Міністерство освіти і науки України

Київський національний університет імені Тараса Шевченка

Кафедра обчислювальної математики факультету кібернетики

**Паралельні обчислення у методі ‘сіток’ розв’язання еліптичних рівнянь на комп’ютері гібридної архітектури**

**Текстова частина до бакалаврської роботи**

**за спеціальністю „Прикладна математика” 6.040301**

#### Керівник бакалаврської роботи

доктор фіз.-мат. наук

Хіміч Олександр Михайлович

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2015 р.

Виконав студент

Оленченко Ілля Андрійович

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2015 р.

Київ 2015

**TODO в текстову частину**

I) Текстова частина (ТЧ):

0) общее оформление http://web.znu.edu.ua/lab/fordep/oformlenie/diplom\_specialista.htm

1) прочитать как должно быть оформлена работа

2) індивідуальне завдання на КР(МР);

3) календарний план;

4) зміст;

5) анотація 1 ст. ;

6) вступ 1-3 ст.;

7) основна частина (її розділи): 30-50 ст.;

– аналіз існуючих методів (алгоритмів) вирішення поставленої задачі;

– обґрунтування вибору рішення;

– вибір принципу дії системи чи обґрунтування методик;

– розробка структурної і (або) функціональної схеми;

– розробка принципової схеми;

– експериментальні дослідження;

– метрологічні характеристики;

– алгоритмічне та програмне забезпечення;

8) висновки;

9) література;

10) глосарій;

11) додатки;

II) носій інформації на якому розміщені текстова частина роботи, програми, матеріали та презентація доповіді.

Наведені заголовки основної частини є рекомендованими для КР(МР). Основна частина КР(МР) повинна мати аналітично-розрахунковий характер.

Дозволяється вводити нові частини за вказівками керівника.

Кожний розділ ТЧ може складатися з підрозділів, пунктів, підпунктів, параграфів.

Обсяг ТЧ повинен складати 30-50 сторінок машинописного тексту на аркушах фор­мату А4, причому об’єм основної частини повинен складати не мен­ше 70 % всієї роботи.

**Вступ**

В сучасний період розвитку обчислювальної техніки актуальність числових методів, що дозволяють розв’язувати широкий клас задач за допомогою ЕОМ, продовжує зростати. Особливо гостро постає питання оптимального використання усіх запропонованих компонентів з архітектури комп’ютера. За останні 10 років розвиток графічних процесорів (GPU) при використанні у сукупності із центральним процесором (CPU) нестримними темпами відкриває все нові можливості ЕОМ. З одного боку, як і раніше продовжується приріст продуктивності ЕОМ за рахунок збільшення кількості процесорів. З іншого боку, гібридні системі стають більш популярні, що зумовлено використанням елементів принципово нової архітектури.

**Актуальність роботи** зумовлена нестримним рухом технологій з плином часу. Таким чином на вже розв’язані задачі можна подивитися під іншим кутом, а саме використання гібридних комп’ютерів для розв’язання диференціальних рівнянь. Значна частина прикладних задач зводиться до математичних моделей, які описуються системами лінійних алгебраїчних рівнянь (СЛАР) з розрідженими матрицями.

Використання комп’ютера встановлює задачу раціонального використання ресурсів, адже навіть в наш час ресурсами є скінчені величини, якими не можна розкидатись. Зараз ми маємо багатопроцесорні системи, які можуть виконувати декілька процесів одночасно, або майже одночасно. Проте архітектура CPU достатньо обмежена, тоді як GPU дає можливості дуже великої кількості процесорів, якої позбавлений CPU. А саме виконання операцій одночасно у великих кількостях. Звичайно використання лише однієї частини комп’ютера гібридної архітектури не може дати оптимального часу, тож є сенс для розв’язання такої задачі використовувати обидва процесора.

Протягом останніх десятиліть на основі розроблених алгоритмів було створено низку бібліотек, до яких увійшли програми для розв’язування СЛАР з розрідженими матрицями: SparseBLAS, SparsPak, SSP, Boeing Library, Bell Laboratories, IMSL, NAG та інші. Серед програмних засобів, призначених для паралельних комп’ютерів, ефективні реалізації алгоритмів з розрідженими матрицями пропонують бібліотеки Aztec, BlockSolver95, Hypre, ILUS, MUMPS, PARMS, PSBLAS, PSPASES, PSparslib, SUPERLU, SPARSKIT.

Розвиває та покращує ринок GPU на сьогодні компанія NVidia та AMD. Її розробка CUDA (Compute Unified Device Architecture) або ATI Stream Technology дає можливості використовувати можливості GPU для запуска обчислювальних програм. Різниця між використанням багатопроцесорного одного керуючого пристрою та гібридної архітектури полягає у наступному:

Перший потребує від алгоритмів більшої степені паралелізма на однотипових процесорних ядрах, які на програмному рівні вирішуються за допомогою спеціальних програмних систем такі як MPI. Другий потребує від алгоритма більш складної багаторівневої паралельної моделі. Такі системи потребують відповідей на додаткові запитання до алгоритма та використовування пам’яті.

Розв’язання диференціальних рівнянь завжди було суттєвою проблемою багатьох задач з моменту існування таких задач. Розв’язуючи ту чи іншу реальну проблему за допомогою математики дослідники будують математичні моделі, які в свою чергу у багатьох випадках зводяться до розв’язання диференціального рівняння бо саме диференційні рівняння краще за будь які інші окреслюють суть процесу.

Було б чудово, якщо ЕОМ мали засоби вирішувати такий великий клас задач аналітично, проте саме лише використання машини для пошуку розв’язка змушує нас відмовитися від точних розв’язків на користь наближених в деякому наборі точок. Замість точної задачі можна використати наближення диференційного оператора у вигляді різницевої схеми, поставити йому у відповідність граничні початкові умови і знайти наближений розв’зок за допомогою системи лінійних алгебраїчних рівнянь.

Використання таких систем і залежність від їх розв’язку має не лише клас диференційних рівнянь, а й багато областей науки й підприємств. Не зважаючи на велику увагу до створення програмного забезпечення з лінійної алгебри багато проблем ефективного його використання залишаються.

Такі машинні алгоритми здебільшого передбачають розв’язання задач із потрібною точністю та будь якою кількістю потрібних точок, де ми можемо знайти наближений розв’язок. Простіше за інші, математичні моделі записуються та обчислюються на ЕОМ за допомогою лінійної алегбри та СЛАР. Список методів для розв’язання систем можемо перераховувати достатньо довго (метод Гауса, метод Гауса-Жордана, метод Гауса-Зейделя, матричний метод розв’язання систем лінійних алгебраїчних рівнянь, метод квадратного кореня, метод Крамера, метод прогонки, метод Якобі, метод релаксації, розгалуження Холецького, проекційні методи, метод регуляризації Тихонова, ітераційні методи, метод Річардсона).

Обирання того чи іншого метода залежить від вигляду отриманої системи, її властивостей, бажаної швидкості знаходження розв’язку. Для використання методу на ЕОМ, маючи різницеву залежність для знаходження наближеного розв’язку, ітеративні матимуть перевагу при розпаралелюванні. Кожна наступна ітерація знаходиться з використанням попередніх обчислень, доки процес не збіжиться в деякій точці. Гарантія збіжності забезпечується кроками методу та початковим наближенням. Такі методи, маючи коректну постановку збігаються при достатній кількості операцій. Тож головне питання для ітеративного процесу складається у швидкості знаходження наближення, бо в деяких умовах використання методів, як для швидкого обчислення у реальному часі, при надточних обчисленнях можуть залежати не тільки отримані розв’язки, приймання рішення, а навіть життя людини.

**Метою роботи** є отримання результату оптимізації за допомогою паралельних обчислень та презентація методів паралельних алгоритмів для таких задач.

**Основна частина**

Зазвичай основним обчислювальним компонентом систем для високопродуктивних обчислень, включаючи кластери, є центральний процесор. Проте, вже починаючи з процесорів, які з’явилися в 1989 році у складі комп’ютерів з’явився такий елемент, як співпроцесор, що можна вважати гібридизацією на апаратному рівні.

У середині 2000-х років для обчислювальних цілей почали використовувати графічний процесор.

Основна проблема складається в тому, щоб знайти спосіб виконувати обчислювальні задачі за допомогою графічного процесора. Та при використанні будь якої з існуючих технологій, це наддасть алгоритмам нових швидкостей, через високі обчислювальні можливості GPU.

Такі особливості GPU пояснюються особливостями архітектури. Якщо сучасні CPU мають декілька ядер (2, 4, 8), графічний процесор спочатку створювався як багатоядерна структура, у якій кількість ядер вимірюється сотнями. Різниця в архітектурі обумовлює й різницю в принципах дії. Якщо архітектура CPU пропонує послідовну обробку інформації, то GPU історично пропонувався для обробки комп’ютерної графіки, тому розрахован на масивно паралельні обчислення.

Кожна з цих двох архітектур має свої переваги. CPU краще працює з послідовними задачами. При великій кількості оброблюваної інформації перевагу має GPU. Умова лише одна – в задачі повинен спостерігатися паралелізм.

«GPU вже досягли тієї точки розвитку, коли багато додатків реального світу можуть з легкістю виконуватися на них, при чому швидше, за багатоядерні системи. Майбутні обчислювальні архітектури стануть гібридними системами з графічними процесорами, які будуть складатися з паралельних ядер працюючи у зв’язку з багатоядерними CPU»[цитата]

На сучасному ринку можна виокремити 2 конкурентні програмно-апаратні архітектури, за допомогою яких можливо використати повну потужність гібридного комп’ютера

* NVidia CUDA
* ATI Stream Technology

Це найпопулярніші та швидко зростаючі системи, які поєднують у собі всі вдалі напрацювання обох компаній та інших досліджень. До цієї двійки прагне долучитися і компанія Intel з їх технологією Intel Larrabee, проте на дану мить, ця пропозиція не має цінності.

В даній роботі була обрана CUDA як досліджувана технологія, з наступних причин:

* Інтерфейс програмування додатків CUDA (CUDA API) заснований на стандартній мові програмування С з деякими обмеженнями. За думкою розробників це повинно спростити та пом’якшити процес вивчення архітектури CUDA.
* Поділена між потокам пам’ять (shared memory) розміром у 16 Кб може бути використана під організований користувачем кеш з більш ширшою полосою пропуску ніж при виборці зі звичайних текстур.
* Більш ефективні транзакції між пам’яттю центрального процесора та відеопам’яттю.
* Повна апаратна підтримка цілочисельних та бітових операцій.
* Підтримка компіляції GPU кода коштами відкритого LLVM (низькорівнева віртуальна машина).

З обмежень маємо:

Усі функції, виконані на пристрої не підтримують рекурсії.

Такі висновки представлені при порівнянні CUDA з традиційним підходом до організації обчислювань загального призначення за допомогою можливостей графічних API.

Використовувана в тестуванні алгоритмів система має наступні характеристики:

* CPU – Intel Xeon E-5606 8 cores 2.13 GHz
* Memory – 12Gb
* GPU - Tesla M 2090, 512 ядер CUDA, 665 GFlops

За класифікацією по Фліну загальна архітектура може мати наступні варианти реалізації:

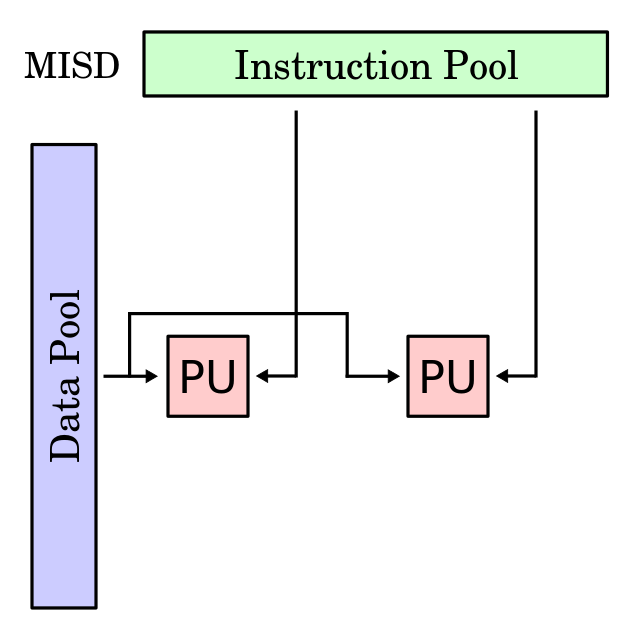
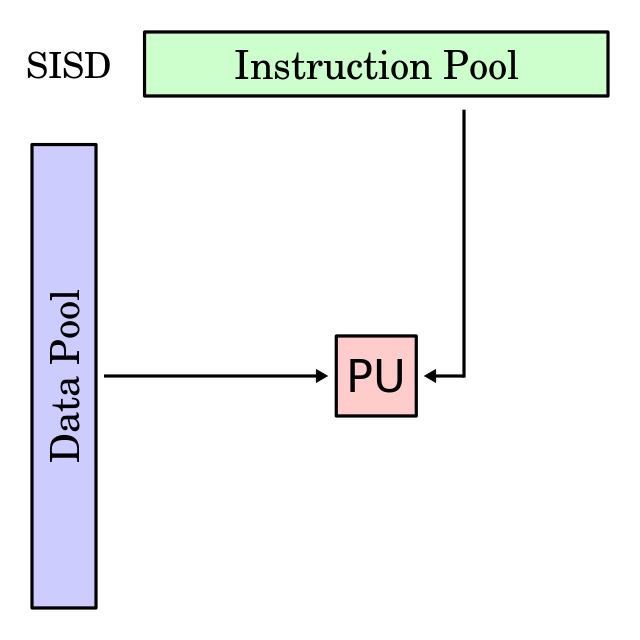
* SISD
* MISD
* SIMD
* MIMD

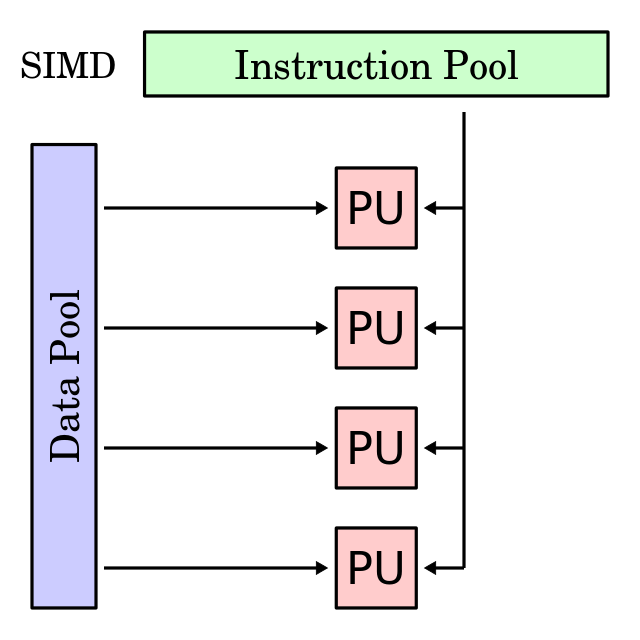
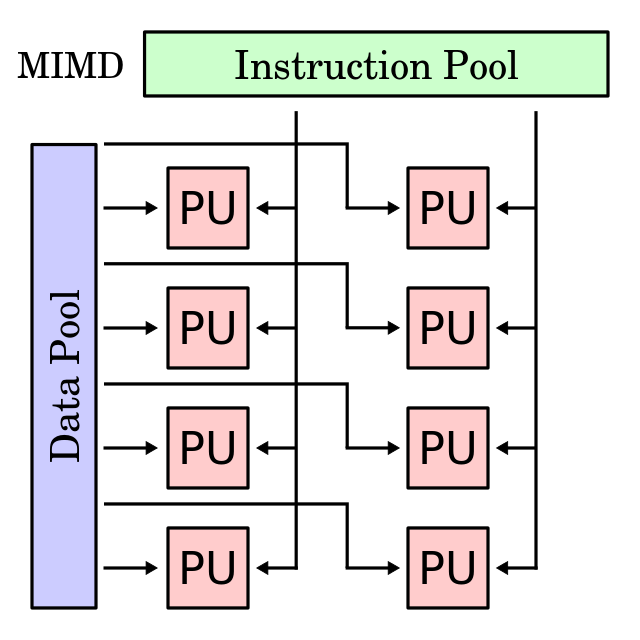
(Single/Multiple Instruction Single/Multiple Data)

За цією класифікацією ЕОМ поділяються на 4 типа за кількістю потоків команд та даних. За звичай попередники сучасних комп’ютерів мали одну головну архітектуру SISD. Паралельно йшла розробка векторних та матричних архітектур MISD, SIMD та багатопроцесорні MIMD.

Кожна архітектура має свої особливості та свої як гарні так і погані сторони. Методи розпаралелювання можуть бути оптимізовані для одних чи інших типів архітектур.

Вигляди архітектур:

Саме MIMD (Multiple Instruction stream, Multiple Data stream) - концепція архітектури комп'ютера, що використовується для досягнення паралелізму обчислень.

Вважатимемо, що паралельний обчислювальний комплекс має такі складові:

1. Хост-комп’ютер для здійснення керування використання багатопроцесного обчислювального ресурсу, проведення загальносистемного моніторингу, комунікації з термінальними мережами користувачів, візуалізації розв’язків задач та реалізації тієї частини обчислювального процесу, яка не розпаралелюється;
2. Обробляюча частина, що містить обчислювальні вузли для розв’язування задачі з паралельною організацією обчислень. Вона є однорідною масштабованою системою, яка складається з багатьох високопродуктивних процесів з власною оперативною та дисковою пам’яттю, об’єднаних комунікаційним середовищем міжпроцесної взаємодії;
3. Дискове сховище для зберігання програмних модулів, великорозмірних даних та результатів обчислень;
4. Комунікаційні середовища та комутаційне середовище, призначені для ефективної взаємодії обчислювальних вузлів при проведенні розрахунків.

Операційна система хост-комп’ютера повинна забезпечувати виконання ряду завдань, таких, як запуск програми на хост-комп’ютері, формування завдання і запуск процесу розв’язування задачі на вибраній кількості процесів, моніторинг виконуваних завдань, збереження і візуалізація протоколів паралельних розрахунків, адміністрування доступних частин розподіленої файлової системи. Також має бути встановлено відповідне середовище міжпроцесної взаємодії та компілятор, що підтримує мову програмування, на якій написано виконувану програму.

Серед основних вимог до програм, написаних для виконання на MIMD-комп’ютерах, виділяють:

1. паралелізм – здатність виконання програми багатьма процесами одночасного;
2. масштабованість – забезпечення можливості виконання програми з використанням різної кількості процесів;
3. локальність – така організація обчислень, при якій звернення до локальних даних відбувається значно частіше, ніж до віддалених.

Додаткові складнощі, які виникають при розробці такого забезпечення мають такі характери:

* Програма повинна бути складена з кода для CPU (на звичайній мові програмування С / С++) та кода для графічного процесора написаного на спеціальній мові, CUDA.
* Друга проблема пов’язана з ефективним використанням обчислювальних ресурсів, з узгодженням розподілу обчислювальних ресурсів на ядрах (GPU та CPU).

Щоб дати відповідь на запитання чи має сенс використовувати той чи інший метод для даних архітектур важливо мати коефіцієнти, що характеризують ефективність паралельних алгоритмів. Розпаралелювання обчислень багатоваріантно, тобто для MIMD машин з однією й тою ж самою структурою міжпроцесорних зв’язків можуть бути побудовані різноманітні варіанти алгоритмів про паралельну організацію обчислень. Ефективність алгоритмів у значній мірі визначається схемою розподілу вихідних даних між процесами. Наприклад, найпростішим способом розподілу є блочний, при якому матриця розподіляється на декілька рівних блоків, кількість яких дорівнює кількості процесів Для порівняння й оцінки якості алгоритмів паралельних обчислень будемо користуватися такими критеріями, як коефіцієнт прискорення та коефіцієнт ефективності:

Деякі автори вводять й інші характеристики:

При побудові паралельних алгоритмів вважається, що необхідна для реалізації обчислювального алгоритму інформація зберігається і обробляється в оперативній пам’яті гіпотетичного послідовного комп’ютера або ж у сумарній пам’яті MIMD-комп’ютера, на якому виконуються *р* процесів, тобто обчислювальний процес здійснюється без використання зовнішньої пам’яті.

Опис задачі:

У якості модельної задачі розглядаємо задачу Діріхлє для самоспряжених рівнянь другого порядку в прямокутнику з заданими граничними умовами.

На сітці поставимо у відповідність різницеву задачу, використовуючи звичайну схему «хрест».

Для запису системи лінійних алгебраїчних рівнянь в матрично-векторному вигляді необхідно спочатку встановити відповідність між впорядкуванням рівнянь та впорядкуванням невідомих. Роздивимось два впорядкування невідомих. Природне впорядкування та червоно-чорне. Природне:

yi\*j\* наступна за yij при j\* > j або якщо j\* = j та i\* > i.

Червоно-чорне:

Нехай червоні невідомі утворюють множину всіх таких yij для яких i+j парне, і нехай чорні невідомі при не парній сумі. Тоді червоно-чорним впорядкуванням може бути будь яке впорядкування, при якому будь яке чорне невідоме йде після червоної невідомої.

Приклади:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 11 | 12 | 13 | 14 | 15 |
|  | 6 | 7 | 8 | 9 | 10 |
|  | 1 | 2 | 3 | 4 | 5 |

Звичайне впорядкування.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 6 | 14 | 7 | 15 | 8 |
|  | 11 | 4 | 12 | 5 | 13 |
|  | 1 | 9 | 2 | 10 | 3 |

Червоно-чорне впорядкування.

При розв’язування СЛАР з матрицями великих порядків не рідко реалізація прямих методів, навіть тих, що враховують розрідженість матриці, на комп’ютері може виявитися досить складною і неефективною. Проблеми машинної реалізації прямих методів пояснюються обмеженням об’єму оперативної пам’яті та зниженням швидкодії при використанні дискової. Запис систем з матрицями, які мають певну специфіку, у вигляді процедури обчислення вектора *Ах* потребує у ряді випадків менше комп’ютерної пам’яті, ніж при реалізації прямих методів. У такому випадку для розв’язування СЛАР доцільно використовувати швидкозбіжні ітераційні методи.

Значного ефекту від використання ітераційних методів можна отримати у задачах з розрідженими матрицями, оскільки при використанні прямих методів відбувається збільшення кількості ненульових елементів, а відтак зростають вимоги до пам’яті та об’єму обчислень. Особливо відчутною може бути різниця у випадку паралельних обчислень, оскільки тоді час розв’язування задачі можна скоротити ще більше.

Ідея типового ітераційного методу полягає у виборі початкового наближення *х*1 до розв’язку *х* і побудові послідовності *х*2, *х*3,…, такої, що . Теоретично при використанні ітераційного методу необхідно виконати нескінченну кількість арифметичних операцій, щоб отримати *х*, але на практиці призупинення відбувається тоді, коли, на наш погляд, чергове наближення достатньо близьке до *х*. Такі методи є досить привабливими з точки зору вимог до машинної пам’яті, оскільки їх реалізація у загальному випадку вимагає зберігати лише матрицю *А*, вектори *b*, *х*(*і*) і ще, можливо, один або кілька векторів. До того ж, варіювання величини бажаної точності розв’язку дає змогу змінювати загальний час розв’язування системи.

Одна за найбільш вагомих частин дослідження випала на аналіз та обирання методу розв’язання системи рівнянь. На вимогу попередніх викладок, умови на обирання методу були наступні:

* Ітеративний
* Можливість розпаралелювання
* Можливість пошуку наближеного розв’язку з деякою точністю
* Збіжність методу
* Схильність до розріджених матриць

Серед розглянутих, були наступні методи:

* Метод Якобі
* Метод Гауса-Зейделя
* Метод релаксацій
* Метод Річардсона (явний Чебишевський метод)

Для досліджень були обрані 2: Метод Річардсона та Метод релаксацій (верхніх). Метод Якобі був відхилений як спрощення Річардсона, а Гауса-Зейделя є модифікацією Якобі, та все ж не є кращим за Річардсона. Метод Релаксації має менше аналогів та спрощень (хоча при особливому додатковому параметрі можна перейти до методу Зейделя), та має 2 види верхні та нижні в залежності від обраного додаткового параметра. Цей метод є представником стаціонарних одно крокових ітераційних методів лінійної алгебри. Розглядається саме 2 методи з причин різного підходу методів до розпаралелювання в залежності від моделі методу, на це впливає саме використання точок даної ітерації при розрахунку тієї ж самої ітерації.

Основні формули для обох методів виглядають наступним чином:

\frac{(D+\omega L)(x^{(s+1)}-x^{(s)})}{\omega}+Ax^{(s)}=b

Метод верхньої релаксації.

Тут: x^{(s)} - наближення, отримане на ітерації з номером s, x^{(s+1)} - наступне наближення, \omega - параметр метода, A, L, D – матриці образовані наступним чином – A – повна матриця (симетрична, додатньо визначена, квадратна), D – діагональна матриця від А. L – нижня трикутна матриця від А, не включаючи діагональну, b – вектор (матриця) правих частин.

Необхідною умовою збіжності метода з будь якого початкового наближення до точного розв’язку задачі є виконання умови:

\omega \in (0,2)

Якщо А відповідає наданим вище умовам це є і достатньою умовою. При різних значеннях параметру говорять о нижніх чи верхніх релаксаціях. При проміжному значенні 1 отримаємо Метод Зейделя.

У загальному випадку немає аналітичної формули для обчислення оптимального параметру методу. Наприклад, при розв’язанні системи рівнянь, отриманих при апроксимації диференційних рівнянь в частинних похідних, можна використати евристичну оцінку вигляду:

\omega_{opt}\approx 2-O(h)

Де h – крок сітки дискретизації.

В деяких випадках можна оцінити більш точно оптимальний параметр:

\omega_{opt}=\frac {2}{1+\sqrt{1-\rho^2(D^{-1}(R+L))}

Де \rho - спектральний радіус матриці а R, D, L верхня, діагональна та нижня матриця від похідної А.

Для методу Річардсона достатньо лише додатної визначеності.

http://old.math.tsu.ru/EEResources/cm/text/5.files/image407.gif

Де τ – оптимальний чебишевський параметр, окремий для кожної ітерації. Записаний наступним чином для найменшої похибки (доведено), і детермінується наступним чином:

http://old.math.tsu.ru/EEResources/cm/text/5.files/image412.gif

http://old.math.tsu.ru/EEResources/cm/text/5.files/image413.gif

http://old.math.tsu.ru/EEResources/cm/text/5.files/image414.gif

Такий метод з таким набором параметрів називається явним ітераційним методом з чебишевським набором параметрів, при використанні лише нульового елементу з вектора отримаємо метод Якобі.

Для програмування цього методу потрібно знати кількість ітерацій, тож за вхідними даними потрібно зробити оцінку, яку можна виписати наступною формулою:

http://old.math.tsu.ru/EEResources/cm/text/5.files/image417.gif

http://old.math.tsu.ru/EEResources/cm/text/5.files/image441.gif

Де ε – бажана похибка.

Для найбільш невдалого випадку, коли η дуже мале, отримаємо наступну нерівність:

http://old.math.tsu.ru/EEResources/cm/text/5.files/image443.gif

Одна з важливих задач тут правильне впорядкування параметрів, бо саме від них залежать збіжність методу. Розв’язок цієї задачі був запропонований Самарським і є достатньо громіздким, опустивши їх можемо виписати конкретну формулу

$ \tau_j = \left[{\frac{{L + l}}{2} + \frac{{L - l}}{2} \cos \frac{{{\pi}(2j - 1)}}{{2i}}}\right]^{- 1}, j = 1, 2, \ldots , i.  $

Де L та l найбільше та найменше власне значення початкової матриці. Виходячи з таких обчислень отримаємо нову оцінку кількості кроків:

$  i  \approx  \left[{\frac{{\sqrt {\mu} }}{2} \ln \cdot {\varepsilon}^{- 1}}\right] + 1.  $

Програмна реалізація використання повного оптимального набору має свої додаткові аспекти, та описана у додатках.

Таким чином обидва методи можуть бути використані, як досліджувані. Вони задовольняють нашим прописаним умовам, мають передумови для паралелізма, тож можемо перейти до реалізації їх послідовних та паралельних алгоритмів.

Послідовні алгоритми повністю відповідають формулам алгоритмів та ніяк не відрізняються від них, більшої уваги слід надати паралельним алгоритмам. Перед описом можливих алгоритмів та дослідження на їх оптимальність під різні умови, потрібно надати аналізу існуючих можливостей паралельної технології CUDA.

Проводити розпаралелювання можна двома способами, за допомогою написання своїх функцій для графічних пристроїв, чи за допомогою програмних паралельних інтерфейсів. Розглянемо ці підходи.

**MPI**

Message Passing Interface (MPI, інтерфейс передачі повідомлень) - програмний інтерфейс(API) для передачі інформації, який дозволяє обмінюватися повідомленнями між процесами, що виконують одну задачу. Розроблено Вільямом Гроуппом, Евін Ласко та іншими.

MPI є найбільш поширеним стандартом інтерфейсу обміну даними в паралельному програмуванні, існують його реалізації для великого числа комп'ютерних платформ. Використовується при розробці програм для кластерів і суперкомп'ютерів. Основним засобом комунікації між процесами в MPI є передача повідомлень один одному. У стандарті MPI описаний інтерфейс передачі повідомлень, який повинен підтримуватися як на платформі, так і в додатках користувача.

У першу чергу MPI орієнтований на системи з розподіленою пам'яттю, тобто коли витрати на передачу даних великі, у той час як OpenMP орієнтований на системи з загальною пам'яттю (багатоядерні із загальним кешем). Обидві технології можуть використовуватися спільно, щоб оптимально використовувати в кластері багатоядерні системи.

**OpenMP**

OpenMP(Open Multi-Processing) — це набір директив компілятора, бібліотечних процедур та змінних середовища, які призначені для програмування багатонитевих програм на багатопроцесорних системах із спільною пам'яттю на мовах C, C++ та Fortran.

Розробку специфікації OpenMP ведуть кілька великих виробників обчислювальної техніки та програмного забезпечення, робота яких регулюється некомерційною організацією, названою OpenMP Architecture Review Board (ARB). Специфікації для мов Fortran і C/C++ з'явилися відповідно в жовтні 1997 року і жовтні 1998 року.

OpenMP можна розглядати як високорівневу надбудову над Pthreads (або аналогічними бібліотеками нитей). POSIX-інтерфейс для організації нитей Pthreads підтримується широко (практично на всіх UNIX-системах).

OpenMP реалізує паралельні обчислення за допомогою багатонитевості, в якій «головна» (master) нить створює набір підлеглих (slave) нитей і завдання розподіляється між ними. Передбачається, що ниті виконуються паралельно на машині з декількома процесорами (кількість процесорів не обов'язково має бути більше або дорівнювати кількості нитей).

Завдання, що виконуються нитями паралельно, так само як і дані, необхідні для виконання цих завдань, описуються за допомогою спеціальних директив препроцесора відповідної мови.

Кількість створюваних нитей може регулюватися як самою програмою за допомогою виклику бібліотечних процедур, так і ззовні, за допомогою змінних оточення.

OpenMP має такі переваги:

1. За рахунок ідеї «інкрементального розпаралелювання» OpenMP ідеально підходить для розробників, що прагнуть швидко розпаралелювати свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.
2. При цьому, OpenMP — досить гнучкий механізм, що надає розробникові великі можливості контролю над поведінкою паралельного застосунку.
3. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана як послідовна програма, тобто немає необхідності підтримувати послідовну та паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких приведений в специфікаціях.
4. Одним з переваг OpenMP його розробники вважають підтримку так званих «orphan» (відірваних) директив, тобто директиви синхронізації і розподілу роботи можуть не входити безпосередньо в лексичний контекст паралельної області.

**CUDA**

**CUDA**(**Compute Unified Device Architecture**) - паралельна обчислювальна платформа і модель програмування, створена NVIDIA і виконувана на графічних процесорах (GPU), які вони виробляють. CUDA дає розробникам програм прямий доступ до великої кількості віртуальних команд і пам'яті на паралельних обчислювальних елементах у CUDA GPU.

Графічні процесори, які використовують CUDA, можуть використовуватися не тільки для обробки графіки; цей підхід відомий як GPGPU. У порівнянні з традиційним підходом до організації обчислень загального призначення за допомогою можливостей графічних API, у архітектури CUDA відзначають наступні переваги в цій області:

* CUDA архітектура
* Використання GPU обчислень для звичайних цілей
* Збереження продуктивності
* CUDA C/C++ мова
* Заснований на стандартизованому C/C++
* Малий набір доповнень для включення можливостей гетерогенного програмування
* Чітке API для управління пристроями, пам’яттю.

Найбільшу увагу наддамо саме мові та її можливостям. У подальших викладках будемо використовувати наступні поняття:

* Host (хост) – CPU та його пам’ять
* Device (пристрій) – GPU та його пам’ять

Гетерогенне програмування складається з двох частин коду котрі записані разом, але мають різний спосіб дії. Окремими функціями пишеться код для пристрою і хоста, після початку дії хоста в деякий момент часу ми викликаємо функції пристрою.

Відмінностями від звичайного програмування мають наступний характер. Функції, змінні пристрою мають декілька специфікаторів які дозволяють бачити функцію чи змінні на хості та пристрою чи лише на пристрої. Викликання функцій пристрою дає нам додаткові змінні – кількість блоків та кількість потоків (ниток) у кожному блоці. Щоб передати аргументи потрібно скопіювати їх з пам’яті хоста у пам’ять пристрою та передати посилання на них у аргументах.

Дуже особливими означеннями тут є блоки та потоки. Кожен блок може мати декілька потоків, кожен блок обчислюється паралельно з іншими блоками та не мають основних можливостей до синхронізації (існують способи синхронізації блоків як динамічне програмування, та вони настільки збільшують час виконання, що не має сенсу їх використовувати). Потоки навпаки мають вбудовані способи синхронізації за необхідністю.

Набір паралельних блоків називається сіткою, основне питання для паралельної задачі буде коректна індексація. Для цього ми маємо індекс блоку, індекс потоку та розмір блоку.

Особливість потоків в кожному блоці полягає у тому, що вони мають можливість синхронізуватися, тобто кожен потік може дочекатися іншого потоку. Придивимось уважніше до індексації при розгляданні наших модельних задач.

За особливістю GPU ми маємо велику кількість процесорів, проте не нескінчену. Для кожного пристрою кількість оптимальних одночасних операцій не перевищує кількості процесорів, проте ламати систему заради оптимальної кількості потоків чи блоків не є суттєвим, бо GPU має змогу самостійно відредагувати поставлену задачу.

Метод Річардсона шукає наступне наближення за допомогою попереднього наближення, оптимальних параметрів, правої частини, основної матриці. Тож для розрахунку наступної ітерації ми можемо паралельно обрахувати одразу всі внутрішні точки нашої сітки. З урахуванням цього запропонуємо наступний паралельний алгоритм:

Розіб’ємо область внутрішніх та приграничних вузлів на p підобластей (за кількістю ПП – процесор з пам’яттю) прямими перпендикулярними до тієї вісі, за якою більше вузлів розбиття (у нашому випадку сітка однорідна, і не має переваги та чи інша вісь). Для визначеності у подальшому будемо вважати, що розбиття проводимо прямими, які паралельні до 1 вісі. Розбиття проводимо таким чином, щоб час на ітерацію у кожній підобласті був приблизно однаковий. Нехай підобласті нумеруються знизу до гори. Розширимо всі підобласті окрім граничних на один додатковий рядок до гори та до низу, граничні розширимо на один рядок у сторону сітки. Таким чином додаткові сіткові прямі, відіграють роль тих сіткових прямих, які знаходяться у сусідніх процесорах. Таким чином можна встановити зв’язок перед кожною ітерацією у ПП.

При такому розбитті можна представити наступний паралельний процес:

1. Задаємо початкове наближення у кожну підобласть. Вводимо в кожний ПП величину бажаної похибки, який детермінує закінчення ітеративного процесу. Обчислюємо і зберігаємо в усіх ПП вектор оптимальних параметрів.
2. У кожному ПП у вузлах сітки одночасно й незалежно знаходимо наступну ітерацію. На цьому ж кроці перевіряємо умову закінчення процесу. Якщо воно не виконується то переходимо на наступний крок.
3. Послідовно встановлюємо два рази зв’язок між ПП таким чином, щоб у результаті цього ПП, який містить у собі сусідні підобласті, з’єдналися кожен раз попарно між собою. Після кожного такту встановлюємо зв’язок між ПП щоб вони обмінялися додатковими значеннями.

Для визначення коефіцієнтів, що характеризують явний чебишевский метод з паралельною організацією обчислень, достатньо розглянути одну ітерацію. Час для реалізації однієї ітерації за основною формулою дорівнює

Де М – кількість арифметичних операцій, необхідних для обчислення наступної ітерації для будь якого вузла. Після цього машині потрібно лише 2 рази послідовно встановити зв’язок між сусідніми ПП. Після встановлення зв’язку ПП обмінюються 2Q1 словами. Отримаємо наступну формулу:

А також:

З чого виходить, що з приростом р ефективність розпаралелювання буде зменшуватися, якщо інші величини, що входять до формули, залишаються незмінними. Також бачимо, що при зростанні М та Q, тобто при збільшенні навантажень на ПП за числом арифметичних операцій збільшується й ефективність. Також отримаємо приріст у швидкості при збільшенні кроків сітки.

Перейдемо до методу верхніх релаксацій. Для розв’язання задачі даним методом при умові звичайного впорядкування невідомих задають початкове наближення у вузлах сітки та величину похибки, яка детермінує закінчення ітераційного процесу. Важливо на кожному вузлі враховувати поточні розрахунки у сусідніх вузлах, які вже були розраховані для даної ітерації. Цей момент змінює нашу схему кардинально наступним чином. Ми вже не в змозі розпаралелити одразу всю сітку. На це ще накладається проблема синхронізації блоків, мі не можем розрахувати першу діагональ, потім послідовно другу і першу, и так далі до закінчення ітераційного процесу. Це питання пропонується розв’язати наступним чином.

Аби надати максимальну паралелізацію такому доволі послідовному алгоритму розв’язувати вузли діагоналями, які проходять наступним чином:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 6 |
| 3 | 4 | 5 | 6 | 7 |
| 4 | 5 | 6 | 7 | 8 |
| 5 | 6 | 7 | 8 | 9 |

Так позначаються лінії процесу. Номер у клітинці позначає належність вузла до будь якої з ліній. Позначимо кожну діагональ як окремий блок. Тож для коректних розрахунків ми маємо розрахувати наступну ітерацію для першої діагоналі. Другий крок першої ітерації розрахує перше наближення для другої діагоналі використовуючи початкове наближення більших діагоналей (з більшим номером) та першу ітерацію менших діагоналей. Проблема синхронізації постає саме тут, на другому кроці першої ітерації ми не в змозі обчислити перший крок другої ітерації для першої лінії бо ми не можемо дочекатися повного обчислення другого блоку. Ми зможемо обрахувати перший крок другої ітерації під час третього кроку першої ітерації. Таким чином отримаємо хвильовий ефект, що надає нам дуже гарну оптимізацію. Хвилі такого процесу будуть виглядати наступним чином:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5\* | 4 | 4\* | 3 | 3\* |
| 4 | 4\* | 3 | 3\* | 2 |
| 4\* | 3 | 3\* | 2 | 2\* |
| 3 | 3\* | 2 | 2\* | 1 |
| 3\* | 2 | 2\* | 1 | 1\* |

Тут позначені ітерації які будуть паралельно обчислюватися, зірочкою позначені ті вузли що будуть обраховуватися на деякому кроці, відповідний номер позначає ітерацію які вони будуть обчислювати в тому чи іншому вузлі. Щоб досягти паралельного обчислення потрібно опрацювати перші декілька кроків. Таким алгоритмом ми скоротимо витрати на синхронізацію блоків, та лишимося можливості оперувати одразу всіма вузлами. Це питання вирішимо наступним чином. Послідовно викличемо алгоритм для не парних блоків та для парних. Це трохи зменшить час виконання.

Наддамо метричні характеристики такого алгоритму, для роботи під час навантаження по всіх вузлах:

Час виконання такого алгоритму можна зменшити наступним чином – зменшити кількість арифметичних операцій, чи підвищуючи кількість процесорів.

Бачимо, що отриманий коефіцієнт ефективності лінійно пропорційний до кількості вузлів, чим більше отримаємо сітку для обчислень тим більш ефективніший буде паралельний алгоритм.

Вочевидь, що описана організація обчислень для розв’язання сіткових рівнянь методом верхньої релаксації чи Річардсона може бути ефективно реалізована на MIMD – машині зі всіма можливими структурами між процесорних зв’язків.

//Проведення роботи алгоритму та опрацювання алгоритмів, отримані дослідницькі прискорення та ефективність.

1. Підготовка та налаштування глобальних змінних
   1. Кількість вузлів
   2. Допустима похибка
   3. Початок часу виконання
   4. Налаштування різницевої матриці
   5. Права частина на сітці
   6. Перше наближення
2. Розрахунок додаткових змінних
   1. Розрахування спектру матриці
   2. Масив оптимального чебишевського набору кроків
   3. Знаходження мінімальної потрібної кількості операцій
3. Обробка головного циклу
   1. Проведення ітерацій по наближенню за формулами
4. Вивід та аналіз результатів

Аналогічний алгоритм використовується й для обробки програми на CUDA. Лише функція головного циклу оптимізується на GPU.

Отримані результати свідчать, що на маленьких системах використання GPU не є раціональною ідеєю, бо саме використання такого пристрою затрачує деякий час, але при зростанні кількості операцій, складності системи оптимізація проявляється і оптимізований алгоритм виконується в декілька разів швидше.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Розмір сітки | 5 | 10 | 25 | 50 |
| Ітерації | 50 | 220 | 1375 | 5500 |
| C++ (сек.) | 0.012 | 0.05 | 5.88 | 95 |
| CUDA (сек.) | 2.27 | 2.28 | 4.77 | 40.5 |

Таким чином вже на цій стадії роботи можна відмітити пропорційність росту часу в одному та іншому випадках і сказати, що оптимізація при використанні гібридних архітектур помітна, і саме за такими обчисленнями може стати майбутнє обчислювальної математики.