УДК 004.4

ДОСЛІДЖЕННЯ ШАБЛОНІВ СИНХРОНІЗАЦІЇ ТА ЇХ ВИКОРИСТАННЯ В ТЕХНОЛОГІЇ GPGPU

С.Д. Погорілий, О.А. Верещинський

Київський національний університет імені Тараса Шевченка,
Україна, 01601, Київ, вул. Володимирська, 60,
тел. (044) 521 3590

В роботі проаналізовано задачу синхронізації паралельних потоків, сформовано критерії, яким має відповідати розв'язок. Описано метод синхронізації з використанням стратегій блокування. Розглянуто підходи на основі динамічного та статичного поліморфізму. Проаналізовано можливість використання шаблонної синхронізації в технології GPGPU на базі операторів САА-М Глушкова.

Problem of parallel thread synchronization analyzed, solution criteria formulated. Synchronization method based on blocking strategies described. Dynamic and static polymorphism approaches investigated. Possibility of use of pattern synchronization in GPGPU based on Glushkov SAA-M operators analyzed.

Вступ

Розробка паралельних застосувань є складною задачею, оскільки неправильне використання блокувань може призвести до виникнення складних для виправлення помилок. Таким чином, створення паралельних компонентів, які можуть повторно використовуватися також є складним процесом, оскільки постійне виникнення нових технологій та підходів призводить до додаткових затрат часу на внесення змін у вже існуючі компоненти. В статті запропоновано використання шаблону «Стратегія» для усунення описаних проблем.

При застосуванні технології GPGPU виникає низка принципово нових засобів синхронізації, оскільки ця технологія вносить особливості (ієрархія робочих потоків, ієрархія пам'яті) не притаманних звичайним багатозадачним системам. Мета роботи – створення узагальненого методу синхронізації об'єктів, що дозволить винести деталі механізму блокування поза межі конкретного алгоритму, а також буде застосовний до GPGPU-застосувань [1–3].

Шаблонна синхронізація

Шаблон «Блокування на основі стратегій» дозволяє збільшити гнучкість та можливість повторного використання без втрати швидкодії чи рівня підтримки коду [4].

Контекст, в якому виникає задача: застосування або система, де компоненти повинні оперувати за умов високого рівня паралелізму.

Проблема: існування набору існуючих компонентів у яких стратегії синхронізації «зашиті» в код, що не задовольняє таким умовам:

1. Простота настроювання швидкодії синхронізації

Настроювання компонентів для різних сценаріїв паралельного використання має бути досить простим. Однак, якщо стратегія синхронізації «зашита» в компонент, його модифікація вимагає додаткового часу та зусиль, наприклад, для використання новіших, більш швидких методів синхронізації.

2. Простота підтримки коду та внесення змін в існуючі стратегії

Вдосконалення існуючого коду та виправлення помилок має бути нескладною процедурою. Однак, якщо в системі вже існує багато копій одних і тих же базових компонентів, то можуть виникати конфліктні ситуації.

Розв'язок. Для розв'язку задачі «стратегізації» аспектів синхронізації запропоновано імплементувати їх як типи даних, які можна «підключати» в інші компоненти. Вони мають бути членами об'єктів, що вимагають синхронізації для того щоб забезпечувати синхронізацію, що надасть можливість вибирати необхідну стратегію на етапі виконання чи компіляції програми.

Імплементація. Шаблон може бути реалізовано за допомогою наступних кроків.

І. Визначити інтерфейс об'єкта та його реалізацію. Основною ідеєю цього кроку є виділення чіткого інтерфейсу основного об'єкта та ефективної його реалізації без синхронізації. Наступний клас задає основний інтерфейс:

```
class ExampleClass
  {
public:
void DoSomething();
    // …
  }
```

© С.Д. Погорілий, О.А. Верещинський, 2014

ІІ. Створення стратегій, що відображатимуть аспекти синхронізації:

В цьому кроці слід визначити, які деталі синхронізації можуть змінюватися і змінити основний інтерфейс згідно з цими деталями.

Багато компонентів використовують доволі просту синхронізацію, що може бути реалізована за допомогою об'єктів типу семафорів та м'ютексів. Синхронізація такого типу може бути виділена у стратегії, що використовують параметризовані типи або поліморфізм.

Підхід на основі поліморфізму. Цей підхід базується на передачі поліморфного Lock-об'єкта в конструктор основного об'єкта і встановлення його приватним членом, якому в подальшому будуть делеговані всі запити на синхронізацію (рис. 1).

Типовим підходом буде використання шаблона «Міст». Для початку визначимо абстрактний клас із поліморфними чисто віртуальними методами aquire та release:

```
class Lockable
  {
  // Aquire the lock
virtual int aquire() = 0;

  // Release the lock
virtual int release() = 0;
  }
```

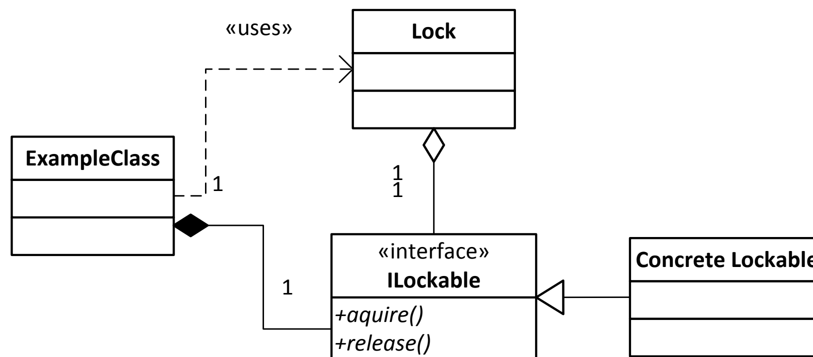

Рис. 1. UML діаграма поліморфної стратегії блокування

Підкласи мають перевантажити віртуальні методи визначаючи таким чином стратегію блокування. Наприклад, наступний клас визначає мютексне блокування:

```
class Thread_Mutex_Lockable : public Lockable
  {
public:
    // Acquire the lock
virtual int acquire (void)
      {
return lock_.acquire ();
      }

    // Release the lock
virtual int release (void)
      {
return lock_.release ();
      }
private:
    // Concrete lock type
    Thread_Mutex lock_;
  };
```

І нарешті створимо не поліморфний клас, що буде пам'ятати вказівник на поліморфний Lockable (застосуємо шаблон «Міст»):

```
class Lock
  {
public:
    // Constructor stores a reference to the
```

```
        // base class.
        Lock (Lockable &l): lock_ (l) {};
        // Acquire the lock by forwarding to the
        // polymorphic acquire() method.
        int acquire (void) { lock_.acquire (); }

        // Release the lock by forwarding to the
        // polymorphic release() method.
        int release (void) { lock_.release (); }

    private:
        // Maintain a reference to the polymorphic lock
        Lockable &lock_;
      };
```

Цей клас призначений для того, аби він міг бути використаним як стековий об'єкт, а не через вказівник. Це зробить можливим використання іншого шаблона – «Блокування за областю видимості», як показано в наступному прикладі:

```
    class ExampleClass
      {
    public:
        ExampleClass (Lock lock) : lock_(lock) {}
    void DoSomething();
          {
          // Use the Scoped Locking idiom to
          // acquire and release the <lock_> automati-cally.
          Guard<Lock> guard (lock_);
          // Implement the method.
          }
        // …
    private:
        // The polymorphic strategized locking object.
        Lock lock_;
      }
```

Параметризовані типи. В цьому підході слід додати відповідний шаблонний параметр LOCK і визначити відповідні об'єкти типу LOCK як приватні члени класу, що відповідають за стратегію блокування, яка використовується в методах компонента (рис. 2).

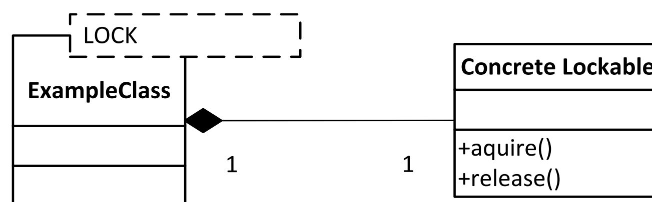

Рис. 2. UML діаграма стратегії блокування на основі параметризованих типів

Наступний код ілюструє запропонований підхід:

```
    template<class LOCK>
    class ExampleClass
      {
    public:
      // A method.
    void DoSomething()
        {
        // Use the Scoped Locking idiom to
        // acquire and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        // Implement the method.
        }
      // ...
    private:
      // The parameterized type strategized locking object.
```

```
LOCK lock_;
// Other data members and methods here...
}
```

При використанні такої реалізації шаблон може бути інстанційованим будь-яким класом, що відповідає сигнатурам aquire та release, які вимагаються блокуванням за областю видимості. Також даний клас не може бути успадкованим від абстрактного базового класу, такого як Lockable.

Підхід на базі параметризованих типів слід застосовувати тоді, коли стратегія блокування відома ще на етапі компіляції програми. З іншого боку поліморфний підхід можна використовувати, якщо стратегія стає відомою лише на етапі виконання або навіть вибирається за допомогою іншого алгоритму. Як завжди, шаблони дають збільшення продуктивності за рахунок повністю прекомпільованого коду синхронізації, а поліморфізм надає більше гнучкості на етапі виконання програми [5, 6].

ІІІ. Визначити сімейство класів-стратегій блокування.

Кожна стратегія має реалізувати певний інтерфейс, що може бути реалізований за допомогою різних специфічних для конкретного застосування паралельних сценаріїв. Якщо відповідні компоненти синхронізації не існують, або вже існуючі компоненти мають інший інтерфейс, шаблон «Фасад» може бути використано для адаптації сигнатур до очікуваних інтерфейсів[7].

Додатково до мютексного блокування можуть бути визначені інші типові стратегії, наприклад блокування читання/запису, семафори, критичні секції, бар'єри тощо. Дуже корисною є так звана стратегія 0-блкування (відповідно що шаблона «Null-об'єкт». Вона визначає ефективну стратегію блокування для непаралельних програм наступним чином:

```
class Null_Mutex
  {
public:
    Null_Mutex (void) { }
   ~Null_Mutex (void) { }
    int acquire (void) { return 0; }
    int release (void) { return 0; }
  };
```

Всі методи даного класу є порожніми встроюваними функціями, які будуть видалені оптимізатором.

IV. Приклади використання

Наступні приклади ілюструють застосування параметризованих типів стратегій блокування для наведеного основного інтерфейсного класу, що може бути настроєний у залежності від конкретного сценарію:

- Однопоточна реалізація:
  typedef ExampleClass<Null_Mutex> EXAMPLE_CLASS;
- багатопоточна реалізація з використанням мютексів:
  typedef ExampleClass<Thread_Mutex> EXAMPLE_CLASS;
- багато поточна реалізація з використанням блокування читання/запису:
  typedef ExampleClass<RW_Lock> EXAMPLE_CLASS.

Слід зауважити, що при створенні цих конфігурацій ExampleClass не потребує внесення додаткових змін. Така прозорість забезпечується використанням блокування на основі стратегій, що абстрагує деталі синхронізації в окремий параметризований тип. Більш того, деталі настроювання можуть бути сховані за допомогою typedef. Таким чином, досить просто оголосити об'єкт типу EXAMPLE_CLASS без розкриття деталей синхронізації.

Синхронізація CUDA-операторів за допомогою стратегій блокування

Як було зазначено в роботах [8, 9] системи алгоритмічних алгебр Глушкова оперують поняттям операторів, пов'язуючи їх у цілісні алгоритми з допомогою формульного запису. Використовуючи шаблон «Команда» для представлення таких «примітивних» алгоритмів можна реалізовувати схеми, подані у вигляді САА, створюючи макрокоманди на рівні об'єктів (окремі оператори можна об'єднувати у послідовності). Таким чином, САА виконують роль абстракції від конкретної платформи, зосереджуючись на особливостях роботи алгоритму.

Схеми, записані з допомогою САА для реалізації алгоритмів допомогою технології NVidia CUDA. Для цього слід використати механізм шаблонів С++:

```
template<class Operation>
__global__ void Kernel(Operation i_operation)
  {
  i_operation.Operate();
  }
```

Використовуючи описаний вище підхід шаблонного блокування можна винести синхронізаційні алгоритми за межі ядра оператора, надаючи таким чином можливість їх заміни на нові, внесення вдосконалень та виправлення помилок (наприклад можна застосувати міжблочну чи міжпотокову синхронізацію.

На діаграмі рис. 3 показано можливість настроювання простого алгоритму з трьох кроків, кожен з яких у схемі представлений абстрактним оператором. Як бачимо, завдяки можливостям поліморфізму можна замінити реалізацію окремого кроку алгоритму на іншу, що дає можливість використовувати багато поточність, GPGPU тощо. Також слід зауважити, що кожен підхід реалізує свою підмножину стратегій блокування, але абстрактна стратегія все одно задається при настроюванні базового оператора, виносячи таким чином блокування на один рівень абстракції із оператором.

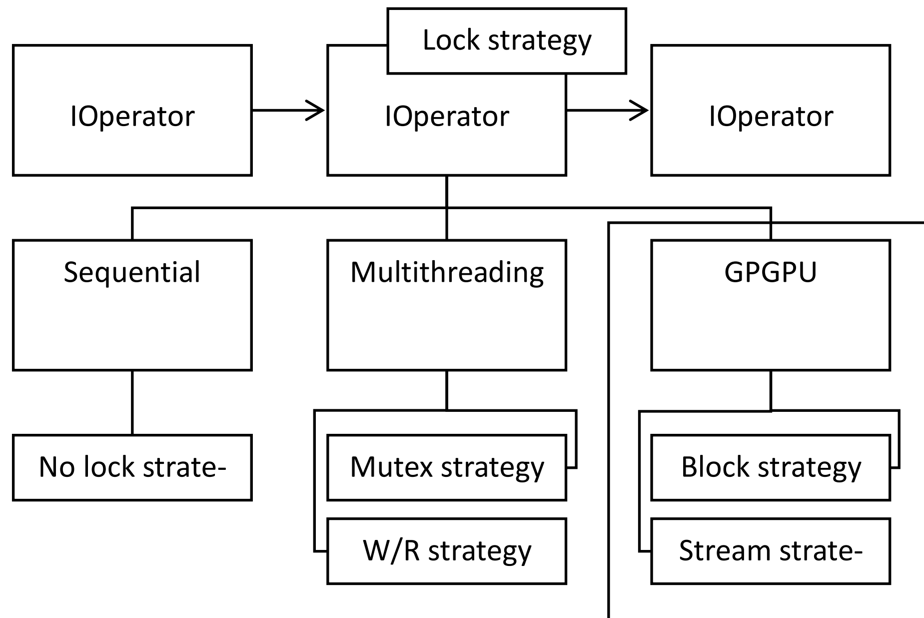

Рис. 3. UML діаграма стратегії блокування на основі параметризованих типів

Висновки

Проаналізовано проблему настроювання механізму синхронізації в об’єктах, що працюють в умовах паралелізму.

Розглянуто використання шаблону «Стратегія» для забезпечення різних сценаріїв синхронізації. Запропоновано підходи на основі принципу поліморфізму та параметризованих типів. Такі шаблони дають збільшення продуктивності за рахунок повністю предкомпільованого коду синхронізації, а поліморфізм надає більше гнучкості на етапі виконання програми.

Запропоновано використання стратегій блокування для гнучкого настроювання синхронізації у реалізації алгоритмів з використанням об’єктів-операторів, які відповідають операторам в аналітичному запису алгоритму в нотації САА-М.

1. *CUDA* Parallel Computing Platform. http://www.nvidia.com/object/cuda_home_new.html
2. *Погорілий С.Д., Верещинський О.А. Вітель Д.Ю.* Новітні архітектури відеоадаптерів. Технологія GPGPU (Ч. 1) // Реєстрація, зберігання і обробка даних, К.: 2012. – Т. 14. – № 4.
3. *Погорілий С.Д., Верещинський О.А. Вітель Д.Ю.* Новітні архітектури відеоадаптерів. Технологія GPGPU (Ч. 2) // Реєстрація, зберігання і обробка даних, К.: 2012. – Т. 15. – № 1.
4. *Douglas C. Schmidt,* Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components, C++ Report, SIGS, Vol. 11, No. 9, September, 1999.
5. *Погорілий С.Д., Калита Д.М.* Оптимізація алгоритмів маршрутизації з використанням систем алгоритмічних алгебр // УСиМ. – 2000. – № 4. – С. 20–30.
6. *Pogorilyy S.D., Gusarov A.D.* Paralleling Of Edmonds-Karp Net Flow Algorithm // Appl. Comput. Math. 5. – 2006. – N 2. – P. 121–130.
7. *Levchenko R.I., Sudakov O.O., Pogorilyy S.D.* DDCI: Simple Dynamic Semiautomatic Parallelizing for Heterogeneous Multicomputer Systems // Proceedings of the 5th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 21 – 23 September 2009, Rende (Cosenza), Italy.
8. *ПогорілийС.Д., ВерещинськийО.А.* Система алгоритмічних алгебр Глушкова як формалізований підхід до шаблонів проектування // К.: Theoretical and Applied Aspects of Program Systems Development, December 3-7, 2012.
9. *Погорілий С.Д., Верещинський О.А.* Створення методики проектування застосувань для програмно-апаратної платформи CUDA // Проблеми програмування. – 2013. – № 3. –С. 47–60.