Кроме того, начальные и конечные точки эквидистанты определяется по формулам

$$x^R_{0,+} = x^{**}_0 + R\frac{\delta y^{**}_1}{S^{**}_1}, \qquad x^R_{n_*,+} = x^{**}_n + R\frac{\delta y^{**}_{n_*}}{S^{**}_{n_*}},$$
$$y^R_{0,+} = y^{**}_0 + R\frac{\delta y^{**}_1}{S^{**}_1}, \qquad y^R_{n,+} = y^{**}_{n,+} - R\frac{\delta x^{**}_1}{S^{**}_{n_*}}. \tag{14}$$

Аналогично определяем

$$x^R_{0,-} = x^{**}_0 - R\frac{\delta y^{**}_1}{S^{**}_1}, \quad x^R_{n_*,-} = x^{**}_{n,-} - R\frac{\delta y^{**}_{n_*}}{S^{**}_{n_*}}$$
$$y^R_{0,-} = y^{**}_0 + R\frac{\delta y^{**}_1}{S^{**}_1}, \quad y^R_{n_*,-} = y^{**}_{n,-} + R\frac{\delta y^{**}_1}{S^{**}_{n_*}}. \tag{15}$$

Таким образом, ломаная, последовательно проходящая через точки $(x^R_{i,+}, y^R_{i,+})$ $(i = \overline{o, n_*})$ и линия, проходящая через точки $(x^R_{i,-}, y^R_{i,-})$ $(i = \overline{0, n_*})$ является внешней и внутренней расчетной эквидистантной ломаной, т.е. расчетной траекторией возможного движения центра фрезы.

Выводы

Приведенный алгоритм дает асимптотически оптимальную траекторию движения режущего инструмента при контурном фрезеровании с асимптотически минимальным числом звеньев, т.е. эквидистантную обрабатываемому контуру. Это позволяет расширить технологические возможности методов расчетов траекторий движения осевых инструментов (например фрез) при подготовке управляющих программ для станков с ЧПУ.

ЛИТЕРАТУРА

1. Гжиров Р.И., Серебреницкий П.П. Программирование обработки на станках с ЧПУ: Справочник. – Л.: Машиностроение. Ленинр. Отд-ние, 1990. – 588с.
2. Молчанов Г.Н. Повышение эффективности обработки на станках с ЧПУ. – М.: Машиностроение, 1979. – 240с.
3. Константинов М.Т. Расчет программ фрезерования на станках с ЧПУ. – М.: Машиностроение, 1985. – 165с.
4. Байков В.Д., Вашкевич С.Н. Решение траекторных задач в микропроцессорных системах ЧПУ/ Под ред. Б.Б.Смолова. – Л.: Машиностроение, Ленингр. Отдние, 1986. – 106с.
5. Леус В.А. Гладкая окружностная интерполяция кривых // Вычислительные системы: Сб. науч. Тр. СО АН СССР. Вып.38. – Новосибирск, 1970. – С.102 – 127.
6. Завьялов Ю.С. и др. Сплайны в инженерной геометрии/ Ю.С.Завьялов, В.А.Леус, В.А.Скороспелов. – М.: – М.: Машиностроение, 1985. – 224 с.
7. Лигун А.А. Асимптотически оптимальный алгоритм кусочно-линейной интерполяции в решениях траекторных задач фрезерной обработки / А.А.Лигун, В.С. Коротков, А.А. Шумейко // Известия вузов. Машиностроение. – 1989. – №7. – С.147–151.
8. Асимптотически оптимальный алгоритм кусочно-линейной интерполяции при подготовке управляющих программ для фрезерных станков с ЧПУ / А.А.Лигун, В.С.Коротков, А.А.Шумейко. – К., 1987. – 27с. – Деп. В УкрНИНТИ 21.09.87, №2620.

пост. 05.12.2017

К.С. КРАСНІКОВ, к.т.н., ст. викладач кафедри програмного забезпечення систем, kir_kras@ukr.net
Дніпровський державний технічний університет, Кам'янське

Комп'ютерні технології швидкого обчислення математичних моделей

У роботі наведено розгляд існуючих комп'ютерних методів прискорення обчислень з прикладами і ілюстраціями, а також проблеми, які виникають при комп'ютерній реалізації. Результати проведених експериментів, викладені на рисунках, свідчать про суттєве підвищення швидкості обчислень при збільшенні числа паралельно працюючих потоків.

The paper provides an overview of existing computer methods for accelerating of computations with examples and illustrations, as well as problems that arise during computer realization. The results of the experiments outlined in the figures indicate a significant increase in the speed of computations with an increase in the number of parallel working threads.

Постановка проблеми

Характерною особливістю комплексних тривимірних математичних моделей, наприклад такої, як у роботі [1], є необхідність здійснення великої кількості математичних операцій і обчислення великих масивів даних, що впливає на час розрахунку, який без прискорення може сягати декілька діб. Завдяки розвитку комп'ютерної техніки, сьогодні існують засоби істотного підвищення швидкості обчислень за рахунок використання паралельних технологій у програмному коді.

Великі кластерні обчислювальні центри і персональні комп'ютери мають процесори, зокрема багатоядерні, з декількома обчислювальними пристроями, які можуть працювати одночасно. Для цього потрібно застосовувати спеціальні алгоритми під час програмної реалізації математичної моделі для прискорення обчислень. Найпростіша комп'ютерна реалізація має вигляд послідовного виконання математичних операцій згідно етапів обчислення, таким чином весь час розрахунку складає суму тривалостей обчислення кожної операції. Для прискорення необхідно виконувати деякі операції одночасно, що означає необхідність пошуку незалежних операцій або їх сукупностей — функцій (для забезпечення однакового результату роботи програми).

Аналіз останніх досліджень та публікацій

Згідно з роботами [2, 3] існують наступні рівні паралелізму:

1) Бітів — пов'язаний із скінченним об’ємом пам’яті для даних, над якими виконується операція, наприклад, добуток двох 16-бітних чисел потребує декількох операцій для 8-бітного процесора, а для 16-бітного — одну.

2) Інструкцій — можливість одночасного виконання процесором інструкцій, наприклад, додавання і добутку замість послідовного. Прикладом є інструкції: FMA (об’єднані добуток і додавання), AVX (одна інструкція виконує одну з арифметичних дій над двома векторами чисел).

3) Даних — розділ масиву даних на частини і виконання однакової операції над означеними частинами окремими обчислювальними одиницями (наприклад, ядрами процесору).

4) Завдань — одночасне виконання декількох функцій, які складаються з багатьох інструкцій, наприклад, одним завданням для операційної системи є завантаження файлу з мережі, а іншим — відображення анімації на моніторі. Для виконання кожного із завдань створюється окремий потік команд або «нитка», яка відповідає окремому ядру процесора (потоки використовуються операційною системою для розпаралелювання обчислень між процесорами і число потоків у комп’ютерній програмі завжди мінімум один — це головний потік).

Згідно з законом Амдала [4] не слід очікувати максимальне прискорення, якщо програма не повністю розпаралелена, оскільки найповільніша її частина буде визначати час розрахунку. Але звичайно середню більшу швидкість все одно можна досягнути відповідно закону Густафсона [5], якщо додати іншу розпаралелену розрахункову задачу, більшу ніж вихідну.

Формулювання мети дослідження

Метою роботи є підвищення швидкості обчислень у комп’ютерній програмі при підвищенні числа ядер (старт на різних комп’ютерах).

Виклад основного матеріалу

Другий рівень згаданої паралелізації обчислень можна використати у розв’язанні системи лінійних алгебраїчних рівнянь для динаміки системи твердих тіл [1]: кожне число в рядку матриці помножується на відповідне число у векторі значень і результат додається. Замість послідовного виконання додавання і добутку рекомендовано застосовувати відповідну інструкцію процесора. Відомо, також що при цьому додатково зменшується помилка арифметичних операцій (рис. 1).

x1		x2		x3	
*	+	*	+	*	…
y1		y2		y3	

Рис. 1. Приклад послідовних операцій, які трапляються при розв’язанні СЛАР

Арифметична операція додавання виконується зазвичай послідовно, але цю послідовність можна розпаралелити для прискорення, якщо набір чисел для визначення суми є достатньо великий (при малому — прискорення буде несуттєве, рис. 2).

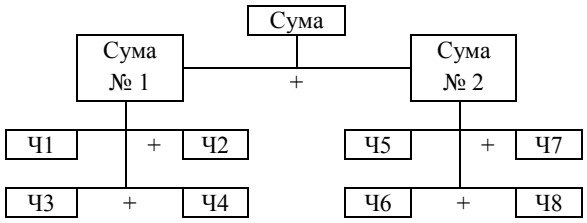

Рис. 2. Додавання в кожній парі чисел можна робити паралельно. Також чотири числа можна додати до інших чотирьох чисел за допомогою інструкції AVX

Третій рівень пропонується використати у обчисленні гідродинаміки, поділивши розрахункову область на частини, кожну з яких обчислювати незалежно на кожному етапі математичної моделі. У даному випадку може виникнути так звана гонка умов — «condition race» — ситуація, в якій декілька потоків намагаються записати значення глобальної змінної або джерела даних. Таке змагання часто відбувається на рівні оперативної пам'яті. Іншим прикладом є одночасна робота з файлом на диску, або з зовнішнім пристроєм, який має з’єднання з комп'ютером. Дану ситуацію краще зрозуміти на прикладі, коли два потоки одночасно обчислюють наведений нижче код. Перший потік виконує операцію добутку mult *= pershii_koef; другий потік також виконує добуток але з іншим числом: mult *= drugii_koef. Після виконання означеного коду на перший погляд в змінній mult повинно бути її значення, помножене на обидва коефіцієнта. Але через означену «гонку даних» так буде не завжди, іноді змінна mult буде мати значення добутку тільки з одним з коефіцієнтів. Розберемо чому так стається — обидві операції множення працюють над спільним ресурсом — змінною mult. Оператор присвоювання на мові програмування записується одним символом =, але на рівні машинного коду (асемблера), після компіляції програми він стане ланцюжком інструкцій. Далі наведено можливу послідовність виконання інструкції на рівні машинного коду для двох потоків:

1) mult одночасно завантажується у деякі регістри P1 і P2.

2) pershii_koef завантажується у регістр P3 і drugii_koef — у регістр P4.

3) виконується добуток значень регістрів P1 і P3 та результат зберігається у P5, і добуток P2 і P4 зберігається у P6.

4) виконується запис з регистру P5 у змінну mult, і також — з регистру P6 у змінну mult.

Під час паралельного виконання два потоки одночасно зчитують із пам’яті значення змінної mult і зберігають у відповідних регістрах, виконують одночасно добуток, але запис результату у змінну mult доведеться робити послідовно. Результат потоку, який виграв «гонку даних», буде втрачений тому, що після нього другий потік запише свій результат операції в змінну mult. В даному випадку проблеми не виникне, якщо послідовно обчислювати набори команд, а в загалі необхідно додатково синхронізувати паралельну роботу потоків.

Четвертий рівень підходить до розмежування обчислення математичної моделі фізичного процесу, який досліджується, і розрахунків, пов’язаних з отри-

манням 3D-зображення отриманих результатів, якщо вони об'єднані в єдиній комп'ютерній програмі. Не має необхідності на кожному кроці за часом отримувати зображення — максимум 30 разів на секунду буде достатньо для користувача. Таким чином головний потік відображає результати розрахунків, отриманих від допоміжних потоків, кількість яких відповідає числу вільних для обчислення процесорів.

У нашому випадку багатоетапного розрахунку математичної моделі синхронізація між етапами розрахунку забезпечується очікуванням виконання своїх дій всіма потоками перед переходом до наступного етапу (рис. 3).

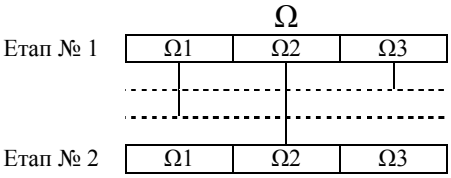

Рис. 3. Приклад розділу масиву чисел на окремі частини у межах кожного етапу обчислення з можливим запізненням розрахунків, що часто має місце при комп'ютерній реалізації математичних моделей

Також можна використовувати блокування потоків. Для цього в необхідному рядку коду встановлюється деякий *об'єкт* з двома станами «відкритий», «закритий». Якщо будь-який потік потрапляє на означений рядок, *об'єкт* переключає стан у «закритий» для зупинки інших потоків, які потрапили на цей рядок також. Після закінчення роботи потік переключає *об'єкт* у стан «відкритий» і наступний в черзі потік починає свою роботу.

Блокування потоків має недолік — «дедлок» (deadlock) – одна з проблем, що виникають при паралельному програмуванні. «Дедлок» виникає, коли два або більше працюючих потоки намагаються одержати вільний доступ до відповідно двох або більше глобальних змінних. При чому кожен з потоків встигає одержати дозвіл для роботи з однією із глобальних змінних, хоча потребує доступ до інших також, але інші вже зайняті і виникає безвихідна ситуація, коли всі потоки не мають можливості продовжити обчислення, всі стоять в нерухомих чергах — користувач бачить, що додаток "завис".

Користувач не розуміє, що трапилося, коли додаток зникає з екрану комп'ютера в результаті виникнення виняткової ситуації, яку програміст не обробив належним чином. Іншим прикладом є той, у якому додаток завершує обчислення, але одержані результати є неправильними. І найгірший випадок, коли додаток "зависає", не видає результатів і закривати його потрібно, використовуючи зовнішнє закриття (наприклад, через системну команду завершення задачі).

Випадок «дедлоку» можна розглянути на прикладі двох потоків P1 і P2 і двох глобальних змінних R1 і R2. Нехай потік P1, входячи в небезпечну частину коду, одержує доступ до змінній R1. Одночасно з цим другий потік P2 не має доступу до змінної R1. Але він має доступ до змінній R2 і користується цим, блокуючи змінну R2. Якщо тепер потік P1 намагатиметься звернутися до змінної R2, то він зупиниться і стане у чергу. Так само і потік P2, намагаючись одержати доступ до змінної R1, можливо зупиниться і стане у чергу через зайнятість доступу до змінної R1. Як два автомобілі без можливості руху назад на дуже вузькій вулиці не в змозі проїхати повз одне одного вимушені зупинитися і чекати на зовнішню допомогу.

З метою уникнення «дедлоку» потрібно уважно розподілити обчислення серед потоків з використанням декількох глобальних змінних. В ситуації коли декілька небезпечних частин коду можуть інколи одночасно використовують декілька глобальних змінних є різні способи розв'язання проблеми «дедлоку»:

1) Кожна небезпечна частина коду одержує доступ до всіх глобальних змінних одразу, гарантуючи собі перше місце у черзі і в будь-який момент часу тільки одна небезпечна частина коду буде виконуватися. Таким чином і «дедлок» уникається. Очевидним недоліком такого блокування є неоптимальне користування глобальними змінними, адже деякі з них одночасно можуть просто не застосовуватися, що спричиняє збільшення часу обчислення у результаті.

2) Вводиться правило: коли робота з глобальними змінними ведеться послідовно в небезпечних частинах коду послідовно, а не одночасно, тоді після доступу до змінної вона одразу ж звільняється, не очікуючи завершення небезпечної частини коду. Це значно ефективніший спосіб, який дозволяє повністю уникати «дедлоку». Але він обмежує програміста у можливості одночасного доступу до декількох глобальних змінних у кожній з небезпечних частин коду.

3) Програма, яка має тільки одну небезпечну частина коду, повністю уникає «дедлоку». Це спосіб дуже схожий на перший одержанням доступу до всіх ресурсів одразу і знаходження тільки одного потоку в небезпечній частині, поки інші потоки чекають у черзі.

4) Якщо використовувати для очікування потоку спеціальний вид функції bool Join (int t), аргументом якої є час очікування. Якщо він минув то функція повертає false і це майже завжди означає появу «дедлоку», але при цьому програміст має можливість обробити так ситуацію необхідним шляхом і зависання додатку не відбудеться. Якщо ж потік встиг відпрацювати за означений час, тоді функція Join повертає значення true і програма продовжує працювати у звичайному режимі.

5) Інтелектуальне використання глобальних змінних, а саме вибіркове обмеження на запис змінної, в той час як зчитування залишається завжди доступним без блокування роботи всіх зацікавлених у читанні потоків. Порівнюючи цей підхід з застосуванням оператору lock, приходимо до більш ефективного використання глобальних змінних. Звичайно він потребує більш уважного програмування, але подібні змінні трапляються майже у кожній програмі, тому цей розв'язок можна рекомендувати якнайчастіше.

І на останок розглянемо підхід із використанням семафорів, які мають такі важливі особливості:

1) Семафори можуть змінювати обмеження на виконання коду потоками, на відміну від звичайних прийомів блокування. Вони також створюють чергу, але без об'єкта синхронізації, якщо частина коду зайнята одним потоком, а інші потоку намагаються перейти в небезпечну частину коду.

2) Семафори, на відміну від звичайних методів, дають дозвіл на виконання небезпечної частини коду визначеному числу потоків.

У загальному випадку семафори схожі на систему масового обслуговування з обмеженим числом вільних місць. Наприклад, у супермаркеті тільки обмежена кількість покупців проходять одночасно оплату товарів на касі, а інші чекають, поки каса звільниться.

Для обмеження кількості комп’ютерів, з якими обмінюється даними сервер також використовуються семафори, зменшуючи навантаження на нього. Таким чином семафори є більш універсальним і високорівневим засобом керуванням потоками з можливістю гнучкого керування своєї пропускної здатності.

Висновки та перспективи подальших досліджень

В результаті впровадження паралельних обчислень у програму досягнуто прискорення (рис. 4).

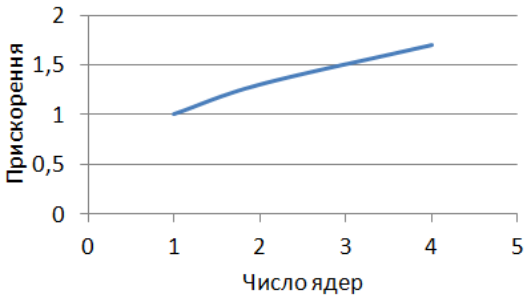

Рис. 4. Одержана залежність прискорення обчислень матмоделі гідродинаміки і динаміки твердих тіл від числа ядер процесора

Задача обробки зображення фільтром 3×3 дуже добре підходить до розпаралелювання тому, що майже 100% обчислень буде розподілено між процесорами (рис. 5). Для вимірювання тривалості обробки з точністю до мілісекунд використовувався клас StopWatch з бібліотеки .NET. На старті екземпляр цього класу перезавантажується і починає відлік з нуля. Після завершення процесу обробки зображення програма показує тривалість.

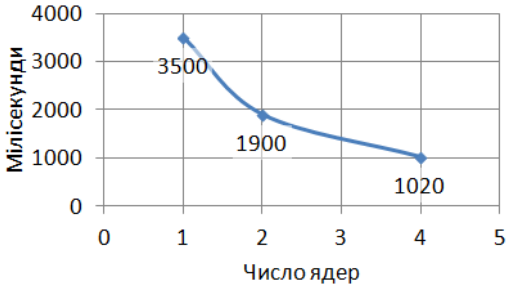

Рис. 5. Залежність часу обробки зображення 8192x4096 від числа використаних ядер процесора

ЛІТЕРАТУРА

1. Красніков К. С. Математична модель тривимірного руху порошкового дроту у розплаві сталі під час продування інертним газом на установці ківш-піч. // Вісник ХНТУ, – № 3(54). Херсон: ХНТУ, 2015. – С. 378-383.
2. Culler D. Parallel computer architecture: a hardware / software approach / David Culler, Jaswinder Pal Singh, Anoop Gupta // Morgan Caufmann Publishers, 1998. – pp. 1056.
3. Quinn M. J. Parallel programming in C with MPI and OpenMP // McGraw-Hill Education, 2003. – pp. 527.
4. Amdahl G. (April 1967) «The validity of the single processor approach to achieving large-scale computing capabilities». In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, 1967. – pp. 483-85.
5. McCool M. D., Robison A. D., Reinders James (2012). "2.5 Performance Theory". Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, 2012. – pp. 61–62. ISBN 978-0-12-415993-8.
6. http://www.intuit.ru/studies/courses/10554/1092/lecture/27095

пост. 23.11.2017