ЛАБОРАТОРНА РОБОТА №5

Назва роботи: функційне програмування.

Мета роботи: ознайомитися з парадигмою функційного програмування.

1. Парадигма функційного прогрмування

Коли в об'єктно-орієнтованому програмуванні, програміст розглядає програму як множину взаємодіючих між собою об'єктів, то у функційному програмуванні програму можна представити як обчислення послідовності функцій, що не зберігають ніяких станів. Тобто терміном функційне програмування (ФП) називають такий стиль програмування, при якому результат функції залежить тільки від її параметрів, а не від зовнішнього стану. Це прямо співвідноситься з поняттям функції в математиці і означає, що якщо два рази викликати функцію з одними і тими ж параметрами, то вийдуть однакові результати. Такою властивістю володіють багато математичних функції в стандартній бібліотеці C++, наприклад sin, cos і sqrt, а також прості операції над примітивними типами, наприклад 3 + 3, 6 * 9 або 1.3 / 4.7. Чиста функція не модифіковує ніяких зовнішніх станів, вона впливає тільки на значення, що повертається.

При такому підході стає простіше міркувати про функції, особливо в присутності паралелізму, оскільки багато проблем, пов'язаних з пам'яттю, просто не виникають. Якщо колективні дані не модифікуються, то не може бути ніякої гонки і, отже, не потрібно захищати дані за допомогою м'ютексів. Це спрощення настільки істотне, що в програмуванні паралельних систем все більш популярні стають такі мови, як Haskell, де всі функції чисті за замовчуванням.

Але функційне програмування застосовується не тільки в мовах, де ця парадигма застосовується за умовчанням. C++ – мультипарадигменна мова, і на ній, безумовно, можна писати програми в стилі ФП. З появою в C++11 ***лямбда-функцій***, включенням шаблону std::bind з Boost та TR1 і додаванням ***автоматичного виведення типу*** це стало простіше ніж в C++98. ***Майбутні результати***(*std::future*) – це останній елемент з тих, що дозволяють реалізувати в C++ паралелізм в стилі ФП; завдяки передачі ***майбутніх результатів*** результат одного обчислення можна зробити залежним від результату іншого без явного доступу до даних, що розділяються.

2. Приклад ФП.

2.1. Швидке сортування в стилі ФП.

Щоб продемонструвати використання ***майбутніх результатів*** при написанні паралельних програм в дусі ФП, розглянемо просту реалізацію алгоритму швидкого сортування. Основна ідея алгоритму проста: маючи список значень, вибрати якийсь опорний елемент і розбити список на дві частини – в одну увійдуть елементи, менші опорного, в іншу – більші чи рівні опорного. Відсортований список виходить шляхом сортування обох частин і об'єднання трьох списків: відсортованої множини елементів менших опорного елементу, самого опорного елементу і відсортованої множини елементів більших або рівних опорному елементу. На рис. 2.1 показано, як цей алгоритм сортує список з 10 цілих чисел. У лістингу нижче приведена послідовна реалізація алгоритму в

стилі ФП; в ній список приймається і повертається за значенням, а не сортується за місцем в std::sort().

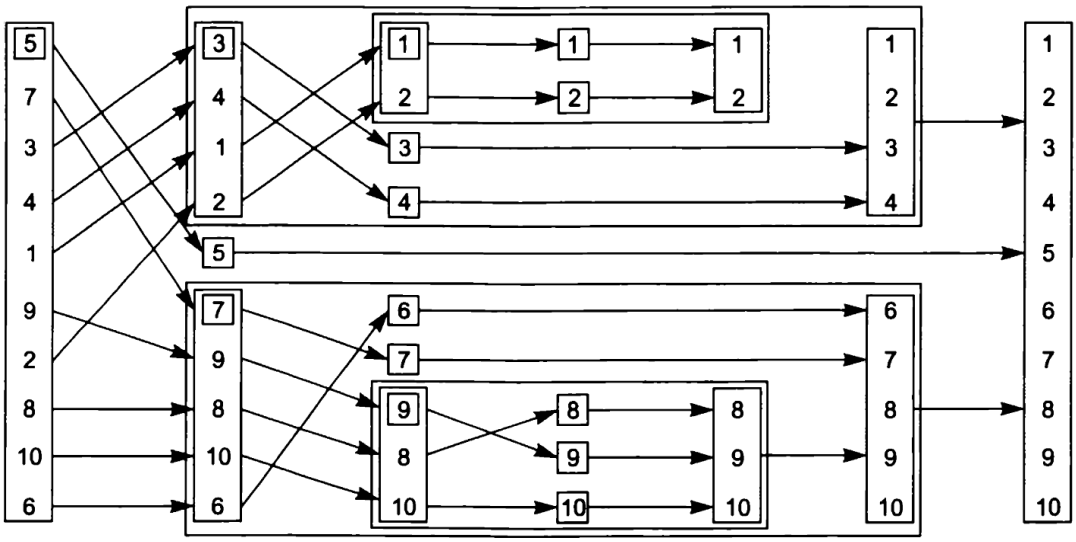

Рис. 2.1. Рекурсивне сортування з використанням ФП

Лістинг 2.1

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input){
   if(input.empty()){
      return input;
   }
   std::list<T> result;
   result.splice(result.begin(), input, input.begin()); // (1)

  T const& pivot=*result.begin(); // (2)
   auto divide_point=std::partition(input.begin(), input.end(), [&](T const& t){return t < pivot;}); //(3)

   std::list<T> lower_part;
   lower_part.splice(lower_part.end(),input,input.begin(), divide_point); // (4)

   auto new_lower(sequential_quick_sort(std::move(1ower_part))); // (5)
   auto new_higher(sequential_quick_sort(std::move(input))); // (6)

   result.splice(result.end(),new_higher); // (7)
   result.splice(result.begin(),new_lower); // (8)

   return result;
}
```

Хоча інтерфейс витриманий в стилі ФП, пряме застосування ФП призвело б до невиправдано великої кількості операцій копіювання, тому всередині ми використовуємо «звичайний» імперативний стиль. В якості опорного ми вибираємо перший елемент і відрізаємо його від списку за допомогою функції *splice()* (1). Потенційно це може привести до неоптимального сортування (в термінах кількості операцій порівняння і обміну), але будь-який інший підхід при роботі з *std::list* може істотно збільшити час за рахунок обходу списку. Ми знаємо, що цей елемент повинен увійти в результат, тому можемо відразу помістити його в список, де буде зберігатися результат. Далі ми хочемо використовувати цей елемент для порівняння, тому беремо посилання на нього, щоб

уникнути копіювання (2). Тепер можна з допомогою алгоритму *std::partition()* розбити послідовність на дві частини: менші опорного елементу і не менші опорного елементу (3). Критерій розбиття найпростіше задати за допомогою ***лямбда-функції***. Тут ми запам'ятовуємо посилання в ***замиканні***, щоб не копіювати значення *pivot*.

Алгоритм *std::partition()* переупорядковує список на місці і повертає ітератор, який вказує на перший елемент, що не менший опорного значення. Повний тип ітератора досить довгий, тому використовуємо специфікатор типу auto, щоб компілятор вивів його самостійно.

Оскільки дотримуємося парадигми ФП, то для рекурсивного сортування обох «половин» потрібно створити два списки. Для цього ми знову використовуємо функцію *splice()*, щоб перемістити значення зі списку *input* включно до *divide_point* в новий список *lower_part* (4). Після цього *input* буде містити тільки всі інші значення. Далі обидва списки можна відсортувати шляхом рекурсивних викликів (5), (6). Застосовуючи *std::move()* для передачі списків, ми уникаємо копіювання – результат в будь-якому випадку неявно переміщується. Нарешті, ми ще раз викликаємо *splice()*, щоб скласти *result* в правильному порядку. Значення зі списку *new_higher* потрапляють в кінець списку (7), після опорного елементу, а значення зі списку *new_lower* – в початок списку, до опорного елемента (8).

2.2. Паралельне швидке сортування в стилі ФП.

Застосувавши функціональний стиль програмування, можна разпаралелити цей код за допомогою ***майбутніх результатів***, як показано в лістингу нижче. Набір операцій той же, що і раніше, тільки деякі з них виконуються паралельно.

Лістинг 2.2

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input){
   if(input.empty()){
      return input;
   }
   std::list<T> result;
   result.splice(result.begin(),input,input.begin());

   T const& pivot=*result.begin();
   auto divide_point=std::partition(input.begin(),input.end(),[&](T const& t){return t < pivot;});

   std::list<T> lower_part;
   lower_part.splice(lower_part.end(), input,input.begin(), divide_point);

   std::future<std::list<T> > new_lower(std::async(&parallel_quick_sort<T>,std::move(lower_part))); // (1)
   auto new_higher(parallel_quick_sort(std::move(input)) ) ;  // (2)

   result.splice(result.end(),new_higher); // (3)
   result.splice(result.begin(),new_lower.get()); // (4)

   return result;
}
```

Істотна зміна тут полягає в тому, що сортування у нижній частині списку проводиться не в поточному, а в окремому потоці – за допомогою *std::async()* (1). Верхня частина списку, як і раніше, сортується шляхом прямої рекурсії (2). Рекурсивно викликаючи *parallel_quick_sort()*, ми можемо задіяти доступний апаратний паралелізм. Якщо *std::async()* створює новий потік при кожному зверненні, то після трьох рівнів рекурсії ми отримаємо вісім працюючих потоків, а після 10 рівнів (коли в списку

приблизно 1000 елементів) буде працювати 1024 потоки, якщо це апаратно підтримується. Якщо бібліотека вирішить, що запущено занадто багато завдань (приміром, тому що кількість завдань перевищила рівень апаратного паралелізму), то може перейти в режим синхронного запуску нових завдань. Тоді нове завдання буде працювати в тому ж потоці, який звернувся до *get()*, а не в новому, так що ми не будемо нести витрати на передачу завдання новому потоку. Варто зазначити, що відповідно до стандарту реалізація *std::async* вправі як створювати новий потік для кожного завдання (навіть при значному перевищенні ліміту), якщо явно не заданий прапорець *std::launch::deferred*, так і запускати всі завдання синхронно, якщо явно не заданий прапоруць *std::launch::async*.

Можна не використовувати *std::async()*, а написати свою функцію *spawn_task()*, яка буде служити обгорткою навколо *std::packaged_task* і *std::thread*, як показано в лістингу 2.3; потрібно створити об'єкт *std::packaged_task* для зберігання результату виклику функції, отримати з нього ***майбутній результат***, запустити завдання в окремому потоці і повернути ***майбутній результат***. Само по собі це не дає великої переваги (і, швидше за все, призведе до значного перевищення ліміту готових до виконання потоків), але прокладає дорогу до переходу на більш хитромудру реалізацію, яка поміщає завдання в чергу, яка обслуговується ***пулом потоків***. Розглядати ***пули потоків*** не будемо, крім того, йти по такому шляху замість використання *std::async* має сенс тільки в тому випадку, коли ви точно знаєте, що робите, і хочете повністю контролювати, як ***пул потоків*** будується і виконує завдання.

Повернемося до функції *parallei_quick_sort*. Оскільки для отримання *new_higher* ми застосовували пряму рекурсію, то і склеїти (*splice*) його можна на місці, як і раніше (3). Але *new_lower* тепер представляє собою не список, а об'єкт *std::future<std::list<T>>*, тому спочатку потрібно витягти значення за допомогою *get()*, а тільки потім викликати *splice()* (4). Таким чином, ми дочекаємося завершення фонового завдання, а потім *перемістимо* результат в параметр *splice()*; функція *get()* повертає посилання на r-значення збереженого результату, отже, його можна *перемістити*.

Навіть припускаючи, що *std::async()* оптимально використовує доступний апаратний паралелізм, наведена реалізація все одно не ідеальна. Основна проблема в тому, що *std::partition* робить багато роботи і залишається послідовною операцією. Якщо потрібна максимально швидка паралельна реалізація, то можна використати підхід з лабораторної роботи №3.

Лістинг 2.3

```
template<typename F,typename A>
std::future<typename std::result_of<F(A&&)>::type>spawn_task(F&& f,A&& a){
   typedef std::result_of<F(A&&)>::type result_type;
   std::packaged_task<result_type(A&&)> task(std::move(f)));
   std::future<result_type> res(task.get_future());
   std::thread t(std::move(task), std::move(a));
    t.detach();
    return res;
}
```

Функційне програмування – не єдина парадигма паралельного програмування, що дозволяє уникнути модифікації поділюваних даних. Альтернативою є парадигма CSP (Communicating Sequential Processes - взаємодіючі послідовні процеси, http://www.usingcsp.com/cspbook.pdf), в якій потоки концептуально розглядаються як повністю незалежні сутності, без будь-яких поділюваних даних, але з'єднані комунікаційними каналами, по яких передаються повідомлення. Ця парадигма покладена в основу мови програмування Erlang (http: //www.erlang.org /) і середовища MPI (Message

Passing Interface, http://www.mpi-forum.org/), яке широко використовується для високопродуктивних обчислень на C і C++.

3. Приклад програми.

Лістинг 3.1

```
// don't forget to use compilation keys for Linux: -lpthread -std=c++11
//#include "stdafx.h" //
#include <iostream>
#include <iterator>
#include <list>
#include <future>
#include <stack>
#include <thread>
//#include <exception>
// #include <mutex>
// #include <memory>
// #include <chrono>
#include <vector>
#include <algorithm>

#define INPUT_DATA { 5, 7, 3, 4, 1, 9, 2, 8, 10, 6 }


template<typename F, typename A>
std::future<typename std::result_of<F(A&&)>::type>
spawn_task(F&& f, A&& a){
	//typedef std::result_of<F(A&&)>::type result_type; // C++14
	typedef std::result_of<F(A&&)> result_type;
	std::packaged_task<result_type(A&&)> task(std::move(f));
	std::future<result_type> res(task.get_future());
	std::thread t(std::move(task), std::move(a));
	t.detach();
	return res;
}

template < typename T >
std::list < T > sequential_quick_sort(std::list<T> input){
	if (input.empty()){
		return input;
	}
	std::list<T> result;
	result.splice(result.begin(), input, input.begin()); // (1)

	T const& pivot = *result.begin(); // (2)
	auto divide_point = std::partition(input.begin(), input.end(), [&](T const&
t){return t < pivot; }); //(3)

	std::list<T> lower_part;
	lower_part.splice(lower_part.end(), input, input.begin(), divide_point); // (4)

	auto new_lower(sequential_quick_sort(std::move(lower_part))); // (5)
	auto new_higher(sequential_quick_sort(std::move(input))); // (6)

	result.splice(result.end(), new_higher); // (7)
	result.splice(result.begin(), new_lower); // (8)

	return result;
}


template < typename T >
std::list < T > parallel_quick_sort(std::list<T> input){
	if (input.empty()){
```

```
            return input;
        }
        std::list<T> result;
        result.splice(result.begin(), input, input.begin());

        T const& pivot = *result.begin();
        auto divide_point = std::partition(input.begin(), input.end(), [&](T const&
t){return t < pivot; });

        std::list<T> lower_part;
        lower_part.splice(lower_part.end(), input, input.begin(), divide_point);

        std::future<std::list<T> > new_lower(std::async(&parallel_quick_sort<T>,
std::move(lower_part))); // (1)
        auto new_higher(parallel_quick_sort(std::move(input)));  // (2)

        result.splice(result.end(), new_higher); // (3)
        result.splice(result.begin(), new_lower.get()); // (4)

        return result;
}

int main()
{
        std::list<unsigned long long int> input(INPUT_DATA);
        std::list <unsigned long long int> output = sequential_quick_sort(input);

        std::cout << "input:" << std::endl;
        std::copy(input.begin(), input.end(), std::ostream_iterator<unsigned long long
int>(std::cout, ", "));
        std::cout << std::endl << std::endl;

        std::cout << "output:" << std::endl;
        std::copy(output.begin(), output.end(), std::ostream_iterator<unsigned long long
int>(std::cout, ", "));
        std::cout << std::endl << std::endl;

        std::cout << "Press any key to continue . . .";
        std::cin.get();

        return 0;
}
```

4. Варіанти завдань

n	Вхідні дані
1	125, 126, 121, 121, 117, 120, 121, 116, 116, 112, 119, 111, 116, 117, 112, 114, 106, 111, 112, 107
2	145, 146, 141, 141, 137, 140, 141, 136, 136, 132, 139, 131, 136, 137, 132, 134, 126, 131, 132, 12
3	165, 166, 161, 161, 157, 160, 161, 156, 156, 152, 159, 151, 156, 157, 152, 154, 146, 151, 152, 147
4	185, 186, 181, 181, 177, 180, 181, 176, 176, 172, 179, 171, 176, 177, 172, 174, 166, 171, 172, 167
5	205, 206, 201, 201, 197, 200, 201, 196, 196, 192, 199, 191, 196, 197, 192, 194, 186, 191, 192, 187

6	225, 226, 221, 221, 217, 220, 221, 216, 216, 212, 219, 211, 216, 217, 212, 214, 206, 211, 212, 207
7	245, 246, 241, 241, 237, 240, 241, 236, 236, 232, 239, 231, 236, 237, 232, 234, 226, 231, 232, 227
8	265, 266, 261, 261, 257, 260, 261, 256, 256, 252, 259, 251, 256, 257, 252, 254, 246, 251, 252, 247
9	285, 286, 281, 281, 277, 280, 281, 276, 276, 272, 279, 271, 276, 277, 272, 274, 266, 271, 272, 267
10	305, 306, 301, 301, 297, 300, 301, 296, 296, 292, 299, 291, 296, 297, 292, 294, 286, 291, 292, 287
11	325, 326, 321, 321, 317, 320, 321, 316, 316, 312, 319, 311, 316, 317, 312, 314, 306, 311, 312, 307
12	345, 346, 341, 341, 337, 340, 341, 336, 336, 332, 339, 331, 336, 337, 332, 334, 326, 331, 332, 327
13	365, 366, 361, 361, 357, 360, 361, 356, 356, 352, 359, 351, 356, 357, 352, 354, 346, 351, 352, 347
14	385, 386, 381, 381, 377, 380, 381, 376, 376, 372, 379, 371, 376, 377, 372, 374, 366, 371, 372, 367
15	405, 406, 401, 401, 397, 400, 401, 396, 396, 392, 399, 391, 396, 397, 392, 394, 386, 391, 392, 387
16	425, 426, 421, 421, 417, 420, 421, 416, 416, 412, 419, 411, 416, 417, 412, 414, 406, 411, 412, 407
17	445, 446, 441, 441, 437, 440, 441, 436, 436, 432, 439, 431, 436, 437, 432, 434, 426, 431, 432, 427
18	465, 466, 461, 461, 457, 460, 461, 456, 456, 452, 459, 451, 456, 457, 452, 454, 446, 451, 452, 447
19	485, 486, 481, 481, 477, 480, 481, 476, 476, 472, 479, 471, 476, 477, 472, 474, 466, 471, 472, 467
20	505, 506, 501, 501, 497, 500, 501, 496, 496, 492, 499, 491, 496, 497, 492, 494, 486, 491, 492, 487
21	525, 526, 521, 521, 517, 520, 521, 516, 516, 512, 519, 511, 516, 517, 512, 514, 506, 511, 512, 507
22	545, 546, 541, 541, 537, 540, 541, 536, 536, 532, 539, 531, 536, 537, 532, 534, 526, 531, 532, 527
23	565, 566, 561, 561, 557, 560, 561, 556, 556, 552, 559, 551, 556, 557, 552, 554, 546, 551, 552, 547
24	585, 586, 581, 581, 577, 580, 581, 576, 576, 572, 579, 571, 576, 577, 572, 574, 566, 571, 572, 567
25	605, 606, 601, 601, 597, 600, 601, 596, 596, 592, 599, 591, 596, 597, 592, 594, 586, 591, 592, 587
26	625, 626, 621, 621, 617, 620, 621, 616, 616, 612, 619, 611, 616, 617, 612, 614, 606, 611, 612, 607
27	645, 646, 641, 641, 637, 640, 641, 636, 636, 632, 639, 631, 636, 637, 632, 634, 626, 631, 632, 627
28	665, 666, 661, 661, 657, 660, 661, 656, 656, 652, 659, 651, 656, 657, 652, 654, 646, 651, 652, 647
29	685, 686, 681, 681, 677, 680, 681, 676, 676, 672, 679, 671, 676, 677, 672, 674, 666, 671, 672, 667
30	705, 706, 701, 701, 697, 700, 701, 696, 696, 692, 699, 691, 696, 697, 692, 694, 686, 691, 692, 687
n – порядковий номер у журналі	

5. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.