პარალელური ალგორითმები

განხილული საკითხები:

- სტანდარტული ბიბლიოთეკის ალგორითმების გაპარალელურება
- განხორციელების ფანდები >>>
- C++ ბიბლიოოთეკის პარალელური ალგორითმები >>>

სავარჯიშოები >>>

სტანდარტული ბიბლიოთეკის ალგორითმების გაპარალელურება

C++17-მა დაიმატა პარალელური ალგორითმის ცნება C++ -ის სტანდარტულ ბიბლიოთეკაში. ამჟამად დიაპაზონებზე მოქმედი ფუნქციების უმეტესობისთვის არსებობს დამატებული გადატვირთვები პარალელური შემთხვევისთვის. პარალელურ გადატვირთვას იგივე ხელმოწერა აქვს, რაც აქამდე არსებულ სერიულს, ოღონდ აქვს დამატებული პირველი პარამეტრი, რაც განსაზღვრავს დაპარალელების განხორციელების ფანდს (პოლიტიკას). მაგალითად:

```
std::vector<int> my_data;
      . . .
std::sort(std::execution::par, my_data.begin(), my_data.end());
```

განხორციელების ფანდი `std::execution::par` მიუთითებს სტანდარტულ ბიბლიოთეკას, რომ ნებადართულია პარალელური ალგორითმის გამოძახება მრავალი დინების გამოყენებით. თუმცა ეს არის ნებართვა და არა მოთხოვნა. ბიბლიოთეკას შეუძლია კოდი შეასრულოს სერიულადაც. ისიც შევნიშნოთ, რომ განხორციელელბის ფანდის მითითებით ალგორითმის სირთულეზე მოთხოვნები იცვლება ხშირად შედარებით მსუბუქი ხდება.

<<< განხორციელების ფანდები

სტანდარტი განსაზღვრავს განხორციელების სამ ფანდს:

- `std::execution::sequenced_policy`
- `std::execution::parallel_policy`
- `std::execution::parallel_unsequenced_policy`

ეს კლასები განსაზღვრულია `<execution>` სათაურში. სტანდარტი განსაზღვრავს სამ შესაბამისი ფანდ ობიექტს ალგორითმისთვის გადასაწოდებლად:

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`

პროგრამისტს არ შეუძლია განსაზღვროს საკუთარი ფანდი პარალელურობის განხორციელებისთვის.

განხორციელების ფანდის განსაზღვრის ზოგადი გავლენები

მას შემდეგ, რაც ალგორითმს გადაეწოდა განხორციელების ფანდი, ეს უკანასკნელი გავლენას ახდენს ალგორითმის სირთულეზე, გამონაკლისების გამოსროლაზე და აგრეთვე თუ სად, როგორ და როდის შესრულდება ალგორითმის ბიჯები.

პარალელური განხორიელების მართვას თავისი ზედნადები ხარჯები აქვს. ამას, გარდა, ბევრი პარალელური ალგორითმი დამატებით კიდევ ასრულებს სხვადასხვა მოქმედებას იმ იმედით, რომ მთლიანობაში გააუმჯობესებს ალგორითმის წარმადობას დახარჯული დროის თვალსაზრისით. წვრილმანები დამოკიდებულია ცალკეულ ალგორითმზე, მაგრამ ზოგადად თუ სერიული ალგორითმი გვიდგენს რომ რაღაცა შესრულდება ზუსტად t დროში, ან არაუმეტეს t დროში, მაშინ განხორციელების ფანდის ზედნადები დაასუსტებს ამ მოთხოვნას O(t)-მდე.

თუ ალგორითმი სრულდება განხორციელების მითითებული ფანდით და გამონაკლისი გამოვარდა, მაშინ შედეგები განისაზღვრება შესაბამისი ფანდით. სამივე ფანდი გამოიძახებს `std::terminate`-ს ნებისმიერი დაუმუშავებელი (uncaught) გამონაკლისის შემთხვევაში. ერთადერთი გამონაკლისი ამ ფანდების შემთხვევაში შესაძლოა იყოს `std::bad_alloc`, როცა ბიბლიოთეკა ვერ მიიღებს საკმარის მეხსიერებას თავისი შიგა ოპერაციებისთვის.

განხორციელების ფანდი განსაზღვრავს თუ რომელი აღმასრულებელი მხარეები არიან ჩარტული ალგორითმის ბიჯების შესრულებაში. შესაძლოა ესენი იყოს, დინებები, ვექტორული ნაკადები, GPU დინებები ან სხვა რამ. განხორციელების ფანდი ასევე განსაზღვრავს რიგობრივ შეზღუდვებს ალგორითმის ბიჯების შესრულებაზე: სრულდებიან ისინი რაიმე კერძო რიგით, ალგორითმის სხვადასხვა ნაწილი ურთიერთმონაცვლეობს (დროში არ იკვეთება) თუ ერთმანეთის პარალელურად მიდის, და ა.შ..

ფანდი `std::execution::sequenced_policy` არაა დაპარალელებისთვის. იგი აიძულებს იმპლემენტაციას რომ ყველა მოქმედება შეასრულოს იმ დინებაზე რომელმაც გამოიძახა ფუნქცია. უფრო მეტი, ოპერაციები უნდა შესრულდეს განსაზღვრული რიგით, ურთიერთმონაცვლეობის გარეშე. ზუსტი რიგი არაა დადგენილი, შესაძლოა იგი იცვლებოდეს ფუნქციის სხვადასხვა გაშვებაში. გარანტია არაა რომ იგივე რიგით შესრულდება, როგორც ფანდის მიტითების გარეშე.

ფანდი `std::execution::parallel_policy` უზრუნველყოფს პარალელურ განხორციელებას დინებების გარკვეული რაოდენობის მიერ. მოქმედებები შესაძლოა შესრულდეს ან იმ დინების მიერ, რომელმაც გამოიძახა ალგორითმი, ან ბიბლიოთეკის მიერ გახსნილი დინებების მიერ. მოქმედებები სრულდება განსაზღვრული რიგით, ურთიერთმონაცვლეობის გარეშე, მაგრამ ზუსტი რიგი განუსაზღვრელია. თითოეული მოქმედება (ოპერაცია) სრულდება ერთი რომელიმე დინების მიერ. ამ ფანდის გამოყენება იწვევს დამატებით შეზღუდვებს იტერატორებზე, მნიშვნელობებზე და გამოძახებად ობიექტებზე დაწვრილებით ქვემოთ ვნახავთ) - ისინი არ უნდა ქმნიდნენ დასწრების ვითარებას.

ფანდი `std::execution::parallel_unsequenced_policy` უზრუნველყოფს ალგორითმის დაპარალელებისთვის უდიდეს შესაძლო მხარდაჭერას, სამაგიეროდ ადევს მკაცრ შეზღუდვებს იტერატორებს, მნიშვნელობებს და გამოძახებად ობიექტებს.

ამ ფანდის მიერ გამოძახებული ალგორითმის ოპერაციები ერთ რომელიმე დინებაში შეიძლება სრულდებოდეს ისე, რომ მეორე დაიწყოს პირველის დამთავრებამდე იგივე დინებაში, შესაძლოა ოპერაცია დაიწყოს ერთ დინებაში და დამთავრდეს მეორეში.

როდესაც ამ ფანდს ვიყენებთ, იტერატორებზე, მნიშვნელობებზე და გამოძახებად ობიექტებზე შესრულებული მოქმედებები არ უნდა იყენებდნენ რაიმე სახის სინქრონიზებას.

<<< C++ ბიბლიოთეკის პარალელური ალგორითმები

`<algorithm>` და `<numeric>` სათაურებიდან აღებული ალგორითმების უმეტესობას აქვს გადატვირთვა, რომელიც იღებს განხორციელების ფანდს. მათ შორისაა:

`all_of, any_of, none_of, for_each, for_each_n, find, find_if, find_end, find_first_of, adjacent_find, count, count_if, mismatch, equal, search, search_n, copy, copy_n, copy_if, move, swap_ranges, transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, is_partitioned, partition, stable_partition, partition_copy, sort, stable_sort, partial_sort, partial_sort_copy, is_sorted, is_sorted_until, nth_element, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, is_heap, is_heap_until, min_element, max_element, minmax_element, lexicographical_compare, reduce, transform_reduce, exclusive_scan, inclusive_scan, transform_exclusive_scan, transform_inclusive_scan, adjacent_difference`.

ეს პარალელიზებადი ალგორითმების თითქმის სრული სიაა. არსებით გამონაკლისს წარმოადგენს ალგორითმი `std::accumulate`. მისი პარალელიზებადი ორეული იწოდება `std::reduce`-ად, თუმცა მისთვის მიწოდებული ბინარული ოპერატორი უნდა იყოს ასოციაციური და კომუტაციური. თუ ასე არაა - პასუხი დამოკიდებულია მოქმედებების მიმდევრობაზე და არის არასანდო.

ამ სიის ყოველი ალგორითმის სერიულ სახეობას (განხორციელების ფანდის გარეშე) აქვს ორი გადატვირთვა. მაგალითად, დახარისხების ალგორითმის გადატვირთვებია:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort(
	RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

ამათ კიდევ ემატება ორი გადატვირთვა განხორციელების ფანდით:

```
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(
	ExecutionPolicy&& exec,
	RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(
	ExecutionPolicy&& exec,
	RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

ამ ალგორითმს არ ეხება, მაგრამ განხორციელების ფანდის გამოყენება შესაძლებელი არის მხოლოდ მრავალი გავლის იტერატორებისთვის. თუ განხორციელების ფანდის გარეშე ალგორითმს სჭირდება *Input Iterators* ან *Output Iterators,*

მის გადატვირთვებს განხორციელების ფანდით დასჭირდებათ *Forward Iterators*. ეს ადვილი ასახსნელია, რამდენიმე დინება რომ დაინაწილებს საქმეებს, მათ დასჭირდებათ თავ-თავისი დიაპაზონების გადაწოდება. ამის გაკეთება შეუძლებელია თუ არ მოხდა იტერატორების დამახსოვრება შესაბამისი ტიპის ცვლადებში. უმარტივეს იტერატორებს, როდესაც ნაკადებთან მუშაობენ (istream_iterator<>, ostream_iterator<>), ფიზიკურად არ აქვთ დამახსოვრების შესაძლებლობა.

მაგალითად, ტიპური გაასლების ალგორითმის ხელმოწერა თუ არის

```
template<class InputIterator, class OutputIterator>
OutputIterator copy( InputIterator first, InputIterator last, OutputIterator result);
```

გადატვირთვა განხორციელების ფანდით არის

```
template<class ExecutionPolicy,class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy
(
	ExecutionPolicy&& policy,
	ForwardIterator1 first, ForwardIterator1 last,
	ForwardIterator2 result

);
```

`ForwardIterator` მოითხოვს, რომ იტერატორის განმისამართება აბრუნებს რეალურ რეფერენსს მნიშვნელობაზე. ამისგან განსხვავებით უმარტივესი იტერატორები საშუალებას იძლევიან რომ დაბრუნდეს შემცვლელი (proxy) ტიპებიც.

<<< სავარჯიშოები