წრფივი პროგრამირების და მატრიცული თამაშების ამოხსნის მეთოდები, დაფუძნებული უპირობო მინიმიზირების ალგორითმების და დაპროგრამების თანამედროვე ტექნოლოგიების გამოყენებაზე

დოქტორანტის I კოლოკვიუმი

დოქტორანტი

ლუკა გორგაძე

ხელმძღვანელი

სრული პროფესორი

კობა გელაშვილი

შესავალი

წრფივი პროგრამირების ამოცანა (შემდეგში LP) არის ერთ-ერთი ყველაზე გამორჩეული მათემატიკური ამოცანა როგორც თეორიული, ასევე პრაქტიკული თვალსაზრისით. „წრფივი პროგრამირება გამოიყენება რესურსების განაწილების, წარმოების დაგეგმვის, საინვესტიციო პორტფელების შედგენის, სამხედრო სტრატეგიების განსაზღვრის მიზნით და ა.შ.. ითვლება, რომ ეკონომიკაში მისი გამოყენების შემდეგ დანახარჯები შემცირდა დაახლოებით 20%-ით, რამაც მათემატიკური მეთოდების გამოყენებაში დაარწმუნა ისეთი მეწარმეებიც და მენეჯერებიც, რომლებიც მიჩვეული იყვნენ მხოლოდ გამოცდილებასა და ინტუიციაზე დაყრდნობას„ (იხ. Jiří Matoušek, Bernd Gärtner. Understanding and Using Linear Programming, Springer-Verlag Berlin Heidelberg 2007).

თავისი არსებობის 80 წლის განმავლობაში, LP–ის გამოყენებათა სფერო გაფართოვდა და ახლა ის ბევრად მეტია ვიდრე მათემატიკური ეკონომიკა. თეორიულ კომპიუტერულ მეცნიერებაში LP წარმოადგენს ალგორითმების აგების ერთ-ერთ ძირითად საშუალებას. გარდა იმისა, რომ მრავალი ამოცანა მთელრიცხვა პროგრამირებიდან, გრაფთა თეორიიდან და ა.შ. უშუალოდ წრფივი პროგრამირების მეთოდებზე დაყრდნობით იხსნება, მთელი რიგი გამოთვლითი ამოცანებისთვის ეფექტური (პოლინომიალურ დროში მომუშავე) ალგორითმის არსებობა მტკიცდება LP–ის საშუალებებით.

ამჟამად ბევრად დიდი მოცულობის LP–ის ამოცანების ამოხსნა შეიძლება, ვიდრე რამდენიმე ათეული წლის წინ. ზოგადად, ითვლება რომ ამას აქვს ორი მიზეზი: კომპიუტერების სიმძლავრის ზრდა და LP–ის ალგორითმების თეორიის განვითარება და რომ ალგორითმებთან დაკავშირებული (მათ შორის პარალელურ ალგორითმებთან) თეორიული პროგრესი უფრო მნიშვნელოვანია. თუმცა, ამჟამად ასევე გააზრებულია დაპროგრამების გარემოს, განსაკუთრებით პარალელური დაპროგრამების გარემოს არსებითი გავლენა. თანამედროვე დაპროგრამების ენების, განსაკუთრებით C++-ის, სტანდარტულ შესაძლებლობებს შორის არის მრავალდინებიანი პროგრამის შექმნის არაერთი ეფექტური მეთოდი. დაპროგრამების ენების გაფართოებები საშუალებას გვაძლევს ვიმუშაოთ ვიდეო ადაპტერების მიერ შექმნილ მრავლობით დინებებთან და მივაღწიოთ მაღალ წარმადობას არაფრით გამორჩეული პარამეტრების მქონე კომპიუტერებზეც. ამიტომ სრულიად ბუნებრივია, რომ LP–ის კომერციული სოლვერები, მაგალითად GUROBI, მათემატიკური ფუნდამენტის (სიმპლექს ალგორითმი, ბარიერული ფუნქციები) იმპლემენტაციაში იყენებს დაპროგრამების გარემოს თანამედროვე მეთოდებს.

თავისი აქტუალობის გამო, LP-ის მიმართულებით ყოველთვის დიდი იყო პუბლიკაციების რაოდენობა. ამჟამად, მისი ამოხსნისთვის გამოიყენება ორი განსხვავებული მიდგომა: კომბინატორული სიმპლექს მეთოდი და ბარიერული ფუნქციების გამოყენებაზე დაფუძნებული მეთოდები. GUROBI, მაგალითად, იყენებს ორივეს. ბარიერული ფუნქციების გამოყენებით ხდება ამოცანის ზომების შემცირება, შემდეგ იხსნება სიმპლექს ალგორითმით. გასათვალისწინებელია, რომ ეს არაა სტანდარტული სიმპლექს-ალგორითმი, არამედ იგი გაჯერებულია მრავალი ევრისტიკით, რაც იცავს პროგრამის მსვლელობას მრავალი შემჩნეული გამონაკლისი (ავარიული) შემთხვევისგან.

საინტერესოა, რომ LP-ის ამოცანების ამოხსნის პროცესში ძირითად დაბრკოლებას ქმნის უმარტივესი შეზღუდვები ცვლადების ნიშანზე.

ამავე დროს, ბოლო ათეულ წლებში გამოჩნდა რამდენიმე პუბლიკაცია, მიძღვნილი კვადრატული პროგრამირების ამოცანებისადმი, სადაც ცვლადების ნიშანზე შეზღუდვები იხსნება ცვლადის გარდაქმნის საშუალებით. მაგალითად, თუ განვიხილავთ LP-ის სტანდარტულ ამოცანას:

$$\begin{cases} cx = c_1x_1 + \cdots + c_nx_n \to \min, \\ A_1x \le 0, \ldots, A_mx \le 0, \\ x_1 \ge 0, \ldots, x_n \ge 0, \end{cases} \quad (1)$$

სადაც A_j მატრიცის j-ურ სტრიქონს აღნიშნავს, მაშინ ახალ $x_i = t_i^2$ ცვლადებზე გადასვლის შემდეგ (1) ამოცანის ნაცვლად შეგვიძლია განვიხილოთ შემდეგი მინიმიზირების ამოცანის:

$$\left(c_1t_1^2 + \cdots + c_nt_n^2\right) + \sum_{i=1}^{m}\left(a_{i1}t_1^2 + \cdots + a_{in}t_n^2 + s_i^2\right)^2 \to \min, \quad (2)$$

რაც აღარ არის ამოზნექილი ამოცანა, მაგრამ სამაგიეროდ არის უმარტივესი სახის უპირობო მინიმიზირების ამოცანა. აქ s_i^2 არის ე.წ. „ღრიჭოს (slack)“ ცვლადები, რისი საშუალებითაც უტოლობის ტიპის შეზღუდვები გარდაიქმნება ტოლობის ტიპის შეზღუდვებად: $A_ix + s_i^2 = 0$.

უნდა შევნიშნოთ, რომ (2) არის საჯარიმო ფუნქცია, რაც დამატებით სირთულეებს ქმნის რადგან (1) და (2) არაა ტოლძალოვანი (ეკვივალენტური) ამოცანები. ეს სირთულე ადვილად გვარდება, თუ ჩვენ ვისარგებლებთ წრფივი პროგრამირების ამოცანის სტანდარტული სახის ეკვივალენტური წარმოდგენით, რასაც ჰქვია სიმეტრიული მატრიცული თამაშები. თუმცა ამჟამად წვრილმანებზე არ შევჩერდებით.

სადოქტორო დისერტაციის პერსპექტივაში, ჩვენი ამოცანა არის (2) ტიპის ამოცანის ამოხსნის ეფექტური რიცხვითი რეალიზება, რაც კონკურენტუნარიანი იქნება ამჟამად მოქმედ საუკეთესო LP-ის ამომხსნელებთან.

ამ მიმართულებით, ვაწარმოებთ შემდეგი სახის კვლევებს:

- უპირობო მინიმიზირების საუკეთესო რიცხვითი ალგორითმების გარჩევა, ზოგიერთი მათგანის გაუმჯობესებაში საკუთარი წვლილის შეტანა;
- უპირობო მინიმიზირების რიცხვითი ალგორითმების ძირითადი დამხმარე პროცედურის - ერთგანზომილებიანი მინიმიზირების (ე.წ. line search) ალგორითმის ადაპტირება მეოთხე რიგის პოლინომის მინიმიზირების ამოცანაზე;
- კერძოდ მეოთხე რიგის პოლინომის მინიმიზირების ეფექტური ალგორითმის იმპლემენტირება, რომელიც ამ ტიპის პოლინომებზე ბევრად უკეთ იმუშავებს, ვიდრე საუკეთესო ზოგადი უპირობო მინიმიზირების ალგორითმები. შევნიშნოთ, რომ 1გ მინიმიზირების მოდიფიცირების გამო, ეს იქნება გლობალური მინიმიზირების ალგორითმი IV რიგის პროლინომისთვის, რასაც ექნება LP-სგან დამოუკიდებელი სამეცნიერო ინტერესი;
- კონკურენტული პროგრამირების მეთოდების გამოყენებით, ეფექტური პარალელური ალგორითმის შემუშავება.

ამ პუნქტების განხორციელების საკითხზე გავამახვილებთ ყურადღებას წარმოდგენილ კოლოკვიუმში.

უპირობო მინიმიზირების ალგორითმების ათვისება და გაუმჯობესება

ჩვენს ვიცით ის საკვანძო მომენტები, რომლებიც მუშავდებოდა 10-წლეულების განმავლობაში და შედეგად მივიღეთ ისეთი ეფექტური ალგორითმები, როგორებიცაა lcg (მცირე მეხსიერებიანი შეუღლებული გრადიენტების მეთოდი) და lbfgs ((მცირე მეხსიერებიანი bfgs). ჩვენი აზრით, აქ გადამწყვეტი მნიშვნელობა აქვს 1გ მინიმიზირების (line search) ალგორითმს, რომელიც შემოთავაზებულია [1]-ში. ექსპერიმენტის სახით, ეს პროცედურა დავამატეთ კარგად ცნობილ (მაგრამ ნაკლებად შესწავლილ) პოლიაკის მძიმე ბირთვის მეთოდს და შედეგად მივიღეთ სრულიად კონკურენტუნარიანი ალგორითმი, რაც დადასტურდა საკმაოდ საფუძვლიანი ტესტირებით (იხ. [2]).

ასევე დიდი გავლენა აქვს ალგორითმების სიწრაფეზე პრეკონდიცირების გამოყენებას. თუმცა, პრეკონდიციონერის შერჩევა ძლიან ფაქიზი საკითხია და მის შერჩევას ვგეგმავთ IV რიგის პოლინომისთვის კერძო მინიმიზირების ალგორითმის დამუშვების შემდეგ, რასაც დიდი ალბათობით განვახორციელებთ lbfgs-ის საფუძველზე.

1გ მინიმიზირების ალგორითმის ადაპტირება IV რიგის პოლინომის მინიმიზირებაზე

თეორიულად, ეს სრულიად გამჭვირვალე და მარტივი საკითხია. თუ დავაფიქსირებთ პოლინომის კლების მიმართულებას (ანტიგრადიენტს, ან შეუღლებულ მიმართულებას), მაშინ მიღებულ 1გ წრფეზე საწყისი პოლინომის შეზღუდვა არის ისევ მეოთხე რიგის პოლინომი, ოღონდ უკვე მხოლოდ ერთი ცვლადის. შესაბამისად, მისი გლობალური მინიმალის მოსაძებნად შეგვიძლია გამოვიყენოთ როგორც ზუსტი მიდგომა (ამ პოლინომის წარმოებულის ნულთან ტოლობა მესამე რიგის განტოლების ამოხსნას ნიშნავს, რაც ფრომულებით გადაჭრადია), ასევე მიახლოებითი. თუმცა, ჩნდება შემდეგი სახის სირთულე, რომლის გადაჭრის საუკეთესო გზის შერჩევაზეც ვმუშობთ ამჟამად. თუ ცვლადების რაოდენობა საკმაოდ დიდია და მიმდინარე მიახლოება საკმაოდ შორსაა ამონახსნისგან, მაშინ ერთი ცვლადის პოლინომის კოეფიციენტები ძალიან დიდია (10-ის მეოცე ხარისხები და უფრო მეტი, მაგალითად), რაც C++ ენის ორმაგი სიზუსტის პირობებში ვერ უზრუნველყოფს კავშირს ერთგანზომილებიანი პოლინომის ფესვებსა და საწყისი პოლინომის 1გ მინიმალს შორის. საკითხი საკმაოდ ფაქიზია, ამიტომ წარმოდგენილი ნაშრომის მოცულობის გამო დეტალებში ვერ შევალთ. თუმცა შეგვიძლია განვიხილოთ პრობლემის გადაჭრის რამდენიმე გზა, რომლებიც ამჟამად ცნობილია ჩვენთვის.

შეგვიძლია მაღალი სიზუსტის მათემატიკის გამოყენება. C++ ენაში ამის საშუალებას გვაძლევს გარე ბიბლიოთეკა boost. მას აქვს მაღალი სიზუსტის რიცხვებისთვის სიმრავლეები:

```
cpp_bin_float_50;
cpp_bin_float_50;
cpp_dec_float_50;
cpp_dec_float_100;
```

ყოველი მათგანის გამოყენება ხსნის დამრგვალების ცდომილებებსი მიერ შექმნილ პრობლემას. თუმცა, მათგან ყველაზე სწრაფის გამოყენებაც (ყველაზე ნაკლები სიზუსტისმქონის) თითქმის 100-ჯერ ნელია ორმაგი სიზუსტის რიცხვების გამოყენებასთან შედარებით იმ შემთხვევაში, როდესაც ორმაგი სიზუსტეც მუშაობს.

თუმცა, ამ მიზნით შესაძლებელია სხვა პროგრამირების ენების გამოყენებაც, მაგალითად Go ან Java. ეს პერსპექტიული მიმართულებაა და ამჟამად კვლევის პროცესშია.

კონკურენტული პროგრამირება C++-ში

იმისათვის რომ პროგრამამ მაქსიმალურად გამოიყენოს მრავალ ბირთვიანი პროცესორის რესურსები, ის უნდა უნაწილებდეს შესასრულებელ სამუშაოს სხვადასხვა დინებებს (threads). შედეგად, ოპერაციულ სისტემას შეეძლება ეს დინებები პარალელურად, პროცესორის სხვადასხვა ბირთვებზე ამუშაოს, რაც პროგრამას საგრძნობლად აჩქარებს (აჩქარება დამოკიდებულია სხვადასხვა ფაქტორებზე, მაგალითად, თუ რამდენ ბირთვიანია პროცესორი, რამდენად თანაბრად არის სამუშაო დანაწილებული და რამდენად დამოუკიდებლები არიან თითოეული დინებისთვის მიცემული დავალებები ერთმანეთისგან).

დინებების შექმნა C++-ში საკმაოდ მარტივია, ამისათვის უნდა შევქმნათ std::thread ტიპის ბიექტი, კონსტრუქტორში პირველ პარამეტრად გადავაწოდოთ ფუნქციაზე მიმთითებელი (ამ ფუნქციის ტანში ვწერთ დინების მიერ შესასრულებელ ბრძანებებს) და მომდევნო პარამეტრებად თვითონ ამ ფუნქციისთვის გადასაწოდებელი პარამეტრები.

განვიხილოთ მარტივი მაგალითი. დავწეროთ პროგრამა რომელიც შექმნის მასივს და დაბეჭდავს მისი ელემენტების ჯამს. ელემენტების ნახევარს პროგრამის მთავარი დინება დააჯამებს, ხოლო მეორე ნახევარს მის მიერ შექმნილი ახალი დინება.

```
void sumArray(int a[], int startIndex, int endIndex, int *sum) {
	*sum = 0;
	for (int i = startIndex; i < endIndex; i++) {
		*sum += a[i];
	}
}

int main()
{
	int a[] = { 1, 2, 3, 4, 5, 6 };
	int aLength = 6, middleIndex = aLength / 2;
	int firstHalfSum = 0, secondHalfSum = 0;
	std::thread secondHalfSumer(sumArray, a, middleIndex, aLength, &secondHalfSum);
	for (int i = 0; i < middleIndex; i++) {
		firstHalfSum += a[i];
	}
	secondHalfSumer.join();
	std::cout << "Sum: " << firstHalfSum + secondHalfSum << std::endl;
}
```

პროგრამირების ხერხს როდესაც პროგრამისტი უშუალოდ ქმნის დინებებს, დინებებზე დაფუძნებული პარალელიზმი (thread based parallelism) ეწოდება. ეს მიდგომა არის მაქსიმალურად ქვედა დონის (low level), რადგან დინებები თვითონ ოპერაციული სისტემების აბსტრაქციები არიან. ისევე როგორც ბევრი სხვა ქვედა დონის მოდელი, ამ მიდგომით პროგრამის შექმნაც არის შრომატევადი და კომპლექსური, რაც თავისმხრივ შეცდომების დაშვების შანსსაც ზრდის. მაგალითად, ასეთი მიდგომით პროგრამირებისას პროგრამისტს უწევს განსაზღვროს რამდენი დინება შექმნას და როგორ დაუნაწილოს მათ გასაკეთებელი სამუშაო. ეს კი საფრთხილო საკითხია, რადგან დინებების შექმნა საკმაოდ დიდ რესურს მოითხოვს. შესაბამისად, არასწორად გათვლის შემთხვევაში, ასეთი კონკურენტული

პროგრამა თანმიმდევრულზე (sequential) უფრო ნელი იქნება და მეტიც, ოპერატიულ სისტემაში გამოყენებადი დინებების ლიმიტის გადაჭარბების შემთხვევაში exception-ით დაამთავრებს მუშაობას. დინებათა ოპტიმალური რაოდენობის განსაზღვრა კი საკმაოდ რთულია, რადგან ის დამოკიდებულია პროცესორზე და დინებებზე გასაშვებ სამუშაოებზე.

ზემოთხსენებული სირთულეების თავიდან ასაცილებლად არსებობს მეორე, მაღალი დონის (high level), მიდგომა - დავალებებზე დაფუძნებული პარალელიზმი (task based parallelism). ამ მიდგომის გამოყენებისას პროგრამისტი უბრალოდ ანაწილებს გასაკეთებელ სამუშაოს დავალებებად (tasks) და აწვდის ბიბლიოთეკის მეთოდს. ბიბლიოთეკა კი თავად უზრუნველყოფს ოპტიმალური რაოდენობის დინებების შექმნას და მათთვის დავალებების დარიგებას.

ამ მიდგომის ფარგლებში C++11 სპეციფიკაცია განსაზღვრავს მეთოდ std::async-ს, რომელსაც იგივე პარამეტრები გადაეცემა რაც thread-ის კონსტრუქტორს. მათ შორის განსხვავება არის ის რომ async-ის შემთხვევაში ბიბლიოთეკა თავად ირჩევს რომელ დინებას (უკვე შექმნილ ძველს, ახალს თუ მიმდინარეს) შეასრულებინოს გადაწოდებული ფუნქცია. გარდა ამისა, async ფუნქცია მოსახერხებელს ხდის ფუნქციიდან ინფორმაციის დაბრუნებას და exception-ებთან მუშაობას. მაგალითად, ჩვენს მიერ მოყვანილ პროგრამაში რომელიც მასივის ჯამს პოულობდა, thread-ის პირდაპირ შექმნის შემთხვევაში არ შეგვეძლო გადაწოდებული ფუნქციის დასაბრუნებელი მნიშვნელობის გამოყენება, ამიტომაც ფუნქცია ჯამს ერთ-ერთი შემავალი პარამეტრის მეშვეობით აბრუნებდა. async-ის გამოყენებისას კი შეგვიძლია ჯამი პირდაპირ ფუნქციის დასაბრუნებელი მნიშვნელობის სახით დავაბრუნოთ, რაც უფრო ბუნებრივია. async მეთოდით იგივე ამოცანა შეგვიძლია შემდეგნაირად გადავწყვიტოთ.

```
int sumArray(int a[], int startIndex, int endIndex) {
	int sum = 0;
	for (int i = startIndex; i < endIndex; i++) {
		sum += a[i];
	}
	return sum;
}

int main()
{
	int a[] = { 1, 2, 3, 4, 5, 6 };
	int aLength = 6, middleIndex = aLength / 2;
	int firstHalfSum = 0;
	std::future<int> secondHalfSum = std::async(sumArray, a, middleIndex, aLength);
	for (int i = 0; i < middleIndex; i++) {
		firstHalfSum += a[i];
	}
	std::cout << "Sum: " << firstHalfSum + secondHalfSum.get() << std::endl;
}
```

სამწუხაროდ დავალებებზე დაფუძნებული პარალელიზმი async-ის საშუალებით საკმაოდ არასტაბილურია, რადგან C++11-ის სპეციფიკაცია არ განსაზღვრავს როგორ და რომელ დინებებს უნდა დაურიგდეთ დავალებები. შედეგად, მაგალითად, როდესაც ვთხოვთ async-ს დავალების კონკურენტულად გაშვებას, Visual C++ 2017 იმპლემენტაცია ამ დავალებას გაზიარებული დინებებიდან (thread pool) ერთ-ერთ დინებას დაავალებს, GCC 4.8 კი ყოველი დავალებისთვის ახალ დინებას შექმნის. ეს კი იმას გამოიწვევს რომ, ერთი და იგივე პროგრამამ შეიძლება პირველი იმპლემენტაციის გამოყენების შემთხვევაში ოპტიმალურად იმუშაოს,

მეორის შემთხვევაში კი რესურსების ამოწურვის გამო exception-ით დაასრულოს მუშაობა. სამწუხაროდ პრობლემა უფრო სიღრმისეულია ვიდრე უბრალოდ რომელიმე იმპლემენტაციის არასრულყოფილება. საქმე იმაშია რომ C++11 სპეციფიკაცია საოცრად ართულებს ნამდვილი, დავალებაზე დაფუძნებული პარალელიზმის იმპლემენტაციას. ნამდვილი დავალება (task) არ უნდა იყოს მჭიდროდ გადაჯაჭვული დინებასთან, მარტივი უნდა იყოს დინების რამდენიმე დავალებისთვის გამოყენება, დავალების ერთიდან მეორე დინებაზე გადატანა, დავალების დაბლოკვისას დინების სხვა დავალებით დასაქმება. პრობლემა იმაშია რომ, C++11 სპეციფიკაციით განსაზღვრული thread_local ცვლადები და mutex-ები, რომლებიც ერთი და იგივე დინებიდან უნდა ჩაიკეტონ და გაიხსნან, ძალიან ართულებენ ნამდვილ დავალებაზე დაფუძნებულ პარალელიზმს. იმის განხილვა თუ რატომ ართულებენ ეს ენის ელემენტები ნამდვილი დავალებების არსებობას მიმდინარე ნაშრომის მიღმაა. იმედია C++-ის ახალ სპეციფიკაციაში ვიხილავთ ამ პრობლემის გადაწყვეტას. (იხ. [4], [5] და [6])

მიმდინარე C++-ის ვერსიაში ნამდვილი დავალებებზე ორიენტირებული პარალელიზმისთვის პროგრამისტმა თავად უნდა შექმნას ან სხვისგან მოიძიოს გაზიარებული დინებების (thread pool) კლასი. ასეთ შემთხვევაში, პროგრამა გახდება მეტად პორტაბელური, აღარ იქნება დამოკიდებული STL-ის იმპლემენტაციაზე. ასევე, იმ შემთხვევაში თუ პროგრამისტმა იცის რომ მისი დავალებები არ იყენებენ thread_local ცვლადებს ან mutex-ებს, მას შეუძლია ამ ფაქტის გათვალისწინებით დამატებითი იპტიმიზაცია გაუკეთოს თავის გაზიარებული დინებების იმპლემენტაციას. ამ ოპტიმიზაციის ხარჯზე, პროგრამა რომელიც async ფუნქციას იყენებს იქნება უფრო ნელი ვიდრე ის პროგრამა რომელიც პროგრამისტის მიერ შექმნილ გაზიარებული დინებების კლასს იყენებს.

ტესტ ფუნქციების კოლექცია კონკურენტული იმპლემენტაციებით

მათემატიკური ტესტ ფუნქციების კოლექციის არსებობა აუცილებელია სხვადასხვა ოპტიმიზაციის ალგორითმების დებაგირებისთვის და მათი ეფექტურობის შეფასებისთვის. თვითონ ტესტ ფუნქციების მუშაობის დროის შემცირება კი ამ პროცესს დააჩქარებს. ამისათვის შევქმენით ტესტ ფუნქციების კოლექცია კონკურენტული იმპლემენტაციებით, რომელშიც ამჟამად 18 ფუნქცია შედის. როგორც ვნახავთ, კონკურენტული იმპლემენტაციები საგრძნობლად უფრო სწრაფად მუშაობენ ვიდრე იგივე ფუნქციების სერიული ეკვივალენტები. თვრამეტივე ფუქნციას აქვს ცვალებადი რაოდენობის პარამეტრები, შესაბამისად მათი გამოთვლის წესი დამოკიდებულია ფუნქციის პარამეტრების რაოდენობაზე. შედეგად, კოლექცია სინამდვილეში შედგება არა 18-ი ფუნქციისგან არამედ 18 მსგავს ფუნქციათა ოჯახებისგან. იმისათვის რომ ჩვენს იმპლემენტაციას ცვალებადი რაოდენობის პარამეტრების მხარდაჭერა ქონოდა, თითოეულ ფუნქციის მნიშვნელობის შესაბამის პროგრამულ მეთოდს პარამეტრები გადაეცემა x მასივის სახით. მაგალითად, თუ 20 პარამეტრის გადაცემა გვინდა, მეთოდს გადავაწოდებთ 20 ელემენტიან x მასივს. ფუნქციის მნიშვნელობას მეთოდი მისი დასაბრუნებელი მნიშვნელობის საშუალებით აბრუნებს, ხოლო თითოეული ცვლადის მიმართ გრადიენტის მნიშვნელობას პარამეტრად გადაცემულ g მასივში წერს. ცხადია მეთოდის გამომძახებელმა უნდა უზრუნველყოს რომ x და g მასივები ერთი ზომის იყვნენ.

თვრამეტივე ფუნქცია ჯამის სახით არის მოცემული. ისე რომ შემავლი პარამეტრების უმრავლესობა (ზოგიერთ შემთხვევაში ყველა პარამეტრი) ფუნქციის მნიშვნელობის და გრადიენტების გამოთვლაში ერთნაირად მონაწილეობენ, მაგალითზე ნათელი გახდება თუ რა იგულისხმება ამაში ზუსტად. ამ საერთო დამახასიათებელი ნიშნის გამო შესაძლებელი გახდა

რამდენიმე განზოგადებული მეთოდის შექმნა რომლებსაც გადაეცემათ გამოსათვლელი მათემატიკური ფუნქციის გამომთვლელ მეთოდზე მიმთითებელი, ისინი კი თავად ახდენენ ამ ფუნქციის გამოთვლისთვის შესასრულებელი სამუშაოს სხვადასხვა დინებებისთვის დანაწილებას. 18 ფუნქციისთვის სულ სამი ასეთი განზოგადებული ფუნქცია გახდა საჭირო, მათმა გამოყენებამ მნიშვნელოვნად შეამცირა ფუნქციების იმპლემენტაციისთვის საჭირო კოდის რაოდენობა.

მაგალითის საშუალებით უფრო მარტივია იმის დანახვა თუ როგორ მუშაობენ ეს ზოგადი ფუნქციები. განვიხილოთ ტესტ კოლექციებიდან ყველაზე მარტივი მათემატიკური ფუნქცია POWER, რომელსაც აქვს შემდეგი სახე.

$$f\left(x_0, \ldots, x_{n-1}\right) = \sum_{i=0}^{n-1} \left((i+1)\, x_i\right)^2$$
$$x_0 = \ldots = x_{n-1} = 1$$

როგორც ვხედავთ, რადგან ის ჯამის სახით არის მოცემული რომლის შემავალი ელემენტები მხოლოდ ერთ ცვლადზე არიან დამოკიდებულნი, საკმაოდ მარტივია მისი გამოთვლისთვის საჭირო სამუშაოს დინებებისთვის დანაწილება. მაგალითად, 4 ბირთვიანი პროცესორის შემთხვევაში, შეგვიძლია პირველი მეოთხედი შემავალი პარამეტრების შესაბამისი ჯამის ნაწილი ერთ დინებას გამოვათვლევინოთ, მეორე - მეორე დინებას და ასე შემდეგ. იგივე პრინციპი ვრცელდება გრადიენტების გამოთვლაზეც.

იმისათვის რომ ზოგადმა ფუნქციამ აღნიშნული სტრატეგიით შეძლოს სამუშაოს დანაწილება, მას სჭირდება მოქნილი მეთოდი რომელიც ითვლის POWER ფუნქციის მნიშვნელობას და გრადიენტებს. მოქნილი იმ გაგებით რომ ასეთი მეთოდი, იმის მიხედვით თუ რა პარამეტრებით გამოვიძახებთ, დაითვლის ფუნქციის მნიშვნელობის და გრადიენტების მხოლოდ იმ ნაწილს რომელშიც პარამეტრებით განსაზღვრული ცვლადები მონაწილეობენ. C++-ში POWER მათემატიკური ფუნქციის გამომთვლელი მოქნილი მეთოდი შემდეგნაირად გამოიყურება.

```
double POWERRanged(double *x, double *g, int begin, int end) {
	double item;
	double fx(0.0);

	for (int i = begin; i < end; i++)
	{
		item = (i + 1)*x[i];
		fx += item * item;
		g[i] = 2.0*item*(i + 1);
	}
	return fx;
}
```

როგორც ვხედავთ, ეს მეთოდი, მაგალითად, 0-ს და 8-ს (begin, end) გადაწოდების შემთხვევაში ფუნქციის მნიშვნელობის და გრადიენტების მხოლოდ იმ ნაწილს გამოითვლის რომელშიც პირველი 8 პარამეტრი მონაწილეობს.

ახლა კი ვნახოთ, როგორ ითვლის ზოგადი მეთოდი, კონკურენტულად, ამ მეთოდის გამოყენებით, POWER ფუნქციის მნიშვნელობას. აღნიშნული მეთოდი იქნებს initializer მეთოდის მიერ გამოთვლილ გლობალურ ცვლადს block_size-ს. რომელშიც წერია თუ რა

რაოდენობის ცვლადებისთვის უნდა გამოვათვლევინოთ თითოეულ დინებას ფუნქციის მნიშვნელობა. მაგალითად, თუ POWER ფუნქციას 32 პარამეტრიანი ვერსიის მნიშვნელობის გამოთვლა გვინდა და პროცესორს რომელზეც პროგრამა ეშვება აქვს 4 ბირთვი, მაშინ თითოეულმა დინებამ მნიშვნელობები 8 პარამეტრის მონაწილეობით უნდა გამოითვალოს. ასეთ შემთხვევაში, ზოგადი ფუნქცია POWERanged-ს 0 და 8 (begin, end) პარამეტრებისთვის დაათვლევინებს ერთ დინებას, 8 და 16 პარამეტრებისთვის - მეორე დინებას, 16 და 24 პარამეტრებისთვის მესამე დინებას, 24 და 32 პარამეტრებისთვის მიმდინარე დინებას. ბოლოს კი თითოეული დინების მიერ დაბრუნებულ მნიშვნელობებს შეკრებს და მათ ჯამს დააბრუნებს.

```
double valgradPool
(
	double(*functionToCompute)(double *x, double *g, INT begin, INT end),
	double *x,
	double *g,
	INT n
)
{
	vector<future<double>> futures(num_threads - 1);
	INT block_start = 0, i;
	for (i = 0; i < (num_threads - 1); ++i)
	{
		INT block_end = block_start + block_size;
		futures[i] = threadPool->enqueue(functionToCompute, x, g, block_start,
block_end);
		block_start = block_end;
	}
	double sum = functionToCompute(x, g, block_start, n);
	for (i = 0; i < futures.size(); i++) {
		sum += futures[i].get();
	}
	return sum;
}
```

იმისდა მიუხედავად რომ თვრამეტივე ფუნქციის გამოსათვლელად გამოყენებულ იქნა ზოგადი ფუნქციები, race condition-ების გამო ყველა შემთხვევაში ისეთი ტრივიალური არ იყო მათი გამოყენება როგორც POWER ფუნქციის შემთხვევაში. მაგალითად, მაგალითად განვიხილოთ ერთერთი ფუნქცია COSINE

$$f\left(x_0, \ldots, x_{n-1}\right) = \sum_{i=0}^{n-2} \cos\left(x_i^2 - 0.5x_{i+1}\right)$$
$$x_0 = \ldots = x_{n-1} = 1$$

როგორც ვხედავთ ამ ფუნქციის შემთხვევაში თითქმის ყველა ცლადის მიმართ გრადიენტი რამდენიმე ჯამის ელემენტზეა დამოკიდებული. შესაბამისად, დინებებისთვის დავალებული ნაწილების საზღვრებზე, როგორებიცაა წინა მაგალითში g[8] და g[16], შეიქმნება race condition-ი, რადგან პირველმა დინება მერვე ჯამის ელემენტის განხილვისას და მეორე დინებამ მეცხრე ჯამის ელემენტის განხილვისას შეიძლება ერთად მოახდინონ g[8]-ს მნიშვნელობის შეცვლა. ასეთი შემთხვევების თავიდან ასაცილებლად დინებებს აღარ ვავალებთ სასაზღვრო ჯამის ელემენტებს, ამ ელემენტებთან დაკავშირებულ გამოთვლებს ვაწარმოებთ მთავარ დინებაში.

```
double COSINE
(
```

```
        double *x,
        double *g,
        const int n
)
{
        double item;
        double fx = valgradPool(COSINERanged, x, g, n);
        for (int i = block_size - 1; i < n - 1; i += block_size)
        {
                item = -0.5*x[i + 1] + x[i] * x[i];
                fx += cos(item);
                g[i] -= 2.0*sin(item)*x[i];
                g[i + 1] += 0.5*sin(item);
        }
        return fx;
}
```

ზოგიერთ ფუნქციაში ასევე გამოიკვეთა სხვა ტიპის race condition-ი, რომლის თავიდან ასაცილებლად სხვა მიდგომა გახდა საჭირო. ერთ-ერთი ასეთი ფუნქციაა LAIRWHD.

$$f\left(x_0, \ldots, x_{n-1}\right) = \sum_{i=0}^{n-1} \left(4\left(x_i^2 - x_0\right)^2 + \left(x_i - 1\right)^2\right)$$

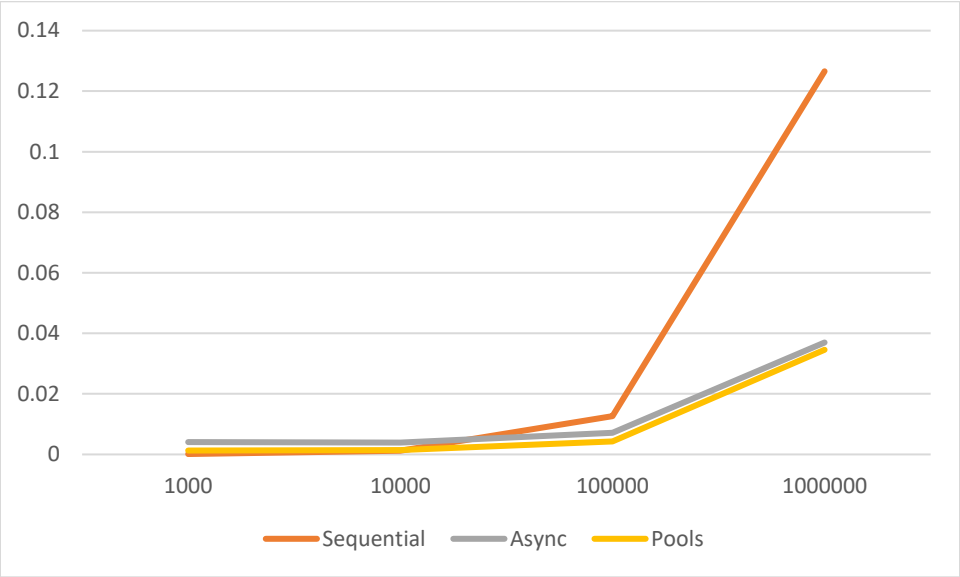
$$x_0 = \ldots = x_{n-1} = 4$$

ამ ფუნქციაში როგორც ვხედავთ g[0] გრადიენტის მნიშვნელობა ჯამის ყველა ელემენტზეა დამოკიდებული. შესაბამისად ყველა დინება ცვლის მის მნიშვნელობას. წინა მიდგომით რომ გადაგვეჭრა ეს პრობლემა მაშინ გამოთვლა თანმიმდევრული (sequential) გახდებოდა. მაგრამ არსებობს უფრო ეფექტური გზაც, კერძოდ, თითოეულ დინებას შეუძლია g[0]-ის მნიშვნელობა, g[0]-ისგან განსხვავებულ, სხვა ცვლადში შეინახოს, მუშაობის დასრულების შემდეგ კი მისი მნიშვნელობა მთავარ დინებას გადააწოდოს. ამის შემდეგ, მთავარი დინება გადმოწოდებულ g[0]-ის ფრაგმენტებს შეკრებს და ჩაწერს g[0]-ში.

ფუნქციების კოლექციის იმპლემენტაციისას ცვადეთ დინებებზე ბრძანებების გაშვების სამივე ხერხი: დინებების პირდაპირ შექმნა, async-ის გამოყენება და გაზიარებული დინებების (thread pool) გამოყენება. ყველაზე ეფექტური გაზიარებული დინებების გამოყენებით დაწერილი იმპლემენტაცია აღმოჩნდა. თუმცა კოლექციაში ასევე შევიტანეთ async-ის გამოყენებით დაწერილი იმპლემენტაციები, რადგან დროთა განმავლობაში ბიბლიოთეკა შეიძლება დაიხვეწოს და შესაბამისად async-ის გამოყენება უფრო ოპტიმალური გახდეს.

ცრილში და შესაბამის გრაფიკზე შეგიძლიათ იხილოთ თანმიმდევრული და კონკურენტული (async-ით და thread pool-ებით) იმპლემენტაციების ეფექტურობის შედარება სხვადასხვა რაოდენობის პარამეტრებიან POWER ფუნქციის მნიშვნელობების და გრადიენტების გამოთვლისთვის. როგორც ვხედავთ, საკმარესად დიდი n-ისთვის კონკურენტული იმპლემენტაციები ბევრად უფრო ეფექტური არიან.

n	Sequential	Async	Pools
1000	0.00013	0.00404	0.00129
10000	0.00126	0.00391	0.00148
100000	0.01261	0.00712	0.00429
1000000	0.12656	0.03697	0.03457



ლიტერატურა

1. William W. Hager and Hongchao Zhang, The Limited Memory Conjugate Gradient Method, SIAM Journal on Optimization, 23 (2013), pp. 2150-2168
2. Speeding up the convergence of the Polyak's Heavy Ball algorithm. https://doi.org/10.1016/j.trmi.2018.03.006 (with Irina Khutsishvili, Luka Gorgadze, Lela Alkhazishvili)
3. Anthony Williams. C++ Concurrency in Action: Practical Multithreading (1st edition). 2012.
4. Async tasks in C++. See https://www.xtof.info/blog/?p=923
5. Eli Bendersky. The promises and challenges of std::async task-based parallelism in C++11. See https://eli.thegreenplace.net/2016/the-promises-and-challenges-of-stdasync-task-based-parallelism-in-c11/
6. Bartosz Milewski. Async Tasks in C++11: Not Quite There Yet. See https://bartoszmilewski.com/2011/10/10/async-tasks-in-c11-not-quite-there-yet/