УДК 004.382.2

Малашонок Г. І., Сідько А. А.

РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ: ДАП-ТЕХНОЛОГІЯ РОЗПАРАЛЕЛЮВАННЯ РЕКУРСИВНИХ АЛГОРИТМІВ

Описано нову технологію динамічного розпаралелювання рекурсивних алгоритмів з розподіленою пам'яттю. Головним поштовхом до її створення було відкриття великого класу матричних рекурсивних алгоритмів за останні десятиліття. Усі вони, включно з алгоритмами В. Штрассена, є розвитком ідей, закладених у відомих працях А. А. Карацуби про швидке множення чисел і поліномів.

Основними об'єктами нової технології розпаралелювання є дроп, амін і пайн. Граф алгоритму розбивається на компактні підграфи (дропи), які можуть бути передані для обчислення на інші процесори. Для обчислення дропа проводиться розгортання його алгоритму і будується відповідна йому структура даних (амін). Для збереження всіх амінів і їх станів у кожному процесорі створюється список (пайн). Протягом обчислень, поки не завершено обрахунки конкретного аміна, цей амін зберігається в пайні. Після завершення обчислень уся пам'ять, відведена під амін, звільняється.

Іншою особливістю ДАП-технології є механізм розподілу дроп-завдань між процесорами. Для управління процесом розподілу завдань ми використовуємо список доступних дроп-завдань, з глибиною вкладеності для кожного завдання. Ми також використовуємо список дочірніх процесорів з інформацією про всі відправлені завдання та список вільних процесорів. Список дочірніх процесорів оновлюється щоразу під час відправлення завдання і отримання результатів обчислень. Список вільних процесорів ділиться порівну між усіма процесорами, які отримують завдання з найменшою глибиною вкладеності.

Для управління обчисленнями та операціями обміну даних створюються два потоки. Перший забезпечує обрахунок і відсилання результатів обчислень, а другий відповідає за управління і операції обміну даними. Диспетчерський потік є провідним, він контролює всі порти і стани, після чого засинає, щоб дати можливість іншому потоку виконувати обчислення.

Крім того, передбачено механізм звільнення процесора. Якщо у процесора немає дроп-завдань, то він додає себе в список вільних та пересилає цей список першому батьківському процесору. Такий процесор може отримати нове дроп-завдання, а також він може отримати результат від дочірнього процесора і продовжити обчислення відповідно до завдань активного аміна.

Для пояснення нової технології розпаралелювання наведено приклади побудови дроп-завдань для алгоритму обернення трикутної матриці і для алгоритму множення матриць.

Ключові слова: паралельне програмування, обчислювальний кластер, розподілена пам'ять, автоматичне розпаралелювання, динамічне розпаралелювання, DAP-розпаралелювання.

Вступ

Зацікавлення динамічними системами розпаралелювання обчислень на кластері з розподіленою пам'яттю виникло давно. Однією з перших у цьому класі систем була T-system, розроблена в Інституті програмних систем РАН під керівництвом С. А. Абрамова. Вона сьогодні відома за назвою OpenTS [5; 6]. За останні роки з'явилася велика кількість рекурсивних матричних алгоритмів, які становлять інтерес для побудови динамічних паралельних програм, тому що дають змогу ефективно керувати процесом розпаралелювання, розгортаючи рекурсії. Технології розпаралелювання таких алгоритмів на кластері з розподіленою пам'яттю активно розвиваються.

Перші підходи до розв'язання цієї задачі мали низку недоліків. Історично першим підходом була схема LLP-управління [3]. Вона передбачала наявність єдиного центру управління. Його основний недолік – це проблема «вузького горлечка» для інформаційних потоків у центрі управління.

Наступною була розроблена схема децентралізованого управління DDP [4; 1; 2]. У цій схемі на кожному процесорному вузлі створювався свій диспетчерський потік. Однак перші програми управління, отримані для цієї схеми, продемонстрували погані результати з сильно розрідженими даними. Це було пов'язано з тим, що не було передбачено можливості переходу до обчислення інших піддерев, поки не завершено обрахунок поточного піддерева. А в умовах,

© Малашонок Г. І., Сідько А. А., 2018

коли потрібно чекати результати від інших процесорів, це призводить до гальмування всього обчислювального процесу.

У цій роботі запропоновано нову схему динамічного управління обчислювальним процесом, у якій знято обмеження, що з'явилися в попередньому варіанті DDP-схеми. Процесори тепер не простоюють, очікуючи завершення обрахунку поточного обчислювального дерева, а беруть на виконання будь-які інші направлені їм піддерева. Нова версія DDP управління має назву ДАП-схеми (дроп-амін-пайн). Вона відрізняється тим, що послідовно розгортає функцію в глибину, зберігаючи всі стани на будь-якому рівні вкладеності доти, доки всі обчислення на поточному піддереві не будуть завершені. Це дає змогу будь-якому процесору вільно переключатись між піддеревами, не очікуючи завершення обрахунку на поточному піддереві.

Як і в попередніх рекурсивних схемах управління, ця схема передбачає декілька етапів розвитку обчислювального процесу для будь-якої рекурсивної функції. Перший етап – це розгортання рекурсивних обчислень від кореневої функції до листових, при цьому будується обчислювальне дерево, у вузлах якого розташовані рекурсивні функції. На другому етапі виконуються обчислення на листових вершинах. І третій етап – це згортання обчислювального дерева, з поверненням результатів обчислення в зовнішні функції. Цей етап завершується, коли результат буде обчислений на кореневій вершині обчислювального дерева.

1. Граф рекурсивного алгоритму

Граф рекурсивного алгоритму в розгорнутому вигляді можна зобразити так, щоб усі вершини першого рівня зображувалися в одній площині. Усі вершини наступного рівня рекурсії – на наступній паралельній площині і так далі. Передача даних у такому графі відбувається від вхідної вершини першого рівня до інших вершин першого рівня, а від них до вершин другого рівня і так далі до листових вершин, а потім у зворотному напрямі. Під час руху у зворотному напрямі, після повернення результатів, обчислення на вершині завершуються і звільняється пам'ять. Результат буде отриманий після завершення обчислень на кореневій вершині.

Нехай є обчислювальний кластер, який містить *n* процесорів, і потрібно організувати обчислювальний процес відповідно до деякого рекурсивного алгоритму. Кожен процесор може виконати обчислення в будь-якому компактному фрагменті графа алгоритму і, крім того, може передати будь-яку компактну частину зі своєї частини графа іншому процесору.

Важливим є те, що розгортання обчислювального процесу супроводжується передачею іншим процесорам фрагментів обчислювального дерева разом зі списком підлеглих їм вільних процесорів кластера. При цьому кожен процесор може отримати будь-яку з вершин дерева алгоритму і управляти всім обчислювальним процесом у цій вершині.

Отже, можна говорити про дві множини – множину вершин графа алгоритму і множину процесорів кластера. Завдання організації обчислювального процесу полягає в тому, щоб «прив'язувати» процесори до вершин дерева в процесі обчислень.

2. Приклад: алгоритм обернення трикутної матриці

Для ілюстрації наведемо рекурсивний алгоритм обернення нижньотрикутної матриці порядку 2^N, $(N>1)$. Нехай матриця A розбита на рівні квадратні блоки, і потрібно знайти обернену до неї матрицю B. Ці дві матриці пов'язані рівнянням: $AB = I$. Введемо позначення для блоків матриць:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ і } B = \begin{pmatrix} x & y \\ z & k \end{pmatrix}$$

З рівняння випливає, що мають виконуватись такі відношення: $b=0$; $y=0$, $x=a^{-1}$, $k=d^{-1}$ і $cx+dz=0$. Отже, $z=-kcx$. Алгоритм обернення такої трикутної матриці можна записати, наприклад, таким чином:

```
proc inv(A)
  While n>1 do
        { A→ (a,c,d) ; x=a^-1; k=d^-1; w=c*x; w=k*w;
          z=(-1) w; (x,z,k) → B; return B;}
  return [[1/A_11]];
```

Тут a^{-1} позначає виклик цієї ж процедури *inv*(a) для матриці вдвічі меншого розміру, а знаком «зірочка» позначається процедура *mult(A,B)* блочного множення матриць, наприклад, така:

```
proc mult(A,B){
  A→ (a,b,c,d); B→ (l,m,q,p) ;
  While n>2 do
        {u1=a*l; v1=b*q; w1=u1+v1; u2=a*m; v2=b*p; w2=u2+v2;
         u3=c*l; v3=d*q; w3=u3+v3; u4=c*m; v4=d*p; w4=u4+v4;
         (w1,w2,w3,w4) → C; return C;}
  (al+bq, am+bp, cl+dq, cm+dp) → C; return C;}
```

Зазначимо, що це загальний блочний алгоритм множення матриць. Якщо ж множаться дві трикутні матриці або ж матриці, в яких тільки перший множник трикутний або тільки другий множник трикутний, то аналогічно можна записати ще три варіанти для блочного множення матриць: *multLL(A,B)*, *multLS(A,B)*, *multSL(A,B)*, відповідно. Якби ми хотіли охопити ще й всі випадки з верхньотрикутними матрицями, то записали б ще три подібних варіанти: *multUU(A,B)*, *multUS(A,B)*, *multSU(A,B)* і два змішаних, коли множаться два різних види трикутних матриць: *multLU(A,B)*, *multUL(A,B)*. Разом 9 різних видів для множення повних і трикутних матриць. Можна, звичайно, визначити одну загальну процедуру та передавати їй ключ із 9 значень, щоб вибирати потрібний варіант.

Поділ рекурсивної частини обчислень від послідовної ми визначили умовами $n > 1$ і $n > 2$. У реальних обчисленнях тут може бути будь-яка умова. Розмір блоку, за якого потрібно переходити до послідовного алгоритму обчислення, залежить від багатьох факторів, і в загальному випадку він повинен налаштовуватися виходячи з реального обладнання і його фізичних характеристик. Розглянемо відповідні цим рекурсивним алгоритмам обчислювальні графи (див. рис. 1 і 2).

Кожен граф має одну вхідну вершину і одну вихідну вершину. На вхідну вершину надходить вектор з вхідними даними inData, і в цій вершині обчислюється «вхідна функція» inF. У вихідній

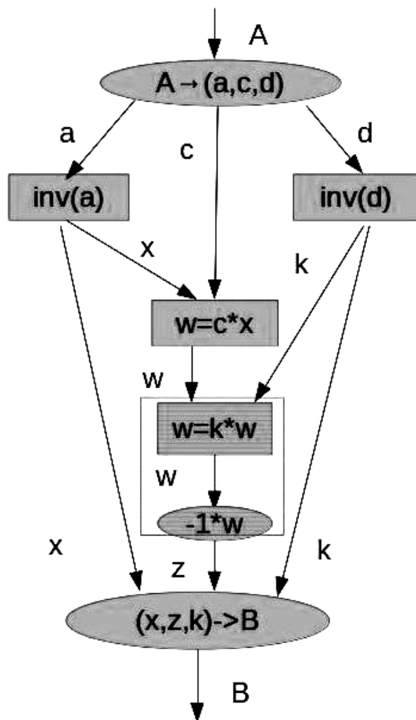

Рис. 1. Граф алгоритму обернення трикутної матриці

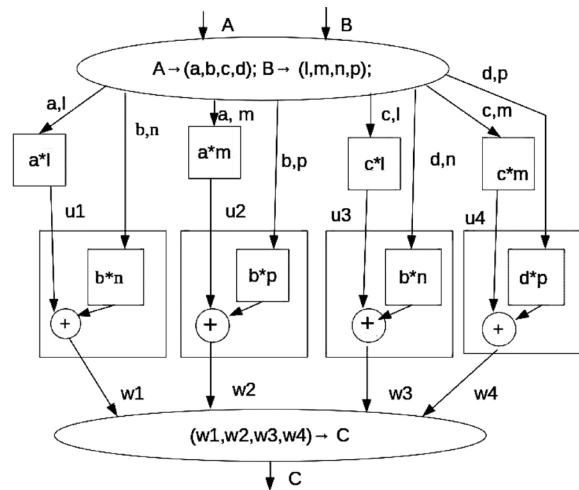

Рис. 2. Граф алгоритму блочного множення матриць

вершині функція outF обчислює вектор з результатом обчислень outData. У цих прикладах вхідна функція виконує поділ матриць на блоки, а вихідна збирає з блоків результат обчислення. Усі інші вершини здійснюють обрахунки згідно з рекурсивним алгоритмом. Усі дуги в графах орієнтовані в напрямку передачі даних від вхідної вершини до вихідної.

3. Дропи, аміни, пайни

Для того щоб керувати рекурсивними обчисленнями і забезпечити передачу обчислювальних функцій на інші процесори, ми розіб'ємо весь обчислювальний граф на окремі компактні підграфи (дропи). Для цього спочатку виділимо всі вершини обчислювального графа, які обраховуються рекурсивно (R-вершини). Такі вершини повинні мати високу, як мінімум нелінійну, складність обчислення у відношенні до обсягу даних. Очевидно, що лінійні за складністю обчислювальні блоки не можна передавати на виконання іншим процесорам, тому що час передачі – це не менше, ніж лінійна функція від обсягу даних.

У кожен дроп-підграф входить одна R-вершина і, можливо, інші вершини обчислювального графа, в які можна потрапити з цієї R-вершини шляхом, що не проходить через інші R-вершини. Ми розглядаємо тільки направлені шляхи, тобто шляхи в напрямку передачі даних. Вибір розбиття обчислювального графа на дроп може бути неоднозначним. Під час розбиття можна керуватися правилом не створювати зайві види дропів, а мати якомога меншу кількість різних видів. Головна особливість дропа полягає в тому, що це обчислювальний підграф, який можна передати на інший обчислювальний процесор.

На рис. 1 і 2 вершини, об’єднані в один дроп, обведено квадратним контуром.

На рис. 1 це три види дропів: *inv*, *A*B* і *-A*B*. Тут *inv* позначає обернення невиродженої трикутної матриці. На рис. 2 це два види дропів: *A*B* і *A*B+C*.

Отже, ми визначаємо дропи як найменші компоненти обчислювального графа, які можуть бути передані на інші процесори. Один із дропів відповідає всьому обчислювальному графу: для задачі обернення трикутних матриць – це *inv*, а для задачі множення матриць – це *A*B*. Всього ж треба мати чотири різних види дропів для повного покриття цих двох обчислювальних графів: *inv, A*B, -A*B, A*B+C*.

Якщо розгортання обчислення рекурсивної функції здійснюється на одному й тому самому процесорі, то потрібно зберігати в пам’яті вхідні і вихідні дані для всіх рівнів рекурсії. Всі дропи, що є складовими обчислювального графа алгоритму на одному рівні, зберігаються в одному масиві дропів, який називається амін. З іншого боку, кожен амін – це розгорнутий стан деякого дропа.

Наприклад, амін *A*B* складається з 4 дропів *A*B*, 4 дропів *A*B+C*, однієї вхідної і однієї вихідної функцій. Амін *inv* складається з 2 дропів *inv*, 1 дропа *A*B*, 1 дропа *-A*B*, однієї вхідної та однієї вихідної функцій.

Всі аміни, які формуються на одному процесорі, зберігаються в загальному списку, який має назву пайн. Цей список зростає з формуванням нових амінів. Після завершення всіх обчислень в аміні і повернення результату амін видаляється з пам’яті, але при цьому всі інші аміни не змінюють своїх порядкових номерів у поточному пайні.

Це дає змогу легко вести облік усіх обчислювальних задач під час організації розподілених обчислень. Кожна обчислювальна задача (дроп) має певну адресу приписки, яка визначається номером процесора (np), номером аміна (na) в списку пайна цього процесора і номером дропа (nd) в цьому аміні. Це адреса, на яку треба повернути результат обчислення поточного дропа: *pad* = [np, na, nd]. Обрахунок цього дропа може здійснюватися на будь-якому процесорі. Це обчислення передбачає, що буде побудований відповідний йому амін і розміщений у деякому пайні. В адресному полі такого аміна має бути вказаний *pad* для відправлення результатів обчислень.

3.1. Основні поля дроп-об’єкта

type – тип дропа (унікальний номер у списку всіх типів дропів).

myAmine – поле, що вказує на амін поточного дропа.

InData і **outData** – вектори для вхідних і вихідних даних.

NumberOfMainComponents – кількість головних компонент в InData, якщо значення цих компонент уже відомі, то можна створювати амін.

daughterProc – номер процесора, на якому обчислюється поточний дроп.

state – стан обчислювального процесу в поточний момент часу:

- не готові вхідні дані для початку обчислень;
- готові вхідні дані, можна починати обчислення;
- відбувається обчислення на поточному процесорі;
- відбувається обчислення на іншому процесорі і додаткові процесори не потрібні;
- відбувається обчислення на іншому процесорі і потрібні додаткові процесори;
- обчислення завершено та записано в **outData**.

3.2. Основні поля амін-об’єкта

pad = (np, na, nd) – адреса, на яку потрібно повернути результат обчислень поточного дропа.

type, inData, outData, NumberOfMain Components – ті самі, що і в дропа.

outFunctionInData – вектор для вхідних даних вихідної функції аміна.

level – глибина рекурсії цього аміна.

amineState – стан обчислення: 0 – відбувається процес обчислення; 1 – відбувається процес обчислення, але додаткові процесори не потрібні; 2 – обчислення завершено.

arcs – топологія графа аміна: в i-му рядку записано зв’язки для кожної компоненти вихідного вектора i-го дропа (номер компоненти вихідного вектора, номер дропа і номер компоненти вхідного вектора в цьому дропі). Кількість рядків дорівнює кількості дропів в аміні, плюс ще нульовий рядок для вихідного вектора вхідної функції аміна.

drop – масив усіх дропів поточного аміна.

3.3. Основні поля пайн-об’єкта

body – список усіх амінів, які обчислювались на цьому процесорі.

pineState – стан обчислень: 0 – відбувається процес обчислення та додаткові процесори потрібні; 1 – відбувається процес обчислення, але додаткові процесори не потрібні; 2 – обчислення завершено.

3.4. Основні функції дроп-об’єкта

Функція **isItLeaf()** оцінює обсяг обчислень для вхідних даних і повертає true, якщо обчислення не потрібно розпаралелювати.

Функція **calcLeafDrop()** обчислює поточний дроп у послідовному режимі.

3.5. Основні функції амін-об'єкта

Функція **inputFunction()** приймає вхідні дані аміна **inData** і формує вхідні дані для дропів.

Функція **outputFunction()** приймає дані від дропів і формує вихідні дані аміна на векторі **outData**.

4. Організація обчислювальних потоків

Ми будемо використовувати два потоки: обчислювальний потік і диспетчерський потік. Ці потоки буде запущено на кожному процесорі кластера. Обчислювальний потік відповідає за виконання обрахунку згідно з обчислювальним алгоритмом. Диспетчерський потік відповідає за всі функції управління обчисленнями, включно з прийомом усіх вхідних повідомлень.

4.1. Обчислювальний потік CalcThread

Обчислювальний потік перебуває в стані циклу, він очікує надходження дроп-завдання і виконує відповідні обрахунки.

Об'єкти обчислювального потоку:

pine – список амінів на поточному процесорі;

availableDropTasks – масив зі списків доступних дроп-завдань з однаковою глибиною рекурсії;

flagOfExistDropTasks – змінна, яка повідомляє, що список доступних дроп-завдань непорожній;

currentDrop – містить дані про поточний дроп: **pad, type, inData;**

stopCalcThread – змінна для зупинки обчислювального потоку;

flagCalcGo – змінна, яка повідомляє, що відбувається процес обчислення;

flagOfReadyDataOutF – змінна готовності даних для вихідної функції аміна;

requestOfMoreProc – змінна запиту додаткових процесорів.

Функції обчислювального потоку

Метод **stopCalcThread()** – встановлює змінну для зупинки обчислювального потоку.

Метод **isCalc()** – повідомляє, чи відбувається процес обчислення.

Метод **setAmine()** – приймає дроп-завдання, створює амін і встановлює змінну isCalc.

Метод **writeResultsToAmine()** – результати обчислень дропа записуються в його амін у вектори вхідних даних інших дропів. Ці дропи перевіряються на готовність до виконання і заносяться у свій список **availableDropTasks** у разі готовності. Встановлюється змінна **isReadyDataOutF**.

Метод **runCalcThread()** – основний метод обчислювального потоку. Якщо **availableDropTasks** порожній і немає дроп-завдань від інших процесорів, то виконується порожній цикл. Після надходження дроп-завдання виконується **setAmine()** і створюється амін у списку пайн, якщо надійшло нелистове завдання. Запускається обчислення вхідної функції, потім усі доступні дропи записуються в **availableDropTasks**. З цього списку вибирається дроп із найбільшою глибиною рекурсії, і створюється новий амін. Якщо ж функція дропа **isItLeaf()** повернула true, то **culcLeafDrop()** обчислює поточний дроп у послідовному режимі. Після чого **writeResultsToAmine()** записує результат у наступні дропи аміна. Доступні дропи записуються в **availableDropTasks**, і якщо змінна **isReadyDataOutF** дорівнює true, то відбувається завершення обчислення поточного аміна, а результат відправляється на pad-адресу аміна. Якщо це адреса на іншому процесорі, то виконується команда міжпроцесорної пересилки. Якщо це адреса у своєму пайні, то **writeResultsToAmine()** записує результат у наступні дропи аміна на pad-адресу і всі доступні дропи записуються в **availableDropTasks**.

4.2. Диспетчерський потік DispThread

Диспетчерський потік відповідає за відправлення завдання, прийом завдання та результатів, обмін вільними процесорами, запис результатів у пайн та додавання доступних дроп-завдань у список. Він реалізований за допомогою нескінченного циклу та очікує, доки не буде сигналу на завершення всієї програми. Диспетчерський потік засинає на деякий час, щоб у роботу увійшов обчислювальний потік. Потім просинається і продивляється всі порти, чи щось надійшло, та займається розсилкою завдань та вільних процесорів. Кожне повідомлення, надіслане чи відправлене іншим процесорам, має своє значення tag, відповідно до якого виконується та чи та робота:

- 0: повідомлення містить дроп-завдання;
- 1: повідомлення містить вільні процесори;
- 2: повідомлення містить стан процесора;
- 3: повідомлення містить результат дроп-завдання;
- 4: повідомлення містить команду на завершення (вся задача порахована).

Роботу диспетчерського потоку можна умовно розбити на 9 процесів:

1) очікування сигналу завершення;
2) прийом завдання;

3) прийом вільних процесорів;
4) прийом та запис стану дочірнього процесора;
5) прийом результату надісланого дропа та запис цих результатів у наступні дропи;
6) прийом неголовних компонент та їхній дозапис у потрібне місце;
7) відправлення доступних завдань вільним процесорам (якщо є завдання і процесори);
8) відправлення вільних процесорів дочірнім (якщо немає доступних дроп-завдань, але є вільні та дочірні процесори);
9) відправлення всього списку вільних процесорів, разом із поточним, батьківському процесору (якщо немає дочірніх процесорів та доступних завдань).

4.3. Поля диспетчерського потоку

Pine – вказівник на пайн.

SleepTime – час, на який засинає диспетчерський потік.

ExecuteTime – час, скільки працював потік.

StopThread – булівська змінна, яка сигналізує про завершення програми.

FreeNodes – список вільних процесорів, його може мати кожен процесор. Вільні процесори надсилає батьківський або повертає дочірній, якщо йому більше не потрібні. Тільки вільним процесорам надсилаються завдання. Процесор є вільним, коли йому ще не надсилалось завдання або коли він повернув результати завдань.

SubNodes – список дочірніх процесорів, ключ-значення, де ключ – це номер дочірнього процесора, а значення – цілочисельний масив, у якому завжди на першій позиції стоїть стан flagOfMyState процесора, а потім ідуть пари – номер аміна та номер дропа, з яких було надіслано завдання цьому дочірньому процесору. Таких пар може бути декілька. Щоразу, як дочірній процесор повертає результат, пара, яка відповідає порахованому завданню, видаляється. Якщо видалено всі такі пари, то цей процесор з дочірнього списку видаляється. Процесор є дочірнім, якщо йому надіслано хоча б одне завдання.

4.4. Головний метод диспетчерського потоку

Execute() – головний метод диспетчерського потоку, де відбуваються всі процеси:

1. Запускаємо диспетчерський потік.

2. Встановлюємо йому максимальний пріоритет.

3. Створюємо об'єкт обчислювального потоку CalcThread(Pine p).

4. Якщо поточний процесор нульовий:

4.1. Додаємо всі процесори у список вільних. На початку всі процесори є вільними, а нульовий процесор є головним.

4.2. Передаємо обчислювальному потоку початкове завдання.

5. Запам'ятовуємо поточний час в **executeTime**.

6. Запускаємо нескінченний цикл.

6.1. Починаємо продивлятись усі отримані повідомлення. Спершу дивимось, чи не прийшло повідомлення на завершення програми, якщо так, завершуємо диспетчерський потік та фіналізуємо.

6.2. Перевіряємо, чи надійшло завдання. Передаємо його обчислювальному потоку **setAmine()**. І якщо номер поточного процесора був у списку вільних, то видаляємо його, бо, поки він рахує, йому вже не можна надсилати завдання.

6.3. Дивимось, чи прийшло повідомлення з вільними процесорами, якщо прийшло, то додаємо їх у свій список. Вільні процесори можуть прийти від батьківського процесора на допомогу або повернутись від дочірнього, якщо вони йому не потрібні.

6.4. Якщо прийшов стан дочірнього процесора:

6.4.1. Записуємо цей стан у список дочірніх процесорів **SubNodes** на нульову позицію масиву.

6.4.2. Записуємо цей стан у відповідний дроп, який було надіслано.

6.4.3. Перевіряємо, чи не змінився після цього стан pine поточного процесора, якщо змінився, то ставимо pine відповідний стан.

6.5. Перевіряємо, чи прийшов результат надісланого дроп-завдання.

6.5.1. Записуємо отриманий результат у відповідний дроп та ставимо йому стан 2 (пораховано).

6.5.2. Якщо в масиві відповідного дочірнього процесора, який надіслав результат, записана лише одна пара амін-дроп, то видаляємо його зі списку дочірніх та додаємо в список вільних.

6.5.3. Якщо в масиві більше ніж одна пара амін-дроп, то знаходимо ту пару, яка відповідає завданню, та видаляємо її з масиву.

6.5.4. Записуємо результати порахованого дропа в амін та записуємо в **IsReadyDataOutF**, чи готові дані, для вихідної функції.

6.5.5. Знаходимо амін, для дропа якого було надіслано результат.

6.5.6. Тепер може виникнути ситуація, коли всі дані будуть готові, для вихідної функції, тоді треба виконати цю функцію та її результат записати в результат аміна. І так може повторюватись декілька разів.

6.5.7. Поки змінна **isReadyDataOutF** дорівнює true:

Виконуємо вихідну функцію **outputFunction()** та записуємо її результат у масив вихідних даних аміна. Якщо завдання для цього аміна надіслав цей процесор:

Записуємо у вихідний масив дропа, з якого було створено порахований амін, вихідні дані цього аміна. Заносимо отримані результати в дропи цього аміна та виконуємо **isReadyDataOutF()**, якщо це можливо. (Виконуємо ті самі дії, поки не дійдемо до аміна, в якого не готові дані для вихідної функції, або завдання для цього аміна надіслано від іншого процесора).

6.5.8. Якщо ми дійшли до аміна нульового процесора на нульовій позиції в pine, то це означає, що всю задачу пораховано, тому ставимо змінній **stopThread** значення true, розсилаємо всім процесорам сигнал на завершення та зупиняємо обчислювальний потік.

6.5.9. В іншому випадку надсилаємо результат аміна, до якого ми дійшли, процесору, який нам надіслав завдання для цього аміна.

6.6. Якщо є вільні процесори і доступні завдання:

6.6.1. Дивимось, чи поточний процесор перебуває в режимі обрахунку, чи **culcThread. isCalc()** дорівнює true.

6.6.1.1. Якщо в режимі очікування завдання, то додаємо його в список вільних процесорів.

6.6.2. Ділимо вільні процесори на доступні завдання.

6.6.3. Надсилаємо завдання вільним процесорам, починаючи з нульової глибини рекурсії.

6.6.3.1. Видаляємо надіслане завдання зі списку доступних та встановлюємо йому **subNodes**.

6.6.3.2. Видаляємо процесор зі списку вільних.

6.6.3.3. Якщо процесор, якому щойно надіслано завдання, є в списку дочірніх процесорів, додаємо йому пару номер аміна – номер дропа **subNodes**, який йому відправили.

6.6.3.4. Якщо цього процесора немає в дочірніх, то додаємо його зі станом 0 та однією парою номер аміна – номер дропа в **subNodes**.

6.6.4. Якщо процесорів більше, ніж доступних дроп-завдань, із завданням може бути надіслано список вільних процесорів:

6.6.4.1. Ідемо циклом по доступних завданнях.

6.6.4.1.1. Беремо кількість процесорів, яка припадає на 1 завдання, записуємо в масив **subNodes**.

6.6.4.2. Видаляємо їх зі списку вільних.

6.6.4.3. Відсилаємо першому процесору зі списку, який ми взяли, завдання і список процесорів, які залишилися в цьому списку як вільні процесори.

6.6.4.4. Додаємо процесор, якому було надіслано завдання, або у список дочірніх, або вже знаходимо його там, та дописуємо йому координати надісланого дропа.

6.6.4.5. Видаляємо відправлений дроп зі списку доступних завдань **availableDropTasks**.

6.7. Якщо в поточного процесора є дочірні та вільні процесори, але при цьому список завдань порожній:

6.7.1. Якщо процесор не в режимі обрахунку, додамо його в список вільних.

6.7.2. Надсилати вільні процесори будемо тільки тим дочірнім, у яких стан 0, тобто яким зараз потрібні вільні процесори.

6.7.3. Якщо вільних процесорів більше, ніж тих, що їх потребують, тоді ділимо порівну всі вільні процесори та відправляємо кожному дочірньому, який у стані 0, відповідну кількість вільних, при цьому надіслані процесори видаляються зі списку вільних.

6.7.4. Якщо вільних процесорів виявилося менше, ніж дочірніх, що їх потребують, тоді по черзі відправляємо по одному вільному процесору дочірнім, у стані 0.

6.8. Якщо поточний процесор не перебуває в режимі обрахунку та не має дочірніх процесорів та доступних завдань, це означає, що йому не потрібні його вільні процесори та він може взяти собі інше завдання. Він додає себе в список вільних та відправляє цей список деякому батьківському процесору.

Батьківським процесором вважається той, який надіслав деяке завдання, яке ще обчислюється. Оскільки завдання йому може надіслати будь-хто, в процесора може бути декілька батьківських процесорів.

Батьківський процесор, якому можна надіслати вільні процесори, він шукає так: ідемо циклом по pine знизу вверх та шукаємо перший ненульовий амін, і якщо завдання для нього надіслав не поточний процесор, то беремо цей номер процесора **amine.parentProc** та надсилаємо йому свій список вільних разом із собою. Відсилати свій список вільних процесорів, якщо він більше не потрібен, батьківському потрібно для того, щоб він зміг віддати ці вільні процесори тому, хто перебуває в стані обрахунку, щоб пришвидшити процес обчислення всієї задачі.

6.9. На цьому кроці, зробивши всю необхідну роботу, диспетчерський потік засинає на деякий час **sleepTime**, віддаючи керування обчислювальному потоку. Після того як диспетчерський процесор прокинеться, він почне цикл спочатку та виконуватиме ті самі процеси, доки не буде вся задача обчислена та не надійде сигнал на завершення.

Висновки

Ми надали опис універсальної динамічної схеми розпаралелювання рекурсивних алгоритмів на кластері з розподіленою пам'яттю, опис основних об'єктів, їхніх полів та функцій, а також пояснили роботу двопотокової системи, яка запускається на кожному ядрі кластера. Її можна застосовувати для будь-яких блочно-рекурсивних алгоритмів, як із щільними, так і з розрідженими матрицями.

Цю схему було реалізовано мовою програмування Java, з використанням пакетів OpenMPI та MathPartner, а її роботу ми перевірили на наведених вище матричних алгоритмах.

Ми плануємо провести докладне експериментальне дослідження ефективності цієї схеми і опублікувати результати експериментів.

Список використаної літератури

1. Ильченко Е. А. Об одном алгоритме управления параллельными вычислениями с децентрализованным управлением / Е. А. Ильченко // Вестник Тамбовского университета. Серия: Естественные и технические науки. – 2013. – Т. 18, вып. 4. – С. 1198–1206.
2. Ильченко Е. А. Об эффективном методе распараллеливания блочных рекурсивных алгоритмов / Е. А. Ильченко // Вестник Тамбовского университета. Серия: Естественные и технические науки. – 2015. – Т. 20, вып. 5. – С. 1173–1186.
3. Малашонок Г. И. Организация параллельных вычислений в рекурсивных символьно-численных алгоритмах / Г. И. Малашонок, Ю. Д. Валеев // Труды конференции ПаВТ'2008 (Санкт-Петербург). – Челябинск : Изд-во ЮУрГУ, 2008. – С. 153–165.
4. Малашонок Г. И. Управление параллельным вычислительным процессом / Г. И. Малашонок // Вестник Тамбовского университета. Серия: Естественные и технические науки. – 2009. – Т. 14, вып. 1. – С. 269–274.
5. Abramov S. T-system: programming environment providing automatic dynamic parallelizing on IP-network of Unix-computers / S. Abramov, A. Adamovitch, M. Kovalenko // Report on 4-th International Russian-Indian seminar and exhibition. Sept. 15–25. Moscow, 1997. – P. 15–25.
6. Abramov S. OpenTS: An Outline of Dynamic Parallelization Approach / S. Abramov, A. Adamovich, A. Nyukhin, A. Moskovsky, V. Roganov, E. Shevchuk, Yu. Shevchuk, A. Vodomerov // Malyshkin V. (Ed.). PaCT 2005: International Conference on Parallel Computing Technologies. – Berlin, Heidelberg : Springer, 2005. – P. 303–312.

Gennadi Malaschonok, Alla Sidko

DISTRIBUTED COMPUTING: DAP-TECHNOLOGY FOR PARALLELIZING RECURSIVE ALGORITHMS

This article addresses the description of a new technology of dynamic parallelization of recursive algorithms on distributed memory. The main impetus for its creation was the discovery of a large class of matrix recursive algorithms in recent decades. All of them, including the Strassen algorithms, are the development of the ideas of Karatsuba's well-known works on the rapid multiplication of numbers and polynomials. The main objects of the new parallelization technology are drop, amine, and pine. The algorithm graph is divided into compact subgraphs (drops) which can be transferred to other processors for calculation. To calculate the drop, the algorithm is unfolded and the corresponding data structure (amine) is constructed. To save all the amines and their states, a list (pine) is created in each processor. During the calculations, until the calculation of a particular amine is completed, this amine is retained on the pine. After completing the calculations, all the memory allocated for the amine is released.

Another feature of ADP-technology is the mechanism of distribution of drop-orders by processors. To manage the process of job allocation, we use a list of available drop jobs, with the depth of nesting for each job. We also use the list of child processors with information about all sent tasks, as well as a list of free processors. This list of child processors is updated each time the job is sent and when the results of the calculations are received. The list of free processors is divided equally among all the processors that receive the task with the lowest depth of nesting.

To manage calculations and data exchange operations, two threads are created. The first will provide calculations and sending out the results of calculations, and the second will take care of all the management and of all the data exchange. The control thread is the master; it controls all the ports and states and falls asleep to allow the second thread to perform calculations.

In addition, there is a mechanism for releasing the processor. If the processor does not have drop jobs, it adds itself to the list of free processors and forwards this list to the first parent processor. Such a processor can receive a new drop-task, and it can also get the result from the child processor and continue the calculations in accordance with the tasks of the active amine.

To explain the new parallelization technology, we provide examples of the construction of drop tasks for the inversion algorithm of the triangular matrix and for the matrix multiplication algorithm.

Keywords: parallel programming, computational cluster, distribute memory, automatic parallelization, dynamic parallelization, DAP-parallelization.

Матеріал надійшов 10.06.2018