, 2018. 209 с.