МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 39.00.00.000 ПЗ

Група ІПм-23-1

Димінський Денис

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Димінський Денис Русланович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі та методи адаптивних фреймворків управління гетерогенними багатоядерними кластерами

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Димінський Д.Р.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Шекета Василь Іванович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Димінському Денису Руслановичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Моделі та методи адаптивних фреймворків управління гетерогенними багатоядерними кластерами ”

керівник проекту (роботи) Шекета Василь Іванович, д.т.н., професор

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних технологій управління кластерами

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Дослідження особливостей управління гетерогенними багатоядерними системами
2. Модель паралельного програмування для великомасштабної обробки MapReduce
3. Моделі та методи адаптивної структури програмування для гетерогенних кластерів
4. Методологія розподілу навантаження з урахуванням можливостей для гетерогенних кластерів

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Приклад гетерогенної багатоядерної архітектури (рис. 1.1)
2. Архітектура системи Cell Broadband Engine (рис. 1.2)
3. Архітектура графічного процесора NVIDIA з підтримкою CUDA (рис. 1.3)
4. Приклад операцій MapReduce у програмі Word Count (рис. 1.4)
5. Високорівнева системна архітектура симетричних і асиметричних кластерів (рис. 2.1)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Дослідження особливостей управління гетерогенними багатоядерними системами	15.10.2024	виконано
4	Модель паралельного програмування для великомасштабної обробки MapReduce	08.11.2024	виконано
5	Моделі та методи адаптивної структури програмування для гетерогенних кластерів	20.11.2024	виконано
6	Методологія розподілу навантаження з урахуванням можливостей для гетерогенних кластерів	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 77 с., 15 рис., 4 табл., 53 джерела.

Тема: Моделі та методи адаптивних фреймворків управління гетерогенними багатоядерними кластерами

Об'єкт дослідження: процес управління та розподілу навантаження у гетерогенних багатоядерних кластерах.

Мета роботи: дослідити адаптивні моделі і методи управління гетерогенними багатоядерними кластерами на основі адаптованої структури програмування, яка дозволить оптимізувати розподіл навантаження та забезпечити ефективне використання ресурсів системи.

Предмет дослідження: моделі, методи та алгоритми управління та розподілу навантаження в умовах асиметричної архітектури гетерогенних багатоядерних кластерів.

Результати дослідження

В роботі розроблено алгоритм динамічного планування робочого навантаження, що враховує специфічні можливості різних обчислювальних прискорювачів і забезпечує високу масштабованість системи.

Висновок

Запропоновано розширений фреймворк на основі MapReduce, адаптований для програмування гетерогенних кластерів, що забезпечує ефективний розподіл навантаження між обчислювальними вузлами різної продуктивності.

ГЕТЕРОГЕННИЙ КЛАСТЕР, БАГАТОЯДЕРНА СИСТЕМА, УПРАВЛІННЯ РЕСУРСАМИ, АДАПТИВНИЙ ФРЕЙМВОРК, MAPREDUCE, РОЗПОДІЛ НАВАНТАЖЕННЯ, АСИМЕТРИЧНА АРХІТЕКТУРА, ДИНАМІЧНЕ МАСШТАБУВАННЯ

ABSTRACT

Master Thesis: 77 pp., 15 fig., 4 tab., 53 sources.

Thesis Subject: Models and methods of adaptive frameworks for managing heterogeneous multicore clusters

The subject of research: the process of management and distribution of loads in heterogeneous multi-core clusters.

The purpose of the work: to investigate adaptive models and methods of managing heterogeneous multicore clusters based on adapted programming structures that allow optimizing load distribution and ensuring efficient use of system resources.

Subject of research: models, methods and algorithms of management and distribution of loads in conditions of asymmetric design of heterogeneous multi-core clusters.

Research results

The work developed an algorithm for dynamic workload planning, which takes into account the specific capabilities of various computing accelerators and ensures high scalability of the system.

Conclusion

An extended framework based on MapReduce is proposed, adapted for programming heterogeneous clusters, which provides efficient distribution of loads between computing nodes of different performance.

HETEROGENEOUS CLUSTER, MULTI-CORE SYSTEM, RESOURCE MANAGEMENT, ADAPTIVE FRAMEWORK, MAP-REDUCE, LOAD BALANCE, ASYMMETRIC ARCHITECTURE, DYNAMIC SCALING

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AMP - Asymmetric Multicore Processors

APU - Accelerated Processing Unit

CUDA - Compute Unified Device Architecture

DMA - Direct Memory Access

DRAM - Dynamic Random-Access Memory

DVFS - Dynamic Voltage and Frequency Scaling

EDF - Earliest Deadline First

EIB - Element Interconnect Bus

FCFS - First Come First Served

FOP - Feature-Oriented Programming

FPGA - Field-Programmable Gate Array

GPGPU - General-Purpose computing on Graphics Processing Units

GPP - General-Purpose Processors

GPU - Graphics Processing Unit

HPC - High Performance Computing

ISA - Instruction Set Architecture

MPI - Message Passing Interface

PCI - Peripheral Component Interconnect

RISC - Reduced Instruction Set Computing

RTL - Register Transfer Level

SIMD - Single-Instruction Multiple-Data

SIMT - Single-Instruction Multiple-Threads

SMT - Simultaneous Multi-Threading

SPE - Synergistic Processing Elements

SPMD - Single-Program Multiple Data

TBB - Threading Building Blocks

TCO - Total Cost of Ownership

ВСТУП

Актуальність теми.

Гетерогенні багатоядерні кластери стали важливою частиною сучасних обчислювальних систем, адже вони здатні забезпечити високу продуктивність та масштабованість, необхідні для обробки великих обсягів даних у наукових дослідженнях, штучному інтелекті, машинному навчанні, обчислювальній біології та інших сферах. Проте різнорідність компонентів цих кластерів, зокрема поєднання процесорів різної архітектури, графічних прискорювачів (GPGPU), і спеціалізованих обчислювальних елементів, створює складності в управлінні, особливо у контексті балансування навантаження та оптимального використання ресурсів.

Водночас традиційні методи управління та програмування для гомогенних кластерів не підходять для гетерогенних систем через специфіку архітектури та асиметрію обчислювальних потужностей вузлів. У відповідь на ці виклики, останні дослідження фокусуються на розробці адаптивних моделей та фреймворків, здатних автоматично враховувати та ефективно розподіляти навантаження між різними обчислювальними елементами кластеру. Це, у свою чергу, дозволяє суттєво підвищити продуктивність та зменшити затрати на виконання складних обчислень.

Додатково, наявність нових технологій, таких як розширені платформи для паралельного програмування на основі MapReduce, відкриває можливості для створення більш гнучких та ефективних підходів до управління гетерогенними ресурсами. Актуальність даного дослідження обумовлена необхідністю розробки таких адаптивних фреймворків та методологій, які б дозволили використовувати всі переваги гетерогенних кластерів, знижуючи при цьому витрати на їх обслуговування та мінімізуючи вимоги до компетенцій програмістів у налаштуванні та керуванні системою.

Отже, дослідження в цій сфері має велике практичне значення, оскільки розвиток таких адаптивних систем може значно розширити

можливості їх застосування, підвищити ефективність обчислювальних ресурсів та скоротити час розгортання нових програмних рішень у середовищах з різнорідною архітектурою.

Мета магістерської роботи - дослідити адаптивні моделі і методи управління гетерогенними багатоядерними кластерами на основі адаптованої структури програмування, яка дозволить оптимізувати розподіл навантаження та забезпечити ефективне використання ресурсів системи.

Об'єкт дослідження - процес управління та розподілу навантаження у гетерогенних багатоядерних кластерах.

Предмет дослідження - моделі, методи та алгоритми управління та розподілу навантаження в умовах асиметричної архітектури гетерогенних багатоядерних кластерів.

Задачі дослідження:

- Провести аналіз особливостей управління гетерогенними багатоядерними системами.
- Дослідити існуючі підходи до програмування таких систем, зокрема розглянути адаптивні фреймворки.
- Розробити моделі та методи адаптивного управління ресурсами на основі розширеного підходу MapReduce.
- Запропонувати та реалізувати методологію розподілу навантаження з урахуванням специфіки гетерогенних кластерів.
- Оцінити ефективність запропонованих моделей та методів у симетричних та асиметричних конфігураціях ресурсів.

Методи дослідження:

- Аналіз існуючих підходів та моделей програмування гетерогенних систем.
- Імітаційне моделювання для дослідження взаємодії гетерогенних кластерів.
- Експериментальний аналіз для оцінки ефективності запропонованих алгоритмів та методів у порівнянні з традиційними підходами.

Наукова новизна отриманих результатів

Запропоновано розширений фреймворк на основі MapReduce, адаптований для програмування гетерогенних кластерів, що забезпечує ефективний розподіл навантаження між обчислювальними вузлами різної продуктивності.

Практичне значення результатів

Розроблені моделі та методи можуть бути застосовані для оптимізації роботи гетерогенних обчислювальних кластерів, що використовуються у наукових дослідженнях, обробці великих обсягів даних та ресурсномістких обчисленнях. Це забезпечить підвищення продуктивності системи, зменшення часу обробки та більш ефективне використання доступних ресурсів, що є важливим для великих наукових та інженерних обчислень.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 77 сторінок, і містить 15 рисунків, 4 таблиці, список використаних джерел із 53 найменувань.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ОСОБЛИВОСТЕЙ УПРАВЛІННЯ ГЕТЕРОГЕННИМИ БАГАТОЯДЕРНИМИ КЛАСТЕРАМИ ТА СИСТЕМАМИ

1.1. Проблеми у використанні гетерогенної багатоядерної системи

Великомасштабні обчислювальні системи, що містять тисячі ядер, зазвичай служать робочими конячками для наукових відкриттів і діяльності підприємств. Зі збільшенням розміру середнього обчислювального кластера та вимогами до оркестрування та керування великими вхідними та вихідними наборами даних, використання обчислювальних прискорювачів, співпроцесорів, прискорених процесорів (APU) та асиметричних багатоядерних процесорів (AMP) із сумішшю загальних Цільові процесори (GPP) і спеціалізовані ядра стали провідною парадигмою для економічно ефективних обчислювальних установок високого класу. Постачальники процесорів незабаром запропонують однокристальні багатопроцесорні процесори з підтримкою Teraflops, інтегруючи багато простих ядер із шляхами даних SIMD (Single-Instruction Multiple-Data) або SIMT (Single-Instruction Multiple-Threads). Стільникові процесори, AMD Fusion APU, програмовані графічні процесори (GPU) від NVIDIA, Larrabee і Single-chip Cloud Computer є представниками цього класу. процесори зі значним ринковим інтересом і продемонстрованим потенціалом [8]. Терафлопсові процесори прискорюють обчислення завдяки своїй конструкції, залишаючись у межах розумного бюджету. Таким чином, вони вважаються одними з найбільш перспективних обчислювальних робочих потужностей [5] для високопродуктивних обчислень (HPC). Використання гетерогенних систем, які поєднують дрібнозернистий паралелізм з грубозернистим паралелізмом, використовуючи десять або тисячі процесорів, допомогло підвищити продуктивність додатків HPC. Як результат, ці різнорідні архітектури стають все більш популярними для споживчих додатків, включаючи комп’ютерні

ігри, мультимедійні програми, настільні комп'ютери, HPC, робототехніку та вбудовані програми.

Використання різнорідних багатоядерних прискорювачів і співпроцесорів у великомасштабному розподіленому середовищі відрізняється від використання традиційних багатоядерних процесорів, які складаються з гомогенних ядер з однаковою архітектурою набору інструкцій (ISA). Незважаючи на те, що ці гетерогенні багатоядерні процесори пропонують кращу обчислювальну продуктивність і енергоефективність порівняно з традиційними багатоядерними процесорами, їх використання у великомасштабному кластері створює додаткові проблеми при проектуванні та виконанні. Ця робота пропонує адаптивну структуру для керування гетерогенними багатоядерними кластерами та представляє нові рішення проблем програмування та управління ресурсами, які пов'язані з використанням гетерогенних багатоядерних процесорів у великих обчислювальних кластерах.

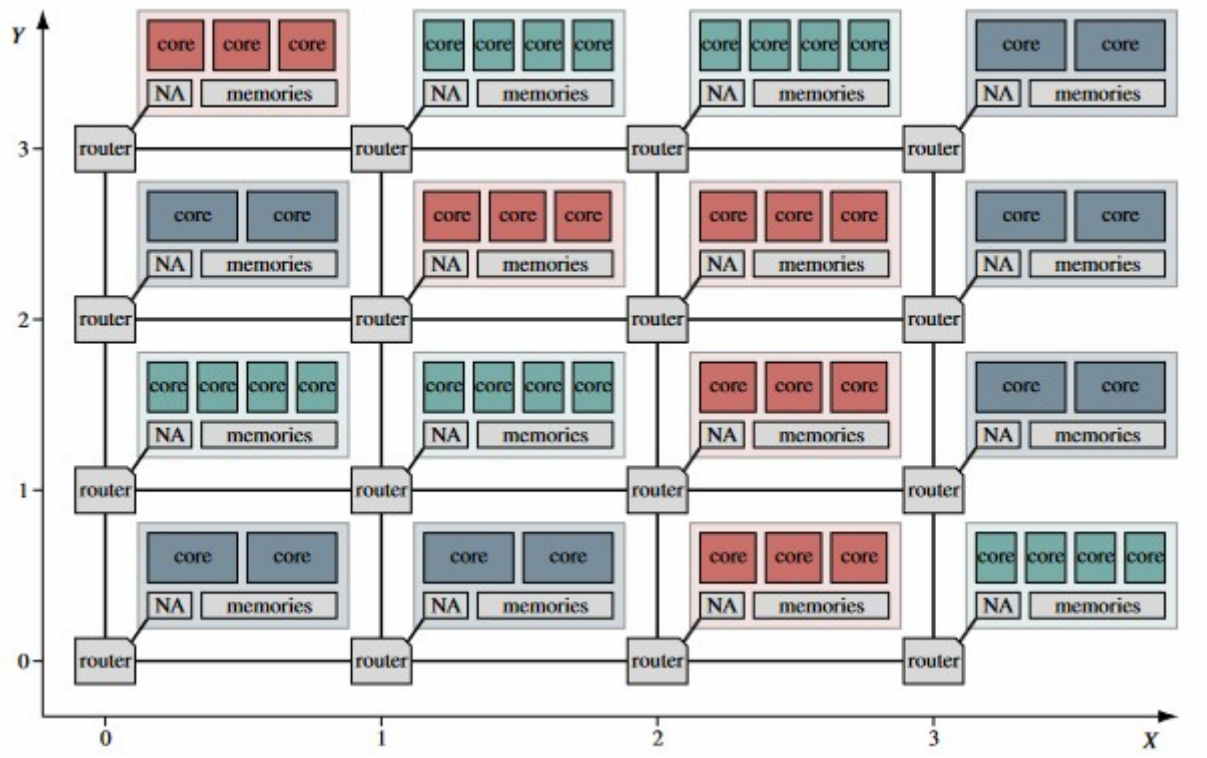


Рис. 1.1. Приклад гетерогенної багатоядерної архітектури

На рисунку 1.1 показано приклад гетерогенної багатоядерної архітектури з плитковою організацією, що складається з 16 плиток з мережею на кристалі (NoC) з топологією 4 × 4. Площина XY визначає розташування кожного маршрутизатора NoC та його локальної плитки. Кожна плитка містить набір ядер, набір модулів пам'яті та мережевий адаптер (NA). Різні типи ядер розрізняються за кольором.

Нова зміна парадигми від однорідних багатоядерних до гетерогенних багатоядерних процесорів у вузлі, а також у великомасштабному кластері створює багато проблем щодо програмування, оркестровки вводу-виводу та управління ресурсами. Багатоядерні архітектури забезпечують паралелізм, однак, щоб скористатися перевагами наявних апаратних ресурсів, необхідно розробити нові парадигми програмування, інтелектуальні методи розподілу даних і ефективні схеми управління ресурсами, щоб використовувати можливості гетерогенних ядер у повній мірі. Далі ми надаємо огляд проблем, пов’язаних із використанням гетерогенних систем, які ми намагаємося розглянути в цій роботі.

1.1.1. Програмування гетерогенних багатоядерних систем

Програмування асиметричних систем, навіть на рівні однієї мікросхеми, вивчено не дуже добре, і є темою багатьох досліджень. Навпаки, моделі програмування для симетричних кластерів, наприклад, MPI [33] і SHMEM, є зрілими, зручними для користувача та звільняють програмістів додатків від деталей низькорівневого управління системою. Програмування гетерогенних систем на основі прискорювачів вимагає роботи з декількома ISA та декількома цілями компіляції . Відсутність простих у використанні та ефективних моделей програмування та інструментів для моделювання неоднорідності системи є однією з ключових проблем у програмуванні неоднорідних кластерів. Крім того, відсутність складних налагоджувачів і підтримки профілювання для кращого розуміння поведінки додатків також

створює перешкоду для використання різнорідних ресурсів у розробці основних додатків.

Хоча приховування архітектурної асиметрії та масштабу системи від моделі паралельного програмування є бажаними властивостями [35], їх складно реалізувати в гетерогенній системі, де використання налаштування та обчислювальної щільності кожного обчислювального блоку є розглядом першого порядку. Відображення високорівневої моделі паралельного програмування на прискорювачі, приховуючи деталі апаратного забезпечення прискорювача, є надзвичайно складним і навіть небажаним, якщо підвищення рівня абстракції призводить до втрати продуктивності. Сучасні підходи до програмування гетерогенних кластерів на основі прискорювача є або спеціальними, або специфічними для інсталяції [21], що створює кілька проблем при застосуванні до загальних установок. Подальші проблеми виникають через неоднорідність, яка проявляється як у моделі програмування, так і в управлінні ресурсами. Реалізація доступної моделі програмування в системах з багатьма ядрами, багатьма процесорами, декількома ISA і декількома цілями компіляції вимагає радикальних модифікацій всього стеку програмного забезпечення. Відповідні моделі програмування, які адаптуються до різних можливостей компонентів типу прискорювача, ще не розроблені. Цей дефіцит змушує авторів додатків, які хочуть використовувати прискорювачі на кластерах, для мікроуправління ресурсами, що стосуються встановлення. Використання специфічних для архітектури рішень є вкрай небажаним, оскільки це ставить під загрозу продуктивність, портативність і стабільність залучених систем і програм.

Вплив альтернативного розподілу робочого навантаження між процесорами загального призначення та прискорювачами також недостатньо зрозумілий. Прискорювачі зазвичай мають набагато вищу щільність обчислень і необроблену продуктивність, ніж звичайні процесори, тому поєднання прискорювачів із звичайними процесорами може викликати дисбаланс між ними. Прискорювачі також зазвичай мають обмежені

можливості для керування зовнішніми системними ресурсами, такими як комунікаційні пристрої та пристрої вводу/виводу, що вимагає підтримки з боку процесорів загального призначення та особливої уваги при розробці програмного забезпечення для керування введенням/виведенням. Щоб забезпечити загальну високу ефективність , керування введенням/виведенням у системах на основі прискорювача має ретельно організовувати передачу даних і розподіл роботи між різнорідними компонентами.

1.1.2. Попередня вибірка даних у гетерогенних багатоядерних системах

Сучасні багатоядерні прискорювачі дотримуються філософії дизайну, яка віддає перевагу багатьом простим ядрам із широкими шляхами передачі даних. Це призводить до більш енергоефективних конструкцій, одночасно зберігаючи експоненціальне підвищення продуктивності. На жаль, цей підхід позбавляє процесори можливостей ефективного виконання коду, що вимагає інтенсивного керування, наприклад, предикторів розгалужень, множинних проблем, динамічного планування інструкцій і спекуляцій. Таким чином, прискорювачі не можуть ефективно підтримувати інтенсивний контрольний код, такий як більшість служб операційної системи. Крім того, хоча прискорювачі мають високу обчислювальну щільність, вони також мають обмежений простір для зберігання, доступний безпосередньо на чіпі або через виділене посилання. Таким чином, будь-яка модель програмування, яка інтегрує прискорювачі, повинна реалізовувати ефективну передачу даних до та з внутрішньої пам’яті прискорювачів. Масштабування неоднорідності ресурсу за межі одного вузла вимагає синтезу паралельного виконання та методів передачі даних. Ці методи повинні збалансувати швидкість обчислення прискорювачів з обмеженою пропускною здатністю передачі даних поза чіпом і вузлом. Досягнення цієї мети при збереженні властивостей, які допомагають підвищити продуктивність програмістів,

таких як агностик щодо кількості та архітектури процесорів, є серйозною проблемою.

Асиметричні багатоядерні процесори та співпроцесори в основному використовуються для інтенсивних обчислювальних навантажень. Однак сучасні додатки часто включають етапи інтенсивного введення-виведення, які обробляють великі дані в потоках, наприклад, стиснення/декомпресію, шифрування/дешифрування, обробку відео та наукові обчислення на основі робочого процесу. Обчислювальні ядра таких додатків можуть виграти від асиметричних кластерів, але властива невідповідність між різнорідними компонентами створює серйозну перешкоду для підтримки високих показників вводу/виводу, необхідних для отримання цих переваг. Це створює критичні дослідницькі проблеми для симетричних , а також асиметричних кластерів, які слід вирішити, щоб повністю реалізувати можливості доступних ресурсів [37-39]. Обмежене приєднане сховище ресурсу на основі прискорювача може перешкодити йому розмістити набір важливих оптимізацій вводу-виводу, наприклад, кешування, попереднє завантаження тощо, які є стандартними для вузлів загального призначення, і змусити їх покладатися на GPP. для попередньої вибірки даних і виконання введення-виведення від їх імені. Крім того, непотрібна серіалізація вводу-виводу може виникнути в менеджері кластера, коли він намагається надати кілька прискорювачів. Враховуючи, що будь-який асиметричний кластер пристойного розміру буде використовувати кілька вузлів, може виникнути конкуренція за сховище та мережеві ресурси, що погіршить продуктивність. Раніше розроблені бібліотеки розподілу даних і керування завданнями для асиметричних архітектур на основі прискорювача, делегують параметризацію передачі даних і планування робочого навантаження програмісту. Незважаючи на те, що це дієвий підхід, він ставить перед програмістом додаткові проблеми при проектуванні та збільшує час розробки програми. Ефективне використання гетерогенних прискорювачів і співпроцесорів у розподіленому середовищі вимагає створення ефективних і

прозорих методів керування введенням/виведенням, попередньої вибірки та постановки даних, які приховують від програміста базову асиметрію та неоднорідність ресурсів, забезпечуючи при цьому бажану продуктивність.

1.1.3. Управління гетерогенними багатоядерними системами

Хоча потенціал багатоядерних систем для каталізації систем HPC і центрів обробки даних очевидний, спроба легко інтегрувати гетерогенні багатоядерні процесори у великомасштабні обчислювальні установки викликає кілька проблем в управлінні цими ресурсами. Існує невід'ємний дисбаланс між ядрами загального призначення, прискорювачами та співпроцесорами в асиметричних налаштуваннях. Тенденція до інтеграції відносно простих ядер із надзвичайно ефективними блоками векторної та потокової обробки призводить до проектів, які за своєю суттю є ефективними для обчислень, але неефективними для керування. Ядра загального призначення ефективні у виконанні коду, що вимагає інтенсивного керування, тому їх, як правило, використовують переважно як контролери паралельного виконання та зв'язку з прискорювачами. З іншого боку, прискорювачі ефективні у виконанні паралельних обчислювальних завдань з даними. Щоб вирішити цю проблему, у великомасштабних системних інсталяціях використовуються спеціальні підходи до поєднання прискорювачів з більш ефективними процесорами, такими як багатоядерні ЦП x86 [21], тоді як архітектура процесорів рухається в напрямку інтеграції ефективних для керування та обчислень ядер на одній мікросхемі [6].

Ще одна фундаментальна проблема управління ресурсами виникає в гетерогенній установці з кількома клієнтами, наприклад, середовищі хмарних обчислень, де гетерогенні ресурси спільно використовуються між декількома одночасно запущеними програмами. Співпроцесори та прискорювачі, як правило, не підтримують виконання кількох програм одночасно, тому вимагають особливого планування, підтримуючи мультитенантність. Щоб збільшити паралелізм між декількома ядрами, новітні графічні процесори

NVIDIA тепер підтримують одночасне виконання кількох ядер, однак ці паралельні ядра мають запускатися з одного процесу. Цієї підтримки недостатньо для підтримки одночасного виконання кількох програм. Крім того, кожен обчислювальний ресурс пропонує унікальні переваги продуктивності для різних додатків, а використання гетерогенних співпроцесорів і прискорювачів в одному кластері, що підтримує багатокористувацький режим, створює додаткові труднощі оптимального застосування для відображення співпроцесора для підвищення загальної продуктивності системи.

Однією з ключових переваг використання різнорідних багатоядерних процесорів у центрах обробки даних і корпоративних установах є енергоефективність, яку пропонують ці ресурси. Використання високопродуктивних енергоефективних співпроцесорів у поєднанні з низькопродуктивними енергоефективними процесорами загального призначення є життєздатним підходом, який можна адаптувати для мінімізації споживання енергії обчислювальним кластером . Однак відсутність ефективного управління ресурсами з урахуванням живлення та методів планування, які можна застосувати до різнорідних мультитенантів кластери загалом зменшують переваги енергоефективності цих ресурсів. Розробка ефективних методів управління ресурсами та планування з урахуванням енергії є складним завданням, оскільки воно вимагає інформації про енергетичний профіль усіх різнорідних обчислювальних ресурсів і характеристики паралельних програм, які виконуються в кластері.

1.2. Багатоядерні обчислювальні прискорювачі

Багатоядерні процесори з тісно пов’язаними прискорювачами стають звичайними, з потенціалом підтримки продуктивності вузла суперкомп’ютерного класу для щільних обчислень за розумного бюджету потужності. У великих центрах обробки даних зазвичай використовуються

готові компоненти, щоб створити економічно ефективне налаштування. У той час як звичайне апаратне забезпечення було звичайним місцем у установках HPC протягом майже двох десятиліть, великі центри обробки даних, що представляють комерційний інтерес, наприклад, Google, Amazon EC2 тощо, дотримуються того самого підходу до створення ефективних систем на масштаб.

Наявність стандартних прискорювачів, таких як Cell [1, 2] у Sony PlayStation 3 (PS3) на базі Cell, а також графічних процесорів NVIDIA робить ці обчислювальні механізми головними кандидатами для розгортання у великомасштабних системах HPC. Швидке зростання високошвидкісних мереж і використання недорогого готового апаратного забезпечення робить такі розподілені асиметричні кластери природними замінниками дорогих високоякісних суперкомп'ютерів. Використання готових компонентів у великих кластерах було продемонстровано як в наукових колах, наприклад, Condor [45], так і в промисловості, наприклад, Tianhe, Google, Roadrunner і Amazon.

1.2.1. Архітектура системи Cell Broadband Engine

Cell показаний на рисунку 1.2, є гетерогенним багатопроцесорним процесором з одним 64-розрядним двостороннім ядром PowerPC SMT загального призначення (Power Processing Element - PPE) і вісьмома векторними процесорами. Процесорні (128-розрядні SIMD-RISC) ядра (Synergistic Processing Elements - SPE), які спеціалізуються на прискоренні паралельних обчислень даних. Мережа взаємозв’язку Cell на кристалі — це кільцеве кільце, яке називається Interconnect Bus (EIB), яке з’єднує всі дев’ять ядер із пам’яттю та зовнішнім каналом введення/виведення для доступу до інших пристроїв, таких як диск і мережевий контролер.

PPE функціонує як зовнішній процесор для планування завдань і розподілу даних між SPE, а також для запуску операційної системи. Він також надає підтримку SPE для виконання системних викликів і послуг. SPE

призначені для прискорення паралельних даних (векторних) обчислень. SPE — це спеціалізовані процесори з приватною пам'яттю, керованою програмним забезпеченням і програміст відповідає за визначення переміщення даних між основною пам'яттю та локальному сховищі кожного SPE за допомогою узгодженого механізму DMA Cell, і може перекривати затримку передачі даних з обчисленням за допомогою кількох асинхронних DMA. Ця можливість дозволяє програмісту явно керувати потоком даних між компонентами Cell, наприклад, для покращення продуктивності введення/виведення.

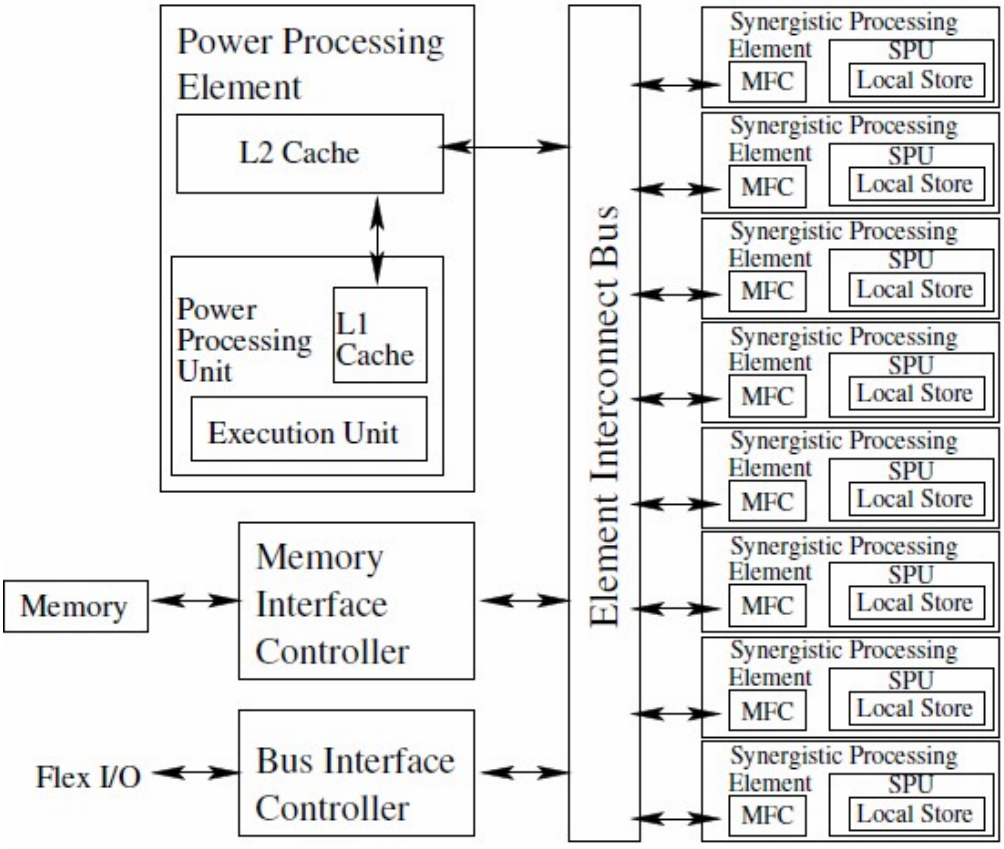


Рис. 1.2. Архітектура системи Cell Broadband Engine

1.2.2. Обчислення загального призначення на графічних процесорах (GPGPU)

Обчислення загального призначення на графічних процесорах (GPGPU) – це поширена техніка, яка використовується для використання паралелізму

за допомогою графічного процесора (GPU) для виконання обчислень загального призначення, якими традиційно керує ЦП. Сучасні найкращі графічні процесори оснащені десятками фрагментних процесорів і мають набагато вищу пропускну здатність, ніж традиційні процесори. Обчислювальна потужність графічних процесорів була використана для багатьох наукових робіт, баз даних , геометричних зображень і показали значну перевагу в продуктивності порівняно з традиційний процесор і багатоядерне програмування. Крім того, збільшення паралелізму всередині процесора також призводить до інших бажаних переваг , таких як зниження енергоспоживання і кращі затримки пам'яті.

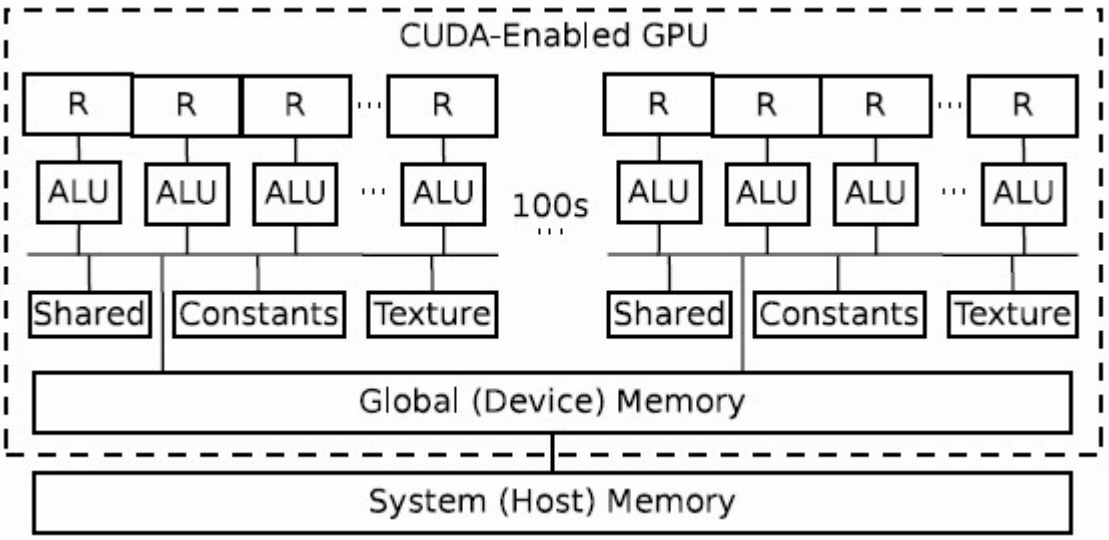


Рис. 1.3. Архітектура графічного процесора NVIDIA з підтримкою CUDA

Графічні процесори з підтримкою CUDA – графічні процесори мають архітектуру SIMT і забезпечують можливості потокової обробки, що дозволяє програмісту виконувати паралельну частину коду на пристроях графічного процесора. На рисунку 1.3 показано високорівневу архітектуру графічного процесора NVIDIA з підтримкою CUDA. Типовий GPU містить сотні мікропроцесорів, згрупованих разом у блоки обробки, що мають спільну та виділену ієрархії пам’яті. Апаратно реалізований планувальник ефективно виконує велику кількість потоків під час кожної обробки блоку.

Фреймворк програмування CUDA [4] зазвичай використовується для програмування графічних процесорів NVIDIA. CUDA надає набір мовних розширень для мови програмування C/C++, щоб розрізняти виконувані багатопотокові функції GPU та однопотокові функції, що виконуються на хості. Завдяки покращеній підтримці програмування графічні процесори тепер впроваджуються в основні кластери.

1.3. Модель паралельного програмування для великомасштабної обробки MapReduce

MapReduce — це модель паралельного програмування для великомасштабної обробки даних у паралельних і розподілених обчислювальних системах. Він забезпечує мінімальні абстракції, приховує архітектурні деталі та підтримує прозору відмовостійкість. Ця модель є високопродуктивною альтернативою для традиційних паралельних мов програмування та бібліотек для обчислювальних середовищ із інтенсивним використанням даних, від корпоративних обчислень до наукових обчислень у пета-масштабі [31]. Декілька дослідницьких заходів були спрямовані на перенесення MapReduce на багатоядерні архітектури, тоді як нещодавно провідні постачальники, такі як Intel, почали підтримувати MapReduce нативно в експериментальних програмних продуктах [13].

```
void map(String document) {
  // document: document contents
  for each word w in document:
    EmitIntermediate(w, '1');
}

void reduce(String word, Iterator partialCounts) {
  // word: a word
  // partialCounts: a list of aggregated partial counts
  int sum = 0;
  for each pc in partialCounts:
    sum += ParseInt(pc);
  Emit(word, AsString(sum));
}
```

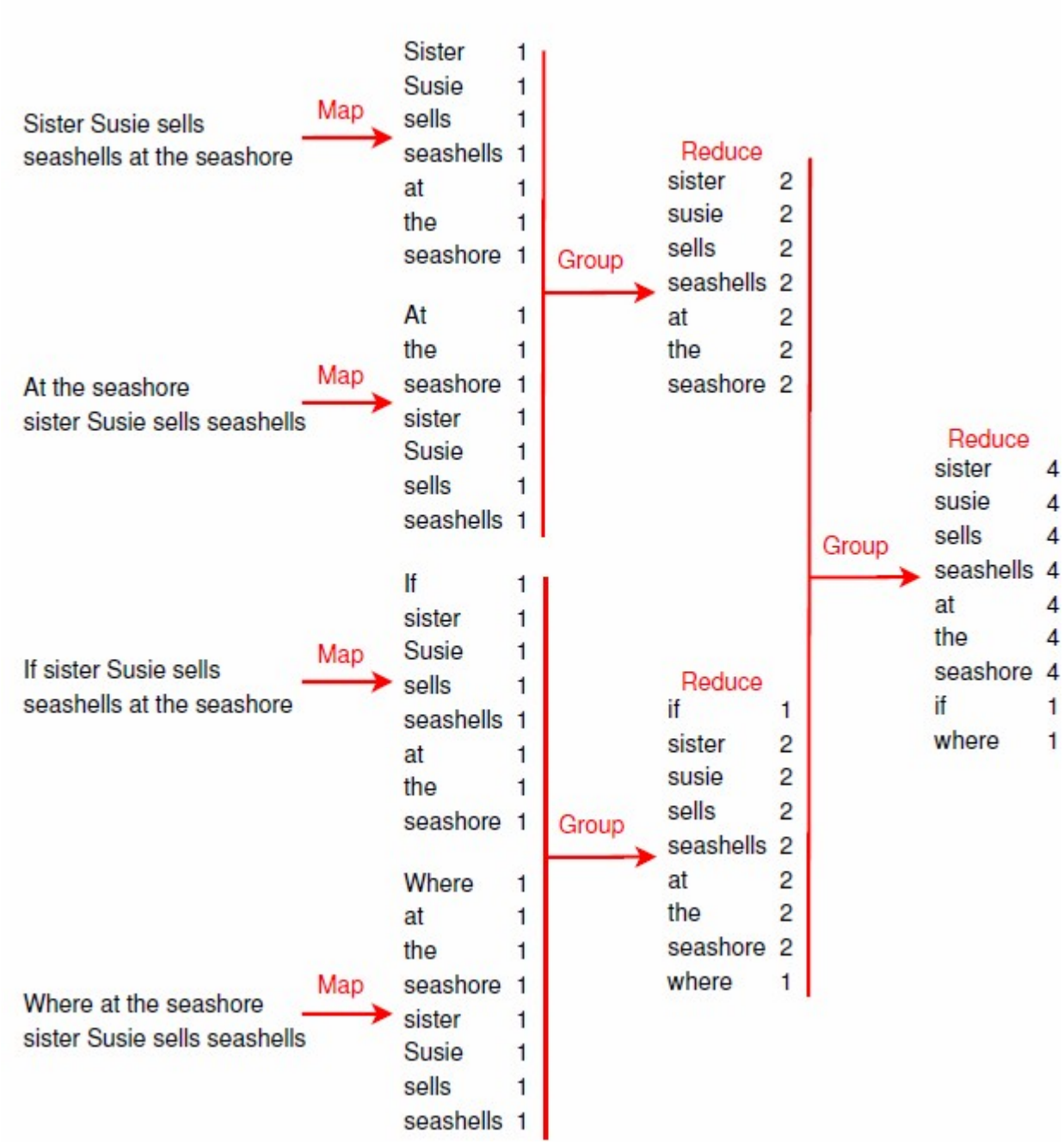


Рис. 1.4. Приклад операцій MapReduce у програмі Word Count

На рисунку 1.4 показано приклад різних операцій MapReduce у програмі Word Count. Перша фаза включає в себе операції карти, які створюють проміжні дані у формі пар (ключ, значення). Потім проміжні дані групуються разом перед виконанням повторних операцій скорочення для отримання остаточного набору результатів. На наступній ілюстрації наведено

псевдокод для простих операцій зіставлення та скорочення програми Word Count, показаної на рисунку 1.4.

MapReduce зазвичай припускає однорідні віртуальні процесори, які чергують ping карт, розділення, сортування та об'єднання даних. Хоча цей підхід є дружнім до програміста, він додає складності системі виконання, якщо остання має керувати різнорідними ресурсами з помітно змінною ефективністю під час виконання коду, що вимагає інтенсивного керування та обчислень. Недавня робота [9] стосується неоднорідності продуктивності, але обмежується лише проблемами, що виникають у зв'язку з використанням віртуальних машин для підтримки обчислювальних вузлів [36]. Внутрішня неоднорідність архітектури залишається основною проблемою, коли включають компоненти кластера спеціалізовані прискорювачі, оскільки в таких налаштуваннях функція відображення повинна враховувати можливості та обмеження окремих компонентів під час планування завдань на цих ресурсах. Ще одна складність виникає через припущення, що дані розподіляються між процесорами до того, як почнеться виконання обчислень MapReduce.

У розподілених системах із прискорювачами, прискорювачі використовують приватні адресні простори, якими потрібно явно керувати системою виконання. Приватні простори ефективно створюють додатковий рівень розподілу даних і кешування (через їх зазвичай обмежений розмір), який невидимий для програмістів, але повинен бути реалізований з максимальною ефективністю системою виконання.

Загальнодоступні реалізації MapReduce for Cell і GPU надають програмісту набір API для написання програм MapReduce для цих архітектур. Середовище виконання для реалізації Cell розділяє потік виконання на п'ять етапів (відображення, розділення, швидке сортування, сортування злиттям і зменшення) і планує ці етапи на ядрах прискорювачів. Ця робота показала, що порівняно зі стандартними багатоядерними налаштуваннями Cell може забезпечити підвищення продуктивності для обчислювально інтенсивних

робочих навантажень із помірними наборами даних. Подібним чином реалізація MapReduce для графічного процесора приховує складність програмування графічного процесора за простими у використанні інтерфейсами MapReduce і дозволяє користувачам писати програми MapReduce для графічних процесорів, вивчаючи графічні API та архітектуру графічного процесора.

Висновки до розділу

В даному розділі проведено комплексний аналіз особливостей управління гетерогенними багатоядерними кластерами та системами, висвітлюючи ключові аспекти, які визначають складність та виклики в цій галузі. Було виявлено, що управління гетерогенними системами значно ускладнюється через відмінності у властивостях і продуктивності компонентів, що вимагає ефективних методів розподілу навантаження. Важливим питанням залишається розробка методів і моделей, що дозволяють гармонійно об'єднувати різнорідні обчислювальні ресурси.

Виклики програмування пов’язані з необхідністю враховувати специфічні характеристики кожного типу ядра та архітектури, що впливає на ефективність і продуктивність. Висока складність програмування потребує використання паралельних моделей, які дозволяють створювати оптимізовані додатки для різнорідного середовища. Для оптимального функціонування гетерогенних багатоядерних систем важливою є ефективна організація процесу попередньої вибірки даних, що дозволяє зменшити затримки доступу до пам'яті та підвищити швидкість обробки інформації.

Розгляд архітектур, таких як Cell Broadband Engine і GPGPU, показав, що ці прискорювачі здатні значно покращити продуктивність обчислень, однак вимагають спеціалізованих підходів для ефективної інтеграції з традиційними CPU.

Модель паралельного програмування MapReduce. Використання моделей, таких як MapReduce, вказує на можливість ефективної обробки великих обсягів даних у великомасштабних системах. Ця модель дозволяє розподілити обробку на декілька ядер та вузлів, що підвищує ефективність виконання паралельних обчислень. У підсумку, розділ сформував базові уявлення щодо необхідності створення адаптивних методів управління та програмування для оптимального використання гетерогенних багатоядерних кластерів.

РОЗДІЛ 2. МОДЕЛІ ТА МЕТОДИ АДАПТИВНОЇ СТРУКТУРИ ПРОГРАМУВАННЯ ДЛЯ ГЕТЕРОГЕННИХ КЛАСТЕРІВ

2.1. Асиметрична архітектура системи

Асиметричні паралельні архітектури швидко впроваджуються в нових системах як обов'язкова умова для досягнення високої продуктивності без шкоди для надійності. Модель реалізовано в асиметричних багатоядерних процесорах, де фіксований транзисторний бюджет витрачається на багато простих, тісно пов'язаних ядер прискорювального типу. Ці ядра забезпечують спеціальні функції, які дозволяють прискорити обчислювальні ядра, що працюють з векторними даними. Ядра прискорювачів контролюються відносно невеликою кількістю звичайних процесорних ядер, які також запускають системні служби та керують зв'язком поза процесором. Дослідники зібрали все більше доказів переваги асиметричних багатоядерних процесорів з точки зору продуктивності, масштабованості та енергоефективності [8, 10]. Поява клітинного процесора та графічних процесорів як HPC і механізмів обробки даних [15-21, 45] додатково підтверджує потенціал асиметричних архітектур.

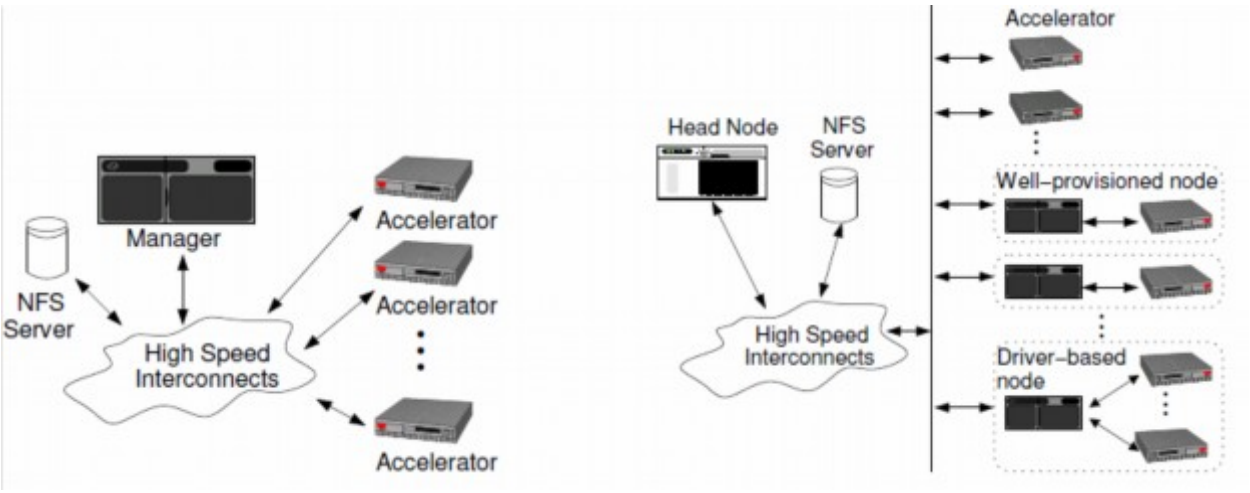
У той час як моделі паралельного програмування для симетричних кластерів були детально вивчені, синтез моделей паралельного програмування для асиметричних паралельних архітектур є відкритою проблемою. Зокрема, головними проблемами залишаються приховування архітектурної асиметрії від моделі програмування та використання величезної обчислювальної щільності прискорювачів у той час, коли вони взаємодіють із за своєю суттю повільнішими компонентами системи. Одним із хороших кандидатів із однорідної моделі паралельного програмування, яку можна адаптувати для асиметричних кластерів HPC, є MapReduce. MapReduce — це проста модель для машинно-незалежного паралельного програмування у великих масштабах. Він забезпечує мінімальні абстракції,

приховує архітектурні деталі, включаючи неоднорідність, і підтримує прозору відмовостійкість. Хоча поточні реалізації MapReduce підтримують автономні прискорювачі, наприклад, Cell [31], і враховують неоднорідність обчислювальних вузлів через віртуалізацію в планувальнику завдань, вони не справляються з асиметрією між обчислювальною щільністю прискорювачів і можливостями обробки та пересилання даних вузлів керування. Ця асиметрія може призвести до серйозного зниження продуктивності через виявлення вузьких місць зв’язку або введення/виведення.

У цьому розділі ми пропонуємо два підходи, які можна використовувати для ефективного програмування гетерогенних кластерів на основі прискорювача. Спочатку ми представляємо CellMR, розширену структуру програмування на основі MapReduce для асиметричних кластерів HPC із головними вузлами загального призначення з великою пам’яттю та обчислювальними вузлами типу прискорювача. CellMR приховує асиметрію та забезпечує високопродуктивну, економічно ефективну та масштабовану обробку даних. Ми націлюємося на кластери HPC із різнорідними архітектурами процесорів, подібними до RoadRunner від LANL [21], однак побудовані з недорогими обчислювальними вузлами, які використовують щільність обчислення графічних та ігрових процесорів. Хоча CellMR можна поширити на довільні гібридні паралельні архітектури, ми спочатку оцінюємо наші зусилля на кластері, який використовує Cell, можливо, один із домінуючих асиметричних багатоядерних процесорів, як прискорювач, а потім ми розширюємо CellMR на різні конфігурації ресурсів, включаючи ієрархічні налаштування. CellMR використовує підхід потокової передачі даних для ефективної підтримки обчислень MapReduce і прагне повністю перекривати затримки введення/виведення та зв’язку. CellMR замінює передавання даних і бібліотеки керування завданнями для асиметричних архітектур на основі прискорювачів, таких як IBM ALF [40], які делегують параметризацію та оптимізацію планування передачі даних розробникам програм. CellMR прозоро адаптує параметри потокової передачі даних і

планування завдань до програми під час виконання, тим самим звільняючи розробників від значних зусиль програмування. CellMR також усуває вузькі місця вводу-виводу за допомогою таких методів, як асинхронний доступ і подвійна буферизація на кількох рівнях системи. По-друге, ми досліджуємо підхід до зниження бар'єрів входу для користувачів і організацій, яким необхідно скористатися перевагами обчислювальної потужності та енергоефективності, які пропонують кластери на основі прискорювачів. У нашому підході використовуються досягнення в розробці програмного забезпечення на основі компонентів, коли складні програмні системи складаються з повторно використовуваних і адаптованих компонентів. Зокрема, ми досліджуємо, як багатошарова архітектура програмного забезпечення може полегшити багато труднощів побудови та підтримки систем на основі компонентів на прискорюваних або базованих кластерах.

Рисунок 2.1 ілюструє високорівневий вигляд гетерогенної системи з симетричним (рис. 2.1 а), а також асиметричним (рис. 2.1 б) обчислювальні вузли. Менеджер і всі обчислювальні вузли з'єднані через високошвидкісну мережу, наприклад, Gigabit Ethernet.



а) Симетричні обчислювальні вузли б) Асиметричні обчислювальні вузли

Рис. 2.1. Високорівнева системна архітектура симетричних і асиметричних кластерів

Дані розміщуються в розподіленій файловій системі, такій як мережева файлова система (NFS) або Lustre. У нашій реалізації ми використовували NFS для базового порівняльного дослідження наших альтернативних підходів. Сервер керує низкою внутрішніх вузлів на основі прискорювачів і відповідає за планування завдань, розподіл даних, розподіл роботи між обчислювальними вузлами та надання інших служб підтримки в якості переднього кінця кластера. Фактичне навантаження обробки даних несуть прискорювачі Cell.

Типове налаштування MapReduce складається з виділеної зовнішньої машини, яка обробляє планування завдань і керування ресурсами для кількох внутрішніх ресурсів. Подібно до типових симетричних кластерів, передній вузол є багатоядерним сервером загального призначення з великим об'ємом DRAM, який діє як менеджер кластера. Різниця полягає в тому, що в CellMR внутрішні обчислювальні вузли є асиметричними прискорювачами на основі клітинок, PS3, а не звичайними комп'ютерами. Менеджер розподіляє та планує навантаження на обчислювальні вузли. Потім загальне ядро на обчислювальних вузлах використовує MapReduce для відображення призначеного робочого навантаження на ядра прискорювача. По суті, модель програмування CellMR нагадує дворівневу MapReduce: інтерфейс відображає робочі навантаження на серверні частини на основі комірок, а базове загальне ядро передає навантаження на свої прискорювачі.

2.1.1. Цільові прискорювачі

Усунення внутрішнього дисбалансу асиметричних кластерів на основі прискорювачів, приховуючи відповідну складність від користувачів, є ключовим для досягнення високої продуктивності та високої продуктивності. Хоча проектування для всіх можливих конфігурацій ресурсів і типів прискорювачів є дуже складним, ми характеризуємо ресурси на основі прискорювачів, які ми плануємо використовувати в цій дисертації, виходячи

з обчислень загального призначення та можливостей керування системою прискорювачів. Конкретніше розглянемо три класи прискорювачів.

Прискорювачі мають високу обчислювальну щільність, а також вбудовані можливості для ефективного виконання керуючого коду та самостійного керування введенням/виведенням і зв'язком. Наприклад, прискорювач у поєднанні з декількома процесорними ядрами загального призначення на одному чіпі потрапляє до цієї категорії. Внутрішня обчислювальна потужність ядер загального призначення та обсяг пам'яті, підключеної до прискорювачів, вважаються достатніми для самостійного керування в тому сенсі, що керуючий код, що виконується для планування завдань і здійснення зв'язку в загальному ядра спеціального призначення не стали для мене основним вузьким місцем продуктивності. Асиметричні багатоядерні процесори, такі як IBM Cell [1], підпадають під цю категорію.

Добре забезпечені прискорювачі з обмеженими ресурсами. Ці прискорювачі мають високу обчислювальну щільність, але недостатню вбудовану обчислювальну потужність загального призначення для виконання керуючого коду та/або недостатньо вбудованої пам'яті для самостійного керування вводом-виводом і зв'язком. Введенням/виведенням і зв'язком керує зовнішній виділений вузол із ядрами загального призначення, який діє як драйвер для прискорювачів. Програмовані FPGA або GPGPU, такі як NVIDIA з підтримкою CUDA та графічні процесори AMD/ATI з підтримкою OpenCL, підпадають під цю категорію. У цих налаштуваннях потрібен звичайний хост-процесор для запуску операційної системи та надання загального призначення вводу-виводу та комунікаційних можливостей для прискорювачів, які спілкуються з хостом через шину вводу-виводу.

Прискорювачі спільного драйвера з обмеженими ресурсами. Ці прискорювачі схожі на попередній випадок, однак драйвери спільно використовуються кількома прискорювачами, щоб отримати потенційно більш економічно ефективний дизайн. Ці прискорювачі схожі на попередній випадок, однак драйвери розподіляються між кількома прискорювачами, щоб

створити потенційно більш економічну конструкцію. Удосконалені багаточіпові програмовані графічні процесори, такі як NVIDIA Ge Force GTX 295 або установки з декількома програмованими прискорювачами або FPGA на одному вузлі драйвера підпадають під цю категорію. Зверніть увагу, що драйвер для цих прискорювачів може сам підтримувати гетерогенні прискорювачі в одному обчислювальному вузлі. Така організація драйверів фактично створює додатковий рівень асиметрії в системі.

2.1.2. Конфігурації ресурсів для асиметричних кластерів

Ми розглядаємо чотири конфігурації ресурсів для цільових асиметричних кластерів, як показано на рисунках 2.2 – 2.5. Конфігурації визначаються типом використовуваних внутрішніх компонентів, а також економічними обмеженнями та цілями продуктивності. У всіх випадках менеджер і всі серверні вузли з'єднані через високошвидкісну мережу, наприклад Gigabit Ethernet. Дані програми розміщуються в розподіленій файловій системі.

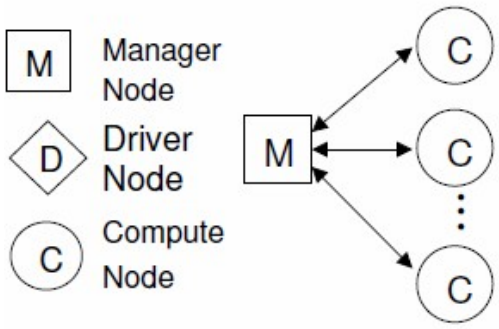


Рис. 2.2. Конфігурації ресурсів для активації асиметричних кластерів (випадок самокерованих добре забезпечених прискорювачів)

Перша конфігурація (рис. 2.2), яку ми розглядаємо, — це конфігурація самокерованих добре забезпечених прискорювачів (Conf I), підключених безпосередньо до менеджера. Ресурс із процесорами Cell, включаючи багатогігабайтну DRAM і високошвидкісне підключення до мережі, підпадає

під цю категорію. Невеликі навчальні заклади також можуть прийняти таку конфігурацію, використовуючи, наприклад, вузли PS3 і зменшуючи робоче навантаження на PS3, щоб не перевищувати обмежену ємність DRAM і не навантажувати обмежені можливості обробки загального призначення PS3. Обчислювальні вузли безпосередньо виконують усі завдання MapReduce, а менеджер об’єднує часткові результати з обчислювальних вузлів.

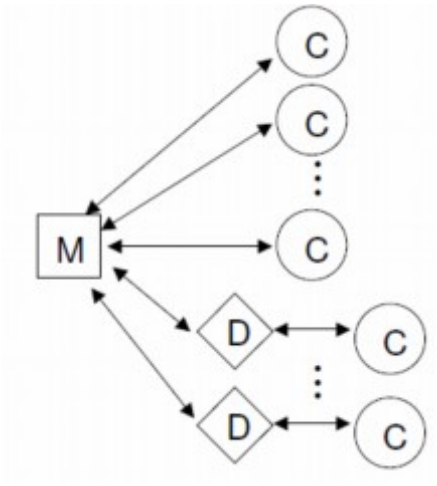


Рис. 2.3. Конфігурації ресурсів для активації асиметричних кластерів (випадок добре забезпечених прискорювачів з обмеженими ресурсами)

Наступна конфігурація (рис 2.3) використовує добре забезпечені прискорювачі з обмеженими ресурсами (Conf II). Кожен драйвер надає великий простір пам’яті, можливості зв’язку та введення/виведення для окремого прискорювача з обмеженими ресурсами, наприклад PS3. Менеджер надсилає дані до вузлів драйвера великими фрагментами. Вузли драйверів передають ці фрагменти на приєднані прискорювачі. Прискорювачі виконують завдання MapReduce, однак часткові результати, створені прискорювачами, об’єднуються у відповідних вузлах драйвера, а менеджер виконує глобальну операцію злиття результатів, отриманих від вузлів драйвера.

Використання одного драйвера на прискорювач з обмеженими ресурсами не завжди є виправданим, оскільки один прискорювач може бути

не в змозі повністю використовувати ресурси драйвера. Навпаки, одного керівника може бути недостатньо, щоб задовольнити потреби в даних багатьох прискорювачів одночасно. Ми виправляємо це, використовуючи ієрархічне налаштування (Conf III), щоб кожен вузол драйвера керував кількома вузлами прискорювача (рис. 2.4) .

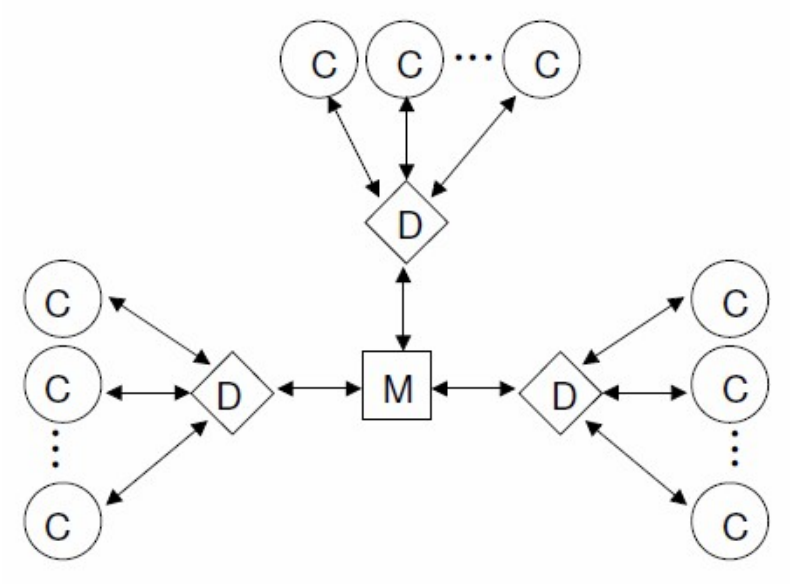


Рис. 2.4. Конфігурації ресурсів для активації асиметричних кластерів (випадок прискорювачів спільного драйвера з обмеженими ресурсами)

Використання одного драйвера на прискорювач з обмеженими ресурсами не завжди є виправданим, оскільки один прискорювач може бути не в змозі повністю використовувати ресурси драйвера. Навпаки, одного керівника може бути недостатньо, щоб задовольнити потреби в даних багатьох прискорювачів одночасно. Ми вирішуємо це за допомогою ієрархічного налаштування (Conf III), щоб кожен вузол драйвера керував кількома вузлами прискорювача (рис. 2.4).

Нарешті, асиметрична система може використовувати поєднання наведених вище конфігурацій на основі конкретних вимог . Ми фіксуємо цю суміш у нашій останній конфігурації (Conf IV) (рис. 2.5). У цьому випадку менеджер не залежить від класу приєднаних обчислювальних вузлів і просто розподіляє вхідне навантаження між доступними обчислювальними вузлами.

Виконання завдань MapReduce і об’єднання часткових результатів автоматично управляються на кожному компоненті, тоді як кінцевий результат створює менеджер, який виконує глобальне об’єднання результатів, отриманих від підключених драйверів.

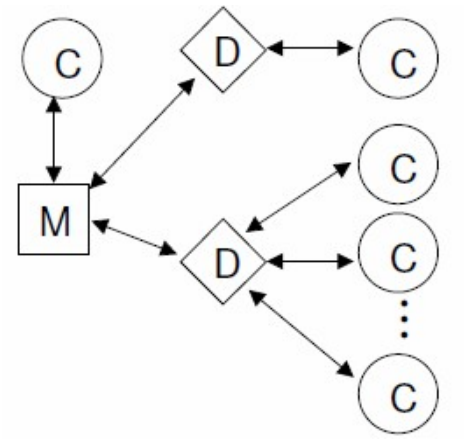


Рис. 2.5 Конфігурації ресурсів для активації асиметричних кластерів (випадок поєднання конфігурацій)

2.2. Застосування фреймворку розширеного програмування на основі MapReduce

Ми використовуємо MapReduce як кластерну модель програмування. Програми, розроблені з використанням MapReduce, паралельно обробляють дані за допомогою двох простих примітивів: Примітив відображення , який відображає вхідні дані пар (ключ, значення) на вихідні дані проміжних пар (ключ, значення); і примітив зменшення , який об’єднує значення, пов’язані з кожним ключем. Система середовища виконання розподіляє вихідні дані етапу карти між вузлами та сортує вхідні дані для кожного вузла перед застосуванням скорочення. Реалізації MapReduce забезпечують бібліотеку часу виконання для вираження інтенсивного паралельного обчислення з використанням карт і примітивів. Ця модель програмування має достатньо високий рівень абстракції, щоб приховати багато складнощів паралельного програмування, таких як розділення, відображення, балансування

навантаження та толерантність до помилок, водночас забезпечуючи адекватні можливості, які можна використовувати для керування різнорідними ресурсами в системі виконання.

Менеджер розділяє завдання MapReduce (карти, скорочення та сортування тощо) на невеликі робочі навантаження та призначає ці робочі навантаження приєднаним вузлам на основі прискорювача. Незалежно від типу серверних вузлів, менеджер прозоро розподіляє і планує навантаження на них. Якщо бек-енд є самокерованим прискорювачем, його ядро загального призначення використовує MapReduce для відображення призначеного робочого навантаження на ядра прискорювача (SPE). На противагу цьому, якщо серверна частина заснована на драйвері, компоненти драйвера далі розподіляють призначене робоче навантаження на підключений(і) вузол(и) прискорювача. Зверніть увагу, що менеджер відрізняється від водія. Драйвери виконують контрольні завдання для зв'язку та вводу-виводу від імені прискорювачів, тоді як менеджер контролює роботу та розподіл даних для всього кластера. Цю модель можна розглядати як ієрархічну MapReduce: кожен рівень відображає робоче навантаження на наступний рівень вузлів, поки він не досягне обчислювального вузла, тобто процесора Cell, де загальне вбудоване ядро відображає робоче навантаження на прискорювачі.

2.2.1. Розширення MapReduce для неоднорідних ресурсів

У типових налаштуваннях MapReduce завдання Map і Reduce плануються окремо на потенційно різних наборах вузлів кластера. У нашому вдосконаленому середовищі виконання MapReduce сегмент даних призначається обчислювальному вузлу, і вся послідовність операцій MapRed uce над цим сегментом даних виконується на цьому обчислювальному вузлі.

Таким чином, наша реалізація не вимагає класифікації ресурсів кластера як картографів або редукторів; сегмент даних залишається на призначеному вузлі, і обидві операції виконуються над ним на цьому вузлі, таким чином забезпечуючи покращену локальність. Одним із недоліків

наявності окремих відтворювачів і редукторів є те, що редуктори не можуть розпочати процес скорочення до завершення всіх відображень. У нашому середовищі виконання MapReduce менеджер не чекає завершення обробки всіх вузлів перед виконанням глобальної операції злиття. Натомість менеджер починає об'єднувати результати, щойно результати отримані від кількох обчислювальних вузлів.

2.2.2. Програмування асиметричних кластерів

З точки зору прикладного програміста, незалежно від використовуваної конфігурації ресурсу, MapReduce використовується для асиметричних ресурсів наступним чином. Додаток складається з трьох частин.

1. Код для ініціалізації середовища виконання. Це відповідає часу, проведеному в програмі MapReduce, але за межами фактичних етапів обробки даних MapReduce і включає ініціалізацію, розподіл даних і фіналізацію. Ця частина є унікальною для нашого дизайну та не має відповідної операції в попередніх реалізаціях MapReduce.

2. Код, який виконується на ядрах прискорювачів і виконує фактичну обробку даних для програми. Це схоже на стандартне налаштування програми MapReduce, що працює на невеликій частині вхідних даних, призначених обчислювальному вузлу. Він включає як фазу карти для розподілу робочого навантаження між ядрами прискорювачів, так і фазу скорочення для об'єднання даних, отриманих від прискорювачів.

3. Код, який запускається в диспетчері для об'єднання часткових результатів із кожного обчислювального вузла в повний результат. Цей код викликається кожного разу, коли результат отримується від обчислювального вузла, і виконує фазу глобального злиття, яка за роботою ідентична фазі зменшення на кожному обчислювальному вузлі.

Усі функції відображення, скорочення та об'єднання залежать від програми та мають надаватися програмістом. Після ідентифікації двійкові файли для вищевказаних компонентів генеруються для всіх доступних цілей

(різних прискорювачів і звичайних багатоядерних процесорів) у системі. Доступність цих двійкових файлів дозволяє нашій системі прозоро планувати завдання в будь-який час, на будь-якому типі прискорювача та приховувати неоднорідність і асиметрію. Крім того, це звільняє програміста від деталей системного рівня, таких як керування підсистемами пам’яті прискорювачів, оркестровка передачі даних між адміністратором і обчислювальними вузлами, а також впровадження оптимізованих механізмів зв’язку між вузлами кластера, і дозволяє програмісту зосередитися на додатках – певна частина коду.

2.2.3. Операції управління даними

У цьому розділі ми описуємо взаємодію під час виконання між різними програмними компонентами в менеджері та кожному з обчислювальних вузлів, як показано на рисунку 2.6.

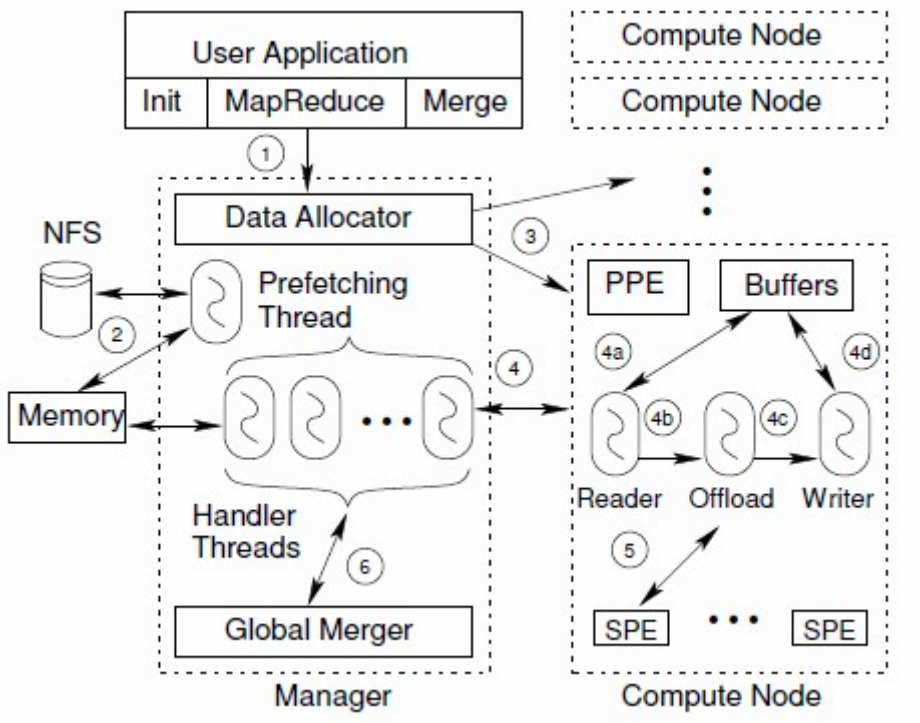


Рис. 2.6. Взаємодія між компонентами CellMR

Менеджер керує плануванням завдань, розміщенням даних і розподілом даних між драйверами або обчислювальними вузлами. Ми

використовуємо добре встановлені стандартні методи для перших двох завдань і зосереджуємось на управлінні обчислювальними вузлами та розподілі даних у цьому обговоренні. Коли програма починає виконуватися (крок 1 на рис. 2.6), менеджер завантажує частину пов'язаних вхідних даних із файлової системи (NFS у нашій поточній реалізації) у її пам'ять (крок 2). Це робиться для того, щоб забезпечити доступність достатньої кількості даних для обчислювальних вузлів і уникнути будь-яких вузьких місць вводу-виводу, які можуть перешкоджати продуктивності. Для добре підготовлених обчислювальних вузлів із драйверами цей крок замінено прямим попереднім завантаженням драйверів.

Далі на доступних обчислювальних вузлах запускаються клієнтські завдання (Крок 3). Ці завдання по суті самостійно планують свою роботу, запитуючи вхідні дані в менеджера, обробляючи їх і повертаючи результати назад менеджеру в безперервному циклі (Крок 4). Для добре забезпечених вузлів результати записуються безпосередньо у файлову систему, а менеджер інформується лише про завершення завдання. Коли менеджер отримує результати, він об'єднує їх (Крок 6), щоб отримати остаточний набір результатів для програми. Коли всі дані, завантажені в пам'ять, оброблено клієнтами, менеджер завантажує іншу частину вхідних даних у пам'ять (Крок 2), і весь процес продовжується, доки не буде спожито весь вхід. Ця модель схожа на використання великої кількості малих операцій з картою в стандартному MapReduce.

Завдання програми викликаються на обчислювальних вузлах (Крок 3) і починають виконувати цикл запиту, обробки та відповіді (Кроки 4a-4d). Ми називаємо обсяг даних програми, оброблених за одну ітерацію на обчислювальному вузлі, як робочу одиницю. За винятком спеціальної для програми функції розвантаження для виконання обчислень над вхідними даними наша структура на обчислювальних вузлах забезпечує всі інші функції, включаючи зв'язок із менеджером (або драйвером) і підготовку буферів даних для введення та виведення. Кожен обчислювальний вузол має

три основні потоки, які працюють на кількох буферах для роботи та передачі даних до/з менеджера або диска. Один потік (Reader) відповідає за запит і отримання нових даних від менеджера (Крок 4a). Дані поміщаються в приймальний буфер. Коли дані отримані, приймальний буфер передається потоку розвантаження (крок 4b), а потім потік Reader запитує додаткові дані, доки не будуть використані всі доступні приймальні буфери. Потік Offload викликає функцію Offload (Крок 5) на ядрах прискорювача з покажчиком на буфер прийому, тип даних робочої одиниці (зазначений програмою користувача на вузлі менеджера) і розмір робочої одиниці. Оскільки вхідний буфер, переданий функції Offload, також є її вихідним буфером, усі ці параметри є параметрами для читання та запису. Це робиться для того, щоб функція розвантаження могла змінювати розмір буфера, змінювати тип даних і розмір даних залежно від програми. Коли функція розвантаження завершується, останній вихідний буфер передається потоку Writer (крок 4c), який повертає результати менеджеру та звільняє буфер для повторного використання потоком Reader (крок 4d). Зверніть увагу, що обчислювальний вузол підтримує робочі одиниці змінного розміру та може динамічно регулювати розмір буферів під час виконання.

Драйвер у наших конфігураціях ресурсів взаємодіє з вузлом прискорювача так само, як менеджер взаємодіє з обчислювальними вузлами. Різниця між менеджером і вузлом драйвера полягає в тому, що менеджеру, можливо, доведеться взаємодіяти та передавати дані на кілька обчислювальних вузлів, тоді як драйвер керує лише одним вузлом прискорювача. Драйвер далі розділяє вхідні дані, отримані від менеджера, і передає їх до обчислювального вузла на фрагменти оптимального розміру, як описано в наступному розділі.

2.2.4. Оцінка

Ми оцінюємо CellMR за допомогою восьми обчислювальних вузлів Sony PS3, підключених через Ethernet 1 Гбіт/с до керуючого вузла. Менеджер

має два чотирьохядерні процесори Intel Xeon 3 ГГц, 16 ГБ оперативної пам’яті, 650 ГБ жорсткого диска та працює під керуванням Linux Fedora Core 8. Менеджер також запускає сервер NFS. З 8 SPE Cell лише 6 SPE є видимими для програміста на PS3.

Ми використовуємо наступні добре відомі програми MapReduce для вивчення впливу різних альтернатив дизайну для симетричного, а також асиметричного неоднорідного кластера. Ці додатки походять із наукових обчислювальних середовищ, включаючи епідеміологію, науку про навколишнє середовище, сегментацію зображень і статистичний аналіз. Для нашої оцінки ми використали чотири поширені програми MapReduce. Ці програми класифікуються на основі фази MapReduce, на якій вони витрачають більшу частину часу на виконання. Нижче наведено короткий опис програм, які ми перенесли на нашу структуру.

• Лінійна регресія: ця програма приймає як вхідні дані великий набір двовимірних точок і визначає лінійну найкращу відповідність заданим точкам. Це програма, у якій домінують карти.

• Підрахунок слів: ця програма підраховує частоту кожного унікального слова у заданому вхідному файлі. Вихідні дані — це список унікальних слів, знайдених у вхідних даних, разом із відповідною кількістю їх входжень. Це програма з домінуванням розділів.

• Гістограма: ця програма приймає як вхідні дані растрове зображення та виробляє підрахунок частоти кожної колірної композиції RGB у зображенні. Це програма з домінуванням розділів.

• K-Means: ця програма бере набір точок у N-вимірному просторі та групує їх у попередньо визначену кількість кластерів із приблизно однаковою кількістю точок у кожному кластері. Це програма з домінуванням розділів.

У таблиці 2.1 показано середній час виконання чотирьох тестів на окремому прискорювачі без використання нашої системи. Зверніть увагу, що лінійна регресія є єдиним тестом, який успішно завершується для всіх

розмірів вхідних даних. Усі інші контрольні тести спричиняють заміну та вичерпують простір підкачки з меншими розмірами вхідних даних. Також зверніть увагу на швидке зростання часу завершення через надмірну заміну, коли розмір вхідних даних збільшується.

Таблиця 2.1.

Час виконання (сек.) на автономній PS3

Input (MB)	Linear Regression	Word Count	Histogram	K-Means
4	0.34	1.95	1.06	1.66
64	2.88	501.76	45.66	167.93
128	12.56	-	318.66	-
192	21.81	-	394.78	-
256	34.89	-	-	-

2.3. Розширений підхід програмної інженерії до програмних гетерогенних кластерів

Вражаючі переваги продуктивності, які забезпечують гетерогенні кластери на основі прискорювачів, роблять їх бажаними обчислювальними засобами як для дослідників, так і для підприємств [21]. Неоднорідність апаратних блоків кластера на основі прискорювача, складність їх API та спеціальна природа їхніх комунікаційних інтерфейсів обмежують використання цих потужних та енергоефективних обчислювальних засобів усіма, крім досвідчених користувачів комп'ютерів. Дослідники в інших галузях — чиї симуляції та комп'ютерні моделі для дослідження безлічі галузей, наприклад, медицини, фізики високих швидкостей тощо, можуть скористатися такими ресурсами — просто залишаються осторонь. Сучасний рівень техніки такий, що буквально потрібно бути досвідченим комп'ютерним спеціалістом, щоб просто налаштувати та використовувати гетерогенний кластер на основі прискорювача, не кажучи вже про його оптимізацію та отримання максимальної продуктивності.

Рівневі архітектури є перевіреним підходом для вираження логіки складних комп'ютерних систем, з кількома методами реалізації. Деякі з цих

методів потребують спеціальних мов або мовних розширень. Ми досліджуємо підхід під назвою mixin-layers, який надає всі переваги багатошарових архітектур у межах стандарту C++. На цей вибір дизайну впливають унікальні вимоги нашого цільового середовища.

Типовий кластер на основі прискорювача потребує координації виконання кількох гетерогенних пристроїв, з'єднаних один з одним через високошвидкісне з'єднання. Завдяки повсюдності C++ можна знайти компілятор C++, який відповідає стандартам, майже на будь-якому комп'ютерному пристрої та операційній системі. Однією з переваг C++ є його природна взаємодія з C. Таким чином, навіть якщо компілятор C++ недоступний на якомусь незвичайному пристрої, завжди можна написати модуль на C і зв'язати його з охоплюючим компонентом C++.

У цьому розділі ми представляємо реалізацію багаторазово використовуваної та адаптованої структури компонентів програмного забезпечення для налаштування гетерогенних систем на основі прискорювача. Фреймворк забезпечує легкість використання та переваги розширення. Ми почали з нашої оптимізованої реалізації, описаної в попередніх розділах, присвячених оптимізації гетерогенних кластерів на основі прискорювача. Як це зазвичай буває, нашою початковою точкою була ручна реалізація для конкретного середовища розгортання, причому отриманий код нелегко використовувати або адаптувати. Ми представляємо результати реархітектури цього початкового коду в реалізацію на основі шару змішування, яка забезпечує повторно використовувані та адаптовані програмні компоненти та скорочує час розгортання програми для гетерогенних кластерів.

2.3.1. Функціонально-орієнтоване програмування та Mixin-Layers

Функціонально-орієнтоване програмування (FOP) — це методологія розробки програмного забезпечення, в якій функції є громадянами першого класу в архітектурі програмного забезпечення. FOP розкладає програми на

набір функцій, які разом забезпечують необхідну функціональність. Створення кількох об’єктів із єдиного набору функцій відокремлює основну функціональність об’єкта від його удосконалень, роблячи отримане програмне забезпечення більш придатним для повторного використання та надійнішим.

Крім того, FOP дозволяє легко змішувати та поєднувати функції модульним способом, оскільки додаток створюється за допомогою поетапного вдосконалення. Загальною стратегією впровадження у FOP є використання багаторівневої архітектури, у якій шари відповідають функціям. Отриманий «стек функцій» складається з багатьох шарів, причому кожен шар забезпечує одну функцію та уточнює існуючі функції в стеку.

Щоб поступово вдосконалювати та гнучко компонувати функції кластерного кластера на основі прискорювача, ми використовуємо mixin-layers, реалізацію нової багаторівневої архітектури, яка використовує вдосконалені методи програмування C++. Змішані шари моделюють різні співробітницькі ролі в межах кожного шару за допомогою внутрішніх класів. Внутрішні класи відображають відносини успадкування класів, що їх охоплюють, на рівні вище. За допомогою шарів змішування програміст може додавати функціональність гнучко, але систематично: кожен доданий рівень забезпечує ті внутрішні класи, які забезпечують необхідну функціональність.

Mixin-layers відповідає нашим цілям дизайну: він реалізує функції як спільну роботу менших капсульованих одиниць, кожну з яких можна вдосконалювати поетапно. У міру розвитку нашої експериментальної інфраструктури нові дослідницькі проблеми можуть вимагати переходу до іншої стратегії впровадження, яка краще задовольнятиме нещодавно відкритий набір вимог. Наприклад, було показано, що предметно-орієнтовані мови ефективні для гнучкого створення функцій. Однак дотримання іншої стратегії впровадження все одно дозволить використати ключові концептуальні внески нашого підходу: демонстрація того, як багаторазово використовувана та адаптована структура компонентів програмного

забезпечення може оптимізувати процес створення гетерогенних кластерів на основі акселератора.

2.3.2. Програмні компоненти для гетерогенних кластерів

Головним завданням при розробці системи на основі компонентів є вирішення того, які функціональні можливості мають бути включені в які компоненти програмного забезпечення. Серед основних критеріїв, які слід враховувати, є інтуїтивність клієнтських інтерфейсів і простота повторного використання. Інші критерії, як правило, є більш предметно-спеціальними, визначеними обмеженнями даної обчислювальної платформи. У цьому випадку наша головна мета полягає в тому, щоб інкапсулювати функціональні можливості для багаторазового використання, які можуть служити зручними будівельними блоками для побудови кластерів на основі прискорювачів. Ми прагнемо приховати більшість деталей низькорівневої реалізації від тих програмістів, які просто хочуть використовувати ці програмні компоненти, щоб швидко зібрати разом кластер механізмів прискорення. У наступному обговоренні ми окреслимо основні програмні компоненти, які ми вирішили зробити доступними як частину нашої інфраструктури. Ми підтримуємо наше рішення представити цей конкретний набір функціональних можливостей як програмні компоненти, описуючи їх функціональні можливості та клієнтські інтерфейси.

Програмні компоненти розподілені між ролями менеджера та обчислювального вузла. На рисунку 2.7 показано різні програмні компоненти менеджера та обчислювальних вузлів, які забезпечують логіку виконання нашого проекту, а також їх відповідну взаємодію один з одним. Далі ми описуємо, як функціональність кластера на основі прискорювача можна розкласти на окремі програмні компоненти, і пояснюємо функціональні можливості цих компонентів. Ці програмні компоненти можна багаторазово використовувати, і їх можна підтримувати з мінімальними зусиллями. Крім того, компоненти можна використовувати для проектування гетерогенних

кластерів на основі прискорювачів, що містять різні типи прискорювачів. Далі ми пояснюємо функціональність цих компонентів.

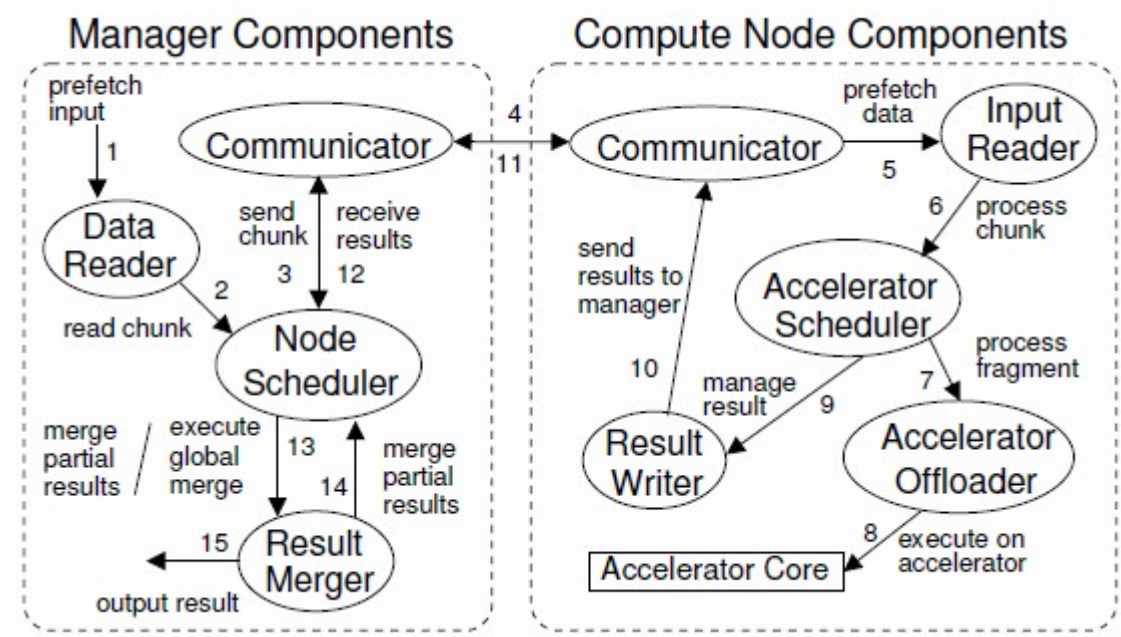


Рис. 2.7. Менеджер компонетів та обчислювальних вузлів та їх взаємодія

Тепер ми опишемо основні компоненти рівня менеджера. Комунікатор інкапсулює методи, необхідні для зв’язку між різними різнорідними ресурсами кластера. Цей компонент також реалізує можливості перекриття зв’язку та обчислення шляхом впровадження таких методів, як подвійна буферизація для кожного кінця комунікаційного пристрою. Комунікатор — це розширюваний модуль, який також можна використовувати для оптимізації апаратних компонентів, наприклад для приховування затримки зв’язку та покращення використання пропускної здатності.

Зчитувач даних - компонент попередньо завантажує вказані користувачем вхідні дані та зберігає їх у структурі відповідно до вимог програми. Дані зчитуються великими порціями із пристрою зберігання — для амортизації витрат на доступ, таких як час пошуку диска — і далі поділяються та впорядковуються відповідно до бажаного макета програми. Потім нові впорядковані дані передаються до планувальника вузлів для потокової передачі на доступні обчислювальні вузли. Зчитувач даних надає

інтерфейси для роботи з попередньо вибраними даними та зміни макета за бажанням.

Планувальник вузлів компонент керує обчислювальним вузлом. Для кожного обчислювального вузла, доступного кластеру, менеджер ініціює один екземпляр Node Scheduler . Планувальник кожного вузла отримує частину необроблених даних від Data Reader і далі оптимально передає їх до відповідного обчислювального вузла на основі прискорювача. Потокова передача налаштована на можливості обробки даних обчислювального вузла та може бути визначена користувачем у цьому компоненті. Процес потокової передачі триває, доки всі вхідні дані не будуть використані та оброблені обчислювальним вузлом.

Планувальник вузлів також отримує результати з обчислювальних вузлів і надсилає часткові результати до обчислювальних вузлів для остаточного об’єднання, як пояснюється далі.

Результат злиття надає програмісту інтерфейси для включення специфічних для програми механізмів для об’єднання часткових результатів з окремих обчислювальних вузлів у об’єднаний набір результатів, а також глобальні критерії об’єднання для отримання кінцевих результатів на вузлі менеджера. Планувальник вузлів отримує та передає часткові результати від обчислювальних вузлів до Result Merger, коли результати стають доступними. Об’єднання результатів об’єднує часткові результати на основі критеріїв конкретної програми, наприклад сортування за порядком, і виконує визначену функцію глобального злиття для отримання кінцевих результатів. Зауважте, що за потреби Result Merger також може знову використовувати Планувальник вузлів для розвантаження операцій об’єднання та злиття на обчислювальні вузли на основі прискорювача для підвищення продуктивності.

Тепер ми опишемо основні компоненти рівня обчислювальних вузлів, як показано на рисунку 2.7, і те, як дані маніпулюються між різними компонентами на обчислювальних вузлах.

Зчитувач вхідних даних на обчислювальному вузлі попередньо вибирає блоки даних із диспетчера та передає їх до компонента Accelerator Scheduler . Ми використали кілька буферів для зчитування даних із менеджера, щоб перекрити затримку зв'язку та обчислення, а також надали підтримку для визначення додаткових оптимізацій, якщо це необхідно. Якщо доступний порожній буфер, зчитувач вхідних даних ініціює запит на інший блок даних. Якщо для вхідних даних немає порожніх буферів , Input Reader просто чекає, поки деякі буфери звільняться.

Компонент Accelerator Scheduler планує дані, зчитані з Input Reader, для виконання на підключеному обчислювальному прискорювачі вузла. Цей компонент є специфічним для типу підключеного прискорювача вузла та надає можливість реалізувати оптимізацію будь-якого пристрою. Вхідний буфер із Input Reader далі розділяється на невеликі фрагменти, які підходять для розміщення в доступній пам'яті відповідного прискорювача. Ці невеликі фрагменти даних потім планується для потокової передачі в компонент Accelerator Offloader для виконання на прискорювачі.

Компонент Accelerator Offloader надає інтерфейс для виконання процедур розвантаження, характерних для пристрою, таких як обробка даних і функції злиття для конкретної програми . Цей компонент надає абстракції для методів, необхідних для інтеграції специфічних для пристрою мов програмування, таких як C для CUDA та C для PS3, із мовою загального призначення, такою як C++. Спеціальні обчислювальні процедури програми та прискорювача можна вказати в окремих файлах, щоб зберегти модульність інфраструктури.

Accelerator Offloader зчитує вхідні дані з Accelerator Scheduler і обробляє їх на підключеному Accelerator Core. Ядро прискорювача представляє певний пристрій прискорення, наприклад Cell або GPU, який виконує специфічне для програми завдання на заданих даних і створює результат. Потім результати повертаються до планувальника Accelerator який передає їх до Result Writer компонент.

Result Writer - компонент отримує результати від Accelerator Scheduler і надсилає їх менеджеру. Після того, як блок даних результату буде надіслано диспетчеру, пов’язаний з ним буфер позначається як вільний і може бути повторно використаний програмою зчитування вхідних даних . Зауважте, що результат із Accelerator Offloader не можна надіслати безпосередньо до автора результатів компонент без залучення Планувальника Accelerator компонент. Компонент Accelerator Scheduler повинен бути повідомлений, коли обробка певного фрагмента даних завершується в Accelerator Offloader щоб можна було запланувати наступний зріз даних. Якщо результати надсилаються безпосередньо до Result Writer компонент без залучення Планувальника Accelerator компонента, тоді потрібно реалізувати певний механізм сигналізації, щоб сповістити планувальник прискорювача компонента, що обробку певного фрагмента даних було завершено, що збільшить накладні витрати на синхронізацію між компонентами.

2.4. Імітаційне моделювання взаємодії гетерогенного кластеру

Зараз ми представляємо ілюстративний приклад справжнього кластера, щоб описати, як різні компоненти нашого дизайну працюють і взаємодіють один з одним. Описуючи функціональні можливості кожного компонента, цей приклад ілюструє, як наш фреймворк організовує вже існуючі програмні компоненти для виконання конкретного обчислювального завдання. Зокрема, ми зосереджуємося на застосуванні підрахунку слів, яке обговорювалося в попередніх розділах, і показуємо, як обчислювально інтенсивну проблему підрахунку частоти слів у текстових файлах можна природним чином розкласти для ефективного виконання в кластері на основі прискорювача.

Гетерогенний кластер, який ми використовували в цьому прикладі, складається з двох обчислювальних вузлів PS3, двох обчислювальних вузлів на основі GPU та менеджера, які з’єднані через високошвидкісну мережу.

Спочатку ми обговоримо компоненти програмного забезпечення, необхідні для програми Word Count. З точки зору програмування, уся функціональність кожного вузла кластера інкапсульована в екземплярі шаблону необхідних компонентів для вузла. Для нашого прикладу програми екземпляр шаблону для вузла менеджера виглядає наступним чином:

```
typedef Manager<WordCount> manager;

#define NUM_PS3          2
#define NUM_GPU          2

manager::DataReader                datReader;
manager::NodeScheduler<PS3>        PS3Schedule[NUM_PS3];
manager::NodeScheduler<GPU>        GPUSchedule[NUM_GPU];
manager::ResultMerger              resMerger;

void main () {
  datReader.startReadingThread();
  for (int i = 0; i < NUM_PS3; ++i) {
    PS3Schedule[i].startSchedulingThread();
  }
  for (int j = 0; j < NUM_GPU; ++j) {
    GPUSchedule[j].startSchedulingThread();
  }
  resMerger.doMerging();
}
```

Відповідний екземпляр шаблону для обчислювального вузла PS3 такий:

```
typedef PS3<ComputeNode<WordCount>> cNode;

cNode::InputReader                 inpReader;
cNode::AcceleratorScheduler        accSchedule;
cNode::AcceleratorOffloader        accOffloader;
cNode::ResultWriter                resWriter;

void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Нарешті, екземпляр шаблону обчислювального вузла на основі GPU виглядає наступним чином:

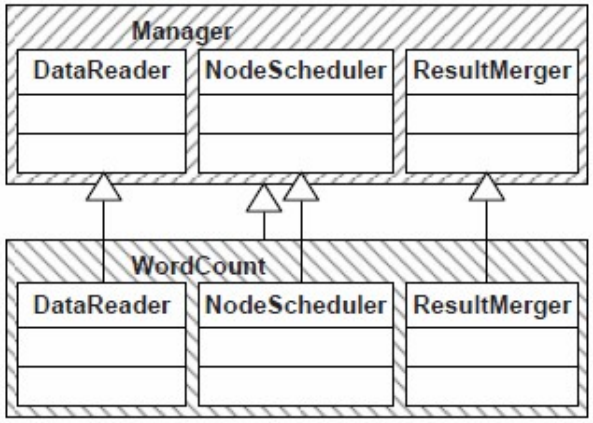
```
typedef GPU<ComputeNode<WordCount>> cNode;

cNode::InputReader                inpReader;
cNode::AcceleratorScheduler       accSchedule;
cNode::AcceleratorOffloader       accOffloader;
cNode::ResultWriter               resWriter;

void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Звернимо увагу на функціональність кожного компонента, наприклад, Input/Data Reader, виражається як внутрішній клас у кожному з наведених вище екземплярів шаблону.

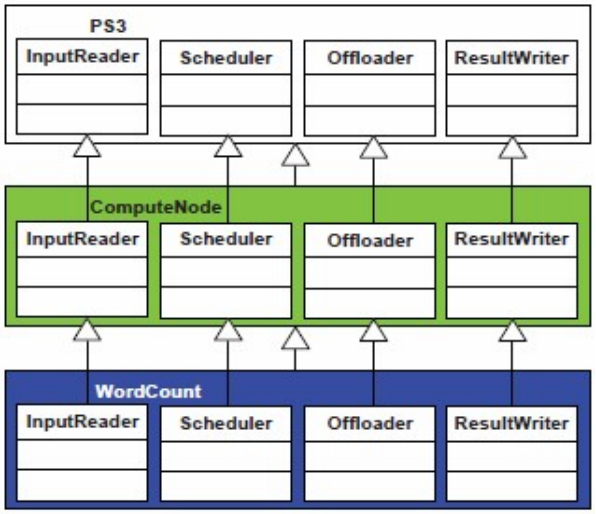
На рисунку 2.8 показано, як різні компоненти програмного забезпечення представлені як шари змішування, як для вузла менеджера, так і для обчислювальних вузлів Cell і GPU. Кожен рівень представляє компонент, визначений як функціональна одиниця з кількома ролями. Наприклад, компонент ComputeNode визначає ролі InputReader, Scheduler, Offloader і ResultWriter. Ці ролі визначають окремі операції, які використовуються під час виконання загального ComputeNode . Кожен шар додається до композиції, щоб удосконалити або розширити наявні компоненти. Наприклад, компоненти графічного процесора або PS3 додають функціональні можливості у своїх ролях, які є специфічними для їхніх відповідних архітектур.

З точки зору реалізації, кожен рівень реалізовано як шаблон C++ класу, чиї внутрішні шаблонні класи містять ролі рівня. Як основні класи компонентів, так і їхні ролі беруть участь у відносинах успадкування з відповідними класами на рівні вище. Таким чином, для повторного використання компонента з усіма його ролями програміст має лише включити цей компонент до екземпляра шаблону. Поки компонент має необхідні ролі (що можна забезпечити, дотримуючись ретельних методів

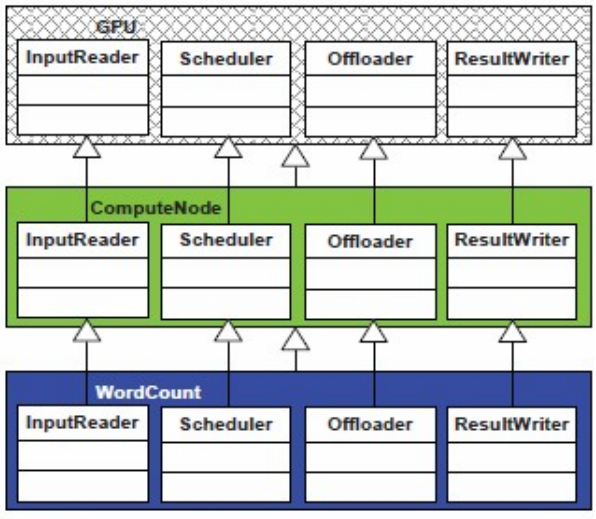
проектування), його функціональність стає негайно доступною для створення будь-якої програми.



(a) Mixin-layers for manager node.



(b) Mixin-layers for Cell-based compute node.



(c) Mixin-layers for GPU-based compute node.

Рис. 2.8. Рівні керуючих та обчислювальних вузлів та визначені ролі

Для програми Word Count, перелік кодів якої наведено вище, два з трьох компонентів для обчислювальних вузлів можна повторно використовувати одразу. На рисунку 2.8 повторно використані компоненти пофарбовані (заштриховані) однаково. Незважаючи на те, що повторно використовувані компоненти потрібно буде перекомпілювати для різних апаратних архітектур, їх функціональні можливості залишаться незмінними.

Цей невеликий, але реалістичний приклад демонструє, як багатошарову архітектуру програмного забезпечення можна використовувати для забезпечення простих у використанні та повторному використанні компонентів програмного забезпечення, які можуть бути як незалежними від архітектури, так і залежними від пристрою. Це спостереження змушує нас повірити, що дотримання цієї парадигми побудови програмного забезпечення має потенціал для полегшення багатьох складнощів реалізації для середнього програміста.

Висновки до розділу

В даному розділі представлено розширену модель програмування на основі MapReduce, яка призначена для ефективної роботи в умовах гетерогенних кластерів із використанням обчислювальних прискорювачів. Розглянуто особливості різних цільових прискорювачів та представлені конфігурації ресурсів гетерогенних кластерів у звичайних та ієрархічних структурах. CellMR застосовується до цих конфігурацій, демонструючи здатність ефективно керувати складністю ресурсних налаштувань з мінімальним втручанням програміста в процес управління ресурсами.

Додатково, CellMR забезпечує чітке розмежування функцій між операціями менеджера та обчислювальних вузлів, реалізуючи їх у середовищі виконання. Проведено оцінювання ефективності CellMR у порівнянні з традиційними підходами, що налаштовуються вручну, у різних симетричних та асиметричних конфігураціях гетерогенних ресурсів.

Результати свідчать, що CellMR демонструє вищу ефективність та високу масштабованість при збільшенні кількості обчислювальних вузлів.

Окрім того, було досліджено потенціал застосування багаторівневої архітектури у розробці програмного забезпечення для прискорення розгортання додатків у гетерогенних кластерах, оснащених обчислювальними прискорювачами.

РОЗДІЛ 3. МЕТОДОЛОГІЯ РОЗПОДІЛУ РОБОЧОГО НАВАНТАЖЕННЯ З УРАХУВАННЯМ МОЖЛИВОСТЕЙ ДЛЯ ГЕТЕРОГЕННИХ КЛАСТЕРІВ

3.1. Представлення гетерогенної архітектури системи

Хоча потенціал багатоядерних прискорювачів для каталізації HPC очевидний, спроба плавно інтегрувати гетерогенні ресурси у великомасштабні обчислювальні установки викликає труднощі щодо керування гетерогенними ресурсами та узгодження обчислень із характеристиками ресурсів. Тенденція до інтеграції відносно простих ядер із надзвичайно ефективними векторними блоками призводить до проектів, які за своєю суттю є ефективними для обчислень, але неефективними для керування. Таким чином, можливості багатоядерних прискорювачів для запуску коду, що вимагає інтенсивного керування, такого як операційна система або комунікаційна бібліотека, за своєю суттю обмежені. Щоб усунути цю проблему, у великомасштабних системних інсталяціях використовуються спеціальні підходи для поєднання прискорювачів із більш ефективними процесорами, такими як багатоядерні процесори x86, тоді як архітектура процесорів рухається в напрямку інтеграції ефективного керування та ефективності обчислень ядер на одному чіпі [6]. Використання специфічних для архітектури рішень є вкрай небажаним в обох випадках, оскільки це ставить під загрозу продуктивність, портативність і стабільність залучених систем і програм.

В цьому розділі ми розширюємо CellMR, щоб вирішити проблеми обмеження пам’яті під час використання різнорідних ресурсів. Ми представляємо вдосконалення в трьох аспектах моделі програмування MapReduce:

а) використовуємо прискорювачі з техніками, які покращують локальність даних і досягають перекривання етапів виконання MapReduce;

б) виконуємо підтримку середовища виконання для використання кількох архітектур прискорювачів (Cell і GPU) в одній установці кластера та адаптації виконання завдань робочого навантаження до різних архітектур прискорювачів під час виконання;

в) представляємо можливості виконання з урахуванням робочого навантаження для віртуалізованих налаштувань виконання програми. Останнє розширення важливе в обчислювальних хмарах, що містять різнорідні обчислювальні ресурси, де ефективний і прозорий розподіл ресурсів для завдань є важливим.

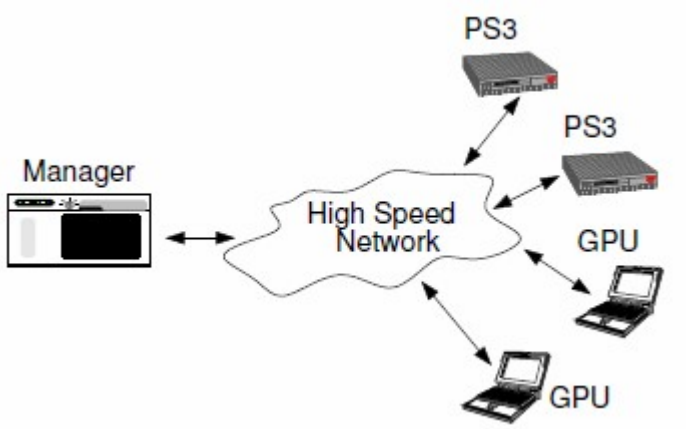


Рис. 3.1. Загальний огляд гетерогенного кластера Cell і GPU

На рисунку 3.1 показано гетерогенну кластерну архітектуру, яку ми досліджуємо в цьому розділі. Багатоядерний сервер загального призначення діє як виділений зовнішній менеджер для кластера та керує низкою внутрішніх вузлів на основі прискорювача. Менеджер відповідає за планування завдань, розподіл даних, розподіл роботи між обчислювальними вузлами та надання інших служб підтримки на передньому кінці кластера. Вузли прискорювача забезпечують кластеру високопродуктивні можливості обробки даних. Щоб ізолювати наше дослідження від впливу численних оптимізацій, доступних для кожного процесора типу прискорювача, ми припускаємо, що легко оптимізований, специфічний для архітектури виконуваний код для різних компонентів програми доступний для всіх типів

прискорювачів, наприклад, через бібліотеки, оптимізовані постачальником. У нашій експериментальній установці цей код зазвичай був би доступний через спеціальні інструменти програмування для прискорювача, такі як CUDA і Cell SDK.

Менеджер розділяє компоненти MapReduce (карту, скорочення, розділення та сортування) на невеликі завдання, придатні для паралельного виконання. Потім він викликає відповідні двійкові файли на прискорювачах і призначає завдання прискорювачам для обробки та агрегації даних. Якщо бек-енд є обчислювальним вузлом на основі клітинок, його загальне ядро використовує MapReduce у вузлі для відображення призначеного робочого навантаження на ядра прискорювача (SPE). Якщо бек-енд базується на графічному процесорі, його загальне ядро x86 використовує MapReduce для виконання робочого навантаження, призначеного підключеному графічному процесору. Коли обчислювальні вузли завершують виконання відповідних робочих навантажень, менеджер порівнює результати, виконує будь-яке необхідне об'єднання даних для конкретної програми та створює кінцевий результат. За потреби менеджер має можливість перевантажувати частину операцій навантаження зі злиття даних на прискорювачі.

Наша структура використовує прозоро оптимізовані двійкові файли для прискорювачів. Таким чином середовище виконання приховує асиметрію між різними доступними ресурсами. Тим не ¬менш, даний прикладний компонент демонструватиме відмінності в продуктивності на різних комбінаціях типів процесорів, систем пам'яті та з'єднань вузлів, доступних у кластері. Щоб покращити використання ресурсів і узгодження між компонентами MapReduce і доступними апаратними ресурсами, система виконання відстежує час виконання завдань на апаратних компонентах і використовує цю інформацію для адаптації планування завдань до компонентів, щоб кожне завдання завершилося виконанням на ресурс, який найкраще для цього підходить. Програміст додатків також може керувати середовищем виконання, забезпечуючи метрику афінності, яка вказує на

найкращий ресурс для даного завдання, наприклад, високе значення афінності для графічного процесора означає, що компонент програми працюватиме найкраще на графічному процесорі, тоді як афінність, рівна нулю, передбачає що додаток бажано виконувати на інших типах процесорів. Система середовища виконання враховує ці значення, приймаючи рішення щодо планування.

3.2. Ефективний розподіл даних програми

Ефективний розподіл даних програми між обчислювальними вузлами є центральним компонентом нашого дизайну. Це створює кілька альтернатив. Підхід «соломинної людини» полягає в тому, щоб просто розділити загальну кількість вхідних даних на таку кількість блоків, як кількість доступних вузлів обробки, і скопіювати фрагменти на локальні диски обчислювальних вузлів. Потім програма на обчислювальних вузлах може отримати дані з локального диска за потреби та записати результати назад на локальний диск. Коли завдання буде виконано, дані про результат можна буде прочитати з диска та повернути менеджеру. Цей підхід простий у реалізації та легкий для вузла менеджера, оскільки він зводить завдання розподілу до єдиного розподілу даних.

Статичну декомпозицію та розподіл даних між локальними дисками потенційно можна використовувати для добре забезпечених обчислювальних вузлів. Однак для вузлів з невеликою пам’яттю є кілька недоліків:

1) це вимагає створення додаткових копій вхідних даних зі сховища менеджера на локальний диск, і навпаки для даних результату, що може швидко стати вузьким місцем, особливо якщо диски обчислювального вузла повільніші, ніж ті, що доступні менеджеру;

2) він вимагає, щоб обчислювальні вузли зчитували необхідні дані з дисків, які мають більшу затримку порівняно з іншими альтернативами, такими як основна пам’ять;

3) це тягне за собою зміну робочого навантаження для врахування явного копіювання, що є небажаним, оскільки обтяжує прикладного програміста деталями системного рівня, таким чином роблячи програму непереносимою для різних налаштувань;

4) це тягне за собою додатковий зв'язок між менеджером і обчислювальними вузлами, що може уповільнити роботу вузлів і вплинути на загальну продуктивність. Отже, це невідповідний вибір для використання з прискорювачами з малою пам'яттю.

Друга альтернатива полягає в тому, щоб розділити вхідні дані, як і раніше, але замість того, щоб копіювати фрагмент на диск обчислювального вузла, як у попередньому випадку, зіставити фрагмент безпосередньо у віртуальну пам'ять обчислювального вузла. Мета тут полягає в тому, щоб використовувати високошвидкісні диски, доступні менеджеру, і уникнути непотрібного копіювання даних. Однак для вузлів з невеликим об'ємом пам'яті цей підхід може створювати блоки, які є дуже великими порівняно з фізичною пам'яттю, доступною на вузлах, що призводить до перевантаження пам'яті та зниження продуктивності. Це посилюється тим фактом, що для доступних реалізацій середовища виконання MapReduce потрібна додаткова пам'ять, зарезервована для системи виконання для зберігання внутрішніх структур даних. Отже, статичний розподіл вхідних даних не є життєздатним підходом для нашого цільового середовища.

Третя альтернатива полягає в тому, щоб розділити вхідні дані на фрагменти, розмір яких залежить від обсягу пам'яті обчислювальних вузлів. Фрагменти все одно мають бути зіставлені у віртуальну пам'ять, щоб уникнути непотрібного копіювання, тоді як розміри фрагментів мають бути встановлені таким чином, щоб у будь-який момент часу обчислювальний вузол міг обробити один фрагмент, одночасно надсилаючи наступний фрагмент для обробки та виводячи попередній фрагмент. обчислена частина. Цей підхід може підвищити продуктивність на обчислювальних вузлах за рахунок збільшення навантаження на менеджер, а також навантаження на

ядра обчислювальних вузлів, які запускають операційну систему та стеки протоколів введення/виведення. Таким чином, ми шукаємо точку проектування, яка збалансує навантаження менеджера, введення/виведення та системні накладні витрати на обчислювальних вузлах, а також необроблену обчислювальну продуктивність на обчислювальних вузлах. Ми використовуємо цей підхід у нашому дизайні.

3.3. Планування робочого навантаження з урахуванням додаткових можливостей

Ми розглядаємо два типи прискорювачів, процесори і графічні процесори з підтримкою CUDA, і розробляємо планувальник, який обслуговує як автономне, так і віртуалізоване виконання програм. В останньому випадку програми спільно використовують ресурси в просторі та/або в часі. Планувальник приймає два параметри як вхідні дані:

1) кількість і тип обчислювальних вузлів у гетерогенному кластері;

2) кількість одночасно запущених програм на гетерогенному кластері.

Далі ми спочатку описуємо різні стани виконання планувальника, а потім представляємо алгоритм планування.

3.3.1. Стани планування

На рисунку 3.2 показано різні стани, що представляють процес навчання та потік виконання планувальника. Спочатку планувальник починає зі статичного призначення завдань вузлам і процесорам на основі наданої користувачем метрики приналежності, продуктивності ресурсів з точки зору часу, витраченого на байт даних, або, якщо інформація недоступна, просто розподіляючи завдання рівномірно між ресурсами. Потім планувальник переходить у фазу навчання, де він вимірює час обробки для різних компонентів програми на ресурсах, на яких вони спочатку заплановані. На основі часу обробки робочого навантаження на кожному з

доступних обчислювальних вузлів планувальник обчислює час обробки на байт для кожного з доступних обчислювальних вузлів. Як тільки швидкість обробки відома, планувальник переходить до фази адаптації, де розклад коригується таким чином, щоб жадібно максимізувати швидкість обробки. Зауважте, що навіть на цьому етапі планувальник продовжує відстежувати свою продуктивність і відповідно коригувати свої рішення щодо планування.

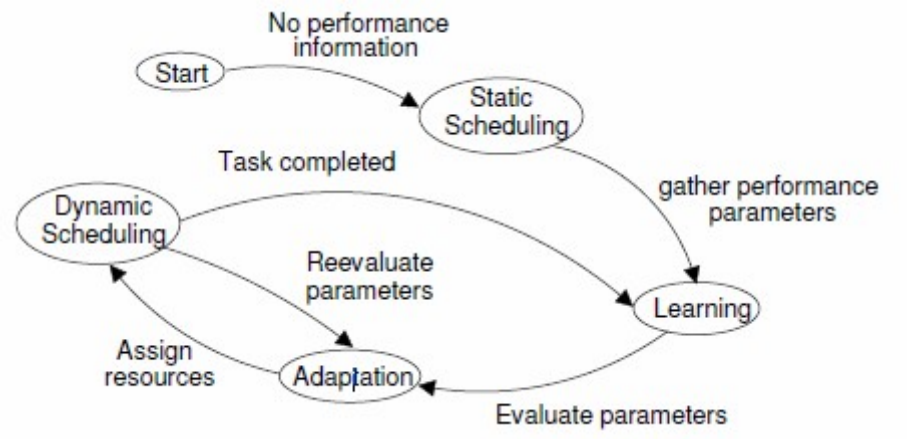


Рис. 3.2. Кінцевий автомат для процесу навчання та виконання планувальника

Для одночасного виконання кількох програм планувальник повинен вирішити, яку програму запускати на якому конкретному прискорювачі. З цією метою планувальник пробує різні призначення, наприклад, починаючи з планування програми A на процесорі Cell і програми B на графічному процесорі на заздалегідь визначений період часу Tlearn , потім скасовуючи призначення для іншого Tlearn, визначаючи призначення, яке забезпечує вищу пропускну здатність, і, нарешті, використання цього призначення для виконання програми, що залишилася. Тим не менш, час для визначення найкращого розкладу збільшуватиметься разом із кількістю програм, що виконуються одночасно можна зменшити за допомогою введення користувача або інформації з попередніх запусків програми. Якщо одна із програм завершує роботу раніше за іншу, планувальник переходить у фазу

навчання та намагається призначити щойно звільнені ресурси програмам, які очікують у черзі, або запущеній програмі, якщо черга порожня. Зауважимо, що не завжди можливо призначити програми найбільш підходящим вузлам, наприклад, коли кілька програм потребують одного типу прискорювачів і доступна лише обмежена кількість прискорювачів. Незважаючи на це, наш підхід гарантує, що всі обчислювальні вузли зайняті, і що призначення додатків обчислювальним вузлам є оптимальним, оскільки загальний час завершення для запланованих додатків мінімізується для даних ресурсів і додатків.

3.3.2. Алгоритм планування

Алгоритм 3.1 описує, як працює наша схема динамічного планування з урахуванням можливостей. Оскільки планувальник не має жодної інформації про те, як доступні обчислювальні вузли виконують задані завдання, на початку вибирається статичний графік. Графік коригується динамічно по мірі виконання завдань та вимірювання їх продуктивності на призначених обчислювальних вузлах.

Алгоритм 3.1. Планування робочого навантаження з урахуванням можливостей.

```
Input: nodeArray, appArray
for node ∈ nodeArray do
    for app ∈ appArray do
        estArray = sendNextChunk(node, app);
    end
end
nodeAppMap = GenNodeAppComb(estArray, nodeArray, appArray);
while incompleteAppExist(nodeAppMap) do
    for appNode ∈ nodeAppMap do
        app = getApp(appNode);
        node = getNode(appNode);
        if appCompleted(app) then
            nodeAppMap = GenNodeAppComb(estArray, nodeArray, appArray);
        end
        else
            estArray = sendNextChunk(node, app);
        end
    end
end
```

Зауважимо, що якщо доступно більше прискорювачів, ніж кількість застосунків, кожен застосунок планується на окремому прискорювачі під час фази навчання. Це усуває будь-які конфлікти ресурсів між різними застосунками та дозволяє визначити точні швидкості обробки.

Наприклад, якщо застосунки A та B призначені кластеру з чотирма обчислювальними вузлами на основі Cell та GPU, система призначає половину вузлів (2 Cell та 2 GPU) кожному з A та B під час фази навчання. Потім призначення переналаштовується з використанням вимірювань продуктивності для покращення загального часу виконання.

3.4. Динамічне масштабування робочих одиниць

Ефективне використання обчислювальних вузлів має вирішальне значення для загальної продуктивності системи. Ключове спостереження в CellMR полягає в тому, що продуктивність обчислювального вузла можна підвищити у багато разів шляхом зменшення навантаження на пам’ять, яка, у свою чергу, пов’язана з розміром робочої одиниці. Дуже велика робоча одиниця призводить до збою на обчислювальних вузлах, тоді як невиправдано мала робоча одиниця збільшує навантаження на менеджера. У будь-якому випадку продуктивність системи знижується. Проблема полягає в тому, щоб знайти робочу одиницю оптимального розміру, яка пропонує найкращий компроміс між продуктивністю обчислювального вузла та навантаженням менеджера.

Оптимальну робочу одиницю для запуску програми на конкретному кластері можна визначити вручну шляхом жорсткого кодування різних розмірів робочої одиниці, виконання програми та вимірювання часу виконання для кожного розміру. Найкращим розміром робочої одиниці є той, для якого мінімізується час виконання програми. Однак це трудомісткий процес, пов’язаний з помилками, і вимагає непотрібного «тестового» доступу до ресурсів, який важко отримати, враховуючи постійно зростаючу потребу у

виконанні «виробничих» завдань у кластері для підтримки високої працездатності.

Щоб виправити це, CellMR надає менеджеру можливість автоматично визначати найкращий розмір робочого блоку для конкретного застосування. Це робиться шляхом надсилання обчислювальним вузлам змінних розмірів робочих одиниць на початку програми та запису часу завершення, що відповідає кожній робочій одиниці. Техніка бінарного пошуку використовується для зміни розміру робочої одиниці, щоб визначити той, який дає найвищу швидкість обробки, обчислену за допомогою (розмір робочої одиниці)/(час виконання). Якщо швидкість обробки однакова для двох розмірів робочих одиниць, перевага віддається більшій, оскільки це мінімізує навантаження на менеджера. Визначений розмір робочої одиниці вибирається як найбільш ефективний для використання з програмою та використовується для решти виконання програми.

Усі доступні обчислювальні вузли беруть участь у пошуку оптимального розміру робочої одиниці. Менеджер надсилає збільшення робочих одиниць до кількох обчислювальних вузлів одночасно, хоча один розмір надсилається принаймні до двох обчислювальних вузлів для визначення середньої продуктивності. Після визначення оптимального розміру робочої одиниці його також можна повідомити користувачеві програми, щоб, можливо, полегшити оптимізацію для майбутнього запуску.

Відомо, що розмір одиниці роботи впливає на продуктивність обчислювальних вузлів, а отже, і всієї системи. У цьому експерименті ми спочатку покажемо, як зміна розмірів одиниць роботи впливає на час обробки на вузлі. Для цього ми використовуємо один вузол PS3, підключений до менеджера, та запускаємо лінійну регресію з розміром вхідних даних 512 МБ.

Рисунок 3.3 показує, що зі збільшенням розміру одиниці роботи час виконання спочатку зменшується до мінімуму, а потім експоненціально зростає. Точка мінімуму (показана пунктирною лінією) вказує на розмір,

після якого обчислювальний вузол починає використовувати підкачку. Використання більшого розміру знижує продуктивність. Використання розміру, меншого за цю точку, призводить до марнування ресурсів: зверніть увагу, що крива майже плоска до точки мінімуму, що вказує на відсутність додаткових витрат на обробку більшої кількості даних.

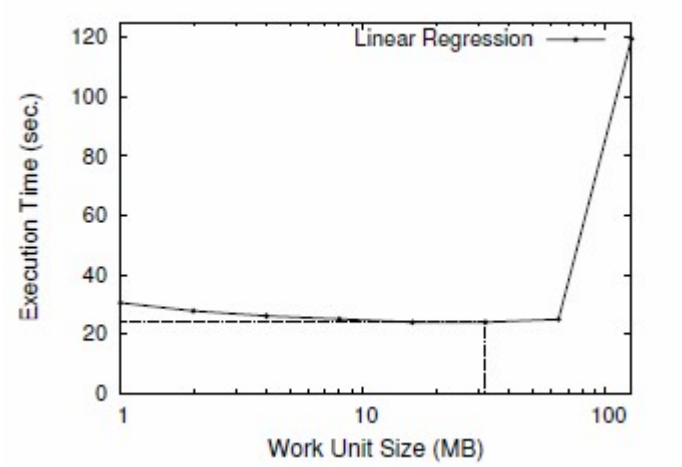


Рис. 3.3. Вплив розміру робочої одиниці на час виконання

Крім того, використання меншого розміру одиниці роботи збільшує навантаження на менеджер, оскільки менеджер тепер повинен обробляти більшу кількість фрагментів для заданого розміру вхідних даних. Використання розміру одиниці роботи, що відповідає точці мінімуму, є оптимальним, оскільки забезпечує найкращий компроміс між продуктивністю обчислювального вузла та менеджера, а також призводить до мінімального часу виконання.

Далі ми оцінюємо здатність CellMR динамічно визначати оптимальний розмір робочої одиниці. В принципі, оптимальний розмір блоку залежить від відносного співвідношення обчислень і передачі даних параметрів програми та машини, зокрема, затримок і пропускної здатності чіпа, вузла та мережевих з'єднань. Ми використовуємо експериментальний процес, щоб знайти оптимальний розмір робочої одиниці. Ми вручну визначили максимальний розмір робочої одиниці для кожної програми, яка може

працювати на одній PS3 без підкачки, як оптимальний розмір робочої одиниці. Ми порівняли розмір блоку ручної роботи з розміром, визначеним CellMR під час виконання.

Таблиця 3.1.

Результативність визначення розміру одиниці роботи

Application	Hand-Tuned Size (MB)	CellMR		
		Size (MB)	# Iterations	Time (s)
Linear Regression	32	30	16	0.65
Word Count	3	2	8	1.82
Histogram	2	1	4	0.15
K-Means	0.37	0.12	16	1.09

Для кожного застосування таблиця 3.1 показує: розмір робочої одиниці, як визначений вручну, так і автоматично, кількість ітерацій, виконаних CellMR для визначення робочої одиниці, і час, потрібний для прийняття цього рішення. Наша структура здатна динамічно визначати відповідну робочу одиницю, близьку до знайденої вручну, і визначення в середньому для наших тестів займає менше 0,93 секунди. Це незначно, тобто менше 0,5% від загального часу виконання програми, коли розмір вхідних даних становить 2 ГБ. Зауважте, що визначення оптимального розміру робочої одиниці не залежить від заданого розміру вхідних даних і має постійну вартість для даної програми. Таким чином, динамічне масштабування робочих одиниць у нашій системі є ефективним і досить точним.

У цьому експерименті ми визначаємо вплив різних розмірів робочих одиниць на продуктивність менеджера. Робиться це наступним чином. Спочатку ми починаємо тривалу роботу (лінійну регресію) на кластері. Далі ми визначаємо час, необхідний для компіляції великого проекту (ядра Linux 2.6) у менеджері, поки виконується завдання MapReduce. Ми повторюємо кроки, коли розмір робочої одиниці зменшується, потенційно збільшуючи вимоги до обробки з боку менеджера. Для кожного розміру робочої одиниці ми повторюємо експеримент 10 разів, використовуючи симетричний

гетерогенний кластер, і фіксуємо мінімальний, максимальний і середній час для компіляції, як показано на рисунку 3.4.

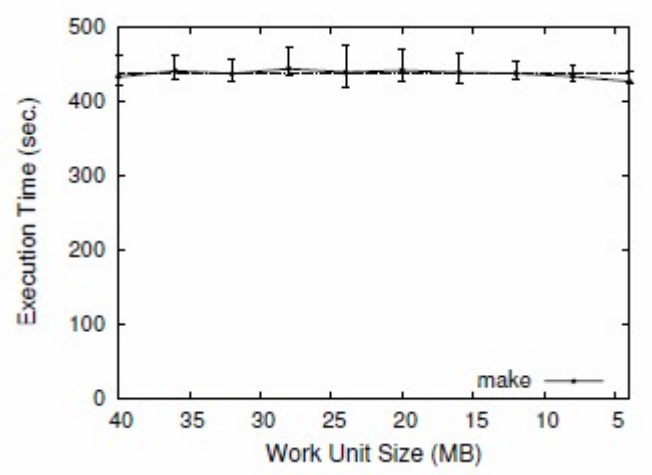


Рис. 3.4. Вплив розміру робочого підрозділу на менеджера

Горизонтальна пунктирна лінія на малюнку показує загальне середнє значення середнього часу компіляції для всіх розмірів робочих одиниць. Враховуючи, що загальне середнє значення залишається в межах мінімального та максимального часу, ми можемо зробити висновок, що варіації кривої часу компіляції знаходяться в межах похибки. Таким чином, відносно пласка крива вказує на те, що CellMR має постійне навантаження на менеджер, і наша структура може підтримувати різні робочі навантаження, не перетворюючись на менеджер у вузьке місце.

Висновки до розділу

У цьому розділі ми представили різні альтернативи розподілу даних для розподілу даних додатків між неоднорідними ресурсами та розробили ефективний підхід до розподілу даних, який розділяє дані додатків між асиметричними ресурсами на основі ємності пам’яті окремих прискорювачів і потоків отриманих даних блоків для кожного обчислювального вузла. Ми також розглянули проблеми асиметричних можливостей ресурсів,

розробивши техніку розподілу робочого навантаження з урахуванням можливостей для одночасного виконання кількох програм на різнорідних ресурсах. Запропонований механізм враховує характеристики вводу/виводу окремої програми та ресурси пам’яті, доступні кожному обчислювальному вузлу, під час виконання паралельних програм на гетерогенному кластері. Наші експериментальні результати показують, що запропоновані підходи значно покращують продуктивність додатків у порівнянні зі статичними методами розподілу даних і розподілу робочого навантаження, а також дозволяють нашій структурі ефективно виконувати програми з інтенсивним об’ємом даних у гетерогенних багатоядерних кластерах .

ВИСНОВКИ

Магістерська робота присвячена дослідженню моделей і методів управління гетерогенними багатоядерними кластерами та розробці адаптивної структури програмування для таких систем, з урахуванням їх асиметричної архітектури та специфічних можливостей.

Аналіз показав, що управління гетерогенними багатоядерними системами вимагає врахування відмінностей у продуктивності та архітектурних особливостях компонентів, що ускладнює розподіл навантаження та викликає потребу в адаптивних методах керування.

У другому розділі розроблено методи, що використовують фреймворк MapReduce для управління гетерогенними ресурсами. Визначено, що асиметрична архітектура потребує розширень для ефективної підтримки неоднорідних ресурсів. Було розглянуто моделі програмування, такі як функціонально-орієнтоване програмування та підхід Mixin-Layers, які дозволяють адаптуватися до динамічних змін у кластері та зменшити складність при розробці додатків для таких систем.

Третій розділ роботи зосереджений на розробці підходів до ефективного розподілу робочого навантаження з урахуванням можливостей окремих компонентів кластеру. Запропоновано алгоритм, що враховує стан планування, адаптивність та специфічні можливості прискорювачів, а також механізми динамічного масштабування робочих одиниць, що дозволяє забезпечити максимальну продуктивність.

У результаті роботи запропоновано методи адаптивного управління та розподілу навантаження, які забезпечують ефективність роботи гетерогенних кластерів, підвищують масштабованість системи та знижують витрати на обробку. Запропоновані рішення є корисними для реалізації паралельних і розподілених обчислень у складних гетерогенних середовищах, забезпечуючи гнучкість і продуктивність при мінімальному втручанні програмістів у процес управління ресурсами.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Jason Cross. A Dramatic Leap ForwardGeForce 8800 GT, Oct 2007. http://www.extremetech.com/article2/0,1697,2209197,00.asp.
2. Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph., 27(3):1–15, 2008.
3. Intel. Single-chip Cloud Computer, 2010. http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf.
4. S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In Proc. of the 32nd Annual International Symposium on Computer Architecture, pages 506–517, June 2005.
5. M. Hill and M. Marty. Amdahl's Law in the Multi-core Era. Technical Report 1593, Department of Computer Sciences, University of Wisconsin-Madison, March 2007.
6. M. Peric`as, A. Cristal, F. Cazorla, R. Gonz´alez, D. Jim´enez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques, pages 13–24, September 2007.
7. David Bader and Virat Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In Proc. of the 14th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science 4873, December 2007.
8. F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-CELL:Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In Proc. Of the 21st International Parallel and Distributed Processing Symposium, March 2007.

9. G. Buehrer and S. Parthasarathy. The Potential of the Cell Broadband Engine for Data Mining. Technical Report TR-2007-22, Department of Computer Science and Engineering, Ohio State University, 2007.
10. Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Cellsort: High performance sorting on the cell processor. In Proc. of the 33rd Very Large Databases Conference, pages 1286–1207, 2007.
11. Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized Data Processing on the Cell Broadband Engine. In Proc. of the Third International Workshop on Data Management on New Hardware, June 2007.
12. Fabrizio Petrini, Gordon Fossum, Juan Fern´andez, Ana Lucia Varbanescu, Michael Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In Proc. of the 21st International Parallel and Distributed Processing Symposium, pages 1–10, 2007.
13. Abadi, M., Barham, P., Chen, J., et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning." Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI).
14. Agarwal, S., Marathe, K., Rountree, B., et al. (2017). "A Framework for Automated Application Performance Tuning Using Machine Learning." Journal of Parallel and Distributed Computing, 103.
15. Al-Fares, M., Loukissas, A., & Vahdat, A. (2008). "A Scalable, Commodity Data Center Network Architecture." Proceedings of the ACM SIGCOMM Conference.
16. Azar, Y., Ben-Basat, R., Bremler-Barr, A., & Pedroni, E. (2019). "Optimal-Rate Bloom Filters and Adaptive Multi-Level Counting for Cluster Management." IEEE INFOCOM 2019 - IEEE Conference on Computer Communications.
17. Barroso, L. A., & Hölzle, U. (2007). The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Synthesis Lectures on Computer Architecture.

18. Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). "Algorithms for Hyper-Parameter Optimization." Advances in Neural Information Processing Systems (NIPS).
19. Binotto, A. P., et al. (2016). "Adaptive Framework for Energy-Efficient Heterogeneous Multi-Core Architectures." IEEE Transactions on Parallel and Distributed Systems.
20. Boettiger, C. (2015). "An Introduction to Docker for Reproducible Research." ACM SIGOPS Operating Systems Review, 49(1).
21. Buyya, R., Yeo, C. S., Venugopal, S., et al. (2009). "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility." Future Generation Computer Systems, 25(6).
22. Cao, J., et al. (2016). "An Adaptive Scheduling Framework for Real-Time Data Analytics in Heterogeneous Clusters." IEEE Transactions on Parallel and Distributed Systems, 27(11).
23. Dean, J., & Ghemawat, S. (2008). "MapReduce: Simplified Data Processing on Large Clusters." Communications of the ACM, 51(1).
24. Fang, B., Yan, J., & Zhou, N. (2015). "Adaptive Framework for Dynamic Resource Management in Cloud Computing." International Journal of Distributed Sensor Networks.
25. Gholami, M., & Lavesson, N. (2018). "An Adaptive Framework for Monitoring Heterogeneous Distributed Systems." Future Generation Computer Systems, 79.
26. Hellerstein, J., et al. (2004). "Adaptive Query Processing: Technology in Evolution." IEEE Data Engineering Bulletin.
27. Hsu, C., et al. (2017). "Heterogeneous Computing Architectures: A Perspective for Embedded System Designers." Proceedings of the IEEE, 105(5).
28. Huang, S., et al. (2019). "Towards an Adaptive Fault Tolerant Framework for Big Data Computing on Heterogeneous Clusters." ACM Transactions on Internet Technology.

29. Hwang, K., Dongarra, J., & Fox, G. (2012). Distributed and Cloud Computing: From Parallel Processing to the Internet of Things. Morgan Kaufmann.
30. Islam, S. M. R., et al. (2018). "Adaptive Management of Heterogeneous Resources in a Fog Computing Environment." IEEE Internet of Things Journal.
31. Isard, M., et al. (2007). "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks." EuroSys 2007.
32. Joukov, N., et al. (2006). "RAID-F: Time- and Energy-Efficient Fuzzy Dedupe for Hybrid Cloud Storage." USENIX Conference on File and Storage Technologies.
33. Kim, H., & Buyya, R. (2006). "Adaptive Framework for Efficient and Scalable Monitoring of Cloud Computing Environments." ACM Computing Surveys (CSUR).
34. Lee, E., & Messerschmitt, D. (1987). "Synchronous Data Flow." Proceedings of the IEEE, 75(9).
35. Maguluri, S., Tsitsiklis, J., & Zhong, Y. (2017). "Scheduling Algorithms for Heterogeneous Multi-core Servers with Synchronous and Asynchronous Processing." ACM Transactions on Modeling and Performance Evaluation of Computing Systems.
36. Mittal, S., & Vetter, J. S. (2015). "A Survey of Methods for Managing Power in High-Performance Computing." ACM Computing Surveys, 47(3).
37. Mohr, R. D., et al. (2020). "Dynamic Resource Scaling for Heterogeneous Clusters Using Adaptive Feedback Loops." Future Generation Computer Systems, 106.
38. Moreno, I., & Xu, J. (2011). "A Framework for Resource Allocation in Cloud Computing Using Machine Learning-Based Predictive Analytics." IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.

39. Nawab, F., et al. (2015). "Adaptive Multi-level Consistency: High Performance and Strong Consistency with Dynamic Adaptation." Proceedings of the VLDB Endowment.
40. Ozcan, R., et al. (2011). "Adaptive and Heterogeneous Query Processing on Multi-core Architectures." ACM SIGMOD International Conference on Management of Data.
41. Park, H., & Ruan, X. (2018). "An Adaptive Workload Management Framework for Heterogeneous Computing Clusters." IEEE Transactions on Parallel and Distributed Systems, 29(12).
42. Radovanović, Z., et al. (2015). "A Framework for Adaptive Energy Management in Heterogeneous Computing Systems." IEEE Transactions on Computers.
43. Ranganathan, P., & Jouppi, N. P. (2008). "Reconfigurable Datacenters for Next-Generation Cloud Services." Proceedings of the IEEE, 99(8).
44. Smith, J. E., & Nair, R. (2005). Virtual Machines: Versatile Platforms for Systems and Processes. Elsevier.
45. Subramaniam, S., & Paul, J. (2015). "Adaptive Load Balancing in Heterogeneous Distributed Systems." IEEE Transactions on Parallel and Distributed Systems, 26(11).
46. Verma, A., et al. (2015). "Large-Scale Cluster Management at Google with Borg." Proceedings of the European Conference on Computer Systems.
47. Xu, X., et al. (2017). "A Machine Learning-Based Adaptive Resource Management Approach in Heterogeneous Cloud Environments." IEEE Transactions on Cloud Computing.
48. Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: The architecture and performance of Roadrunner. In Proc. Supercomputing, 2008.
49. James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In Proceedings of the

2008 ACM/IEEE conference on Supercomputing, SC'08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press.

50. Duc Vianney, Gad Haber, Andre Heilper, and Marcel Zalmanovici. Performance analysis and visualization tools for cell/b.e. multicore environment. In IFMT'08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies, pages 1–12, New York, NY, USA, 2008. ACM.
51. Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. pages 1–15, 2008.
52. Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 287–296, Seattle, WA, March 2008.
53. Perry Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multi-threaded System. In Proc. of the 2007 ACM SIGPLAN Conference on Program