Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ННЦ ЗФН_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

Дослідження алгоритмів виконання функціональних паралельних програм
(тема)

Виконав: студент 2 курсу, групи ПЗСзм-18-1
спеціальності _121- Інженерія програмного забезпечення_____
(код і повна назва спеціальності)
освітньо-професійної програми _Програмне__
_____забезпечення_систем_____
(повна назва освітньої програми)

_____Крамаренко І.Д._____
(прізвище, ініціали)

Керівник _____проф. Дудар З.В._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет _Комп'ютерних наук_____

Кафедра _Програмної інженерії_____

Рівень вищої освіти _____другий (магістерський)_____

Спеціальність _121– Інженерія програмного забезпечення_____
(код і повна назва)

Тип програми освітньо-професійна програма_____

Освітня програма_ Програмне забезпечення систем_____

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові _Крамаренко Івану Дмитровичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _Дослідження алгоритмів виконання функціональних паралельних програм_____

затверджена наказом по університету від «___» _____ 2019 р № __Стз__
заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії
10 грудня 2019 р.

3. Вихідні дані до роботи __*проаналізувати існуючі алгоритми, що використовуються для моделювання та Застосування динамічних засобів розпаралелювання функціональних програм, заснованих на використанні паралельних платформ у реальному часі з використанням оптимізованих методів*_____

4. Перелік питань, що потрібно опрацювати в роботі _*мета роботи, аналіз проблемної галузі і постановка задачі, опис запропонованих варіантів оптимізації, використовувані методи та алгоритми, опис розробленої програмної системи, опис застосованих оптимізацій, аналіз можливих застосувань*_____

5. Консультанти розділів роботи

Найменування розділу	Консультант (посаду, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	проф. Дудар З.В.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Аналіз предметної галузі	10 жовтня 2019 р.	
2.	Огляд існуючих методів	27 жовтня 2019 р.	
3.	Проектування та розробка ПЗ	15 листопада 2019 р.	
4.	Підготовка пояснювальної записки	25 листопада 2019 р.	
5.	Спецчастина	26 листопада 2019 р.	
6.	Підготовка презентації та доповіді	30 листопада 2019 р.	
7.	Попередній захист	10 грудня 2019 р.	
8.	Нормоконтроль, рецензування	11 грудня 2019 р.	
9.	Занесення диплома в електронний архів	12 грудня 2019 р.	
10.	Допуск до захисту в зав. кафедри	14 грудня 2019 р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання 2019 р.

Студент _____
(підпис)

Керівник роботи _____ __проф. Дудар З.В._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 75 с., 36 рис., 6 табл., 4 додатки, 26 джерел.

ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ, АЛГОРИТМ, ВІЗУАЛІЗАЦІЯ, МЕТОД, МОДЕЛЮВАННЯ, ПРОГРАМНА БІБЛІОТЕКА, FPTL.

Об’єктом дослідження є функціональні мови програмування, які дозволяють абстрагуватися від особливостей використовуваної обчислювальної системи

Метою роботи є розробка й дослідження ефективності системи виконання функціональних паралельних програм мовою FPTL на багатоядерних комп'ютерах.

Методи розробки базуються на мові програмування FPTL, методах математичного моделювання.

У результаті роботи розроблено інтегровану систему паралельного виконання FPTL-програм на багатоядерних комп'ютерах.

FUNCTIONAL PROGRAMMING, ALGORITHM, VISUALIZATION, METHOD, MODELING, SOFTWARE LIBRARY, FPTL

The object of the study are functional programming languages that allow you to abstract from the features of the computer system used

The purpose of this work is to develop and investigate the performance of a system of running functional parallel programs in FPTL on multicore computers.

Development methods are based on FPTL programming language, mathematical modeling methods.

As a result, an integrated system of parallel execution of FPTL programs on multi-core computers was developed.

ЗМІСТ

ВСТУП

Збільшення швидкодії комп'ютера шляхом підвищення тактової частоти його єдиного процесора наблизилося до технологічної межі [1]. Тому провідні виробники процесорів почали нарощувати продуктивність комп'ютерної системи шляхом збільшення числа процесорів або ядер – незалежних обчислювальних блоків усередині процесора. На сьогоднішній день даний підхід міцно закріпився при виробництві процесорів як для персональних комп'ютерів і серверів, так і для процесорів портативних і мобільних обладнань.

Для того щоб максимально використовувати ресурси багатоядерного комп'ютера для вирішення на ньому різних задач, потрібне використання підходів, що передбачають паралельне виконання [2]. Широко використовувані в промисловому програмуванні сучасні об'єктно-орієнтовані мови програмування, такі як Java і C++, засновані на імперативній парадигмі і не мають вбудованих у мову програмування засобів для розпаралелювання програм, «прозорих» для програміста, а лише надають набір низькорівневих засобів завдання паралелізму. Практичне використання цих засобів часто приводить до того, що програма виходить погано масштабованою і прив'язаною до конкретної операційної системи, мови програмування або архітектури обчислювальної системи.

Одним із засобів рішення цієї проблеми є застосування функціональних мов програмування [2, 3], які дозволяють абстрагуватися від особливостей використовуваної обчислювальної системи і складного завдання керування паралельними процесами, сконцентруватися тільки на рішенні поставленого завдання й розробляти ефективні паралельні програми.

Мова FPTL (Functional Parallel Typified Language) [3], реалізації якої на багатоядерних комп'ютерах, створювалася з метою ефективної розробки функціональних програм і наступного їх паралельного виконання на багатоядерних комп'ютерних системах із загальною пам'яттю й кластерах. На відміну від відомих мов функціонального програмування, таких як LISP [4], ML

[5], Haskell [6], які у великій ступені засновані на λ-вирахуванні [7], FPTL заснований на використанні традиційної математичної форми визначення функцій шляхом їх композиції за допомогою кінцевої безлічі операцій і загальної форми їх завдання у вигляді систем функціональних рівнянь. Операції композиції функцій прості й дозволяють описувати паралелізм на семантичному рівні, не використовуючи спеціальні процесні примітиви, як це робиться в інших функціональних мовах. Однак реалізація такої «чисто» функціональної мови програмування вимагає розробки методів і алгоритмів, які можуть забезпечити ефективне паралельне виконання програм на багатоядерних комп'ютерах. Це створює необхідні передумови створення відповідної системи виконання мови FPTL на сучасних багатоядерних комп'ютерних системах.

Метою роботи є розробка й дослідження ефективності системи виконання функціональних паралельних програм мовою FPTL на багатоядерних комп'ютерах. Для досягнення цієї мети в магістерській роботі вирішуються наступні завдання:

- порівняльний аналіз методів розпаралелювання обчислень у сучасних функціональних мовах програмування;
- дослідження методів і алгоритмів ефективного паралельного виконання функціональних FPTL програм на багатоядерних комп'ютерах;
- створення інтегрованої системи паралельного виконання FPTL програм на багатоядерних комп'ютерах, яка включає наступні підсистеми.

1 АНАЛІЗ СТАНУ РОЗВ'ЯЗАННЯ ПРОБЛЕМИ ТА ОБҐРУНТУВАННЯ ЦІЛЕЙ ДОСЛІДЖЕННЯ

1.1 Статичні засоби розпаралелювання

Існуюча класифікація мов програмування, що відносить їх до імперативних, функціональних, логічних, об'єктно-орієнтованих, процесних, є досить довільною. Вона відбиває або їхню проблемну орієнтацію, наприклад, функціональна, логічна або процесна, або певний спосіб побудови чи поведінки програми. Для імперативних програм центральним аспектом є таке впорядкування фрагментів програми (операторів, блоків, процедур і ін.), щоб їх послідовне виконання гарантувало одержання запланованого результату.

Об'єктно-орієнтоване програмування, залишаючись послідовним, як і імперативне, претендує на статус більшої декларативності в описі програми й виражається в можливості оперування класами, як узагальненими деклараціями певних об'єктів або форм, по яких можна породжувати об'єкти з різними властивостями.

С іншої сторони, паралельне й процесне програмування робить основний акцент на описі такого упорядкування фрагментів програми, при якому певна їхня частина може виконуватися одночасно. Подвійність програми, як декларативного опису того, що вона повинна робити, і опису того, як вона повинна виконуватися, зафіксована у обох відомих її семантиках, а точніше моделях програми: денотаційної і операційної. Паралельне програмування актуалізувало проблему створення моделей, які могли б бути досить універсальними, щоб можна було або явно відбивати в програмі, або реалізовувати при її виконанні виникаючі можливості розпаралелювання. Мережі Петри, моделі Милнера [8] і Хоара [9] – відомі процесні моделі для цього.

В більшості сучасних мов програмування, як імперативних, так і функціональних, при написанні програми програміст супроводжує її на деякій універсальній процесній мові, що описує її паралельне виконання. Для цього

можуть використовуватися спеціальні оператори мови програмування, анотації, синтаксичні конструкції й процедури. Природня й інша постановка завдання: як автоматично визначати й реалізовувати паралелізм у програмі й при цьому звести до мінімуму участь програміста в цій роботі. Для цього, по-перше, денотаційна модель і мова програмування повинні бути влаштовані так, щоб паралелізм у програмі міг виражатися засобами цієї моделі й мови, не прибігаючи до процесних примітивів для його вказівки. По-друге, повинна існувати досить універсальна процесна модель, щоб на її основі паралелізм у програмі можна було ефективно реалізувати на практиці, використовуючи існуючі засоби комп'ютерної системи.

Одним з найпоширеніших методівпрограмування багатоядерних комп'ютерів є застосування статичної багатопоточності (static threading) [12]. Вона надає програмну абстракцію «віртуальних процесорів», або потоків (threads), що спільно використовують загальну пам'ять. Кожний потік підтримує пов'язаний з нею лічильник команд і може виконувати машинний код незалежно від інших потоків. Операційна система завантажує потік в процесор для виконання і перемикає потік, коли виконання вимагає інший потік. Хоча операційна система й дозволяє програмістам створювати й знищувати потоки, ці операції є відносно повільними. Таким чином, для більшості додатків потоки зберігаються протягом усього часу обчислень, чому вони й одержали назву «статичні».

Для роботи з потоками в мові Haskell у складі стандартної бібліотеки є функція forkio [13]. Дана функція створює новий потік, у якому проводиться обчислення вираження, переданого в якості вхідного параметра даної функції. Для реалізації взаємодії між декількома працюючими потоками в мові є механізм загальних змінних і механізм каналів синхронізації. Канали синхронізації дозволяють організовувати тільки однобічну взаємодію між потоками за допомогою обміну повідомлень через черги. Загальні змінні в мові Haskell мають спеціальний тип Mvar [13] і використовуються для читання й запису даних з різних потоків. Вони відрізняються від звичайних змінних у мові Haskell, значення яких не може бути змінене після їхньої ініціалізації. Наявність змінних у

принципі суперечить основній концепції функціональних мов програмування - незмінюваності даних (data immutability) [14].

В контексті паралельного програмування це може привести до виникнення неузгодженого доступу до даних, або, по-іншому, до стану перегонів [16]. Паралельний алгоритм є детермінованим, якщо він завжди приводить до тих самих результатів при тих самих вхідних даних, незалежно від того, у якій послідовності вибудовуються друг щодо друга інструкції, виконувані різними потоками в багатоядерному комп'ютері. Алгоритм є недетермінованим, якщо його поведінка може мінятися від виконання до виконання. Паралельні програми, що приводять до стану гонки, є недетермінованими.

Одним з можливих рішень проблеми використання змінюваних даних у паралельнім середовищі є використання механізмів статичного типового контролю. Таке рішення було застосоване в експериментальній функціональній мові програмування LinearML [11]. Ця мова створювалася на основі мови ML і її ключовою особливістю є так звана лінійна система типів [15]. Така система типів не дозволяє створити кілька посилань на ті самі дані – кожна змінна посилається на свій власний екземпляр даних. Це гарантується системою типів на етапі компіляції програми.

let $_1$= [1; 2; 3]

let $_2$=List.rev

List.**irelease**

Наприклад, у наступному фрагменті програми мовою LinearML спочатку створюється список цілих чисел 1, потім він використовуються для створення списку 2, що представляє собою реверсивну версію списку 1. Після цього подальше використання в програмі змінної 1 буде вважатися порушенням типізації, і на етапі компіляції буде видаватися відповідна помилка. Крім того, помилкою типізації буде вважатися, якщо змінна взагалі не була використана, тому наприкінці прикладу застосовується оператор **`irelease`** для звільнення виділеної пам'яті.

Такий підхід до організації роботи з даними при виконанні програми дає наступні переваги. По-перше, виключаються всі проблеми організації взаємодії між потоками, пов'язані з можливою неузгодженою зміною стану (стану перегонів). По-друге, виключається необхідність використання автоматичної системи керування пам'яттю (усі інші мови сімейства ML, а також Haskell, використовують для керування пам'яттю автоматичне прибирання сміття).

Можливим рішенням проблеми організації конкурентного доступу до загальних даних є використання механізму програмної трансакційної пам'яті [17]. При даному підході взаємодія з даними в оперативній пам'яті комп'ютера організується по принципах, схожих з організацією Acida транзакцій у системах керування базами даних. Використання даного підходу сильно ускладнює сприйняття програми програмістом, незнайомим з механізмами роботи програмної трансакційної пам'яті.

Іншою проблемою статичної многопоточності є складність забезпечення рівномірного розподілу роботи між потоками. Для будь-яких (крім самих найпростіших) програм для збалансованого завантаження потоків програміст повинен розробляти складні протоколи взаємодії між потоками. Такий стан справ привів до створення паралельних платформ, що надають шар програмного забезпечення, який координує ресурси паралельних обчислень, планує їх і управляє ними. Проте, потоки в функціональних мовах програмування знайшли успішне застосування при організації асинхронної роботи із системою введення і виведення, наприклад, для роботи з мережею або для читання великих файлів.

1.2 Динамічні засоби розпаралелювання

Для вирішення проблем, що виникають при використанні статичних засобів розпаралелювання, була запропонована модель динамічного розпаралелювання. Вона дозволяє програмісту вказувати рівень паралелізму в програмі, не

турбуючись про комунікаційні протоколи, збалансованість завантаження та інші проблеми, які виникають при використанні паралельного програмування. Паралельна платформа містить планувальник, який у свою чергу автоматично забезпечує завантаження потоків, суттєво спрощуючи роботу програміста. Застосування паралельних платформ дозволяє використовувати у функціональній мові програмування спеціальні оператори або синтаксичні конструкції, що дозволяють виділяти вираз, значення яких повинні обчислюватися паралельно.

В мові Haskell у якості таких конструкцій використовуються оператори. Денотаційна семантика цих операторів може бути представлена в такий спосіб:

$$(par\ f\ g)d = g(d)$$

$$(pseq\ f\ g)d = \begin{cases} \bot, \text{если } f(d) = \bot \\ \text{иначе } g(d) \end{cases},$$

де f і g є собою функціональні змінні мови Haskell,

d – елемент даних.

Паралельна семантика цих операторів полягає в наступному: оператор ухвалює як аргументи два вирази, обчислення значень яких гарантовано проводиться паралельно. В якості результату оператор повертає значення другого із зазначених виразів. Оператор також ухвалює на вхід два вираза, але, на відміну від оператора par, їхнє обчислення гарантовано проводиться послідовно. Цей оператор використовується в тих випадках, коли необхідно гарантовано дочекатися завершення паралельних обчислень, породжених оператором. У якості результату повертається значення першого виразу, якщо воно визначене, інакше повертає невизначене значення.

Далі приводиться простий приклад застосування операторів і в мові Haskell для обчислення n-го числа Фібоначчі.

```
nfib :: Int -> Int
     nfib n | n<= 1 = 1 | otherwise = f (g (f + g))where
f = nfib ( - 1) g = nfib (- 2)
```

В наведеному прикладі оператор забезпечує паралельне обчислення чисел Фібоначчі для − 1 і − 2. Після одержання цих значень, що гарантується

оператором, проводиться їхнє підсумовування для одержання n-*го* числа Фібоначчі.

Внутрішня реалізація оператора полягає в тому, що при обчисленні значення (*par xy)d*, формуються дві спеціальні структури даних, що називаються завданнями [19]. Кожне із завдань містить опис виразу й відповідно посилання на вихідні дані. Планувальник паралельної платформи, у свою чергу, робить призначення завдань на виконання програмно-апаратними ресурсами комп'ютерної системи. При цьому в процесі виконання завдання можуть рекурсивно породжувати інші завдання.

Приведемо інший приклад використання паралельних платформ на прикладі мови F# [10]. Розглянемо функціональну програму для паралельного обчислення n-*го* числа Фібоначчі.

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ ->
        let x = Future<int>.Create(fun _ -> fib (n − 2))
        let y = Future<int>.Create(fun _ -> fib (n − 1))
        x.Value + y.Value
```

В розглянутому прикладі функція Future.<int>.Create з бібліотеки TPL [20] використовується для породження двох паралельних завдань, що представляють відповідно паралельне обчислення значень функцій **fun** _->*fib* (− 2) і **fun** _ ->*fib* (− 1). У наступній частині програми x.Value + y.Value відбувається очікування готовності цих завдань і отримані значення використовуються для формування результату. Як видно із прикладу, замість спеціальних операторів розпаралелювання використовуються безпосередні виклики методів паралельної платформи TPL, а очікування готовності обчислень проводиться неявно при звертанні до полів завдань.

Застосування динамічних засобів розпаралелювання функціональних програм, заснованих на використанні паралельних платформ, проте, має свої недоліки. Написання коректних паралельних програм мовою Haskell з використанням операторів відносно легко, оскільки відсутність побічних ефектів

означає, що програмісту не доводиться замислюватися про вирішення таких проблем, як стан перегонів або взаємоблокування, які можуть значно ускладнити написання й налагодження паралельних програм за допомогою засобів паралельного програмування. Однак, написання паралельної програми, яка має гарну масштабованість на цілому ряді паралельних архітектур, набагато складніше. Використання операторів розпаралелювання вимагає від програміста більшої уваги до вибору фрагментів програми, що вимагають розпаралелювання. Наприклад, в мові Haskell, що використовує стратегію відкладених обчислень, часто важко зрозуміти, чому «ідеальна» програма не виконується так, як очікує програміст. При некоректному використанні операторів розпаралелювання може виникнути ситуація, при якій те саме завдання буде виконано кілька раз. Шляхи вирішення цієї проблеми в системі виконання мови Haskell докладно описані в [21].

1.3 Аналіз особливостей мови FPTL

Мова FPTL додержується класичного підходу до завдання функцій, що бере свій початок з моделі обчислюваних функцій Черча. У цій моделі функції створені за допомогою певної безлічі операцій шляхом їхньої композиції й задання простих базисних функцій.

Нагадаємо, що мови LISP, ML, Haskell засновані на λ нотації завдання функцій, яка створювалася з метою явного розрізнення в математичних контекстах зв'язаних вільних змінних. Її функціональна семантика (однозначність редукції λ виразів) була результатом доказу одержання однозначного результату редукування λ виразів, якщо цей результат існує.

В мові FPTL існує чотири прості операції композиції функцій, три з яких мають паралельну семантику. Загальна форма завдання функцій – це система

функціональних рівнянь. Цього інструмента, як доведено, досить щоб можна було визначити будь-яку функцію обчислювану над певною безліччю даних.

В загальному випадку, програма мовою FPTL задається у вигляді трійки <S,I,M> деS– схема програми (система певних у ній функцій), I – інтерпретація, що дозволяє зіставляти схемі різні функції шляхом перевизначення в схемі конкретних базисних функцій, M – модель обчислення значень функцій. Перші два компоненти дозволяють суттєво розширити в одному визначенні безліч функцій, що цікавлять програміста (змінюючи інтерпретацію схеми можна одержати різні функції). Модель обчислення значень функцій, зокрема паралельного обчислення, формально витягає виходячи із властивостей операцій композиції функцій [3].

Таким чином, програміст у загальному випадку не повинен, як це має місце в розглянутих мовах Haskell і F#, явно визначати й указувати ті фрагменти програми, які будуть виконуватися паралельно. В FPTL це реалізує інтерпретатор відповідної моделі паралельного обчислення значень функцій. Як наслідок, програміст, розробляючи програму, має справу тільки із завданням даних і необхідних функцій над ними. Більше того, шляхом еквівалентних перетворень схеми програми, програміст може регулювати ступінь паралелізму в ній [22].

1.4 Обгрунтування задач дослідження

Статичні засоби розпаралелювання засновані на використанні потоків – низькорівневих процесних засобів операційної системи. Вони беруть свій початок з імперативних мов програмування (C, C++, Java і т.д.) і вимагають при написанні паралельних програм використання засобів взаємодії між паралельними процесами, які складні в написанні й приводять до численних помилок, що важко виявляються.

Динамічні засоби розпаралелювання, т.зв. паралельні платформи, беруть на себе всю роботу з організації й планування паралельних обчислень. Це дозволяє позбавити програміста від реалізації протоколів взаємодії між потоками. З іншого боку, застосовувані в них засоби розпаралелювання за допомогою спеціальних операторів, доступні в більшості паралельних платформ для функціональних мов програмування, вимагають від програміста явного виділення в програмі паралельних частин.

Мова FPTL у свою чергу не вимагає від програміста використання спеціальних операторів розпаралелювання. Паралельна семантика задається в ньому неявно за допомогою використання операцій композиції функцій. У той же час, модель паралельного обчислення значень функцій у мові FPTL може бути реалізована як з використанням статичних засобів розпаралелювання, так і з використанням концепцій паралельних платформ.

2 ОПИС ПРОВЕДЕНИХ ТЕОРЕТИЧНИХ ДОСЛІДЖЕНЬ

2.1 Теоретичні основи мови функціонального програмування

Теоретичну базу мови FPTL становлять дослідження з функціональної схематології й функціональних систем [3, 5], які узагальнені в теорії спрямованих відносин, що поєднує функціональний і логічний стилі програмування [6; 7]. Мова FPTL має дві семантики: денотаційну й паралельну операційну семантику, які будуть розглянуті в наступних розділах.

Функції в FPTL розглядаються, як типізовані $t_1' \times t_2' \times \times t_m' \rightarrow t_1'' \times t_2'' \times \ldots \times t''$ (m, n) -арні відповідності ($t_1' \geq 0, n \geq 0$) міжкортежами даних, де t′, = 1, і t″, = 1, - типи елементів вхідного і вихідного кортежів, - довжина кортежу на вході функції, n – на виході. Функції арності (0,1) розглядаються в FPTL як константи.

Для рівного 0 маємо кортеж нульової довжини, позначуваний, із властивостями $\alpha\lambda=\lambda\alpha=\lambda\alpha$, де α – довільний кортеж. Кортеж даних у мові представляється у вигляді послідовного запису його елементів.

В FPTL на відміну від загальноприйнятої форми завдання функцій з явною вказівкою її аргументів (так звана форма завдання загального значення функції) строго різниться властива функція як відображення одної безлічі в іншу і її аплікація до конкретних даних. Роль змінних у завданні функцій в FPTL виконують функції вибору необхідного елемента з кортежу даних. Формально функція вибору аргументу, позначувана l (i,m) (у мові позначається скорочено як [i]), при застосуванні до кортежу довільного типу даних довжинim(m> 0) вибирає його i-*й* елемент,i = 0, … n, > 0. Функції в FPTL є в загальному випадку частковими, причому невизначене значення функції може бути виражене або як необмежений процес обчислення її значення, або як спеціальне обчислене невизначене значення, позначуване ω, із властивостями $\omega\alpha = \alpha\omega = \omega$для будь-якого кортежу α.

Формально функції визначаються як системи функціональних рівнянь $F= \tau_i$, $i= 1, \ldots, n$ де τ_i – функціональні терми, побудовані із заданих (базисних) функцій і

функціональних змінних шляхом застосування чотирьох бінарних операцій композиції функцій: →, +, *, •. Для функцій і функціональних змінних задана арність, а для базисних функцій – також їх тип. Тип функціональних змінних збігається з типом визначальних їх термів і однозначно визначається із завдання типів базисних функцій і правил висновку типів для функцій, побудованих шляхом застосування операцій композиції.

Синтаксис і семантика операцій композиції визначаються в такий спосіб. Послідовна композиція (•): даних і-*го* аргументу функції. Тут і далі спочатку задається синтаксис операції композиції, а потім її семантика, обумовлена через застосування функцій до кортежу даних. Передбачається, що типи кортежу значень функції 1 і кортежу аргументів функції 2 однакові.

В FPTL використовується префіксна форма запису операції послідовної композиції, що задає послідовний характер обчислення значень функцій 1 і 2 і еквівалентна послідовному характеру завдання проходження операторів при виконанні послідовних програм.

Операція конкатенації кортежів значень функцій (*): передбачається, що типи аргументів функцій 1і 2 однакові.

Операція умовної композиції: визначене й відмінно від значення «неправда», інакше. Передбачається, що типи кортежів аргументів функцій однакові.

Операція об'єднання (графіків) ортогональних функцій: передбачається, що типи аргументів і значень функцій 1 і 2 однакові. Функції 1 і 2 вважаються ортогональними, якщо для всякого кортежу даних α визначена не більш ніж одна з них. Операція об'єднання ортогональних функцій була введена з метою вистави паралельних функцій (відомий приклад – функція голосування в телефонії) [22].

Усі операції композиції є асоціативними, а операція + виступає як комутативна. Пріоритет операцій композиції визначається наступному образом (у порядку зростання): +, →, *, •.

Терми в завданні функцій у вигляді систем функціональних рівнянь являють собою композиції, побудовані з базисних функцій і функціональних

змінних шляхом застосування операцій композиції. Передбачається, що арності й типи терма й обумовленої їм функціональної змінної в системі функціональних рівнянь однакові. Функціональні змінні виконують подвійну роль при завданні функції (побудові функціональної програми): одні з них з'являються як необхідні елементи при завданні рекурсивних функцій, інші визначаються далі (у наступнім рівнянні функції, що уточнюється), дозволяючи просто реалізувати покрокову розробку функціональної програми за технологією проектування «зверху донизу».

В FPTL можна представляти будь-який структурний тип даних, що називається абстрактним типом даних (АТД), визначаючи його за аналогією з визначенням функцій у загальному випадку через систему рекурсивних рівнянь. При визначенні абстрактних типів даних використовуються функції конструктора й зворотні до них функції деструктора, які разом з функціями вибору аргументу утворюють повний набір базисних функцій у тому розумінні, що будь-яка обчислювана функція над даними розглянутого типу може бути виражена засобами мови FPTL [3].

При визначенні абстрактних типів даних у якості вихідних можна також використовувати вбудовані типи: *bool*, *real*, *int*, *string* і інші разом з певними в конкретній комп'ютерній системі операціями над ними. Для визначення типів даних застосовуються ті ж операції композиції, які застосовуються для завдання функцій, за винятком того факту, що операція + трактується як операція об'єднання двох безлічей даних (у мові позначається як ++). Помітимо, що ω можна використовувати, як спеціальне позначення «невизначеного» значення функції, яке визначається конструктивно. Із цього випливає, що базисні функції, що витягають із визначення абстрактного типу даних, можуть розглядатися як усюди певні структури.

Приведено приклад визначення в FPTL абстрактного типу даних (списку натуральних чисел):

```
Nat = c_null ++ Nat • c_succ;
Listofnat = c_nil ++ (Nat * Listofnat) • c_cons;
```

Тут функції конструктора *c_null, c_succ, c_nil* і *з_cons* мають арності (0,1), (1,1), (0,1) і (2,1) відповідно й наступні типи: {λ} → nat, Nat → Nat, {λ} → *Listofnat*, *Nat * Listofnat → Listofnat*.

Зворотні до них функції (деструктори), позначувані як *~c_null,~c_succ, ~c_nil, ~c_cons*,автоматично витягають із визначення типу ймають наступну інтерпретацію:

Приведено приклад функціїпредиката, яка перевіряє приналежність кортежу типу даних *Listofnat*:

```
Islistofnat = ~c_nil + ~c_cons •([1] •isnat * [2] •islistofnat);
Isnat = ~c_null + ~c_succ.Isnat.
```

Функція *Islistofnat* визначена на будь-якому кортежі даних, що належать *Listofnat*, і її значенням є те, що може трактуватися як «істина». Для інших відмінних від *Listofnat* даних як результат застосування *Islistofnat* буде невизначене значення, яке може також трактуватися як «неправда». Помітимо, що будь-яке значення функції предиката, відмінне від «неправда» може трактуватися як «істина».

Наприклад, модель пошуку по дереву є недетермінованою, оскільки в загальному випадку можливе застосування декількох правил до стану обчислень дерева, і тому залежно від використовуваного правила будуть виходити різні послідовності обчислень. Модель також паралельна, тому що можливо одночасне застосування декількох правил до незв'язних гілок стану дерева. Джерело паралелізму – властивості операцій ∗, →, +. Практично це означає, що процесобчислень розвивається незалежно від різних галузей стану дерева, що надало право назвати модель обчислень асинхронною [27].

Важливо відзначити, що не всякий порядок застосування правил перетворення станів приводить до коректного обчислення значення функції. Наприклад, при обчисленні $(_1+_2)(\ )$, якщо спочатку робиться спроба обчислення $(\ _1)(\ )$, яке не визначене й процес триває нескінченно, а значення $(\ _2)(\ )$ визначене, потрібний результат не буде отриманий. Таким чином, умовою коректності

реалізації моделі є паралельне обчислення значень функцій, що з'єднуються операцією +.

Не менш істотним є вміння переривати непотрібні обчислення. При одержанні на одній з галузей значення, відмінного від ω, або значення ω обчислення, пов'язані з іншими галузями, необхідно перервати.

Крім цього, модель надає можливість підвищення ефективності обчислень за рахунок умовних конструкцій з попередженням. Якщо є достатня кількість ресурсів, то при визначенні значення можна обчислювати одночасно прагнучі максимально розпаралелювати процес.

Ці особливості моделі є принциповими при її реалізації на обчислювальних системах і при розробці ефективних алгоритмів планування паралельного виконання функціональних програм.

Інша модель організації процесу паралельного обчислення значень функцій створювалася спеціально для ефективної реалізації на багатоядерних процесорах.

2.2 Аналіз основних властивостей мови FPTL

Спочатку треба навести загальну структура функціональної програми мовою FPTL, після чого буде розглянута структура окремих її елементів (блоків).

Структурно FPTL програма складається з наступних блоків:

- блок опису імпорту зовнішніх функцій;
- блок опису даних;
- блок опису функцій, заданих у вигляді функціональних рівнянь;
- блок інтерпретації функціональних рівнянь і завдання статичних параметрів функціональної програми.

Обов'язковою частиною програми є тільки блок опису функціональних рівнянь, інші блоки можуть бути відсутніми.

Далі буде описаний синтаксис кожного з перерахованих блоків. При описі синтаксису будуть використовуватися наступні умовні позначки:

- курсивом будуть позначатися ідентифікатори, що задаються програмістом;
- напівжирним шрифтом будуть позначатися ключові слова мови FPTL;
- запис <*E*> позначає, що конструкція *E* може бути опущена, або може повторюватися 1, 2 і більш раз;
- запис <<*E*>> позначає, що конструкція *E* може повторюватися не менш 1, 2 і більш раз.

Дана конструкція призначена для опису функцій, реалізація яких перебуває в зовнішніх бібліотеках. Такі функції можуть бути реалізовані на різних мовах програмування (наприклад, C або C++). Опис зовнішніх функцій являє собою список наступного виду:

```
<import ім'я_зовнішньої_функції from "ім'я_файлу_бібліотеки"; >
import some_function from "some_lib.dll";
```

В даному прикладі імпортується функція *some_function* з файлу бібліотеки з іменем *some_lib.dll*.

аний блок призначений для опису абстрактних типів даних і їх конструкторів і складається зі списку конструкцій виду:

```
data ім'я_АТД
{
     << ім'я_АТД = конструктор < ++ конструктор > ; >>
}
```

Синтаксис опису конструкторів наступний:

```
ім'я_типу <* ім'я_типу> . ім'я_конструктора
```

Тут ім'я типу може бути як іменем вбудованого типу даних (`int, real, boolean, string` і т.д), так і іменем абстрактного типу даних (включаючи й обумовлений АТД). Можливо також завдання конструктора, що ухвалює на вхід порожній кортеж даних. У цьому випадку опис конструктора полягає тільки з його імені.

В FPTL програмах можна також визначати типи даних, що параметризуються.

Базисні функції в мові FPTL мають зарезервовані імена. Їхній повний список наведено в додатку 1. Операції композиції ∗, · і → визначаються в мові через символи «*», «.» і «->» відповідно. Для зручності читання, у подальших прикладах операція умовної композиції як і раніше буде позначатися через символ →.

Блок опису функцій є основною частиною FPTL програми і називається схемою. Синтаксис опису системи функціональних рівнянь наступний:

```
scheme ім'я_схеми
{
        <<ім'я_функціональної_перем. = функціональний терм; >>
}
```

Одне з функціональних рівнянь, описаних у схемі, називається головним функціональним рівнянням. Ім'я обумовленої їм функціональної змінної повинне збігатися з іменем схеми. Функціональні терми будуються на основі базисних функцій, функцій,імпортованих із зовнішніх бібліотек, функціональних змінних, конструкторів і деструкторів. Слід зазначити, що спеціальні значення λ іω не можуть бути явно використані в програмі, а можуть бути отримані тільки в результаті обчислень.

```
scheme Factorial
{
     Factorial = ([1] * 0) . equal→1, (([1] * 1) . sub .
Factorial * [1]) .
     mul;
}
```

Для розмежування лексичного контексту в мові FPTL існує ще один варіант конструкцій – конструкції **fun**. Синтаксис їх опису аналогічний синтаксису опису схеми.

```
fun ім'я_функції
{
        <<ім'я_функціональної_перем = функціональний терм; >>
```

```
}
```

Кожна конструкція **fun** має свій лексичний контекст. Іншими словами усередині неї можуть бути перевизначені імена функціональних змінних описаних у схемі. Конструкції **fun** можуть бути вкладеними.

```
scheme Integral
{
     Sqr = ([1]*[1]).mul;
     Integral = (0.0 * 2.0).Calc;
     fun Calc
     {
          Fs = ([1].Sqr * [2].Sqr).add;
          Dx = ([2] * [1]).sub;
          Trp = ((Fs * Dx).mul * 0.5).mul;
     }
}
```

В даному прикладі приводиться проста функціональна програма обчислення значення інтеграла функції f (x) на відрізку [0, 2] методом трапецій.

Іншим варіантом використання конструкції **fun** є завдання функціоналів. Воно має в FPTL наступний синтаксис:

```
fun ім'я_функціонала[ім'я_параметра<, ім'я_параметра> ]
{
          <<ім'я_функціональної_перем =функціональний терм; >>
}
```

Синтаксис використання функціонала в правій частині функціонального рівняння наступний:

```
ім'я_функціонала ( параметр <, параметр> )
```

У якості фактичнихпараметрів можуть виступати функціональні змінні, базисні функції, константи, конструктори й деструктори. Використання виразів із операціями композиції не допускається.

```
scheme Functional
{
     Sqr = ([1]*[1]).mul;
     Functional = Trp(Sqr);
```

```
        fun Trp[F]
        {
            Fs = ([1].F * [2].F).add;
            Dx = ([2] * [1]).sub;
            Trp = ((Fs * Dx).mul * 0.5).mul;
        }
}
```

В прикладі наведена програма, що написана з використанням функціонала.

2.3 Блоки інтерпретації й застосування схеми

В загальному випадку схема, подібно конструкції **fun**, задає функціонал. Інтерпретація дозволяє одержати із цього функціонала конкретну функцію шляхом підстановки конкретної функціональної змінної замість функціонального параметра. Зберігаючи загальну структуру програми, описувану схемою, але застосовуючи до неї різні інтерпретації, можна одержувати різні кінцеві програми.

Синтаксис опису блоку інтерпретації наступний:

```
Interpretation
<<ім'я_інтерпретації>>
{
        <<ім'я_функціонального_параметра_схеми = значення ; >>
}
```

В якості значень функціональних параметрів можуть виступати базисні функції, константи, конструктори й деструктори.

Використання виразів із операціями композиції не допускається. Ім'я конкретної інтерпретації, яка буде використана при виконанні функціональної програми, задається програмістом при запуску виконання програми через параметри командного рядка.

Для передачі схемі початкового кортежу даних в FPTL програмі може бути використаний блок завдання констант. Він має наступний синтаксис:

```
application
<ім'я_константи = значення_константи ; >
%ім'я_схеми(ім'я_констранти <, ім'я_константи>)
```

В даному блоці описуються константи, які будуть використані в якості кортежу даних, що надходить на вхід схеми.

Приклад FPTL програми з використанням у ньому блоків `application` і `interpretation`.

```
data Listofcomplex
{
    Complex = (real * real).c_complex;
    Listofcomplex = c_empty ++ (Listofcomplex * Complex).c_list;
}
scheme Accumulate
{
    Accumulate = ~c_empty → 0.0, ~c_list.([1].Accumulate * [2].Part).add;
}
interpretation
Real { Part = ~c_complex.[1]; }
Img { Part = ~c_complex.[2]; }
application
stack = (((c_empty * ((0.1*0.1).c_complex)).c_head) * ((0.2*0.2).c_complex)).c_head;
% Accumulate(stack)
```

В даному прикладі приводиться функціональна програма, що обчислює суму уявних або речовинних частин списку комплексних чисел. У секції **scheme** описується рекурсивне функціональне рівняння *Accumulate*, секція **interpretation** задає дві інтерпретації з іменами *Real* і *Img* (які визначають функцію **`Part`** в схемі), що витягають значення речовинної або уявної частини відповідно.

2.4 Організація вводу-виводу і робота з масивами в мові FPTL

Для здійснення операцій введення і виведення в мові FPTL є наступні функції, описані в табл. 2.1. Ці функції ставляться до базисних.

Таблиця 2.1 – Базисні функції введу/виводу

Ім'я функції	Вхідні дані		Вихідні дані		Опис
fread	string	Ім'я файлу	string	Лічений рядок	Робить читання даних з текстового файлу
fwrite	string	Ім'я файлу			Робить читання структури даних із двійкового файлу
	string	Дані для запису			
print	string	Дані для запису			Робить вивод кортежу даних на дисплей

Більш складні операції введення і виведення можуть бути при необхідності реалізовані на інших мовах програмування за допомогою механізму виклику зовнішніх процедур. Також слід зазначити, що засобу введення і виведення не захищені від побічних ефектів і відповідальність за коректну роботу з ним несе розроблювач функціональної програми.

Для завдання масивів у мову FPTL був уведений спеціальний тип даних – Array['*t*] і наступні процедури для роботи з ним.

В якості типового параметра '*t* масиву можуть виступати вбудовані типи даних, АТД і масиви інших типів (допускається завдання вкладених масивів). Помітимо, що, подібно функціям введення і виведення, робота з масивами в мові FPTL не захищена від можливих побічних ефектів. Коректність роботи з масивами лежить на програмісті функціональної програми. Приведемо приклад роботи з масивами в мові FPTL.

Таблиця 2.2 – Базисні функції для роботи з масивами

Ім'я функції	Вхідні дані		Вихідні дані		Опис
arraycreate	int	Довжина масиву.	Array['t]	Створений масив.	Створює масив заданої довжини й заповнює всі його елементи початковим значенням.
	't	Початкове значення.			
arrayassign	Array['t]	Масив.	Array['t]	Змінений масив.	Привласнює елементу масиву с заданим індексом задане значення.
	int	Індекс елемента.			
	't	Значення.			
arrayget	Array['t]	Масив.	't	Елемент масиву.	Повертає елемент масиву з заданим індексом.
	int	Интдекс елемента.			

Приклад дії.

```
scheme Fillarray
{
     Fillarray = ([1] * ([1] * 0).arraycreate).Fill;
      F = ([2] * [1] * rand).arrayassign;
     fun Fill
     {
          N = [1];
          Arr = [2];
          Fill = (N * Arr * 0).Recurse.[1];
          Recurse = ([3] * N) . equal -> Arr * Arr, (N * ([3] *
Arr) . F * ([3] * 1).add).Recurse;
     }
}
```

В даному прикладі приводиться функція заповнення масиву випадковими числами. Тут функціональне рівняння *F* відповідає безпосередньо за

присвоювання одному елементу масиву значення, згенеровано базисною функцією rand. Рекурсивне функціональне рівняння **`Fill`** використовується для того, щоб послідовно зробити заповнення всього масиву.

FPTL є типізованою мовою програмування й умови визначення типів, отриманих шляхом застосування операцій композиції функцій. При побудові функціональних програм на FPTL, зокрема при визначенні функцій у загальному виді як систем функціональних рівнянь $\boldsymbol{F}_i = \tau_i$, =1, …, ny термах можуть використовуватися базисні функції, щовитягають із визначень, що вводяться, абстрактних типів даних конструктора й деструктора, а також так звані вбудовані функції, реалізовані в комп'ютерах: арифметичні, логічні й інші. Ці функції можуть мати як конкретні типи (конструктор натуральних чисел), так і бути поліморфними, як, наприклад, арифметичні функції й функція вибору аргументів. Тому крім перевірки синтаксису програми, виникає проблема перевірки її типової коректності.

На практиці в реалізації мови типовий контроль може бути зроблений до виконання програми, тобто статично, що можливо не для всіх мов програмування, або динамічно, тобто в процесі виконання програми [21]. Природно, що для паралельних програм, метою яких є зменшення часу їх виконання, доцільно застосовувати статичний типовий контроль (якщо він можливий). У початковій реалізації мови FPTL на багатоядерних комп'ютерах застосовувався динамічний алгоритм типового контролю, який, вносячи істотні накладні витрати, значно збільшував час виконання програми. У той же час в FPTL типовий контроль можна здійснювати до виконання програми. Із цією метою в рамках роботи був розроблений і реалізований алгоритм перевірки типової правильності FPTL програми.

3 АНАЛИЗ РЕЗУЛЬТАТІВДОСЛІДЖЕННЯ

3.1 Алгоритм типового контролю

Загальне завдання функцій в FPTL є у формі системи функціональних рівнянь $\boldsymbol{F}_i = \tau_i$, =1, …, n. Вважаємо, що типи всіх базисних функцій задані.

Перед проведенням типового контролю проводиться ряд перетворень вихідної системи функціональних рівнянь: спочатку для кожної функціональної змінної F_iвиконується для всіх τ_i у функціональних змінних підстановки τ_j замість кожного входження F_i. Будуються $F_i^{(1)}= [\tau_j / F_j \mid j = 1,\ldots,n, j \neq i]\ \tau_i$, =1, .Процес повторюється,будуючи$F_i^{(2)}$шляхом підстановок замість усіхнових функціональних змінних, що входять в $F_i^{(1)}$(змінних, які з'явилися в $F_i^{(1)}$ у результаті здійснених на першому кроці підстановок), що відповідають їх термів. Описаний процес триває до того моменту, поки для кожного не будуть отримані$F_i^{(ki)}$ такі, що в $F_i^{(kj)}$ не існує входжень функціональних змінних, для яких раніше не виконувалася підстановка. Отриману систему функціональних рівнянь $F_i = F_i^{(kj)}$, I= 1,…, nназвемоприведеною.Отримана система функціональних рівнянь еквівалентна вихідній системі.

Далі кожний терм $F_i^{(ki)}$ представляється в еквівалентній формі:

$$F_i = \tau_1^{(i)} + \tau_2^{(i)} \ldots + \tau_{ni}^{(i)},$$

де терми $\tau_j^{(i)}$не містять входжень операції +.Для цього використовуються наступні правила еквівалентності.

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$
$$(A + B) \cdot C = (A \cdot C) + (B \cdot C)$$
$$A * (B + C) = (A * B) + (A * C)$$
$$(A + B) * C = (A * C) + (B * C)$$
$$A \to (B + C) = (A \to B) + (A \to C)$$
$$(A + B) \to C = (A \to C) + (B \to C)$$

Систему функціональних рівнянь nht,f вважати семантично коректною, якщо у представленні, отриманому вище, у правій частині для кожного терма $F_i^{(ki)}$

існує принаймні один +- терм $\tau_j^{(i)}$, що не містить входжень функціональних змінних.

Вимога семантичної коректності на практиці не сильно обмежується, оскільки програми, які не мають цю властивість, можуть виконуватися нескінченно, не приводячи до одержання результату. Очевидно, що у випадку, якщо програма семантично коректна, процес підстановки, описаний вище, не може тривати нескінченно. Надалі будемо розглядати тільки семантично коректні системи функціональних рівнянь.

Після побудови наведеної системи функціональних рівнянь для семантично коректної програми в кожнім рівнянні$\boldsymbol{F}_i=\tau_i$, є хоча б один терм,такий що не міститьфункціональних змінних. У випадку, якщо в процесі побудови наведеної системи рівнянь вийшло більш одного, що не містить функціональних змінних, перевіряється виконання умов правильної типізації для операції +, тобто рівність типів доданків.

Якщо ця умова не виконується, dважається, що типізація не є правильною. Якщо ж ця умова виконується в кожнім рівнянні наведеної системи рівнянь, то покладається, що тип, F_i , i =1,…, nдорівнює типу одного з його доданків і для кожного входженняфункціональної змінної, F_j = 1, у відповідний доданок $\boldsymbol{F}_i^{(ki)}$ визначається його тип. Якщо після цього всі доданки в $\boldsymbol{F}_i^{(ki)}$, мають тип, τ_j дорівнює типу F_i, уважається, що типізація розглянутої системи функціональних рівнянь правильна. А якщо ні, то вважається, що правильність типізації порушена.

В реалізації мови FPTL на багатоядерних комп'ютерах замість двох операцій композиції функцій → та + використовується більш ефективна, з погляду паралельного виконання програм, тернарна операція, еквівалентна умовному оператору в послідовних мовах програмування. Ця операція записується в формі $\tau_1 \rightarrow \tau_2,\tau_3$, де τ_1 – терм-умова.

Вона еквівалентна представленню $(\tau_1 \rightarrow \tau_2) + (\tau_+ \rightarrow \tau_3)$. Тому перед застосуванням алгоритму контролю типової правильності функціональної програми всі входження в неї термів, отриманих шляхом застосування тернарної операції, приводяться до даної еквівалентної форми.

Як приклад розглянуто роботу алгоритму типового контролю на прикладі наступної програми:

$$\begin{cases} F_1 = f_1 \cdot F_1 + f_2 \cdot F_2 + f_3, \\ F_2 = f_4 \cdot F_2 \cdot f_5 + F_1 \cdot f_6 \end{cases}$$

Нехай

$$\{int \to int\}, type(f_4) = \{int \to real\},\ type(f_5) = \{real \to int\}, type(f_6) = \{int \to real\}.$$

Робиться підстановка замість F_2 терм τ_2 у правій частині рівняння

Перевіряємо виконання умов типової коректності для всіх доданків в $F_1^{(1)}$ і $F_2^{(1)}$, що містять входження функціональних змінних, вважаючи їх тип таким, що определен раніше type(F_1) = type(f_3) ={int → int}.При перевірці типової коректності зустрічається помилка типізації – тип функції f_2 рівний { real → real}, як і тип функції F_2 рівний type(f_6) тобто{ int → real }. Робиться висновок про невірну типізацію. Приклад можна змінити так, щоб одержати вірну типізацію.

Описаний вище алгоритм типового контролю реалізований як додатковий модуль, що підключається, у системі виконання FPTL програм. Застосування алгоритму статичного типового контролю дозволило в середньому на 30% скоротити час виконання функціональних програм мовою FPTL, у порівнянні з попередньою реалізацією, заснованої на перевірці типів безпосередньо під час виконання програми.

3.2 Архітектура системи виконання FPTL програм на багатоядерних комп'ютерах

Мова FPTL перетерпіла кілька реалізацій. Одна з ранніх реалізацій [3] виявилася недостатньо ефективною, тому що вона була виконана на комп'ютерах з розподіленою пам'яттю (багатоядерні комп'ютери в той час ще не були широко поширені). Крім того, модель паралельного обчислення значень функцій, як

показала практика, вимагає більших (за часом) накладних витрат. Це пов'язане з тим, що при здійсненні підстановок дерев замість функціональних змінних щоразу доводилося виявляти частини дерева, які готові до обчислення (редекси), що вимагає значного часу й пам'яті. Спроба позбутися цього недоліку спочатку привела до ідеї статичного складання пріоритетних списків, у яких були б записані ці редекси. Одним з варіантів рішення даної проблеми була розробка такої моделі, у якій був би відсутній (або був суттєво спрощений) пошук частин дерев. Дослідження цього питання привело до описуваного в цій главі рішення.

Для підвищення ефективності процесів керування паралельним виконанням FPTL програм були внесені зміни в синтаксис і семантику самої мови, основними з яких є введення тернарної операції умовної композиції замість двох операцій → і +, які, головним чином, призначені для опису умовних конструкцій. Нова тернарна операція є аналогом умовного оператора *if-then-else* у традиційних мовах програмування й представляється тепер в FPTL як (p –>f_1,f_2)(x), де *p* – предикат, а f_1 і f_2– функції, значення однієї з яких буде використано залежно від того, істинно або неправдиво значення p(x). Хоча це звужує виразні можливості мови, проте програміст повертається до прийнятого в мовах програмування завданню умовних конструкції й, що більш важливо, це дозволяє більш ефективно їх виконувати.

Описано алгоритм, створенийдля ефективного виконання функціональних програм на багатоядерних комп'ютерах.

В складі системи виконання функціональних програм можна виділити наступні підсистеми:

- лексичний аналізатор – його призначення – перетворення вихідного тексту програми в список лексем [29];
- синтаксичний аналізатор. Завдання цієї підсистеми – синтаксичний розбір списку лексем. У випадку відсутності в тексті програми синтаксичних помилок будується абстрактне синтаксичне дерево [29].

Підсистема семантичної перевірки. Дана підсистема робить перевірку семантичної коректності функціональної програми: проводиться пошук функціональних змінних і типів даних, не певних у тексті програмі.

Підсистема пошуку рекурсивних визначень. Робить пошук вузлів синтаксичного дерева, відповідних до рекурсивних функцій.

Генератор мережного представлення будує мережне представлення функціональної програми, визначає арність всіх вхідних і вихідних кортежів даних. Також створює базисні функції, які відповідні до конструкторів і деструкторів. Крім того, на цьому етапі проводиться перевірка типової коректності FPTL програми, алгоритм якої описано в главі 3, і визначаються типи кортежів на всіх вузлах мережного представлення функціональної програми.

Інтерпретатор. Інтерпретатор обчислює значення функцій, представлених у вигляді мереж.

Підсистема керування паралельним виконанням здійснює динамічне планування породжуваних при роботі інтерпретатора паралельних процесів (завдань) і призначає їх на виконання відповідних до них ресурсів (потокам).

Бібліотека базисних функцій vістить програмну реалізацію базисних функцій мови FPTL.

4 ОПИС РОЗРОБЛЕНОЇ ПРОГРАМНОЇ СИСТЕМИ

4.1 Опис підсистеми для багатоядерних обчислювань

Підсистема виклику зовнішніх процедур робить динамічне підключення модулів, що виконуються, містять реалізацію зовнішніх процедур.

Підсистема керування пам'яттю відповідає за виділення й звільнення пам'яті при виконанні функціональної програми, робить автоматичне керування пам'яттю (прибирання сміття). На рівні програмної реалізації всі дані підсистеми об'єднані в один файл, що виконується. Архітектура системи виконання FPTL програм представлена на рис. 4.1.

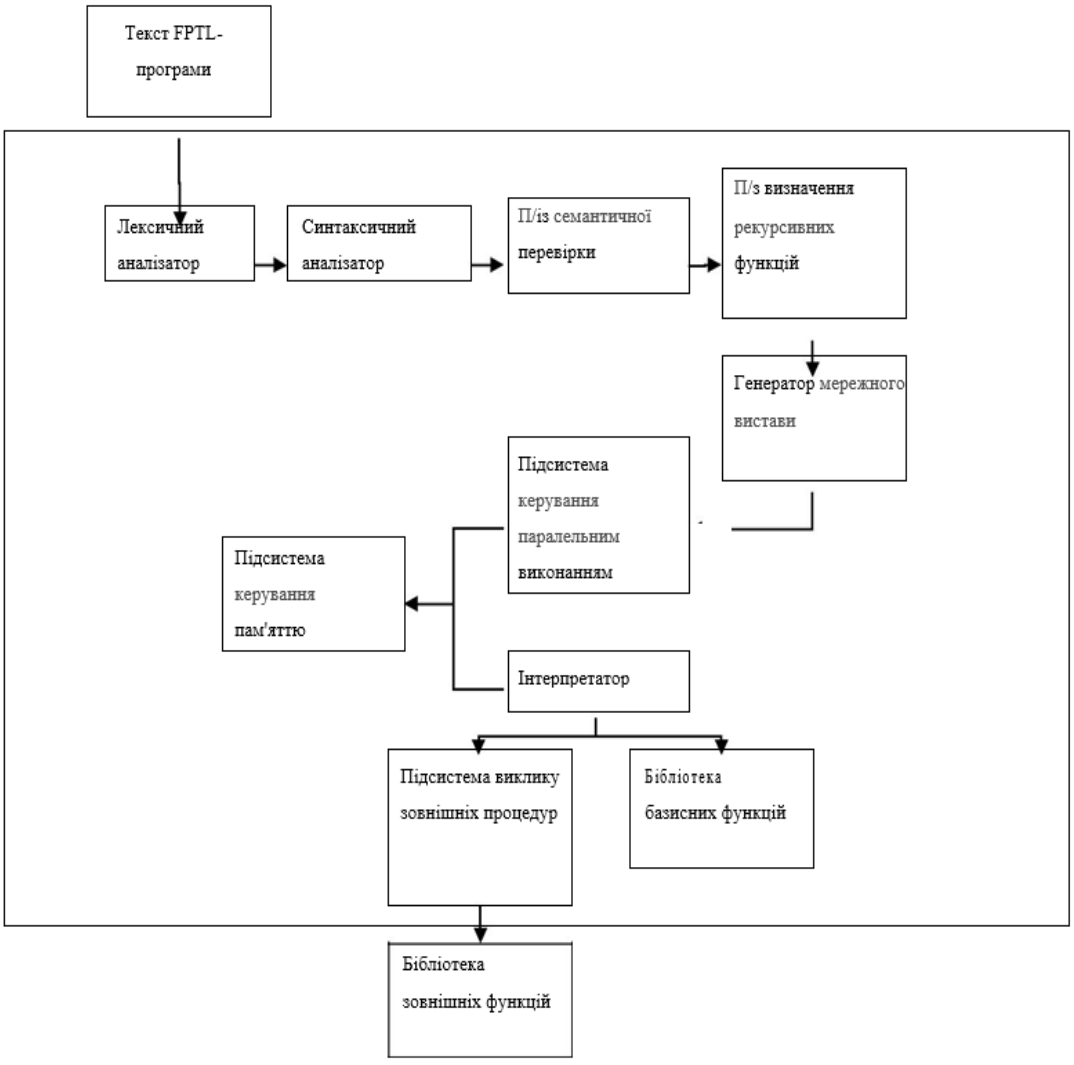

Рисунок 4.1 – Структура системи виконання FPTL програм

4.2 Мережне представлення функціональних програм

Інтерпретатор безпосередньо працює з мережнним представленням функціональної програми.

Формально, мережа – це графічне представлення функцій у програмі, які задаються у вигляді системи рівнянь. Мережне представлення заданих у такий спосіб функцій являє собою безліч мереж, однозначно пов'язаних з термами й що стоять індуктивно в такий спосіб. Якщо терм τ є базисна функція або функціональна змінна, то її представлення має вигляд (див.рис. 4.2, 4.3):

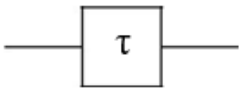

Рисунок 4.2 – Представлення базисної функції або функціональної змінної

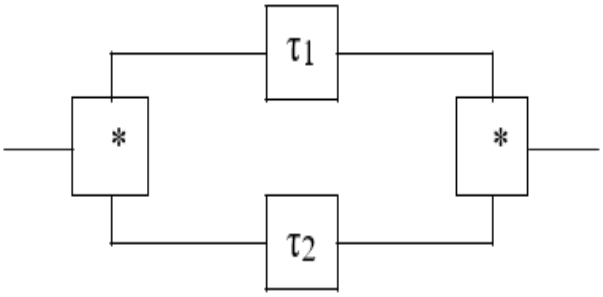

Рисунок 4.3 – Представлення операції паралельної композиції

Якщо $\tau=\tau_1 \rightarrow \tau_2,\tau_3$, то графічне подання має вигляд(рис. 4.4):

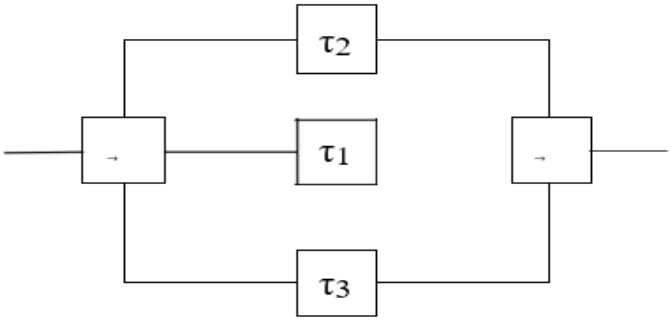

Рисунок 4.4 – Подання операції умовної композиції

В пам'яті комп'ютера мережа представляється багатозв'язковою списковою структурою. Розглянемо докладно внутрішнє представлення кожного з вузлів мережі. Для кожного з них уведемо поняття типу вузла – тега, по якому вузли можна однозначно ідентифікувати.

Вузол, що містить базисну функцію: поле – це адреса базисної або бібліотечної функції в довіднику базисних функцій, поле – адреса наступного вузла мережі. Вузол, що містить функціональну змінну: Тут – ім'я відповідної функціональної змінної,– адреса наступного вузла мережі.

Вузол умовної композиції $\tau\rightarrow$. (τ_1, τ_2) тут τ_1– покажчик на вузол мережі, відповідний до умови, τ_2 – покажчик на вузол мережі, що відповідає then частини умовної конструкції, – покажчик на вузол, відповідний до else-частини. Закриваючий вузол операції умовної композиції є фіктивним і в реалізації замість нього ділянки мережі, і безпосередньо посилаються на елемент, що випливає за закриваючим вузлом. Крім того, уведений додатковий вузол, який завжди розташовується в правому кінці схеми.

Вузол повернення з рекурсивної функції не містить полів.

Приведено приклад мережного представлення функції, що обчислює значення факторіала числа (рис. 4.5).

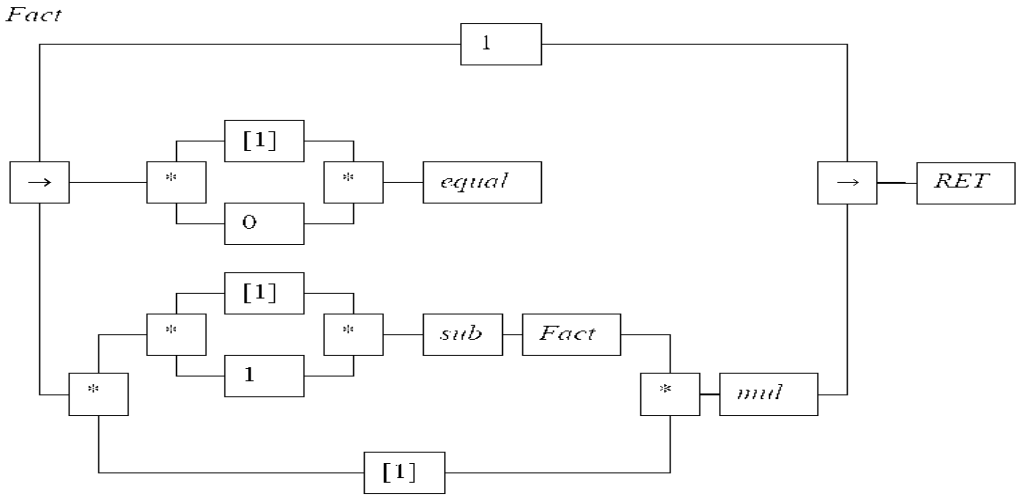

Рисунок 4.5 – Мережне подання функціональної програми обчислення факторіала числа

4.3 Реалізація інтерпретатора

Для обчислення значення функцій, представлених у вигляді схем, у системі виконання FPTL програм використовується інтерпретатор.

Для опису алгоритму роботи інтерпретатора й наступного опису системи керування паралельним виконанням FPTL програм, уведемо також поняття завдання – самостійного процесу, який породжується при обчисленні значення функції.

Завдання має наступну структуру:Task (τ,RS,DS, ready).

Тут – адреса вузла мережі, – стік адрес повернення, – стік даних, – прапор готовності. Завдання являє собою обчислювальний контекст інтерпретатора. Розглянемо алгоритм роботи інтерпретатора (див. рис. 4.6). В описі алгоритму спеціально опущені деякі елементи – вони будуть розкриті далі.

```
EVALUATE (τ, T):
1  switch τ
2    case of BF(f, next):
3       f(T.DS)
4       EVALUATE(next, T)
5    case of FV(F, next):
6       PUSH(T.RS, next)
7       EVALUATE(τ_F, T)
8    case of FORK(τ_top, τ_bottom, next):
9       T_c := CREATE-TASK(T, τ_top)
10      EVALUATE(τ_bottom, T)
11      WAIT-TASK(T_c)
12      EVALUATE(next, T)
13   case of JOIN(next):
14      return
15   case of COND(τ_p, τ_t, τ_e):
16      EVALUATE(τ_p, T)
17      D := CHECK-COND(T.DS)
18      if D = true
19         EVALUATE(τ_t, T)
20      else EVALUATE (τ_e, T)
21   case of RET():
22      τ_next := POP(T.RS)
23      EVALUATE(τ_next, T)
```

Рисунок 4.6 – Приклад роботи інтерпретатора

Пояснимо роботу алгоритму. Процедура EVALUATE ухвалює на вхід вузол мережного представлення функції й завдання . Наступні дії виконуються залежно від типу переданого вузла (конструкція **switch - case of** реалізує семантику зіставлення з зразком). У рядках 2-4 проводиться обчислення значення базисної або зовнішньої функції. Вхідні дані функції беруться зі стека даних завдання, вихідні дані також містяться в цей стек, після цього відбувається перехід до наступного вузла мережі (рядок 4) за допомогою рекурсивного виклику процедури EVALUATE. У рядках 5-7 проводиться підстановка правої частини функціонального рівняння, замість функціональної змінної . При цьому вузол, на який повинен бути здійснений подальший перехід після виконання підстановки, запам'ятовується на стеці адрес повернення завдання. У рядках 8-12 проводиться породження нового завдання. У рядках 15-20 проводиться інтерпретація умовної конструкції: спочатку обчислюється значення функції предиката, заданого мережею, потім перевіряється умова й робиться вибір подальшого перехід до вузла мережі або . Фіктивний вузол *RET* позначає закінчення обчислення значення функції й робить дії для поновлення подальшого процесу обчислень: витягає адресу наступного елемента мережі зі стека адрес повернення (рядки 21-23) і робить перехід до його інтерпретації.

В наведеному вище псевдокоді були використані наступні допоміжні процедури:

- PUSH (S, v) – додавання значення в стек;
- POP (S) – витяг верхнього елемента зі стека;
- COPY(D, S) – копіювання результату обчислення зі стека даних S у стек даних;
- CHECK-COND(S) – перевірка результату на стеці. Повертає неправду, якщо результат представляє значення або невизначеність ω, істину а якщо ні, то.

В основі реалізації системи керування паралельними процесами лежать засоби паралельного програмування, причому в будь-який момент виконання FPTL програми використовується фіксоване число потоків. Кожний потік, далі буде називатися робочим потоком (РП), виконує завдання – процес обчислення

значення функції, заданої мережею. При цьому під час інтерпретації *-вузлів схеми, можуть породжуватися інші завдання. Їм буде відповідати один з паралельних процесів обчислень, породжених у програмі операцією *. По суті, з'єднані операцією * два терма еквівалентні породженню двох процесів обчислень, перший з яких продовжує виконуватися в контексті поточного завдання, а для другого породжується нове завдання. Породжені завдання додаються в спеціальну робочу чергу (РЧ), що існує окремо для кожного робочого потоку. Крім того, у системі виконання FPTL програм є планувальник завдань,який робить пошук нових завдань дляробочого потоку, якщо він простоює.

У якості обґрунтування використання саме такого підходу (використання фіксованої кількості потоків) при реалізації паралельного виконання функціональних програм приведемо наступні аргументи.

Час створення потоку може бути досить великим. У дійсності, якщо не використовувати концепцію завдання, а перейти безпосередньо до породження потоків, то такий підхід приведе до масштабного збільшення накладних витрат, пов'язаних зі створенням потоків.

Велика кількість одночасна працюючих у системі потоків приводить до різкого збільшення частки часу роботи планувальника ядра операційної системи щодо загального часу виконання програми.

Розглянемо детально реалізацію кожного з перерахованих вище компонентів системи: організацію роботи робочих потоків, робочих черг, взаємодія між потоками й чергами й алгоритм роботи планувальника.

Кожний робочий потік являє собою незалежний процес виконання процедури інтерпретатора, і алгоритму планування. Для збереження стану інтерпретатора (збереження значень локальних змінних процедури EVALUATE) використовується власний стек даних робочого потоку [12].

Кількість робочих потоків задає програміст перед виконанням функціональної програми й у процесі виконання воно не змінюється. Для того

щоб повністю задіяти наявні в комп'ютері процесори (ядра), число РП повинне бути не менше, чим кількість фізичних ядер у комп'ютері.

Оскільки паралельна робота з даними в єдиному адресному просторі в загальному випадку вимагає використання механізмів синхронізації, доцільно розділити структури даних, використовувані всіма робітниками потоків на дві групи.

Незмінні дані всіх потоків. До них ставляться мережні представлення функціональних рівнянь, бібліотека базисних і зовнішніх функцій, реалізація конструкторів і деструкторів. Ці структури даних використовуються інтерпретатором і створюються в єдиному екземплярі перед запуском функціональної програми, не змінюються в процесі її виконання. Доступ до цих даних надається в режимі тільки для читання. Важливо відзначити, що такий порядок роботи з даними при паралельній роботі потоків не може привести до стану перегонів, тому використання механізмів синхронізації доступу до пам'яті в цьому випадку не потрібно.

Локальні дані потоків – до цих даних ставляться робочі черги потоків і породжені інтерпретатором завдання. Доступ до них надається як для читання, так і для запису. При цьому якщо при звертанні до даних є можливість гарантувати, що в будь-який момент часу робота з ними ведеться з одного єдиного потоку, використання елементів синхронізації також не потрібно. А якщо ні, то для синхронізації доступу потрібно застосування семафорів (м’ютексів), атомарних операцій або бар'єрів читання-запису [30; 31].

Перед початком виконання функціональної програми в РЧ одному з потоків додається первісне завдання. У процесі виконання функціональної програми, стан робочих потоків може змінюватися в такий спосіб.

РП активний (виконується), якщо є завдання хоча б в одній черзі всіх потоків.Переривання потоку й пошук нового завдання відбувається у двох наступних випадках:

- РП повністю виконав призначене йому завдання. Цьому стану відповідає повернення із процедури EVALUATE;

- при досягненні закриваючої операції * і тільки в тому випадку, якщо інша лінійна ділянка цієї операції, що виходить від відповідної відкриваючої операції *, ще виконується іншим потоком. Цьому стану відповідає рядок 11 описаної вище процедури EVALUATE.

Як тільки буде виконано первісне завдання, а це свідчить про те, що були виконані й усі рекурсивно породжені завдання, усім РП подається команда зупинки.

Як було відзначено в розділі, кожний робочий потік має свою чергу породжених завдань, називану робочою чергою (РЧ). РП бере нове завдання для виконання з кінця своєї черги за принципом LIFO. Однак якщо ця черга порожня, то потік змушений шукати нове завдання в черзі іншого потоку, який витягається із черги за принципом FIFO. Це, по-перше, дозволяє здійснювати міграцію більш складних з обчислювальної точки зору завдань на інші потоки й, по-друге, локалізувати дані при виконанні завдань, узятих із власної черги.

З обліком вищесказаного, приведено алгоритм роботи процедур CREATE-TASK і WAIT-TASK, які були використані раніше в процедурі EVALUATE.

```
CREATE-TASK (T, τ):
1. hch := Task(τ, [],.T,DS, [], false)
2. WQ-PUSH (PO, h)
3. return h
```

Процедура CREATE-TASK створює новий будинок h, який містить покажчик на вузол схеми й додає його в чергу РЧ робочого потоку .

```
WAIT-TASK (    ):
     1 while ≠ true
     2 SCHEDULE()
```

Процедура WAIT-TASK робить очікування готовності породженого в процедурі CREATE-TASK завдання. Якщо завдання не готове, то відбувається виклик алгоритму планування (рядок 2) для пошуку іншої роботи. Таким чином, вдається уникнути простою потока. Процедура SCHEDULE реалізує алгоритм планування.

Помітимо, що можна було б використовувати загальну чергу для завдань, зберігаючи описану логіку керування їх виконанням. Однак звертання до загальної черги безлічі потоків вимагає синхронізації доступу до неї, яка є досить витратною за часом операцією. Також слід зазначити, що принципи використання роздільних черг були описані в, а також використовується в системах паралельного програмування [20].

Для організації доступу до черги використовуються наступні процедури:

WQ-PUSH(,) – додає завдання в кінець РЧ.

WQ-TAKE() – витягає завдання з кінця РЧ (витяг за принципом LIFO). Якщо черга порожня, вертається значення.

WQ-STEAL( ) – витягає завдання з початку РЧ. Якщо черга порожня, вертається значення .

Схематично, робота описаних процедур представлена на рис. 4.6.

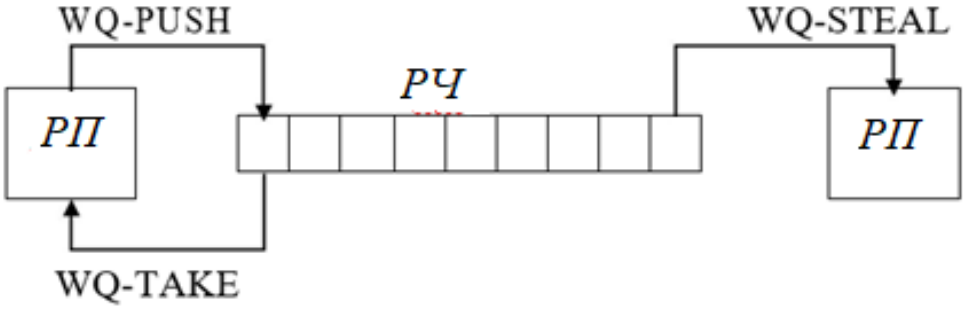

Рисунок 4.6 – Організація взаємодії потоків із чергою завдань

Робочий потік $РП_i$ додає й витягає завдання з кінця черги, у той час як інші робочі потоки $РП_k$, $k \neq$ і можуть тільки витягати завдання з початку черги. Така організація роботи з РЧ дозволяє звести до мінімуму використання елементів синхронізації паралельного доступу до пам'яті. При даному підході стан перегонів виникає лише у двох конкретних випадках:

- черга містить одне єдине завдання. Два потоки намагаються одночасно витягти його з використанням методів WQ-TAKE і WQ-STEAL;
- два потоки одночасно намагаються витягти завдання за допомогою методу WQ-STEAL.

Два вищенаведені випадки приводять до ситуації, коли те саме завдання буде виконано різними потоками. Така поведінка робить алгоритм керування паралельним виконанням недетермінованим, що неприпустимо для його ефективної роботи. Для усунення стану перегонів в цих двох випадках використовується синхронізація з використанням м'ютекса.

Слід зазначити використовуваний принцип, по якім регулюється розмір РЧ. Споконвічно РЧ має фіксований розмір в 32 елемента. Якщо при додаванні чергового завдання, у черзі не виявляється вільного місця, відбувається повне блокування доступу до черги для всіх РП (проводиться за допомогою м'ютекса), створюється нова РЧ вдвічі більшого розміру, у яку копіюються всі завдання зі статичної черги, після цього доступ потоків відновляється вже до нової РЧ.

Процедура додавання в кінець черги виглядає в такий спосіб:

```
WQ-PUSH(Q, T):
1   tail := Q.tail
2   if tail < Q.size
3       Q.D[tail] := T
4       Q.tail := tail + 1
5   else LOCK(Q.M)
6         Увеличить размер массива D в 2 раза.
7         Q.D[tail] := T
8         Q.tail := Q.tail + 1
9         UNLOCK(Q.M)
```

Спочатку проводиться перевірка наявності вільного місця в черзі (рядка 1-2). У випадку якщо черга не повна, проводиться додавання завдання в кінець черги (рядка 3-4). А якщо ні, то, проводиться збільшення розміру масиву, що утворює чергу, в 2 рази й додавання завдання в кінець збільшеної черги.

Розглянуто псевдокод процедури узяття завдання з кінця черги.

Стоки 1-3 відповідають ситуації, коли черга порожня. Інакше, якщо черга містить більш одного завдання, проводиться його витяг (рядка 5-7). Якщо в черзі втримується рівно одне завдання, проводиться одержання до неї ексклюзивного доступу за допомогою блокування м'ютекса, після цього спроба витягу завдання із черги повторюється (стоки 10-13). Процедури LOCK( ) і UNLOCK( ) використовуються для захоплення й звільнення м'ютекса відповідно [12].

```
WQ-TAKE(Q):
1   tail := Q.tail
2   if tail ≤ Q.head
3       return nil
4   else tail := tail − 1
5       Q.tail := tail
6       if tail ≥ Q.head
7           return Q.D[tail]
8       else T := nil
9           LOCK(Q.M)
10          if tail ≥ Q.head
11              T := Q.D[tail]
12          else tail := tail + 1
13              Q.tail := tail
```

Псевдокод процедури одержання завдання з початку черги виглядає в такий спосіб.

```
WQ-STEAL (Q):
1   T := nil
2   LOCK(Q.M)
3   head := Q.head
4   Q.head := Q.head + 1
5   if head < Q.tail
6       T := Q.D[head]
7   else Q.head := head
8   UNLOCK(Q.M)
9   return T
```

В початку наведеного псевдокоду проводиться блокуванням’ютекса Мдля забезпечення ексклюзивного доступу до початку черги. Після цього, якщо черга не порожня, проводиться витяг завдання з її початку.

4.4 Алгоритм роботи планувальника

Алгоритм працює в такий спосіб. Спочатку потік виконує процедуру SCHEDULE, намагається одержати завдання зі своєї РЧ (рис. 4.7а). Якщо РЧ порожня, то потік послідовно переглядає робочі черги інших потоків. (мал. 4.7б).

Якщо в якій-небудь із черг інших потоків, припустимо, є завдання, то потік бере його на виконання й потім виходить із процедури SCHEDULE. У випадку, якщо черги всіх робочих потоків порожні, потік віддає свій квант часу системі. Це дозволяє уникнути повного завантаження ядер обчислювальної системи у випадку виконання чисто послідовних програм, при яких нові завдання не породжуються.

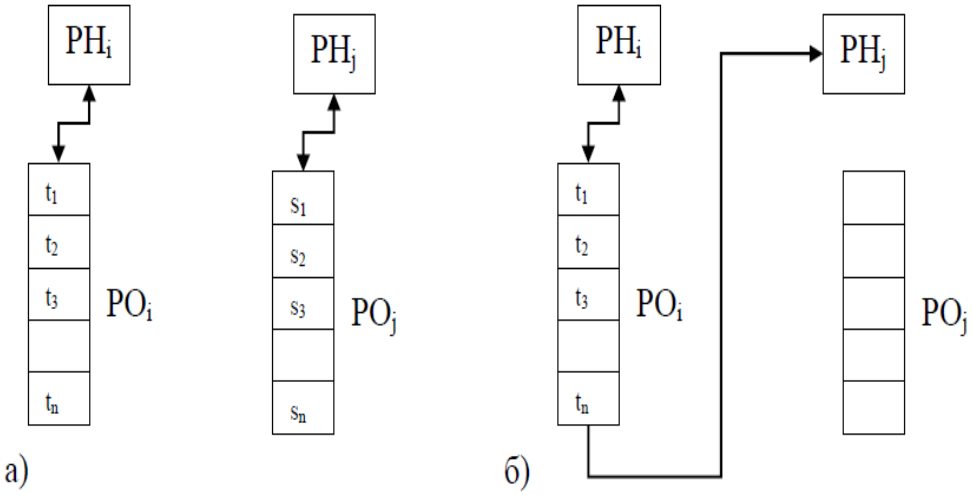

Рисунок 4.7 – Дії робочих потоків при: а) наявності завдань у своїх чергах, б) порожньої робочої черги в РП

Повний алгоритм дій РП можна представити в такий спосіб.

```
THREAD-PROC ():
1   T := «Первоначальное задание»
2   while T.ready = false
3       SCHEDULE()
```

Тут потік в циклі виконує процедуру SCHEDULE (рядки 1-2), поки не буде досягнута умова зупинки – готовність первісного завдання.

4.5 Аналіз складності завдання

Описаний вище алгоритм паралельного обчислення значень функцій має один істотний недолік: кожна операція паралельної композиції приводить до породження нового завдання. При цьому не враховується обчислювальна

складність частини мережі, що представляє це завдання. У реальних умовах це неминуче приведе до ситуації, коли накладні витрати на планування паралельного виконання завдання виявляться вище, чим час виконання цього завдання. Такий випадок виникає, наприклад, якщо ліва й права частина оператора * містить тільки базисні функції (рис. 4.8а), і зустрічається досить часто. Одним зі способів рішення даної проблеми є підхід, при якім для паралельного виконання призначаються тільки ділянки мереж, що містять дві й більш рекурсивні функції по кожному зі шляхів.

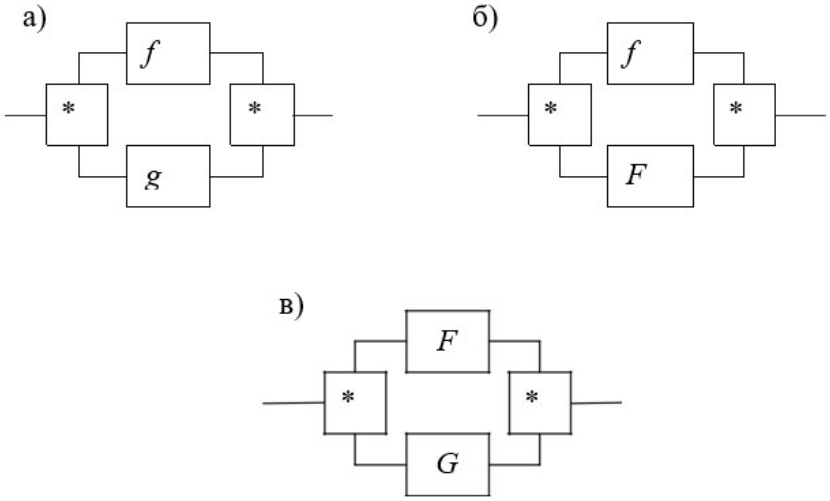

Рисунок 4.8 – Можливі варіанти мереж з оператором *, *f* і *g* – базисні функції; *F* і *G* – функціональні змінні, праві частини яких містять рекурсію

Для реалізації даного підходу, у кожному відкриваючому вузлі операції * будемо зберігати ще додатковий прапор, що сигналізує про те, чи вимагається паралельне обчислення. Значення цього прапора визначається після додаткового аналізу мережної вистави функціональної програми перед її виконанням. Внутрішня вистава модифікованого *-вузла мережі має такий вигляд.

Тут поле ухвалює значення «істина», якщо потрібне паралельне обчислення двох ділянок мережі. Частина процедури `EVALUATE`, відповідальна за роботу з модифікованим вузлом, реалізується в такий спосіб:

```
EVALUATE(,     ):
case of      (,,,          ):
if=
```

```
h:= CREATE-TASK(,)
EVALUATE()
WAIT-TASK( h)
else EVALUATE()
EVALUATE(,)
EVALUATE()
```

В цьому псевдокоді для кожного відкриваючого *-вузла перевіряється прапор (рядок 2) і залежно від його стану або відбувається породження нового завдання (рядка 3-5), або обидві частини мережі обробляються послідовно в контексті одного завдання (рядка 6-8).

Для прикладу, на малюнку 4.8 ситуаціям «а» і «б» буде відповідати послідовний випадок, у той час, як у ситуації «в» для обчислення значень функцій, що задаються рівняннями, буде породжуватися нове завдання.

Здатність будь-якої системи керувати паралельними процесами, динамічно регулювати складність (зернистість) породжуваних процесів є надзвичайно важливою проблемою для досягнення мінімального часу виконання паралельних програм [36].

4.6 Внутрішнє представлення даних

Дані в мові FPTL, як було сказано вище, можна розділити на дві категорії: вбудовані типи даних, що й задаються користувачем. До вбудованих типів даних ставляться типи int, real, bool, string, Array. До типів, що задаються користувачам, ставляться абстрактні типи даних (АТД). Для подальшого опису реалізації внутрішньої вистави і роботи з даними в мові FPTL доречно ввести поділ типів даних на дві групи.

Елементарні типи даних – називатиме такий тип даних, який може бути представлено однієї неподільною областю пам'яті. До цих типів даних ставляться int, real і bool.

До складених типів даних в FPTL ставляться string, Array і АТД. Ці типи даних не можуть бути представлені, як неподільні області пам'яті і їх реалізація вимагає враховувати деякі аспекти роботи інтерпретатора (наприклад, систему прибирання сміття). Складені типи даних реалізуються через проміжну структуру даних – дескриптор складеного типу даних. Усього є 3 типу дескрипторів: дескриптор рядка, дескриптор масиву й дескриптор АТД. Приведемо опис кожного з них.

Дескриптор строкового типу:поле – покажчик на масив символів рядка, індекс першого символу рядка в масиві, індекс останнього символу рядка в масиві. Така вистава дозволяє використовувати один загальний буфер для декількох різних рядків, наприклад, у випадку якщо один рядок є підрядком іншої.

Дескриптор масиву – кількість елементів масиву, покажчик на буфер, в якім утримуються елементи даних масиву.

Дескриптор АТД – ім'я конструктора, за допомогою якого АТД був створений, - покажчик на буфер елементів даних, до яких конструктор був застосований.

Для зручності реалізації інтерпретатора, внутрішнє представлення всіх типів даних у мові FPTL упаковується в одну структуру, розмір якої рівний максимальному з розмірів усіх елементарних типів даних і дескрипторів складених типів. Такий спосіб упакування даних дозволяє уникнути роботи з областями пам'яті різного розміру, спрощуючи реалізацію інтерпретатора. Крім цього, з метою спрощення налагодження програм структура елемента даних також містить інформацію про його тип.

В мові FPTL усі базисні функції роблять операції над кортежами даних, реалізація внутрішнього представлення яких суттєво впливає на зменшення часу виконання операцій над ними. Двома основними операціями, виконуваними над кортежами даних, є операція конкатенації двох кортежів і операція вибору одного елемента з кортежу. Були досліджені наступні варіанти реалізації внутрішньої вистави кортежів.

Реалізація кортежів на основі масивів фіксованої довжини. При даному підході елементи кортежу даних упаковуються в масив, довжина якого відповідає розміру кортежу. Операція вибору елемента з кортежу при цьому реалізується тривіально - це вибір елемента з масиву. При конкатенації, дані із двох вихідних кортежів копіюються в новий масив з довжиною рівній сумі довжин вихідних масивів. Тому що створення результуючого масиву вимагає виділення нової області пам'яті, кожна операція конкатенації кортежів вимагає звертання до системи керування пам'яттю. Це є головним недоліком даного способу реалізації внутрішнього представлення кортежів.

Реалізація кортежів на основі однозв'язних списків. У даній реалізації кортеж являє собою однозв'язний список елементів даних: кожний елемент даних зберігає в собі посилання на наступний елемент. Кінцевий елемент посилається на самого себе. Для реалізації конкатенації двох кортежів потрібно пройтися за списком від головного до кінцевого елемента, і привласнити посилання на наступний за останнім елементом лівого кортежу адресу першого елемента правого кортежу. Щоб уникнути пошуку останнього елемента лівого кортежу, можна на додатку зберігати адресу останнього елементу кортежу. Перевагою даної реалізації є проста реалізація конкатенації кортежів. Недоліками є: необхідність послідовного перегляду списку для реалізації операції витягу елемента кортежу й потреба в збереженні додаткового покажчика на наступний елемент кортежу. Крім того, створення кожного нового елемента кортежу вимагає звертання до системи керування пам'яттю.

Реалізація кортежів на основі динамічно розширюваних масивів. Дана реалізація концептуально нагадує роботу з локальними змінними в стеці, застосовувану в імперативних мовах програмування. У мові FPTL у якості локальних змінних виступають вхідні кортежі даних. Саме таке представлення кортежів і було використане в поточній реалізації. Схема роботи з даними при стековому представлені кортежів описується наступними положеннями.

Статично виділяється масив невеликого розміру для зберігання елементів кортежів. Робота з масивом ведеться за принципом стека;

Для кожного елемента або ділянки мережного представлення функціональної програми відомі n-арності (e_n) вхідного кортежу і вихідного кортежу. Підрахунок арностей проводиться на етапі створення мережного представлення функції.

Нехай базисна функція, ухвалює на вхід кортеж даних арності, а результатом її обчислень є кортеж даних арності. Тоді при обчисленні значень функції вхідним кортежем даних є верхній елемент стека. При обчисленні значення базисної функції кортеж вхідних даних не витягається зі стека, а робиться його копія. Після обчислення значення базисної функції елементів результуючого кортежу містяться в стек. (див. рис. 4.9)

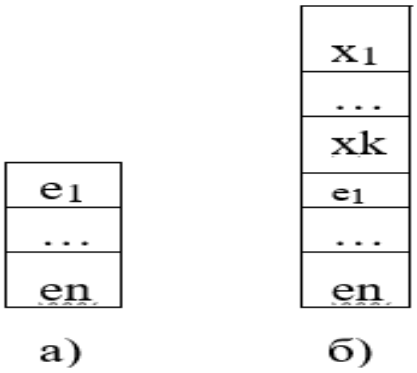

Рисунок 4.9 – Стан стека даних

а) до б) після обчислення значень базисної функції.

При просуванні по мережному виставі функції ліворуч після одержання результуючого кортежу, проводиться згортання стека, тому що вихідний кортеж лівої частини схеми стає більше не потрібний. При цьому проводиться переміщення елементів кортежу на місце елементів кортежу й розмір стека зменшується на кількість елементів. Ця операція дозволяє виключити розростання стека: при такій організації роботи стек не буде містити даних,використання яких не потрібно для подальших обчислень. (рис. 4.10)

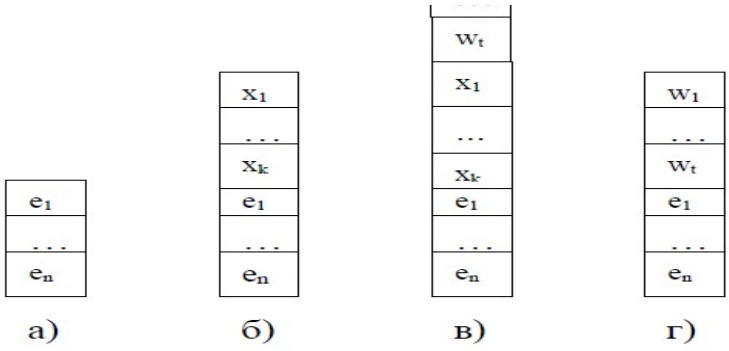

Рисунок 4.10 – Стан стека даних: а) – споконвічне, б) – після обчислення значення баз. функції *f*, в) – після обчислення значення баз. функції *g*, г) – після згортки стеку

Якщо при обробці *-вузла мережі породжується нове завдання, то воно містить свій власний стек даних, у який статично копіюється вхідний кортеж, а після виконання завдання витягається результуючий кортеж і додається в стек вихідного завдання.

Якщо при обробці *-вузла нове завдання не породжується, то спочатку обчислюється кортеж, відповідний до результату інтерпретації нижньої частини *-вузла. Цей кортеж витягається зі стека й запам'ятовується в локальній змінній інтерпретатора. Потім обчислюється й додається в стек результуючий кортеж,отриманий після виконання верхньої частини *-вузла. Після цього в стек додається збережений раніше верхній кортеж.

Якщо при додаванні чергового елемента з'ясовується, що в стеці більше немає місця, відбувається його збільшення: виділяється новаобласть пам'яті в 2 рази більшого розміру, у неї копіюються елементи старого стека, потім стара область пам'яті повертається системі керування пам'яттю.

Організація роботи з кортежами даних за допомогою стека дозволяє добитися наступних переваг. Знижується кількість запитів до системи керування пам'яттю, оскільки звертання до неї проводяться тільки при заповненому стеці;
Зберігається локальність даних: пам'ять виділяється великою безперервною областю й елементи даних кортежів розташовуються «поблизу». Таке розміщення

даних добре сполучається із принципами організації роботи кеш-пам'яті процесора.

4.7 Обчислення значень конструкторів і деструкторів

Перед етапом створення мережного представлення функціональної програми для кожного конструктора й деструктора створюються спеціальні процедури-перехідники. Алгоритм їх роботи може бути представлений у такий спосіб:

```
EVAL-CONSTRUCTOR (,        ):
1. := ALLOCATE(( ))
2. COPY(,,( ))
3. := ADT(,   )
4. PUSH(,  )
EVAL-DESTRUCTOR(,        ):
1. := GET(      )
2. if  .=
3. for each   in  .  do
4. PUSH(,   )
5. else PUSH(, ω)
```

Тут – стек даних завдання, - ім'я конструктора, процедура ALLOCATE(( )) виділяє буфер розміру, рівної *n*-арності вхідного кортежу конструктора, процедура COPY робить копіювання ( ) елементів зі стека даних у буфер, GET( ) одержує копію верхнього елемента стека. Адреси створених процедур перехідників заносяться в довідник базисних функцій і використовуються при обчисленнях (у процедурі EVALUATE).

Виклик зовнішніх функцій (foreign functions) також реалізований через спеціальні процедури-перехідники. Для кожної зовнішньої функції створюється процедура перехідник, яка витягає вхідні дані зі стека даних завдання, перетворює їх у формат вхідних параметрів процедури, робить виклик процедури, перетворює

її вихідні параметри й поміщає їх назад у стек даних завдання. В описі вузлів мережної вистави під адресою зовнішньої функції розуміється адреса її процедури-перехідника. Усі використовувані в програмі бібліотеки зовнішніх функцій завантажуються в оперативну пам’ять безпосередньо перед виконанням FPTL програми.

5 ОПИС МОЖЛИВОСТІ ВИКОРИСТАННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

5.1 Перевірка алгоритму прибирання сміття

Потік, що ввійшов в стан прибирання сміття, подає сигнал на зупинку всіх інших потоків, що виконуються, чекає їхньої зупинки й починає безпосередньо процес прибирання сміття.

У ході цього процесу, відбувається сканування областей пам'яті, які використовуються потоком. Після цього вся виділена пам'ять ділиться на дві категорії: досяжні області пам'яті й сміття. Уся виділена пам'ять, позначена як сміття, повертається в купу. Після цього виконання всіх потоків відновляється.

Головним недоліком прибирання сміття є необхідність зупинки виконання програми, так звана пауза прибирання сміття (garbage collector pause). Ця пауза сильно збільшує час виконання паралельних програм, що активно використовують динамічну пам'ять (наприклад, що проводять операції над списковими структурами даних). Для грубої оцінки максимального прискорення, якого можна добитися при паралельному виконанні програм, що використовує прибирання сміття, можна скористатися законом Амдала.

Для зменшення часу паузи прибирання сміття прибігають до наступних технік. Розпаралелювання процесу прибирання сміття. У цьому випадку процес пошуку досяжних областей пам'яті проводиться паралельно. Також може бути розпаралелено повернення сміття в купу. Це дозволяє деяким чином скоротити час паузи прибирання сміття. Для прибирання сміття в середовищі виконання FPTL програм використовується саме такий варіант оптимізації.

Використання фонового прибирання сміття, що працює без зупинки виконання програми. Існуючі алгоритми досить складні в реалізації й вимагають підтримки з боку апаратного забезпечення або зміни ядра операційної системи. Один з варіантів реалізації фонового прибирання сміття використовується в комерційній віртуальній машині JVM Azul.

До всього вищесказаного, слід зазначити, що прибирання сміття не потрібно для даних елементарних типів: пам'ять для їхнього зберігання виділяється й звільняється разом зі стеком даних завдання.

5.2 Мови й методи реалізації

В табл 5.1 даний опис засобів розробки, використаних для реалізації кожної підсистеми, що входить до складу системи паралельного виконання FPTL програм.

Таблиця 5.1 – Програмні засоби, використані для реалізації компонентів системи виконання FPTL програм

Підсистема	Реалізація	Мова реалізації	Використовувані бібліотеки
Лексичний аналізатор	GNU Flex	C	
Синтаксичний	GNU Bison	C	
Семантичний	Власна	C++	
Підсистема пошуку рекурсивних рівнянь	Власна	C++	
Інтерпретатор	Власна	C++	
Робочі черги	Власна	C++	boost/atomic
Підсистема керування паралельним виконанням	Власна	C++	boost/thread
Складання сміття	Boehm GC	C	
Бібліотекабазиснихфункцій	Власна	C++	
Підсистемавикликузовнішніхпроцедур	Власна	C	libffi

Система виконання FPTL програм реалізована мовою C++ за винятком підсистем, виконаних на основі сторонніх бібліотек. Була обрана саме ця мова програмування з ряду наступних причин:

- програми написані мовою C++ транслюються безпосередньо в машинний код для конкретної процесорної архітектури. Це помітно збільшує швидкодію інтерпретатора, у порівнянні з можливим варіантом його реалізації на мовах Java або C#, програми на яких транслюються в проміжне представлення й виконуються надалі під керуванням віртуальних машин;

- у мові C++ є всі необхідні засоби й бібліотеки,які потрібні для реалізації низькорівневих роботи з потоками й забезпечення синхронізації доступу до пам'яті (бібліотеки boost/thread і boost/atomic);

- засоби компіляції й компонування, а також перераховані бібліотеки мови C++ є кросплатформовою й присутні практично на всіх широко використовуванихопераційних системах і процесорних архітектурах. Недоліками мови C++, використаного для реалізації системивиконання FPTL програм є:

- відсутність вбудованих засобів для автоматичного керування пам'яттю. Для усунення цього недоліку як реалізації збирача сміття була використана модифікована бібліотека Boehm GC. Модифікація полягала в переході від сканування стека потоку при пошуку досяжних об'єктів до сканування безпосередньо стеків даних завдань. Дана модифікація дозволила позбутися «консервативного» характеру роботи збирача сміття й зробити процес прибирання сміття більш ефективним.

Відсутність вбудованого інтерфейсу для роботи із зовнішніми програмними модулями, який необхідний для реалізації механізму виклику зовнішніх функцій у мові FPTL. Для усунення цього недоліку була використана стороння бібліотека libffi, що дозволяє реалізовувати зовнішні програмні модулі, що підключаються під час виконання програми.

Мови програмування, такі як Java і C#, позбавлені перерахованих недоліків. Однак відсутність у них підтримки компіляції програми в машинний код і прямої роботи з пам'яттю може привести до істотних накладних витрат, що робить їхнє використання для розробки подібного роду систем досить ризикованим.

Реалізація системи виконання FPTL програм є кросплатформовою. Є варіанти складань під операційні системи Windows, Linux і OS X.

ВИСНОВКИ

Наведений опис системи виконання FPTL програм. Перелічимо найбільш важливі результати, які при цьому отримані.

Розроблена багатокомпонентна архітектура системи й описана організація взаємодії її підсистем.

Розроблена внутрішня вистава функціональних програм у вигляді мереж ефективного виконання, що забезпечує FPTL програми на багатоядерних комп'ютерах, і представлення всіх структур даних, що використовуються у мові.

Розроблені алгоритми:

- інтерпретатора обчислення, що реалізує, значення функцій, представлених у вигляді мереж і породження, що виробляє нові паралельні процеси (завдання);
- планувальника паралельних процесів, що виконує розподіл завдань по робочих потоках;
- черг, що використовуються для ефективного зберігання породжених інтерпретатором завдань;
- керування складністю (зернистістю) породжених паралельних процесів (завдань).

Розглянуто допоміжні аспекти організації роботи системи виконання FPTL програм, а саме: автоматичне керування пам'яттю, представлення даних і виклики зовнішніх функцій.

Було зроблено огляд методів і програмних засобів, використаних для реалізації наведених алгоритмів і підсистем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. 1. Borkar S.Y. and others. Platform 2015: Intel processor and platformevolution for the next decade. // URL: http://goo.gl/43dbG3 .

1. Филд А., Харрисон П. Функциональное программирование. // М.Ж Мир,1993

2. Бажанов С.Е., Кутепов В.П., Шестаков Д.А. Язык функционального параллельного программирования и его реализация на кластерных системах. // Программирование. 2015, № 5.

3. McCarthy J. Recursive functions of symbolic expressions and their computation by machine. // Cambridge, Mass.: MIT, 2006

4. Milner R.G. The standard ML core language. Polymorphism // TheML/LCF/Hope Newsletter 2005. V. 2 № 2.

5. Peyton Jones S. L. The implementation of functional programming languages. // London: Prentice Hall, 1987

6. Church A. The calculi of lambda-conversion. // Ann. of Math. Studies, Princeton, N.J.: Princeton University Press, 1991. V. 6.

7. Milner R. A calculus of communicative systems. // LNCS, vol. 92. Springer, Heidelberg, 2018

8. Hoare C.A.R. Communicating sequential processes. // Communications of the ACM. Vol. 21 No. 9, 1998

9. F# Historical Acknowledgements. // URL: http://research.microsoft.com/ en-us/um/cambridge/projects/fsharp/ack.aspx

10. Peyton Jones S., Singh S. A Tutorial on Parallel and Concurrent Programming in Haskell // Microsoft Research, Cambridge, 2008

14. Petricek T., Skeet J. Real world functional programming. // Manning Publications Co., 2009

14. Rui Shi, Hongwei Xi. A Linear Type System for Multicore Programming, 2009

15. Lee E. A. The Problem with Threads. // Electrical Engineering and Computer Sciences University of California at Berkeley, 2006

16. Shavit N., Touitou D. Software transactional memory. // Distributed Computing, Vol. 10, No. 2, 1997

17. Marlow S., Newton R., Peyton Jones S. A monad for deterministic parallelism. // Haskell'11, 2011, Tokyo, Japan.

18. Ахо А., Лам М., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструментарий. 2 изд. // Москва, Вильямс, 2008.

19. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. // IEEE Transactions on Computers, Vol. C-28 No. 9, 1979

20. Dijkstra E.W. Solution of a problem in concurrent programming control. // Communications of the ACM, Vol. 8 No. 9. 1965

21. Improving the Automated Testing of Web-based Services by Reflecting the Social Habits of Target Audiences /I.Shubin, I. Turevska // Proceeding of 2015 Information Technologies in Information Business Conference (ITIB) 7 – 9 October, 2015, IEEE Catalog Number CFP15D13-PRT pp. 93-96

22. Frigo M., Leiserson C.E., Randall K.H. The implementation of the Cilk-5 multithreaded language. // In proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018.

23. Cilk Arts, Inc., Burlington, Massachistetts. Cilk++ Programmer's Guide,2018. // URL: http://goo.gl/MzbhKI

24. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 3-е издание. // Вильямс, 2016

25. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. // СПб.: БХВ-Петербург, 2012.

26. Дудар З.В. Порівняння методів прогнозування часових рядів / З.В. Дудар, М.С. Широкопетлєва, О.А. Пономаренко // Бионика интеллекта. – Харьков: ХНУРЭ, 2018. – Вип.2 (91). – С.41-47.