რამდენიმე პარალელური ალგორითმი იტერატორების დიაპაზონისთვის

საკითხები:

ორი ერთდროული დინების პირველი მაგალითი. განვიხილოთ კარგად ნაცნობი პროგრამა, რომლითაც, როგორც წესი, იწყება ყველა ახალი თემის შესწავლა პროგრამირებაში - პროგრამა „გამარჯობა, სამყარო“. მისი ერთდინებიანი სახე არის:

```
#include <iostream>
int main()
{
	std::cout << "Hello World\n";
}
```

მისი სხვა ვარიანტი არის:

```
#include <iostream>
#include <thread>                  //  #1
void hello()                       //  #2
{
	std::cout << "Hello Concurrent World\n";
}
int main()
{
	std::thread t(hello);      //  #3
	t.join();                  //  #4
}
```

პირველი განსხვავება არის #1. გასაგებია რასაც აკეთებს. შემდეგ, გზავნილის დასაბეჭდი გადატანილია ცალკე ფუნქციაში - #2. ეს იმიტომ, რომ ყოველ დინებას უნდა ჰქონდეს საწყისი ფუნქცია, რომელშიც (thread of execution) აღმასრულებელი დინება დაიწყება. მოცემულ დანართში, საწყისი დინებისთვის ასეთი ფუნქცია არის `main()`. ყველა სხვა დანარჩენი დინებისთვის, ასეთი ფუნქცია განსაზღვრული უნდა იყოს `std::thread` ობიექტის კონსტრუქტორში. ჩვენს შემთხვევაში #3. მას შემდეგ რაც ახალი დინება გაეშვება, საწყისი დინება აგრძელებს შესრულებას. თუ იგი არ დაელოდება ახალი დინების დასრულებას, მაშინ პროგრამა დამთავრდება და ახალ დინებას არ მიეცემა დამთავრების საშუალება. ამის გამო არის #4 სტრიქონი ასეთი.

<<< რა ღირს ახალი დინების გაშვება? როგორც აღვნიშნეთ, ახალი დინების გაშვება ზედნადებ ხარჯებთანაა დაკავშირებული. მიკროწამებში ამის გაზომვა შეგვიძლია შემდეგი მარტივი კოდის გამოყენებით:

```
#include <iostream>
#include"chrono"
#include <thread>
using namespace std;

void func() {}
int main()
{
```

```
	thread t;
	auto st = chrono::high_resolution_clock::now();

	t= thread (func);
	t.join();
	auto diff = chrono::high_resolution_clock::now() - st;
	auto time = chrono::duration_cast<chrono::microseconds>(diff);
	cout << time.count() << endl;
}
```

<<< **დინების გაშვება.** C++ -ის ყოველ პროგრამას აქვს სულ მცირე ერთი დინება, რომელიც გამოიძახებს main() პროგრამას. ახალი დინების გაშვება იწყება `std::thread` ობიექტის აგებით, რომელიც განსაზღვრავს დინებაში გასაშვებ ამოცანას. მარტივ შემთხვევაში, ეს ამოცანა შესაძლოა იყოს ფუნქცია void მნიშვნელობით, რომლის დამთავრების შემდეგ დინება გაჩერდება. უფრო რთულ შემთხვევებში, გასაშვები ამოცანა შესაძლოა იყოს ფუნქციური ობიექტი, რომლის სხვადასხვა ნაწილის შესრულება და დინების გაჩერება უნდა მოხდეს გზავნილების რაიმე სისტემით. ვნახოთ ორივეს მაგალითი:

```
void some_func();
std::thread my_thread(some_func);
```

ხოლო გამოძახებადი ობიექტისთვის:

```
class background_task
{
public:
	void operator()() const
	{
		some_func1();
		some_func2();
	}
};
background_task f;
std::thread my_thread(f);
```

აქ, გადაწოდებული ობიექტი ასლდება ახალი დინების საცავში, საიდანაც ხდება მისი გამოძახება. შეგვიძლია დროებითი ობიექტის გადაწოდებაც, თუმცა ამ დროს ყურადღებაა საჭირო, რომ დინების ნაცვლად არ მივიღოთ ფუნქციის განაცხადი

```
std::thread my_thread(background_task());
```

რომელიც ქმნის ფუნქციას ერთადერთი პარამეტრით (უპარამეტრო ფუნქციის პოინტერი). იმის მისაღებად, რაც სინამდვილეში გვინდა, უნდა ავიღოთ შემდეგი ორიდან ერთ-ერთი განაცხადი:

```
std::thread my_thread((background_task()));
std::thread my_thread{ background_task() };
```

აღნიშნულ პრობლემას აგრეთვე ადვილად ვაღწევთ თავს, თუ დინებაში გასაშვებად გადავაწვდით ლამბდა ფუნქციას. მაგალითად, შემდეგი ფუნქტორის

```
class background_task
{
public:
	void operator()() const
	{
		do_something();
		do_something_else();
	}
};
```

შესაბამისი ობიექტის ნაცვლად, დინებას გადავაწოდოთ როგორც ანომიმი ლამბდა ფუნქცია, რომელიც იგივე საქმეს აკეთებს:

```
std::thread my_thread([]{
	do_something();
	do_something_else();
});
```

<<< **დინების დასრულება.** ვნახოთ რა მოხდება, თუ გავუშვით დინება, მაგრამ არ ვიზრუნეთ მის დასრულებაზე:

```
#include <iostream>
#include <thread>
using namespace std;

void func() { cout << "Hello!" << endl; }
int main()
{
	thread t = thread(func);
	this_thread::sleep_for(std::chrono::seconds(5));
}
```

როგორც ვხედავთ, პროგრამა ავარიულად მთავრდება, მიუხედავად იმისა, რომ ჩვენ ვიზრუნეთ და გამოვყავით საკმარისი დრო ძირითად დინებაში, რომ ფონურ რეჟიმში გაშვებული დინებას დაესრულებინა მუშაობა. აუცილებელია, რომ ცხადად ან არაცხადად ვიზრუნოთ თვით დინების დასრულებაზე.

ცხადად ეს კეთდება ორნაირად: ან უნდა დაველოდოთ მის დასრულებას `join()` მეთოდის გამოყენებით, ან უნდა გამოვცალკევდეთ მისგან (detaching). ამ ბოლო შემთხვევაში, `std::thread` ობიექტი აღარაა დაკავშირებული აქტიურ აღმასრულებელ დინებასთან და ამიტომ არაა შემოერთებადი (joinable). ამიტომ შემდეგი კოდი დაგვიწერს, რომ `assert` -ის არგუმენტი არის მცდარი:

```
#include <iostream>
#include <thread>
#include <assert.h>
using namespace std;

void func() { cout << "Hello!" << endl; }

int main()
{
	std::thread t(func);
	t.detach();
	assert(t.joinable());
}
```

გარდა გზავნილისა, კიდევ რაღაც ჩანს ტექსტში. თუ კოდის ბოლო სტრიქონს გავაკომენტარებთ, დავინახავთ, რომ ვიდრე ფონურ რეჟიმში გასაშვები დინება განცალკევდებოდა, მანამდე მან ტექსტის ნაწილის დაბეჭდვა მოასწრო მხოლოდ (თუ თქვენმა კომპიუტერმა მოასწრო და მთლიანად დაბეჭდა მისალმება, მაშინ გაზარდეთ ტექსტის რაოდენობა ან ზომა).

როდესაც დინებას ვუშვებთ ფონურ რეჟიმში (და არ ველოდებით მის დამთავრებას), უნდა ვიზრუნოთ რომ ყველა მონაცემი, რომლებთანაც დინებას აქვს წვდომა, იყოს გამართულ მდგომარეობაში. აი მაგალითი, როდესაც საქმე ფუჭდება იმის გამო, რომ დინების ფუნქცია რეფერირებს ცვლადზე, რომლის არსებობა დამთავრდა დინების დამთავრებამდე:

```
#include <iostream>
#include <thread>
using namespace std;

struct func
{
	int& i;
```

```
        func(int& i_) :i(i_) {}
        void operator()()
        {
                for (auto j = 0; j< 500000; ++j)
                {
                        ++i;                                    // #1
                        cout << i << endl;
                }
        }
};
void Uh()
{
        int n = 0;
        func my_func(n);
        std::thread t(my_func);
        t.detach();                                             // #2
}                                                               // #3
int main()
{
        Uh();
        cout << "A little time for background thread" << endl;
}
```

ლისტინგი 1: ფუნქცია ბრუნდება იმ დროს, როდესაც დინებას კვლავ აქვს წვდომა ლოკალურ ცვლადზე

პროგრამის ყოფაქცევა განუსაზღვრელია. სხვადასხვა კომპიუტერზე მისი ყოფაქცევა განსხვავებულია. თუმცა, თუ რამდენჯერმე გავუშვებთ, დიდი ალბათობით იგი დაეკიდება, რადგან `Uh()` ფუნქციის დამთავრების შემდეგ `n` ცვლადი წყვეტს არსებობას, შესაბამისად, მისი ცვლილების გამო პრობლემები უნდა შეიქმნას, რადგან მეხსიერებაში არაა ადგილი ამ სახელით. დაკიდებების სიხშირე გაიზრდება, თუ მთავარ ფუნქციას ოდნავ გადავაკეთებთ:

```
int main()
{
        Uh();
        int n = 11;
        cout << "A little time for background thread" << endl;
}
```

ამ ამოცანაში, არსებობს სამი საფრთხე:

#1 წვდომა რეფერენსზე, რომელიც სავარაუდოდ მაწანწალა გახდება
#2 არ ველოდებით დინების დამთავრებას
#3 შესაძლოა ახალი დინება ისევ მუშაობდეს...

ამავე დროს, ვიცით რამდენი სიკეთე ახლავს რეფერენსების გამოყენებას და მასზე უარის თქმა დინებების შემთხვევაშიც ძნელია. პროგრამირებაში არსებობს გამოთქმა (idiom): Resource Acquisition Is Initialization (RAII). ამ მიდგომის მიხედვით, დინების გამოყენება რომ უსაფრთხო გახდეს, ამისთვის იქმნება კლასი, რომელიც იძახებს loin() მეთოდს მის დესტრუქტორში. კლასი, რომლიდანაც ამოღებული არის ასლის და მინიჭების კონსტრუქტორები, რეფერენსით მიიღებს დინებას. ლისტინგი 1-ის თემა შემდეგ განვითარებას იღებს:

```
#include <iostream>
#include <thread>
using namespace std;

class thread_guard
{
        std::thread& t;
public:
        explicit thread_guard(std::thread& t_) :
                t(t_)  {}
```

```
        ~thread_guard()
        {
                if (t.joinable())
                {
                        t.join();
                }
        }
        thread_guard(thread_guard const&) = delete;
        thread_guard& operator=(thread_guard const&) = delete;
};
struct func
{
        int& i;
        func(int& i_) :i(i_) {}
        void operator()()
        {
                for (auto j = 0; j< 50000; ++j)
                {
                        ++i;
                        cout << i << endl;
                }
        }
};
void Uh()
{
        int n = 0;
        func my_func(n);
        std::thread t(my_func);
        thread_guard g(t);
        cout << "What can I do?" << endl;
}
int main()
{
        freopen("data.out", "w", stdout);
        Uh();
}
```

ლისტინგი 2: RAII იდიომის გამოყენება დინების დასამთავრებლად

აღსანიშნავია Uh() ფუნქციის მეოთხე (დამატებული) სტრიქონი, რომელიც გაშვებულ დინებას დაარქმევს ახალ სახელს, რომელიც მას არ მისცემს დაშლის საშუალებას ვიდრე არ შემოიერთებს მას (თუ უკვე შემოერთებული არ დახვდა ამ დროს).

<<< არგუმენტების გადაწოდება დინების არგუმენტისთვის. ეს არის ვრცელი და ფაქიზი თემა, რაც ცალკე განხილვის თემაა. მისი სირთულის შესახებ წარმოდგენას შეგვიქმნის ორი ამოცანა, რომელსაც განვიხილავთ ქვემოთ.

<<< დაგროვების პარალელური ალგორითმი იტერატორების დიაპაზონისთვის. C++ ის სტანდარტული ბიბლიოთეკის ფუნქცია std::thread::hardware_concurrency() აბრუნებს დინებების იმ რაოდენობის მაჩვენებელს, რაც ნამდვილად შეიძლება ერთდროულად გაეშვას პროგრამის მოცემული შესრულების დროს. მრავალბირთვიან სისტემაში ეს შეიძლება იყოს CPU -ს ბირთვების რაოდენობა, მაგალითად. ეს მაინც მხოლოდ მინიშნებაა და ფუნქციამ შესაძლოა დააბრუნოს 0 თუ თუ მონაცემების აღება ვერ მოახერხა. მაინც, იგი სასარგებლოა ამოცანის მცირე ნაწილებად დასახლეჩად.

შემდეგი ლისტინგი (4-ე) წარმოგვიდგენს პარალელური std::accumulate-ის მარტივ ვერსიას. კურსის ბოლო ნაწილში ვნახავთ თუ როგორ შეიძლება სტანდარტული ფუნქციის უკვე

არსებული std::reduce ალგორითმი გამოვიყენოთ ამ მიზნით. საკუთარი იმპლემენტაცია გვიჩვენებს რამდენიმე ძირითად იდეას. განვიხილოთ კოდი:

```
template<typename Iterator, typename T>
struct accumulate_block
{
	void operator()(Iterator first, Iterator last, T& result)
	{
		result = std::accumulate(first, last, result);
	}
};
template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
	unsigned long const length = std::distance(first, last);
	if (!length)                                                    // #1
		return init;
	unsigned long const min_per_thread = 25;
	unsigned long const max_threads =
		(length + min_per_thread - 1) / min_per_thread;             // #2
	unsigned long const hardware_threads =
		std::thread::hardware_concurrency();
	unsigned long const num_threads =                               // #3
		std::min(hardware_threads != 0 ? hardware_threads : 2, max_threads);
	unsigned long const block_size = length / num_threads;          // #4
	std::vector<T> results(num_threads);
	std::vector<std::thread> threads(num_threads - 1);              // #5
	Iterator block_start = first;
	for (unsigned long i = 0; i<(num_threads - 1); ++i)
	{
		Iterator block_end = block_start;
		std::advance(block_end, block_size);                        // #6
		threads[i] = std::thread(                                   // #7
			accumulate_block<Iterator, T>(),
			block_start, block_end, std::ref(results[i]));
		block_start = block_end;                                    // #8
	}
	accumulate_block<Iterator, T>()(
		block_start, last, results[num_threads - 1]);               // #9
	std::for_each(threads.begin(), threads.end(),
		std::mem_fn(&std::thread::join));                           // #10
	 return std::accumulate(results.begin(), results.end(), init);  // #11
}
```

ლისტინგი 4: std::accumulate -ის სწორხაზოვანი, გულუბრყვილო ვერსია

თუ დიაპაზონი ცარიელია (#1), უბრალოდ ვაბრუნებთ საწყის მნიშვნელობას init. სხვა შემთხვევაში, დიაპაზონსი არის ერტი მაინც ელემენტი და გვიწევს დასამუშავებელი ელემენტების დაყოფა მინიმალური ზომის ბლოკებად და აქედან გამომდინარე, დინებების მაქსიმალური რიცხვის გაგება (#2). ეს იმიტომ, რომ არ გავუშვათ ბევრი დინება მაშინ როდესაც სულ ხუთი ელემენტია დიაპაზონში, რისთვისაც ერთი დინებაც საკმარისია. დინებების რიცხვი არის ამ რაოდენობისად და აპარატურული დინებების რიცხვის მინიმუმი წარმოადგენს გასაშვები დინებების რაოდენობას (#3). აპარატურული დინებების რაოდენობაზე მეტის გაშვება გამოიწვევს გადართვებს და შეანელებს წარმადობას. თუ გამოძახება std::thread::hardware_concurrency() დააბრუნებს 0-ს, მასინ უბრალოდ ავიღებთ რაიმე რიცხვს ჩვენი შეხედულებისამებრ. ამ შემთხვევაში აღებული გვაქვს 2. ძალიან ბევრი დინების გაშვება შეანელებს ერთ-ბირთვიანი კომპიუტერზე პროგრამის წარმადობას. #3 განსაზღვრავს მონაცემების რაოდენობას ერთი დინებისთვის.

ამ მომენტში ვიცით რამდენი დინება გვჭირდება, ე.ი. შეგვიძლია შევქმნათ ორი ვექტორი: საშუალედო შედეგებისთვის `std::vector<T>`, ხოლო დინებებისთვის `std::vector<std::thread>` (#5). დინება ერთით ნაკლებია, რადგან ამ დროს უკვე მუშაობს ერთი.

დინებების გაშვება ხდება განმეორების შეტყობინებაში: `block_end` იტერატორს მივასწრაფებთ მიმდინარე ბლოკის დასასრულისკენ (#6) და ვუშვებთ ახალ დინებას ამ ბლოკის შედეგების დასაგროვებლად (#7). შემდეგი ბლოკის დასაწყისი არის ამ ბლოკის დასასრული (#8).

მას მერე რაც ყველა დინება გაეშვა, ძირითადი დინება დაამუშავებს ბოლო ბლოკს (#9). ამის შემდეგ `std::for_each` -ის საშუალებით ველოდებით ყველა გაშვებული დინების შემოერთებას (#10). `std::accumulate` –ის ბოლო გამოძახება შეაჯამებს შედეგებს (#11).

ყურადღება მივაქციოთ რამდენიმე გარემოებას. ისეთ `T` ტიპებზე, რომლებზე შეკრების ოპერაცია არ არის ასოციაციური (ნამდვილი რიცხვების ტიპები), `parallel_accumulate` -ის შედეგები შესაძლოა განსხვავდებოდეს `std::accumulate`-ის შედეგებისგან, რადგან შესაკრებები სხვადასხვანაირად დაჯგუფდება.

გამკაცრებულია მოთხოვნები იტერატორებზე. `std::accumulate`-ისთვის საკმარისია ერთჯერადი გავლის *input* იტერატორები, ხოლო მის პარალელურ ვერსიას სჭირდება მრავალჯერადი გავლის იტერატორები, *forward* მაინც.

`T` -ს აგება უნდა იყოს შესაძლებელი ნაგულისხმევი კონსტრუქტორით. სხვანაირად შედეგების ვექორს ვერ შევქმნით.

ასეთი ცვლილებები ზოგადია პარალელური ალგორითმებისთვის.

<<< დაგროვების პარალელური ალგორითმის განხილვა. უპირველეს ყოვლისა, კოდი გვიჩვენებს რომ შესაძლოა მისი აჩქარება სწარაფი წვდომის იტერატორების შემთხვევაში. მაგალითად, სტრიქონი

```
    std::advance(block_end, block_size);                    // #6
```

შეიცვლება პოინტერების არითმეტიკის გამოყენებით.

ჩვენ ვნახეთ, თუ როგორ შეგვიძლია პროგრამის მსვლელობის დროის გაზომვა. სავარჯიშოს სახით, სასარგებლო იქნება ამ ალგორითმის, აგრთვე მისი სწრაფი წვდომის კონტეინერებზე მორგებული ვერსიების სისწრაფის შედარება შემთხვევითი მონაცემებით შევსებულ დიდ ვექტორებზე. აგრეთვე, პარალელური ალგორითმების შემუშავება count, count_if -ებისთვის, გამოყენებული ტექნიკის საფუძველზე.

თუ მსგავის ალგორითმის გამოყენება გვინდა სიისთვის, არის ვარიანტი სია გავყოთ ორად და ისე დავამუშავოთ. როგორც საწყისი მიახლოება, შეგვიძლია ვისარგებლოთ შემდეგი კოდით:

```
#include <iostream>
#include <random>
#include <list>
#include <future>
using namespace std;

template<class InputIt>
typename std::iterator_traits<InputIt >::value_type my_sum_direct
(
       InputIt  first,
       int n
)
{
       typename std::iterator_traits<InputIt >::value_type  sum{};

       for (int i = 0; i<n; ++i)
       {
```

```
		sum += *first;
		++first;
	}
	return sum;
}

template<class BidirIt>
typename std::iterator_traits<BidirIt >::value_type my_sum_inverse
(
	BidirIt  first,
	int n
)
{
	typename std::iterator_traits<BidirIt >::value_type  sum{};

	for (int i = 0; i<n; ++i)
	{
		sum += *first;
		--first;
	}
	return sum;
}
const int K = 10000;

int main()
{
	default_random_engine dre;
	uniform_int_distribution<int> di(0, 100);
	list<int> lst;
	for (int i = 0; i < K; ++i)
		//lst.push_front(di(dre));
		lst.push_front(1);                  //ეს მარტივია გასასინჯად

	int n = K / 2;
	auto result1= std::async(std::launch::async, my_sum_direct<list<int>::iterator>,
			lst.begin(), n);

	auto it = lst.end();
	--it;
	int result2 = my_sum_inverse(it, K - n);

	int result = result1.get() + result2;
	cout << result << endl;
}
```