УДК 519.6

А. І. Кардаш, к. фіз-мат. н., доц.;
С. М. Левицька;
А. Т. Дудикевич, к. фіз-мат. н., доц.

РОЗПАРАЛЕЛЮВАННЯ ОБЧИСЛЕНЬ НА КЛАСТЕРАХ

Досліджено ефективність розпаралелювання обчислень на кластерах на прикладі алгоритму методу Жордана—Гауса розв'язування систем лінійних алгебраїчних рівнянь. Запропоновано практичні поради доцільності розпаралелювання обчислень.

1. Система розділеного програмування MPI

Тепер на зміну дуже дорогим традиційним машинам з невеликою кількістю процесорів приходять відносно недорогі кластери (ферми), які дуже легко (і головне дешево) наростити. Кластер — це мультикомп'ютер, який складається з багатьох окремих комп'ютерів (вузлів) зв'язаних між собою єдиною комунікаційною системою. Кожен вузол має власну оперативну пам'ять. Спільної пам'яті для них не існує. Це породжує масу проблем, пов'язаних з різним представленням даних і роботою з файлами. Переважно в такі ферми об'єднують тисячі процесорів. Потенційний виграш і доступність (багато обчислювальних центрів виставляє такі ресурси для загального користування) викликає бажання в користувачів перенести свої програми на такі ферми. Проте в більшості таке бажання зникає, коли вони дізнаються, що їм доведеться серйозно попрацювати над своїми програмами для їх розпаралелювання.

Тим, хто все-таки не відмовився від такої ідеї, дається на вибір два підходи до розпаралелювання. Перший, найлегший в реалізації, полягає в тому, що користувач лише вказує компілятору в яких ділянках програми можлива паралельна обробка даних, а в яких дії повинні виконуватись послідовно. Такий підхід базується на використанні систем зі спільною пам'яттю. Тому при роботі з кластерами не використовується. Другий підхід набагато складніший: користувач повинен сам, вручну, змінити код програми, явно вказуючи як розподіл обчислень між процесорами, так і міжпроцесорну комунікацію. Зазвичай, поняття «процесор» підміняють поняттям «процес». Така заміна дозволяє працювати з віртуальними об'єктами, які можуть виконуватись як на різних процесорах, так і на одному.

З такою схемою паралельної роботи програміст повинен вручну організовувати зв'язок між процесами, що призводить до ускладнення коду програми і є джерелом помилок. Дуже корисним в даній ситуації був би автоматичний засіб. Але, нажаль, такий засіб є неможливим (принаймні на сьогодні) з декількох причин.

По-перше, час виконання найпростішої операції взаємодії між процесами через комунікаційну систему занадто великий, порівняно з часом виконання найпростішої команди процесора. Тому дані повинні пересилатись між процесорами великими порціями. А це потребує аналізу якщо не алгоритму загалом, то, принаймні, великих, логічно завершених його частин.

По-друге, розділення даних між процесами потребує додаткового контролю за коректністю доступу до даних.

По-третє, потрібно контролювати синхронність обміну даних. Навіть невеликі простої у циклах можуть призвести до сповільнення роботи програми. Потрібно визначити оптимальне співвідношення між величиною порцій інформації, що пересилається і можливими простоями для загального прискорення роботи програми. Дана умова є дуже важливою. Тому, розробляючи програми, необхідно проаналізувати програму взагалі на предмет мінімізації часу, що затрачається на міжпроцесорну організацію.

Враховуючи всю складність розпаралелювання, перед програмістом постає ключове запитання — доцільності такої оптимізації:

— Якщо програма легко розпаралелюється, то чому б цього не зробити?
— Якщо програма не надто важлива, то чи варто витрачати час?
— Якщо програма просто незамінна, то можливо варто спробувати?

© А. І. Кардаш, С. М. Левицька, А. Т. Дудикевич, 2009

Хоча бажання (чи потреби) достатньо для написання програми, це зовсім не означає, що паралельний варіант програми буде корисним.

Якраз автори досліджують ефективність розпаралелювання на кластерах за допомогою реалізації стандарту MPI для окремих класів задач [1, 2].

2. Постановка демонстраційної задачі

Нехай маємо систему з n лінійних алгебраїчних рівнянь з n невідомими

$$\begin{cases} a_{1,1}\cdot x_1 + a_{1,2}\cdot x_2 + \ldots\ldots\ldots\ldots\ldots + a_{1,n}\cdot x_n = b_1; \\ a_{2,1}\cdot x_1 + a_{2,2}\cdot x_2 + \ldots\ldots\ldots\ldots\ldots + a_{n,n}\cdot x_n = b_1; \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n-1,1}\cdot x_1 + a_{n-1,2}\cdot x_2 + \ldots\ldots\ldots\ldots\ldots + a_{n-1,n}\cdot x_n = b_{n-1}; \\ a_{n,1}\cdot x_1 + a_{n,2}\cdot x_2 + \ldots\ldots\ldots\ldots\ldots + a_{n,n}\cdot x_n = b_n. \end{cases} \quad (1)$$

Виникає задача пошуку такого набору чисел

$$x_1,\ x_2,\ldots x_{n-1},\ x_n, \quad (2)$$

які, підставляючи в (1), перетворюють дану систему в тотожність. Розв'язок (2) шукається методом Жордано–Гауса. Ідея методу полягає у вилученні i-ї змінної з усіх рівнянь крім i-го, шляхом лінійного додавання і домноження рівнянь на константи. Метод працює n кроків. На першому кроці перше рівняння ділиться на коефіцієнт при першій змінній (якщо коефіцієнт рівний нулю, то рівняння потрібно переставити з будь-яким іншим, в якого перший коефіцієнт не рівний нулю). Від усіх рівнянь, крім першого, віднімається перше, помножене на коефіцієнт при першій змінній цього рівняння. В результаті виконання першого кроку система (1) набуде вигляду:

$$\begin{cases} x_1 + a_{1,2}^{(1)}\cdot x_2 + a_{1,3}^{(1)}\cdot x_3 + \cdots\cdots\cdots + a_{1,n}^{(1)}\cdot x_n = b_1^{(1)}; \\ a_{2,2}^{(1)}\cdot x_2 + a_{2,3}^{(1)}\cdot x_3 + \cdots\cdots\cdots + a_{2,n}^{(1)}\cdot x_n = b_2^{(1)}; \\ \\ a_{n,2}^{(1)}\cdot x_2 + a_{n,3}^{(1)}\cdot x_3 + \cdots\cdots\cdots + a_{n,n}^{(1)}\cdot x_n = b_n^{(1)}. \end{cases}$$

Усі рядки, крім першого, на першій позиції містять нуль. На другому кроці виконуються усі вищенаведені дії, але з другим рядком. Оскільки на першій позиції в усіх рядках, крім першого, містяться нулі, то в результаті другого кроку отримаємо систему, в якій в усіх рядках, крім першого, на першій позиції міститься нуль, і в усіх рядках, крім другого, на другій позиції міститься нуль.

$$\begin{cases} x_1 + 0\cdot x_2 + a_{1,3}^{(2)}\cdot x_3 + \cdots\cdots\cdots + a_{1,n}^{(2)}\cdot x_n = b_1^{(2)}; \\ x_2 + a_{2,3}^{(2)}\cdot x_3 + \cdots\cdots\cdots + a_{2,n}^{(2)}\cdot x_n = b_2^{(2)}; \\ \\ a_{n,3}^{(2)}\cdot x_3 + \cdots\cdots\cdots + a_{n,n}^{(2)}\cdot x_n = b_n^{(2)}. \end{cases}$$

Наступні кроки аналогічні попереднім з відповідним рядком. Тому, в результаті виконання алгоритму, вихідна система зведеться до вигляду:

$$\begin{cases} x_1 = b_1^{(n)}; \\ x_2 = b_2^{(n)}; \\ \cdots\cdots\cdots \\ x_n = b_n^{(n)}. \end{cases}$$

Вектор вільних членів буде розв'язком задачі.

3. Паралельна реалізація методу

Запишемо задачу у матричній формі $Ax = b$, де A — матриця коефіцієнтів, x — вектор змінних, b — вектор вільних членів. Суть паралельного варіанту методу полягає в тому, що для віднімання на *i*-му кроці *i*-го рядках від інших не потрібно жодної інформації про інші рядки.

На початку *i*-го кроку, після ділення *i*-го рядка на *i*-й коефіцієнт, віднімання відбувається паралельно. За найкращих умов (усі рядки опрацьовувались окремо), віднімання відбудеться за час віднімання одного рядка.

4. Паралельний алгоритм методу Жордано–Гауса

Нехай матриця системи буде опрацьовуватись на *n* процесах. Процеси нумеруються від нуля до *K*, де *K* — це кількість процесів.

Початковий крок

Перший процес зчитує матрицю коефіцієнтів і вектор вільних членів з файлу, розділяє на *K* смуг, по n/K рядків матриці і вектор розмірності n/K в кожній. Після цього кожна смуга (від другої до *K*-ї) пересилається відповідному процесу. При великих розмінностях матриці і кількості процесів доцільно використовувати певне запакування даних — або вмонтоване в MPI (MPI_Pack і MPI_Unpack), або власне. Використовуючи власне запакування потрібно контролювати довжину повідомлення (параметр count функції MPI_Send типу int, тобто повідомлення, більші за 32768 елементи будуть передаватись некоректно). Така оптимізація важлива, тому що часті виклики функції пересилання даних можуть знівелювати весь виграш від подальшої паралельної роботи.

Далі виконується *n* кроків.

i-й крок

Процес, який відповідає за *i*-й рядок, тобто з номером ik/n назвемо *активним*.

Активний процес перевіряє, чи на *i*-му місці в *i*-му рядку не знаходиться нуль. Якщо так, то робота продовжується, якщо ні, то здійснюється заміна за наступним алгоритмом. Перевіряються усі рядки, які належать даному процесу і знаходяться після *i*-го. Перший з них, в якому на *i*-й позиції знаходиться не нульове значення, міняється з *i*-м місцями. Якщо жоден з рядків процесу не підходить для заміни, то починається прийом прапорців про готовність обмінятись рядками від процесів № № $(i\,k/n+1)-K$. Якщо прапорець *M*-го процесу негативний, то перевіряється прапорець наступного процесу, якщо позитивний, то *M*-му процесу надсилається позитивний прапорець про готовність до обміну, номер процесу зберігається, дані (рядок і вільний член) копіюються в буфер, приймається рядок для обміну в буфер і замінюють дані у *i*-му рядку і вільному члені. Перевірка прапорців від наступних процесів припиняється. Якщо по закінченню перевірки прапорців *i*-й рядок все ще містить нуль на *i*-й позиції, то це означає, що між рядками є лінійна залежність, а тому програма аварійно закінчує роботу. Усім процесам, крім того, з яким відбувся обмін надсилається прапорець про відмову обміну. Якщо обміну не було, то негативний прапорець надсилається усім процесам.

Активний процес ділить *i*-й рядок на *i*-й його коефіцієнт і надсилає усім решта процесам.

Не активний процес, якщо його номер більший за номер активного процесу, шукає перший рядок, який належить даному процесу і в якому *i*-та компонента не рівна нулеві. Якщо такий знайдено, то активному процесу надсилається його номер (слугує позитивним прапорцем про готовність до обміну), і очікується відповідь, чи потрібно здійснювати обмін. Якщо отриманий прапорець про обмін позитивний, то дані копіюються у вихідний буфер, надсилаються, приймається інформація у вхідний буфер, копіюється у відповідний рядок. Якщо ж не знайдено рядка, який би можна було обміняти, то надсилається негативна відповідь (—1).

Усі процеси приймають дані від активного процесу (рядок матриці з одиницею на *i*-й позиції і вільний член) для роботи. Кожен процес множить усі свої рядки (розмірності n/k на *n*) на прийнятий вектор (розмірності *n*). Для кожного рядка матриці, прийнятий вектор потрібно поділити на *i*-й коефіцієнт поточного рядка.

Кінцевий крок.

Усі процеси, крім нульового, пересилають вектор вільних членів нульовому процесу, а також записують його у власний файл стану процесу. Також пересилається кількість байт, переданих даним процесом по мережі.

Нульовий процес: приймає частини розв'язку від кожного процесу; приймає і підсумовує загальну кількість байт пересланих по мережі усіма процесами; записує у файл стану; повертає загальний час роботи програми, як результат роботи функції main ().

Для прикладу розглянемо виконання алгоритму розв'язування системи розмірності 6×6 на 3-х процесах. Деталі опишемо лише для нульового, першого та шостого кроків, враховуючи аналогічність інших кроків.

5. Приклад роботи програми

Початковий крок. Процес № 0 виконує такі дії: зчитує 6 рядків коефіцієнтів і вектор вільних членів довжини 6 з файлу; процесу № 1 надсилає рядки №№ 2, 3; процесу № 2 надсилає рядки №№ 4, 5.

Перший крок. Процес № 0 — активний. Активний процес виконує такі дії: перевіряє чи коефіцієнт при першій змінній в першому рядку не рівний нулеві (припустимо, що дана умова виконалась). Якщо так, то ділить рядок № 1 на коефіцієнт при першій змінній; надсилає рядок № 1 процесам №№ 2, 3; віднімає рядок № 0 від рядка № 2 з відповідним множником. Надсилає процесам №№ 2, 3 негативний прапорець обміну.

Процеси №№ 2, 3 виконує такі дії: знаходять перший із своїх рядків, який містить ненульовий перший коефіцієнт, пересилають його номер процесу № 0, отримують негативний прапорець про обмін, приймають рядок № 1 від процесу № 0; віднімають рядок № 0 від рядків № 3, 4 (процес № 1) і № 5, 6 (процес № 2) з відповідними множниками. В результаті системи кожного процесу перетворились таким чином:

Система процесу № 0

$$\begin{cases} x_1 + a_{1,2}^{(1)} \cdot x_2 + a_{1,3}^{(1)} \cdot x_3 + a_{1,4}^{(1)} \cdot x_4 + a_{1,5}^{(1)} \cdot x_5 + a_{1,6}^{(1)} \cdot x_6 = b_1^{(1)}; \\ 0 + a_{2,2}^{(1)} \cdot x_2 + a_{2,3}^{(1)} \cdot x_3 + a_{2,4}^{(1)} \cdot x_4 + a_{2,5}^{(1)} \cdot x_5 + a_{2,6}^{(1)} \cdot x_6 = b_2^{(1)}. \end{cases}$$

Система процесу № 1

$$\begin{cases} 0 + a_{3,2}^{(1)} \cdot x_2 + a_{3,3}^{(1)} \cdot x_3 + a_{3,4}^{(1)} \cdot x_4 + a_{3,5}^{(1)} \cdot x_5 + a_{3,6}^{(1)} \cdot x_6 = b_3^{(1)}; \\ 0 + a_{4,2}^{(1)} \cdot x_2 + a_{4,3}^{(1)} \cdot x_3 + a_{4,4}^{(1)} \cdot x_4 + a_{4,5}^{(1)} \cdot x_5 + a_{4,6}^{(1)} \cdot x_6 = b_4^{(1)}. \end{cases}$$

Система процесу № 2

$$\begin{cases} 0 + a_{5,2}^{(1)} \cdot x_2 + a_{5,3}^{(1)} \cdot x_3 + a_{5,4}^{(1)} \cdot x_4 + a_{5,5}^{(1)} \cdot x_5 + a_{5,6}^{(1)} \cdot x_6 = b_5^{(1)}; \\ 0 + a_{6,2}^{(1)} \cdot x_2 + a_{6,3}^{(1)} \cdot x_3 + a_{6,4}^{(1)} \cdot x_4 + a_{6,5}^{(1)} \cdot x_5 + a_{6,6}^{(1)} \cdot x_6 = b_6^{(1)}. \end{cases}$$

Шостий крок. Процес № 2 — активний.

Активний процес ділить рядок № 6 на коефіцієнт при шостій змінній, надсилає його процесам №№ 0, 1, віднімає його від рядка № 5 з відповідним коефіцієнтом.

Процеси №№ 0, 1 приймають рядок для віднімання, віднімають його від своїх рядків з відповідними коефіцієнтами. Процеси № 0, 1 отримують рядок для віднімання, проводять віднімання.

Системи процесів виглядатимуть так:

Система процесу № 0

$$\begin{cases} x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 0 \cdot x_5 + 0 \cdot x_6 = b_1^{(6)}; \\ 0 \cdot x_1 + x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 0 \cdot x_5 + 0 \cdot x_6 = b_2^{(6)}. \end{cases}$$

Система процесу № 1

$$\begin{cases} 0 \cdot x_1 + 0 \cdot x_2 + x_3 + 0 \cdot x_4 + 0 \cdot x_5 + 0 \cdot x_6 = b_3^{(6)}; \\ 0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + x_4 + 0 \cdot x_5 + 0 \cdot x_6 = b_4^{(6)}. \end{cases}$$

Система процесу № 2

$$\begin{cases} 0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + x_5 + 0 \cdot x_6 = b_5^{(6)}; \\ 0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 0 \cdot x_5 + x_6 = b_6^{(6)}. \end{cases}$$

Кінцевий крок. Процеси №№ 1, 2 надсилають процесу № 0 частини вектора *b*. Процес № 0 виводить вектор вільних членів (який одночасно є і розв'язком) на консоль і записує його у файл результату. Демонстраційна програма надсилає також інформацію, яка кількісно характеризує ефективність роботи — час роботи, об'єм інформації, переданої по мережі, інше.

Примітки. Важливим є порядок роботи з прапорцями, які надсилаються дуже часто. Спочатку прапорець повинен бути надісланим, і лише потім очікуватись відповідь, а не навпаки. Нагадаємо, що функція отримання блокувальна, а функція надсилання — ні. Така послідовність запобігає колізії, коли прапорець не може бути надісланим, бо його не отримано і т. д.

За бажанням користувача у файл стану усіх процесів можна писати розширений коментар про дії, виконані процесом. Для цього потрібно запустити програму з параметром — comment. Це корисно для аналізу роботи програми, але вкрай негативно впливає на час виконання. Тому, для отримання результатів прискорення дана опція була відключена.

Аналіз результатів

Таблиця 1

Результати часу роботи (с)

Розмірність матриці	Кількість процесів			
	1	2	4	10
100	0,012	0,097	0,141	0,435
500	1,226	1,328	1,308	2,119
1000	8,592	6,307	5,992	6,102
1200	14,87	9,977	11,452	8,599
1300	17,761	13,571	9,448	10,3
1400	22,114	15,649	11,092	11,74
1500	27,142	17,914	13,214	13,487
1600	32,943	21,247	15,326	18,316
1700	39,619	25,188	18,806	18,981
1800	46,078	28,82	20,233	19,317

Як видно з таблиці, ефективність паралельного обчислення проявляється лише при великих розмірностях матриць. Це пояснюється зменшенням частки часу на пересилання даних і зростанням частки обчислень, які проводяться паралельно, в загальному часі роботи програми.

Також кожна розмірність системи має власну оптимальну кількість процесів, при якій час роботи мінімальний. Збільшення процесів для розмірності, наприклад, 1600 з чотирьох до десяти призводить до збільшення часу обчислень. На перший погляд, парадоксальний висновок, проте це цілком логічно. Такий ефект буде пояснено трохи згодом.

Проте, абсолютні значення не дають повної картини. Результат наочніший, якщо розглянути відносне прискорення

Таблиця 2

Відносне прискорення роботи програми

Розмірність матриці	Кількість процесів			
	1	2	4	10
100	1	0,12	0,09	0,03
500	1	0,92	0,94	0,58
1000	1	1,36	1,43	1,41
1200	1	1,49	1,30	1,73
1300	1	1,31	1,88	1,72
1400	1	1,41	1,99	1,88
1500	1	1,52	2,05	2,01
1600	1	1,55	2,15	1,80
1700	1	1,57	2,11	2,09
1800	1	1,60	2,28	2,39

Як видно з таблиці, робота двох процесів дійшла до межі зростання прискорення при розмірності 1500, в той час як 4 і 10 процесів все ще має тенденцію до зростання, причому стрімкіше зростає прискорення на більшій кількості процесів.

Враховуючи те, що декілька повідомлень може бути відправлено в одному циклі, а циклів може бути досить багато, потрібно використовувати певний механізм однозначної ідентифікації повідомлень. В демонстраційній програмі використано розпізнавання за дескриптором. Першому повідомленню циклу надано дескриптор «початкове зміщення + номер циклу»; другому — «початкове зміщення + розмірність матриці (вона ж і кількість циклів) + номер циклу»; третьому — «початкове зміщення + подвоєна розмірність матриці + номер циклу», і т. д. Перевагою такого методу є його зручність для виявлення помилок. Дескриптор неприйнятого повідомлення вказує точне місце помилки. Недоліком можна назвати додаткове звуження максимально можливої розмірності. При потребі код програми можна легко змінити. Замість формули для обчислення дескриптора можна кожному повідомленню назначити окрему область зв'язку.

Такий вигляд даних дозволяє зробити певні висновки.

Висновки

Використання паралельних програм корисне лише для дуже великих розмірів задачі.

Також воно ефективне для великих обсягів обчислень, і зовсім неефективне — для малих.

Збільшуючи розміри задачі, потрібно збільшувати кількість процесів для отримання кращих результатів прискорення. Як видно з таблиці № 2, кожна конфігурація процесів досягає потенційного максимуму. Такі дані можна вважати експериментальним доказом закону *Амдаля*.

Збільшення кількості процесів необов'язково приводить до зменшення часу роботи. В діапазоні 100…1000 два процеси справляються краще, ніж десять. А в діапазоні 1200…1700 чотири процеси працюють краще. Це можна пояснити задачкою про землекопів. Якщо 2 землекопи викопають яму 1×1×1 м швидше, ніж один, то 10 — набагато повільніше. Процеси починають «штовхатись», сповільнюючи виконання завдання. Отже, для кожного діапазону розмірностей потрібно вибирати оптимальну кількість процесів.

Значне зростання (в рази) швидкості роботи програми, навіть на відносно невеликих розмірностях матриці, дозволяє сказати, що використання стандарту MPI, зокрема його реалізації MPICH, є ефективним способом комунікації для паралельних обчислень. Проте, варто зазначити, що погано продумана схема роботи програми не може компенсуватись оптимізацією MPICH. В цих словах автори пересвідчилися на власному досвіді, коли одне додаткове пересилання даних в кожному циклі привело до негативних результатів. Тому, MPI — це радше засіб, а не панацея. Даний стандарт полегшує завдання, але розв'язувати проблему розпаралелювання потрібно програмісту, а не машині.

СПИСОК ЛІТЕРАТУРИ

1. Кардаш А. Розпаралелювання алгоритмів та програм в мережах персональних комп'ютерів / А. Кардаш, С. Левицька, А. Дудикевич // Інформаційні технології та комп'ютерна інженерія. — 2005. — № 3. — С. 229—233.

2. Концепції PVM та MPI в задачах розпаралелювання алгоритму LU-факторизації матриць / Автоматика–2006 : матеріали XIII Міжнародної конференції з автоматичного управління / А. І. Кардаш, С. М. Левицька, А. Т. Дудикевич. — УНІВЕРСУМ–Вінниця. — 2007. — С. 200—204.

Рекомендована кафедрою автоматики та інформаційно-вимірювальної техніки

Надійшла до редакції 21.10.08
Рекомендована до друку 20.11.08

Кардаш Андрій Іванович — доцент; ***Левицька Софія Михайлівна*** — старший викладач.

Кафедра програмування;

Дудикевич Анна Теодорівна — доцент кафедри обчислювальної математики.

Львівський національний університет імені Івана Франка