УДК 519.6

А. І. Кардаш, канд. фіз.-мат. наук, доц.;
С. М. Левицька;
А. Т. Дудикевич, канд. фіз.-мат. наук, доц.

ЧИСЕЛЬНИЙ ЕКСПЕРИМЕНТ ПІД ЧАС ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОЗПАРАЛЕЛЮВАННЯ АЛГОРИТМІВ ТА ПРОГРАМ НА СОКЕТАХ

Досліджено ефективність розпаралелювання обчислень на сокетах для прикладу алгоритму методу ортогоналізації розв'язування систем лінійних алгебричних рівнянь. Запропоновано практичні поради доцільності розпаралелювання обчислень.

Вступ. Система розділеного програмування на сокетах

Для виконання операцій розпаралелювання існують спеціальні технології, допоміжні програми та бібліотеки, що полегшують роботу програміста [1—3]. Тепер пропагується для цих цілей використання мінімальних можливостей середовища програмування Visual Studio 2008 на мові програмування C# [4, 5]. В пропонованій роботі якраз буде досліджено ефективність розпаралелювання в цьому середовищі.

Яким би потужним не був комп'ютер, два таких як він, об'єднані в єдину машину, завжди будуть потужніші. Навіть збільшення кількості периферійних частин комп'ютера, таких як оперативна пам'ять, відеопам'ять, додатковий жорсткий диск або багатоядерний процесор, беззаперечно зробить комп'ютер потужнішим.

Однак потужний — ще не означає швидший. Інколи вся модернізація призведе лише до того, що деякі частини комп'ютера будуть працювати у півсили, тоді як інші будуть викладатись на повну. Наприклад, встановлення додаткової оперативної пам'яті додасть навантаження на центральний процесор. Об'єднання декількох комп'ютерів за допомогою локальної мережі додасть навантаження на мережеві пристрої та знову ж на центральний процесор.

Якщо потрібно збільшити швидкість запису/зчитування інформації на жорсткий диск та збільшити її обсяг, то для цього достатньо встановити додатковий жорсткий диск. Якщо нам потрібно збільшити швидкість її обробки — варто придбати багатоядерний центральний процесор та оперативну пам'ять.

Так і розпаралелювання програм може дати різні результати і треба знати, що нам потрібно. Саме розпаралелювання доцільне лише тоді, коли є змога опрацьовування декількох частин інформації одночасно. В протилежному випадку ми можемо тільки отримати ілюзію розпаралелювання, коли здається, що задіяні декілька комп'ютерів, але сам час виконання не зменшується.

Розпаралелювання — це досить складний процес. Спочатку треба розпаралелити сам алгоритм розв'язання нашої задачі. Потім налаштувати на це програму, що включає в себе декілька підпунктів. У програмі треба передбачити метод передачі повідомлення по мережі. Крім того повідомлення передаються у вигляді послідовності бітів, то ж слід реалізувати конвертацію даних у масив бітів та зворотний процес.

Практична реалізація програмного коду здійснюється за допомогою сокетів. Сокети (англ. socket — заглиблення, гніздо) — назва програмного інтерфейсу для забезпечення обміну даними між процесами. Процеси за такого обміну можуть виконуватись як на одному ЕОМ, так і на окремих ЕОМ, з'єднаних між собою мережею. Сокет — абстрактний об'єкт, що є кінцевою точкою з'єднання. Розрізняють клієнтські і серверні сокети. Клієнтські сокети грубо можна порівнювати з кінцевим пристроєм телефонної мережі, а серверні — з комутаторами. Клієнтські прикладні програми (наприклад, браузер) використовують тільки клієнтські сокети, а серверні (наприклад, вебсервер, якому браузер надсилає запити) — як клієнтські, так і серверні сокети.

Сокет на мові системних адміністраторів означає комбінацію IP-адреси та номера порту, наприклад, 10.10.10.10:80. На мові C# створення та використання сокетів виглядає так:

Для сервера виконуємо:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
// створення кінцевої точки, пари: IP-адреса і номер порту.
Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
// Створюємо сокет типу Stream, це один з найпростіших
// сокетів, тому пізніше ми будемо його удосконалювати.
newsock.Bind(ipep);
newsock.Listen(10);
// Listen(10) — означає, що сервер буде одночасно працювати не //більше ніж, з 10-ма клієнтами.
Socket client = newsock.Accept();
IPEndPoint clientep = (IPEndPoint)client.RemoteEndPoint;
```

Створення самого з'єднання, програма очікує, доки клієнт надішле запит про під'єднання та створює сокет, за допомогою якого і буде передаватись інформація.

Надалі дані надсилають за допомогою команди:

```
client.send(data);
```

Де data — масив бітів.

Отримання даних:

```
client.Receive(data);
```

Для клієнта виконуємо:

```
Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
IPEndPoint ipep = new IPEndPoint(IPAddress.Parse(ip.ToString()), 9050);
// створення кінцевої точки, пари: IP-адреса і номер порту. Так як на //сервері, з тією відмінністю, що IP-адреса сервера повинна бути відомою в
//іншому випадку доведеться пробувати підключитись до сервера за всіма
//доступними адресами. Номер порту також повинен співпадати.
try{
```

Необхідність багатопоточного програмування

Основним недоліком сокету типу Stream є те, що кожне надіслане повідомлення повинно бути прийнятим до того, як буде надіслано наступне. Тобто, якщо надіслати відразу два повідомлення, то отримувач отримає тільки останнє з них.

Ця проблема є непомітною для роботи чату, коли програма більше нічого не виконує, окрім як пересилає повідомлення. Але в загальному випадку, тобто в нашому випадку, потрібно буде надіслати серію повідомлень і тільки після цього отримати повідомлення-відповідь.

Найнадійніший спосіб отримання кожного надісланого повідомлення — це постійно на нього чекати, але при цьому інші задачі не виконуються. Щоб ця проблема не впливала на час виконання всього завдання, використовують багатопоточне програмування.

Потік — це керована одиниця коду, що виконується. В багатозадачному середовищі, напрямленому на потоки, всі процеси мають в будь-якому випадку один потік, але можлива і більша кількість потоків. Це означає, що одна програма може виконувати відразу дві або більше задач.

В нашому випадку це виконувати обчислення та очікувати на нове повідомлення.

Необхідно розрізняти поняття «процесорно-орієнтована багатозадачність» та «поточно-орієнтована багатозадачність». Процесорно-орієнтована багатозадачність забезпечує виконання декількох програм, а поточно-орієнтована багатозадачність передбачає одночасне виконання частин однієї програми.

Багатопоточна система C# вбудована в клас Thread.

Щоб створити потік, необхідно створити об'єкт типу Thread.

```
public Thread(ThreadStart entryPoint)
```

Наприклад:

```
Thread th = new Thread(new ThreadStart(this.run));
```

Де run, метод, який і буде виконувати новий потік. Цей метод повинен повертати тип void і не мати жодних аргументів.

Виконання заданого потоку не розпочнеться до тих пір, доки не буде виконаний метод Start(), який оголошений в класі Thread. Його визначення виглядає так:

```
public void Start()
```

Після того, як новий потік розпочнеться, його можна розглядати як окрему програму з тією відмінністю, що коли завершиться головний процес, другорядні процеси теж припинять свою роботу, а також тим, що ці потоки можна синхронізувати, використовуючи статичні змінні. Про це описано детальніше у наступному пункті.

Якраз автори досліджують ефективність розпаралелювання на сокеах на відміну від реалізації стандарту MPI та на кластерах [3—5].

Алгоритм безперервного отримання даних

Для надійної передачі даних замало вчасно отримати повідомлення, потрібно ще й отримати підтвердження про вдалу передачу. Тому метод run, про який ми згадували під час створення потоків, матиме такий вигляд:

```
void run() {
byte[] echo = new byte[] { 1 };  byte[] data;
while (ContinueWork) {
try {
data = new byte[short.MaxValue];
socket.Receive(data);
if (!MustGiveEcho) {
Received.Add(data);
socket.Send(echo); }
else {
EchoGived = true;
MustGiveEcho = false; } }
catch {
ContinueWork = false; } } }
```

MustGiveEcho, EchoGived, ContinueWork статичні змінні типу bool.

В C# під поняттям статичний (static) мають на увазі член, який можуть використовувати усі екземпляри класу, в якому він оголошений. Завдяки тому, що вони статичні, їх використовують для синхронізації між потоками.

Received — колекція повідомлень, також статична, метод run записує повідомлення в її кінець, а головний потік зчитує їх з початку колекції та видаляє їх.

Метод відправлення даних виглядатиме таким чином:

```
public void SendAndVait(Socket Socket1,byte[] data) {
MustGiveEcho = true; EchoGived = false;
//Цими змінними ми повідомляємо допоміжному потоку, що
//наступне повідомлення буде лише підтвердженням про
//отримання, відлунням (Echo), тому його зберігати не треба.
Socket1.Send(data);
while (!EchoGived); }
//Очікуємо завершення операції.
```

Серіалізація

Перед тим, як надсилати дані, потрібно звести їх до вигляду бітової послідовності. Це стандартний процес, що не потребує ніякої додаткової імпровізації.

Серіалізація — процес переводу будь-якої структури даних у послідовність бітів. Оберненою до операції серіалізації є операція десеріалізації — відновлення початкового стану структури даних з бітової послідовності.

Серіалізацію можна виконати однією простою функцією:

```
byte[] ConverDataToByteArrey(object DataClass) {
BinaryFormatter formatter = new BinaryFormatter();
MemoryStream stream = new MemoryStream();
formatter.Serialize(stream, DataClass);
return stream.ToArray(); }
```

Десеріалізація:

```
object ConverByteArreyToData(byte[] binaryData) {
```

```
BinaryFormatter formatter = new BinaryFormatter();
MemoryStream ms = new MemoryStream(binaryData);
return formatter.Deserialize(ms); }
```

Ще одним цікавим моментом є те, що спочатку необхідно надати класу (екземпляру, який ми хочемо конвертувати) властивість Serializable та його інтерфейс:

```
[Serializable()]
class MyClass : ISerializable
```

{ Після цього компіляція програми видасть помилку про відсутність методів інтерфейсу ISerializable. Додамо їх:

```
public MyClass(SerializationInfo info, StreamingContext ctxt)
{ Coef = (List<int>)info.GetValue(«Coef», typeof(List<int>));}
 public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
{ info.AddValue(«Coef», Coef);      }
```

Перший метод — це конструктор для нашого класу, який обов'язково повинен бути присутнім саме у такому вигляді. В ньому ми отримаємо потрібні нам параметри.

В другому методі записуються методи для подальшого його кодування у біти. При цьому назви параметрів у закодованому вигляді можуть відрізнятись від справжніх, але повинні збігатися в першому та другому методах. Якщо деякі параметри належать до інших класів, то їх також потрібно серіалізувати таким же чином.

Висновки

Детально описано, як реалізувати багатопоточну програму з синхронізацією потоків та алгоритмів програмної реалізації передачі повідомлень між процесами, що працюють на різних обчислювальних машинах. Все це реалізовано без допомоги будь-яких допоміжних програм чи технологій, крім тих, що реалізовані у програмному середовищі Visual Studio 2008 на мові програмування C#.

В результаті практичної реалізації розпаралелювання алгоритму методу ортогоналізації [4] для розв'язування систем лінійних алгебричних рівнянь, встановлено, що використання розпаралелювання алгоритмів значно зменшує час виконання завдань, але існує оптимальна кількість процесів, які можуть бути використані для розв'язання кожної конкретної задачі.

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

3. Кардаш А. І. Розпаралелювання обчислення на кластерах / А. І. Кардаш, С. М. Левицька, А. Т. Дудикевич // Вісник Вінницького політехнічного інституту. — 2009. — № 3. — С. 74—79.

4. Джеффри Рихтер. Программирование на платформе Microsoft .NET Framework / Джеффри Рихтер. — М. : Изд-во Русская Редакция, 2003. — ISBN: 5-7502-0208-9.

5. Douglas Gregor : MPI.NET Tutorial in C# / Douglas Gregor and Benjamin Martin. — Open Systems Laboratory, Indiana University. Published September, 2008.

6. Григорій Цегелик. Чисельні методи / Григорій Цегелик. — Львів : Видавничий центр ЛНУ ім. Івана Франка, 2004.

Рекомендована кафедрою комп'ютерних систем управління

Стаття надійшла до редакції 30.03.11
Рекомендована до друку 8.06.11

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

Кафедра програмування;

Дудикевич Анна Теодорівна — доцент кафедри обчислювальної математики.

Львівський національний університет імені Івана Франка, Львів