ჰ.მელაძე პ. წერეთელი

პარალელური ალგორითმები

(ლექციების კურსი)

2014

შესავალი

პირველი კომპიუტერების შექმნის დღიდანვე დღის წესრიგში დადგა უფრო მძლავრი გამოთვლითი ტექნიკის შექმნის საკითხი. ერთ-ერთი პირველი კომპიუტერი ENIAC (Electronic Numerical Integrator and Calculator), რომელიც შეიქმნა 1943 წელს დიდ ბრიტანეთში, წამში ასრულებდა 300 ოპერაციას, რაც 1000-ჯერ მეტი იყო იმ დროის ყველა ელექტრომექანიკური კალკულატორის სიჩქარეზე. მაგრამ ასეთი უზარმაზარი სიჩქარეც კი არასაკმარისი აღმოჩნდა შემქმნელებისა და მომხამრებლისათვის. დღეს ჩვენ გვაქვს კომპიუტერები, რომლებიც მილიონჯერ და მილიარდჯერ უფრო სწრაფია, ვიდრე იყო ENIAC, მაგრამ მოთხოვნა კომიუტერების უფრო მეტ სწრაფქმედებაზე ისევ აქტუალურია. ასტრონომები ცდილობენ შეიცნონ სამყარო, ბიოლოგები ცდოლობენ ადამიანის გენის გაშიფრვას, ინჟინრები ცდილობენ შექმნან უფრო საიმედო და ეკონომიური თვითმფრინავები, ავტომობილები, კოსმოსური ხომალდები და ა.შ. მათ ყველას სჭირდებათ დიდი კომპიუტერული სიმძლავრეები. რა დიდი სიმძლავრეც არ უნდა იყოს ხელმისაწვდომი მოცემულ მომენტში, იგი ყოველთვის არის არასაკმარისი, ყოველთვის არსებობს ამოცანები, რომლეთა გადაჭრისათვის თუნდაც ყველაზე მძლავრი კომპიუტერის შესაძლებლობები არ არის საკმარისი.

წარსულში კომპიუტერის სიმძლავრის გაზრდის პრობლების გადაწყვეტა ძირითადად მდგომარეობდა მისი ტაქტური სიხშირის გაზრდაში. მართალია, ეს არ იყო ერთადერთი საშუალება კომპიუტერის სწრაფქმედების გაზრდისა (მნიშვნელოვან როლს თამაშობდა აგრეთვე ცვლილებები კომპიუტერის არქიტექტურაში), მაგრამ იგი არის უმნიშვნელოვანესი ფაქტორი. სამწუხაროდ, დღეს უკვე მივუახლოვდით ტაქტური სიხშირის გარკვეულ ფუნდამენტალურ ზღვრებს, რომლის იქეთაც ტაქტურის სიხშირის გაზრდა თეორიულადაც შეუძლებელია. საქმე იმაშია, რომ აინშტაინის ფარდობითობის სპეციალური თეორიის თანახმად არავითარი სიგნალი (მათ შორის ელექრონული) არ შეიძლება გავრცელდეს სინათლის სიჩქარეზე უფრო სწრაფად, რომელიც დაახლოებით ტოლია 300 000 კმ/წმ ვაკუუმში და 200 000 კმ/წმ სპილენძის მავთულსა და ოპიკურ-ბოჭკოვან კაბელში. ეს ნიშნავს, რომ ჩვენ რომ გვქონდეს კომპიუტერი, რომლის ტაქტური სიჩქარე არის 10 GHz, სიგნალი ერთი ტაქტის განმავლობაში ვერ გავრცელდება 2 სმ-ზე უფრო შორს, რადგანაც მისი ტაქტის ხანგრძლივობა იქნება 10^{-10} წმ, ხოლო 100 GHz–იან კომიუტერში სიგნალი ერთი ტაქტის განმავლობაში შესძლებს გაიაროს არა უმეტეს 2 მმ-ისა. ამიტომ ასეთ კომიუტერიულ მოწყობილობებს უნდა ჰქონდეთ შესაბამისი ზომები.

არის მეორე პრობლემაც. ჩვენ შეიძლება წარმოვიდგინოთ 2 სმ ზომის პროცესორი და თეორიულადაც დავუშვათ მისი შექმნის შესაძლებლობა, მაგრამ წავაწყდებით მეორე ფუნდამენტურ პრობლემას – სითბოს გამოყოფას. ახლაც კი, როდესაც პროცესორის ტაქტური სიხშირე საშუალოდ არის 3 GHz, მისი გაგრილებისთვის საჭირო ვენტილატორი არის უფრო დიდი, ვიდრე თვითონ ცენტრალური პროცესორი. უფრო მაღალი ტაქტური სიჩქარის მქონე პროცესორებში სითბოს გამოყოფა იქნება უფრო დიდი და, შესაბამისად, უფრო სრულყოფილი გაგრილების საშუალებები იქნება საჭირო.

კომპიუტერის მწარმოებლობის გაზრდის მეორე გზა არის მონაცემთა პარალელური დამუშავება, რაც დაკავშირებულია კომიუტერის არქიტეტურულ ცვლილებებთან. აქ იგულისხმება არამარტო რამდენიმე მანქანური ბრძანების პარალელურად შესრულება, არამედ თითოეული ბრძანების შესრულების დროსაც გარკვეული პარალელიზმის შემოღება და აგრეთვე კომპიუტერის სხვადასხვა მოწყობილობების პარალელურად მუშაობის შესაძლებლობა. ამ გზით კომპიუტერული ტექნიკის განვითარება ხდებოდა ელემენტური ბაზის განვითარების (ანუ ტაქტური სიხშირის გაზრდის) პარალელურად. ასე მაგალითად, თუ ერთ-ერთი პირველ კომიუტერის - 1949 წელს აგებული EDSAC-ის, რომლის ტაქტის ხანგრძლივობა იყო 2 მილიწამი (რაც ნიშნავს 0,5 MHz პროცესორის სიხშირეს) და რომელსაც შეეძალო საშუალოდ 100 ოპერაციის შესრულება წამში, შევადარებთ XXI საუკუნის დასაწყისის ერთ-ერთი მძლავრი კუმპიუტერის HP Superdome-ს ერთ კვანძს, რომლის ტაქტის ხანგრძლივობა არის 1,3 ნანოწამი (რაც ნიშნავს 750 MHz პროცესორის სიხშირეს) და რომელსაც შეუძლია წამში შეასრულოს 192 მილიარდი არითმეტიკული ოპერაცია, ვნახავთ რომ ამ ხნის განმავლობაში პროცესორის სიხშირე გაიზარდა $1{,}5\cdot10^3$-ჯერ, ხოლო კომპიუტერის მწარმოებლობა (ანუ წამში შესრულებული ოპერაციების რაოდენობა) გაიზარდა $1{,}9\cdot10^9$-ჯერ. თუ განვითარება მოხდებოდა მხოლოდ ელემენტური ბაზის გაუმჯობესების (ტაქტური სიხშირის გაზრდის) საფუძველზე, მაშინ მწარმოებლობაც ზუსტად იმდენჯერვე ($1{,}5\cdot10^3$-ჯერ) უნდა გაზრდილიყო. მაშ, საიდან მოვიდა $1{,}9\cdot10^9$-ჯერ ზრდა? ეს ზრდა გამოწვეულია კომპიუტერის არქიტექურაში ახალი გადაწყვეტების გამოყენებით, რომლეთა შორის არცთუ უკანასკნელი ადგილი უკავია მონაცემთა პარალელური დამუშავების პრინციპს, რომელიც რეალიზაციას უკეთებს სხვადასხვა მოქმედებების ერთდროულად შესრულების იდეას.

მაღალი მწარმიებლობის მქონე კომპიუტერები გამოიყენებიან რთული გამოთვლებიათვის. ისეთი ამოცანები, როგორიცაა ამინდის პროგნოზი, თვთმფირანავის ფრთის გარშემო ჰაერის ნაკადის მოდელირება, მსოფლიო ეკონომიკის, წამლის სხვადასხვა რეცეპტორთან ურთიერთქმედების მოდელირება და მრავალი სხვა ამოცანა მოითხოვს ძალიან დიდ გამოთვლით რესურსებს. ასეთი ამოცანების გადასაჭრელად საჭიროა ხანგრძლივი გამოთვლები მრავალ ცენტრალურ პროცესორზე ერთდროულად. ამისათვის გამოიყენება მრავალპროცესორული გამოთვლითი სისტემები. ასეთ სისტემებში პროცესორების რაოდენობა შეიძლება ათასობით იყოს.

კიდევ ერთი გზა გამოთვლითი სისტემის სიმძლავრის გასაზრდელად არის ლოკალური ან გლობალური ქსელით შეერთებული დიდი რაოდენობის ცალკეულ კომპიუტერებზე პარალელური გამოთვლების ჩატარება ერთი ამოცანის გადაჭრისათვის. გამოთვლითი სიმძლავრის თვალსზრისით სისტემა, რომლეც შესდგება მთელ მსოფლიოში გაფანტული 1000 კომპიუტერისგან არაფრით არ განსხვავდება ერთ შენობაში განთავსებული ასეთივე 1000 კომპიუტერისგან შემდგარი სისტემისაგან. ისინი განსხვავდებიან მხოლოდ ინფორმაციის გადაცემის სიჩქარით. არსებობენ ამოცანები, რომლებიც ინფორმაციის გადაცემა გაცილებით ნაკლებია ვიდრე არითმეტიკული გამოთვლების რაოდენობა, ამიტომ მათთვის ინტერნეტით შეერთებული კომპიუტერების (სადაც ინფორმაციის გადაცემის სიჩქარე ნაკლებია ვიდრე ლოკალურ ქსელში) გამოყენება შესაძლებელია. თუკი

გადასაცემი ინფორმაციის რაოდენობა არ არის მცირე, მაშინ უმჯობესი იქნება ლოკალური ქსელით შეერთებული კომპიუტერების გამოყენება. ასე რომ, ორივე ტიპის სისტემები გამოყენებადია სხვადასხვა ტიპის ამოცანებისათვის.

ალბათ ძნელი არ არის 1 მილიონი კომპიუტერი განათავსოთ ერთ შენობაში, რა თქმა უბდა, თუ გაქვთ შესაბამისი ფინანსური რესურსები და შესაბამისი სიდიდის შენობა. 1 მილიონი კომიუტერის განთავსება მთელი მსოფლიოს მასშტაბით კიდევ უფრო ადვილია, რადგან დაგჭირდებათ დიდი ზომის შენობის მოძებნა. ასევე არ იქნება პრობლემა ამ კომიუტერების ქსელში გაერთიანება (ლოკალურში ან გლობალურში). პრობლემები იწყება მაშინ, როდესაც თქვენ დააპირებთ ამ კომპიუტერების გამოყენებას ერთი ამოცანის ამოსახსნელად. შეიძლება მოხდეს რომ 1 მილიონ კომპიუტერზე ან თუნდაც 10000 კომპიუტერზე ამოცანის ამოხსნას უფრო მეტი დრო დასჭირდეს ვიდრე 100 კომპიუტერზე, ან თუნდაც 1 კომპიუტერზე. ბევრი კომპიუტერის (პროცესორის) გამოყენების მიზანი არის ამოცანის ამოხსნის დროის შემცირება. ნებისმიერი ამოცანისათვის არსებობს პროცესორების ზღვრული რაოდენობა რომლის იქეთაც მათი რაოდენობის გაზრდით ამოცანის ამოხსნის დრო არ მცირდება. ხოლო კიდევ უფრო თუ გავზრდით პროცესორების რაოდენობას ამოცანის ამოხსნის დრო ხშირ შემთვევაში გაიზრდება კიდეც. ყველაფერი დამოკიდებულია პარალელური დაპროგრამებისათვის გამოყენებულ პროგრამულ საშუალებებზე და ალგორითმის შინაგან სტრუქტურასა და თავისებურებებზე. ამიტომ გაწეული იყო უზარამაზარი სამუშაო პარალელური დაპროგრამების ტექნოლოგიების დასამუშავებლად და ალგორითმების შინაგანი სტრუქტურის შესასწვლად.

ამრიგად, პარალელური გამოთვლითი სისტემის გამოყენება კონკრეტული ამოცანის გადასაწყვეტად ეფექტური იქნება იმ შემთხვევაში, თუ გამოყენებული ტექნოლოგია და დაპროგრამების პარადიგმა შეესაბამება სისტემის არქიტექტურას, ასევე შეესაბამება ალგორითმი სტრუქტურა გამოთვილითი სისტემის სტრუქტურას, რისთვისაც შეიძლება საჭირო გახდეს ალგორითმის გარკვეულწილად გარდაქმნა. სწორედ ამ საკითხებზე იქნება საუბარი მოდევნო თავებში. განხილული იქნება თანამედროვე პარალელური გამოთვლითი სისტემების სტრუქტურული ნაირსახეობები, დაპროგრამების პარადიგმები, პარალელური დაპროგრამების ტექნოლოგიები.

მაგრამ დავიწყოთ ისტორიით. ვნახოთ თუ როგორ შემოდიოდა პარალელური დამუშავება კომპიუტერში ინფორმაციის დამუშავების სხვადასხვა ეტაპზე.

1. პარალელური გამოთვლითი სისტემები

1.1. მონაცემთა პარალელური დამუშავების ისტორია.

დღეს პარალელური დამუშავების შესაძლებლობის არსებობა კომპიუტერის არქიტექტურაში არავისთვის გასაკვირი არ არის. თავი რომ დავანებოთ ორ- და მეტ ბირთვიან პროცესორებს, სადაც პარალელური დამუშავება “თავლში საცემია”, თითოეული ბირთვის შიგნით და კომპიუტერის სხვა შემადგენელ კომპონენტებშიც გამოიყენება პარალელური დამუშავების ესა თუ ის სახეობა, რომლებიც

რეალიზებულია ერთი მინიატურული კრისტალის შიგნით. მაგრამ თვითონ პარალელური დამუშავების იდეები წარმოიშვა დიდი ხნის წინ. თავდაპირველად ამ იდეების განხორციელება ხდებოდა იმ დროის ყველაზე მოწინავე კომპიუტერებში, რომელთა რაოდენობაც დიდი არ იყო. შემდეგ როდესაც ტექნოლოგია გარკვეულ წილად დამუშავდებოდა და წარმოება გაიაფთებოდა, ისინი ინერგებოდნენ საშუალო დონის კომპიუტერებში. დღეს კი ყველაფერი ეს მთლიანად ინტეგრირებულია თანამედროვე კომპიუტერებში.

იმაში დასარწმუნებლად, რომ ყველა ძირითადი სიახლეები რომლიებიც არის თანამედროვე გამოთვლითი სისტემების არქიტექტურაში გამოიყენება იმ დროიდან, როდესაც არ არსებობდა არც მიკროპროცესორია და არც სუპერკომიუტერის ცნება, გავაკეთოთ მცირე ექსკურსი ისტორიაში და დავიწყოთ პრაქტიკულად პირველი კომიუტერების შექმნის მომენტიდან.

პირველ კომიუტერებში (EDSAC, EDVAC, UNIAC და სხვა, XX საუკუნის 40-50-იანი წლები) მეხსიერების ორგანიზაცია იყო ისეთი, რომ მეხსიერებაში შენახული ინორმაციის (დავუშვათ ბაიტის ან სიტყვის) ბიტები თანმიმდევრობით ერთი მეორის მიყოლებით გადაეცემოდა მოწყობილობას დასამუშავებლად. ამდაგვარადვე, თანრიგ-თანრიგ ხდებოდა არითმეტიკული ოპერაციების შესრულება. ასე რომ, ორი 32-ბიტიანი რიცხვის შესაკრებად საჭირო იყო 32 მანქანური ტაქტი. ეს იყო თანრიგობრივ-მიმდევრობითი მეხსიერების და თანრიგობრივ-მიმდევრობითი არითმეტიკის ეპოქა გამოთვლითი ტექნიკის ისტორიაში.

ელექტრონიკის განვითარებამ საშუალება მისცა ინჟინრებს შეექმნათ თანრიგობრივ-პარალელური მეხსიერება და შესაბამისად, წარმოშვა თანრიგობრივ-პარალელური არითმეტიკა. მანქანური სიტყვის ყველა თანრიგი ერთდროულად წაიკითხება მეხსიერებიდან და მონაწილეობენ ოპერაციის შესრულებაში. 1953 წელს შეიქმნა კომპიუტერი IBM 701 – პირველი კომერციული კომპიუტერი, რომელიც მუშაობდა ასეთ პრინციპზე. მაგრამ ყველაზე უფრო წარმატებული პროექტი იყო IBM 704, რომელშიც პირველად იქნა გამოყენებული მეხსიერება ფერიტულ გულანებზე და ჰქონდა ჰქონდა მცოცავმძიმიან არითმეტიკა.

როგორც IBM 704-ში, ასევ იმ დროის ყველა კომიუტერში შეტანა-გამოტანის ოპერაციები სრულდებოდა არითმეტიკულ-ლოგიკური მოწყობილობის (ანუ პროცესორის იმ ნაწილის, რომელიც უშუალოდ ოპერაციებს ასრულებს) საშუალებით. გარე მოწყობილობები იყო ცოტა. ხოლო მათ შორის ყველაზე სწრაფი – ლენტის წამკითხავი მოწყობილობა მუშაობდა სიჩქარით – დაახლოებით 15 000 სიმბოლო წამში, რაც ბევრად უფრო ნაკლები იყო პროცესორის მიერ მონაცემების დამუშავების სიჩქარეზე. ამის გამო ინფორმაციის შეტანისა და გამოტანის დროს კომიუტერის მწარმოებლობა მნიშვნელოვნად მცირდებოდა. ამ პრობლების ერთ-ერთი პირველი გადაწყვეტა იყო სპეციალიზირებული კომიუტერის შემოღება, რომელსაც ჰქვია შეტანა-გამოტანის არხი და რომლის ერთადერთი დანიშნულებაა ინფორმაციის გარე მოწყობილობებზე ჩაწერა და იქედან წაკითხვა. არითმეტიკულ-ლოგიკურ მოწყობილობას კო შეეძლო ემუშავა შეტანა-გამოტანის არხის პარალელურად. 1958 წელს გამოვიდა კომპიუტერი IBM 709, რომლეიც აგებული იყო IBM 704-ის ბაზაზე და მიერთებული ჰქონდა 6 შეტანა-გამოტანის არხი.

შემდეგი განხორციელებული იდეა კომიუტერის მწარმოებლობის გაზრდისა მდგომარეობდა იმაში, რომ დროში შეთავსებულიყო მეზობელი ბრძანებების შესრულების სხვადასხვა ეტაპები. პირველი ასეთი კომიუტერი იყო IBM STRECH, რომელიც 1961 წელს შევიდა ექსპლუატაციაში. მას ჰქონდა ორი თავისებურება. პირველი – ეს არის რამდენიმე ბრძანებისათვის ოპერანდების წინასწარი წაკითხვა. მეორე თავისებურება მდგომარეობს იმაში, რომ მეხსიერება დაყოფილი იყო ორ დამოუკიდებელ ნაწილად (ბანკად) ისე, რომ არითმეტიკულ-ლოგიკურ მოწყობილობას შეეძლო დამოუკიდებლად ემუშავა ორივე ბანკთან. ეს თავისებურება, რომლესაც ეწოდა მეხსიერების განშრევება, შემდგომში გამოყენებულიიქნა თითქმის ყველა დიდ გამოთვლით სისტემაში. ძირითადი ამოცანა, რომლის გადაჭრაც ამ შემთხვევაში უხდებათ ინჟინრებს მდგომარეობს არითმეტკულ-ლოგიკური მოწყობილობის და მეხიერების შეთანხმებული მუშაობის ორგანიზებაში. საქმე იმაშია, რომ თუ მიმართვა მუდმივად იქნება ერთ და იმავე ბანკზე, ამ შემთხვევაში ორი ან მეტი ბანკის არსებობა ეფექტს არ მოგვცემს. ეფექტისათვის მაქსიმალურად უნდა ვუზრუნველყოთ, რომ მიმართვა ბანკებზე იყოს მონაცვლეობით. თუ მეხსიერება დაყოფილია k დამოუკიდებელ ბანკად (ხანდახან მას სექციასაც უწოდებენ) საჭიროა მეხსიერება ისე იყოს ორგანიზებული რომ მისამართები მონაცვლეობით იყვნენ განთავსებული ბანკებში. ეს ნიშნავს რომ ნებისმიერი k მიმდევრობით განთავსებული მეხსიერების უჯრედები ხვდებიან სხვადასხვა ბანკებში. ასეთ შემთხვევაში მიმდევრობით განთავსებულ მონაცემებთან მუშაობა ხდება მაქსიმალურად ეფექტურად. პრაქტიკაში ეს ნიშნავს მუშაობას ვექტორის ელემენტებთან. ხშირად ბანკების რაოდენობა არის 2-ის ხარისხი $k = 2^m$. ასეთ შემთხვევაში ბანკის ნომერი, რომელშიც მოთავსებულია მისამართი უდრის ამ მისამართის ორობითი წარმოდგენის უკანასკნელ m თანრიგში ჩაწერილ რიცხვს.

1962 წელს ექსპუატაციაში შევიდა კიდევ ერთი კომპიუტერი, რომლემაც მნიშვნელოვანი როლი ითამაშა გამოთვლითი ტექნიკის ისტორიაში. ეს იყო კომპიუტერი ATLAS. ჩვენთვის ეს კომპიუტერი საინტერსოა იმით, რომ მასში პირველად იქნა გამოყენებული ბრძანებების დამუშავებისას კონვეიერის პრინციპი. ბრძანების დამუშავების ციკლი დაყვეს ოთხ ნაწილად: ბრძანების წაკითხვა, ოპერანდის მისამართის გამოთვლა, ოპერანდის წაკითხვა და ოპერაციის შესრულება. ბრძანების შესრულების თითოეულ ამ ეტაპს დამოუკიდებლად ასრულებდა ცალკე მოწყობილობა, რომელთაც შეეძლოთ ემუშავათ პარალელურად. ერთი ეტაპის დასრულების შემდეგ ბრძანება გადაეცემა მომდევნო მოწყობილობას. ასე რომ, დროის მოცემულ მომენტში შესრულების პროცესში (ოღონდ შესრულების სხვადასხვა ეტაპზე) იქნებოდა 4 ბრძანება. ეს არის კონვეიერული დამუშავების პრინციპი, რომლის შესახებაც მომავალში გვექნება საუბარი დაწვრილებით. ბრძანებების დამუშავების ასეთმა მეთოდმა საშუალება მოგვცა ATLAS-ში ხელსაყრელი პირობების შემთხვევაში ბრძანების შესრულების საშუალო დრო შემცირებულიყო 6 მიკროწამიდან 1,6 მიკროწამამდე. ხოლო კომპიუტერის მწარმოებლობა გაიზარდა 200 ათას ოპერაციამდე წამში.

1964 წელს კომპანია Control Data Corporation გამოუშვა კომპიუტერი CDC 6600. ამ კომპიუტერის გენერლური კონსტრუქტორი და იმ პერიოდში კომპანიის ვიცე-პრეზიდენტი იყო სეიმურ ქრეი (Seymour Cray), რომლეიც მომვალში გახდა ბევრი უნიკალური სუპერკოპიუტერის ავტორი. ამ კომიუტერის ცენტრალური პროცესორი

არითმეტიკულ-ლოგიკური მოწყობილობის ნაცვლად, რომელიც ასრულებდა ყველა მანქანურ ოპერაციას, შეიცავდა 10 დამოუკიდებელ ფუნქციონალურ მოწყობილობას. მათ შეეძლოთ ემუშავათ ერთმანეთისგან დამოუკიდებლად და იყვნენ სპეციალიზირებულები, ანუ თითოეული მათგანი ასრულებდა რაიმე კონკერულ ოპრაციას ან ოპარაციათა ერთობლიობას. მაგალითად, მცოცავმძიმიანი შეკრება-გამოკლებისთვის არის ცალკე მოწყობილობა, მთელი რიცხვების შეკრება-გამოკლებისთვის – ცალკე, გამრავლებას ასრულებს – მესამე მოწყობილობას, გაყოფას – მეოთხე, ლოგიკურ ოპერაციებს – მეხუთე და ა.შ. ზოგიერთი ოპერციისთვის მოწყობილობა შეიძლება ყოფილიყო დუბლირებული. თუ აქამდე პარალელური დამუშავება იყო, ასე ვთქვათ, “შეფარული სახით”, ეს იყო პარალელიზმი ცხადი სახით. ამ კომპიუტერის საშუალო მწარმოებლობა იყო 2-3 მლნ. ოპერაცია წამში.

1969 წელს Control Data Corporation–მა გამოუშვა CDC 7600. ეს იყო გაუმჯობესებული CDC 6600. მისი ცენტრალური პროცესორი შეიცავდა 8 ფუნქცინალურ მოწყობილობას და თითოეული მათგანი იყო კონვეიერული, რაც ნიშნავს, რომ დროის მოცემულ მომენტში ერთ ასეთ მოწყობილობაში შეიძლება დამუშავების პროცესში იყოს რამდენიმე ოპერაცია. ე.ი. ამ კომიუტერში ერთდროულად გამოიყენებოდა პარალელიზმიც (8 დამოუკიდებელი მოწყობილობა) და კონვეიერული დამუშავებაც (თითოეული მოწყობილობა კონვეიერულია). მისი საშუალო მწარმოებლობა გახდა 10-15 მლნ. ოპერაცია წამში.

შემდეგი მნიშვნელოვანი მოვლენა პარალელური გამოთვლითი ტექნიკის განვითარებაში იყო კომპიუტერ ILLIAC IV-ის შექმნა. თავდაპირველი პროექტით მას უნდა ჰქონოდა 256 პროცესორული ელემენტი, რომლებიც გაერთიანებული იქნებოდნენ 4 მატრიცაში თითოეულში 64 პროცესირული ელემენტით (მატრიცის ზომა 8 X 8). თითოეულ მატრიცას ექნებოდა თავისი მართვის მოწყობილობა, რომელიც მართავდა 64-ვე სინქრონულად მომუშავე პროცესორს. თითოეულ პროცესორს ექნებოდა თავისი ლოკალურო მეხსიერება. ამრიგად, მატრიცაში 64 პროცესორი შეასრულებდა ერთი და იმავე ოპერაციას სხვადასხვა (თავის ლოკალურ მეხსიერებაში განთავსებულ მონაცემებზე). გარდა ამისა იქნებიდა ერთი საერთო მართვის მოწყობილობა, რომელსაც უნდა ემართა ეს 4 მატრიცა. ტაქტის ხანგრძლივობა პროექტის მიხედვით უნდა ყოფილიყო 40 ნანოწამი, ხოლო მანქანის მწარმოებლობა 1 მლრდ. ოპერაცია წამში – ფანტასტიკური სიჩქარე იმ დროისათვის! და ასევე ფანტასტიკური სირთულის პროექტი! ILLIAC IV-ის პროექტმა ძლიერ გაუსწრო დროს, რამაც გამოიწვია შეფერხებები და გეგმების მრავალგზის კორექტირება. პროექტზე მუშაობა დაიწყო 1967 წელს. 1972 წლისთვის დაამზადეს ერთი მატრიცა 64 პროცესორით, რომელიც დააყენეს NASA-ს სამეცნიერო-კვლევით ცენტრში. მისი მონტაჟი გრძელდებოდა 1975 წლამდე. ეს სისტემა რეალურად გამოიყენებოდა 1982 წლამდე, რის შემდეგაც ეს ერთადერთი ეგზემპლარი გადაიტანეს მუზეუმში. რა გამოვიდა? დაგეგმილი იყო 256 პროცესორიანი სისტემა, დამზადდა 64 პროცესორიანი. ტაქტის ხანგრძლივობა 40 ნანოწამის მაგივრად 80 ნანოწამამდე გაიზარდა, ხოლო მწარმოებლობა დაეცა 50 მლნ. ოპერაციამდე წამში. თანაც პროექტის მეოთხედის (1 მატრიცა 4-ის მაგივრად) დამზადება დაჯდა 4-ჯერ უფრო ძვირი ვიდრე მთელი პროექტისთვის იყო დაგეგმილი. თითქოსდა, პროექტმა განიცადა სრული კრახი. მაგრამ, არა! ამ პროექტმა უზარმაზარი გავლენა იქონია როგორც მომდევნო თაობის

კომპიუტერების არქიტექტურაზე, ისე პარალელური გამოთვლითი სისტემის პროგრამული უზრუნველყოფის მრავალი კომპონენტის განვითერებაზე. პროექტის ფარგლებში დამუშავებული ტექნოლოგიები გამოყენებული იქნა PEPE, BSP, ICL და სხვა სისტემების შექმნისას. იდეები, რომლების ჩადებული იყო დაპროგრამების ენა FORTRAN-ის გამპარალელებელ კომპილატორში საფუძვლად დაედო შემდგომში შექმნილი პარალელური კომპიუტერებისთვის სხვდასხვა სახის პროგრამულ ანალიზატორებს და კომპილატორებს.

კომპიუტერის მანქანურ ბრძანებათა სისტემაში ვექტორული ოპერაციების ჩართვის იდეა დიდი ხნის განმავლობაში იზიდავდა მკვლევარებს. ვექტორული ოპერაცებით უფრო მარტივად ჩაიწერება ფრაგმენტები მანქანურ ენაზე. თუ საჭიროა ორი ვექტორის შეკრება $A = B + C$, საჭირო აღარ იქნება ისეთი დამხმარე ბრძანებები, როგორიცაა ინდექსის გაზრდა, ციკლიდან გამოსვლის პირობის შემოწმება, ციკლის დასაწყისზე გადასვლა და სხვა – საკმარისია შესაბამისი ვექტორული ბრძანების გამოყენება. ასეთი ფრაგმენტის ეფექტურობა გაცილებით უფრო დიდია, ვიდრე ტრადიციული მეთოდებით ჩაწერილის.

ვექტორული ბრძანებების მქონე გამოთვლითი სისტემების წარმატებული პროექტების მაგალითები არცთუ ცოტაა. მაგრამ ყველაზე ნათელი მაგალითები ასეთი კომპიუტერებისა არის Cray-ის ვექტორულ-კონვეიერული კომპიუტერების ხაზი.

1972 წელს სეიმურ ქრეიმ მრავალი კომპიუტერის გენერალურმა კონსტრუქტორმა და Control Data Corporation-ის ვიცე-პრეზიდენტმა დატოვა კომპანია და დაარსა ახალი კომპანია Cray Research, რომელმაც 1976 წელს გამოუშვა პირველი ვექტორულ-კონვეიერული კომპიუტერი Cray-1. მის განსაკუთრებულ თავისებურებებს წარმოადგენდა ვექტორული მანქანური ბრძანებები და დამოუკიდებელი კონვეიერული ფუნქციონალური მოწყობილობები. Cray-1-ის ტექნიკური მახასიათებლები: ტაქტის ხანგრძლივობა 12,5 ნანოწამი, 12 კონვეიერული ფუნქციონალური მოწყობილობა, რომლებსაც შეიძლიათ ერთმანეთისგან დამოუკიდებლად პარალელურად მუშაობა, ოპერატილი მეხსიერება – 1 მლნ. სიტყვა. დღევანდელი გადმოსახედიდან არცთუ ისე შთამბეჭდავი მახასიათებლებია, მაგრამ იმ დროს ეს იყოს ყველზე სწრაფქმედი და ყველაზე განვითარებული არქიტექტურის მქონე მანქანა.

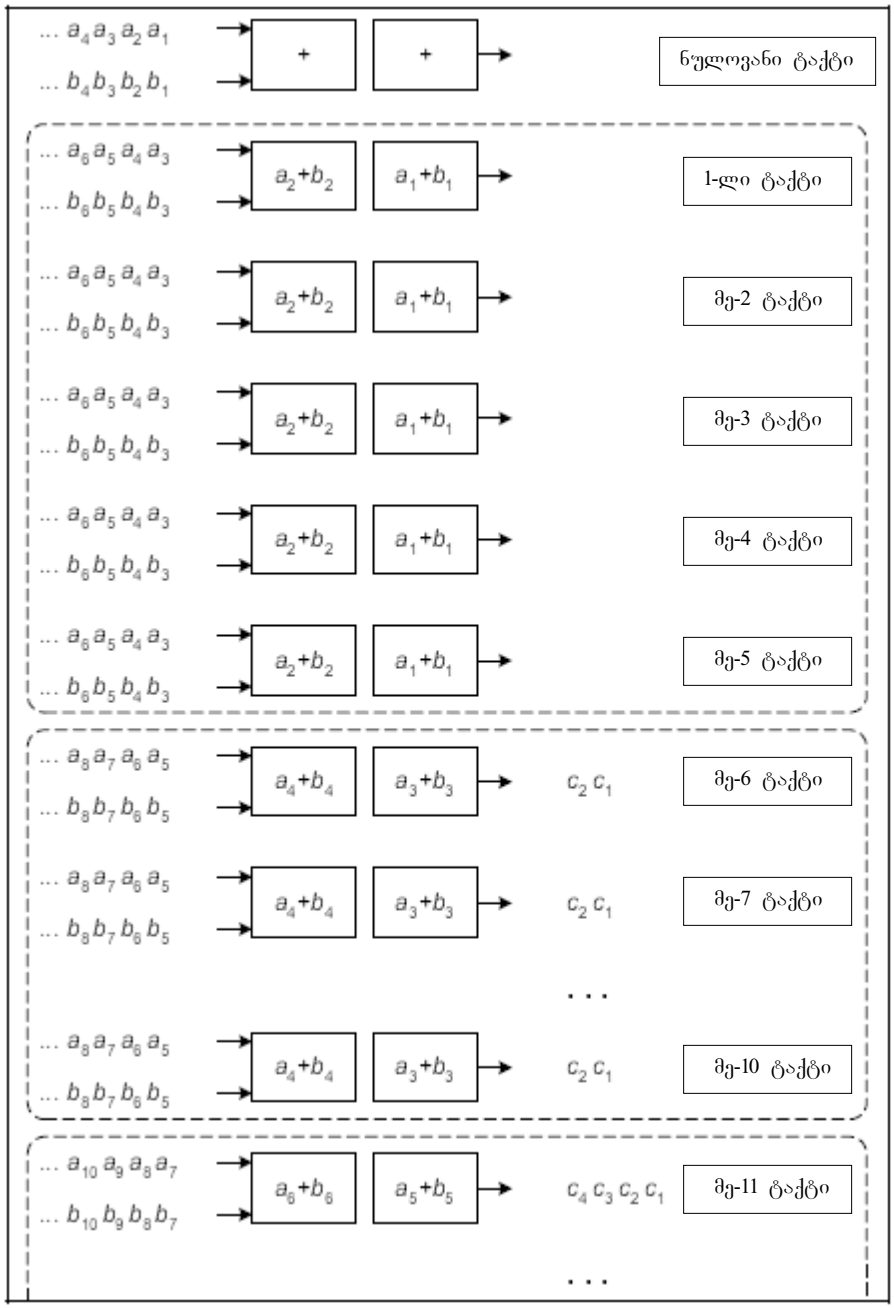
1.2. მონაცემთა პარალელური დამუშავების სახეები

მიუხედავად კომიუტერის არქიტექტურაში პარალელიზმის გამოვლინების მრავალფეროვნებისა არსებიბს მონაცემთა პარალელური დამუშავების მხოლოდ ორი სახე – საკუთრივ პარალელური დამუშავება და კონვეიერული დამუშავება.

1⁰. პარალელური დამუშავება

დავუშვათ, უნდა ვიპივოთ ორი 100-ელემენტიანი ვექტორის ჯამი. ვაქვს მოწყობილობა, რომელიც გამოითვლის ორი რიცხვის ჯამს 5 ტაქტში. მოწყობილობა აგებულია ისე, რომ ამ 5 ტაქტის განმავლობაში იგი ბლოკირებულია და სხვა არაფერი სასარგებლო სამუშაოს შესრულება არ შეუძლია. ასეთ პირობებში ვექტორებზე ოპერაცია დასრულდება 500 ტაქტის

განმავლობაში. ახლა დავუშვათ, გაქვს 2 ასეთი მოწყობილობა, რომლებსაც შეუძლიათ მუშაობა ერთდროულად და ერთმანეთისგან დამოუკიდებლად. სიმარტივისთვის განვიხილოთ სიტუაცია სადაც არ არის სხვა არაფერი დამატებითი ხარჯები დაკავშირებული მონაცემთა მიღებასთან და შენახვასთან. თუ ორივე მოწყობილობას მუდმივად დავტვირთავთ სამუშაოთი, ცხადია, შედეგს მივიღებთ 250 ტაქტის შემდეგ. სამუშოს შესრულება აჩქარდა 2-ჯერ. 10 ასეთი მოწყობილობის შემთვხევაში ორი ვექტორის ჯამს გამოვითვლით 50 ტაქტში. ზოგადად, N ასეთი მოწყობილობა მიიღებს შედეგს დაახლოებით $500/N$ ტაქტის შემდეგ.



2^0. კონვეიერული დამუშავება

წინა მაგალითში შეკრების ოპერაციის შესასრულებლად მოწყობილობა ბლოკირებული იყო 5 ტაქტის განმავლობაში და არავითარ სხვა სამუშაოს არ ასრულებდა. შესაძლებელია თუ არა რომ შემავალი მასივების ელემენტების დამუშავება მოხდეს უფრო ეფექტურად? ამ კითხვაზე პასუხის გასაცემად უნდა გაიხსენოთ თუ როგორ ხდება ნამდვილი რიცხვების წარმოდგენა კომპიუტერში და მათზე ოპერაციების ჩატარება. შეგახსენებთ, რომ კომპიუტერში ნამდვილი (მცოცავმძიმიანი) რიცხვები წარმოდგება 2-ის ფუძიანი მახასიათებლისა და მანტისის სახით. თითოეული წყვილის შეკრება ხდება რამოდენიმე მიკროოპერაციის საშუალებით – რიგების შედარება, რიგების გათანაბრება, მანტისების შეკრება, ნორმალიზაცია და ა.შ. და რაც მნიშვნელოვანია, თითოეული წყვილის დამუშავების დროს მიკროოპერაციები სრულდება მხოლოდ ერთხელ და ერთი და იგივე თანმიმდევრობით. ეს ნიშნავს, რომ თუ, მაგალითად, პირველი მიკროოპერაცია უკვე შესრულდა და მონაცემები გადაეცა მეორე მიკროოპრეციას პირველი მიკროოპერაცია ამ წყვილის დაუშავებისთვის უკვე საჭირო აღარ არის. ამიტომ მან შეიძლება დაიწყოს როგში მდგარი მომდევნი წყვილის დამუშავება.

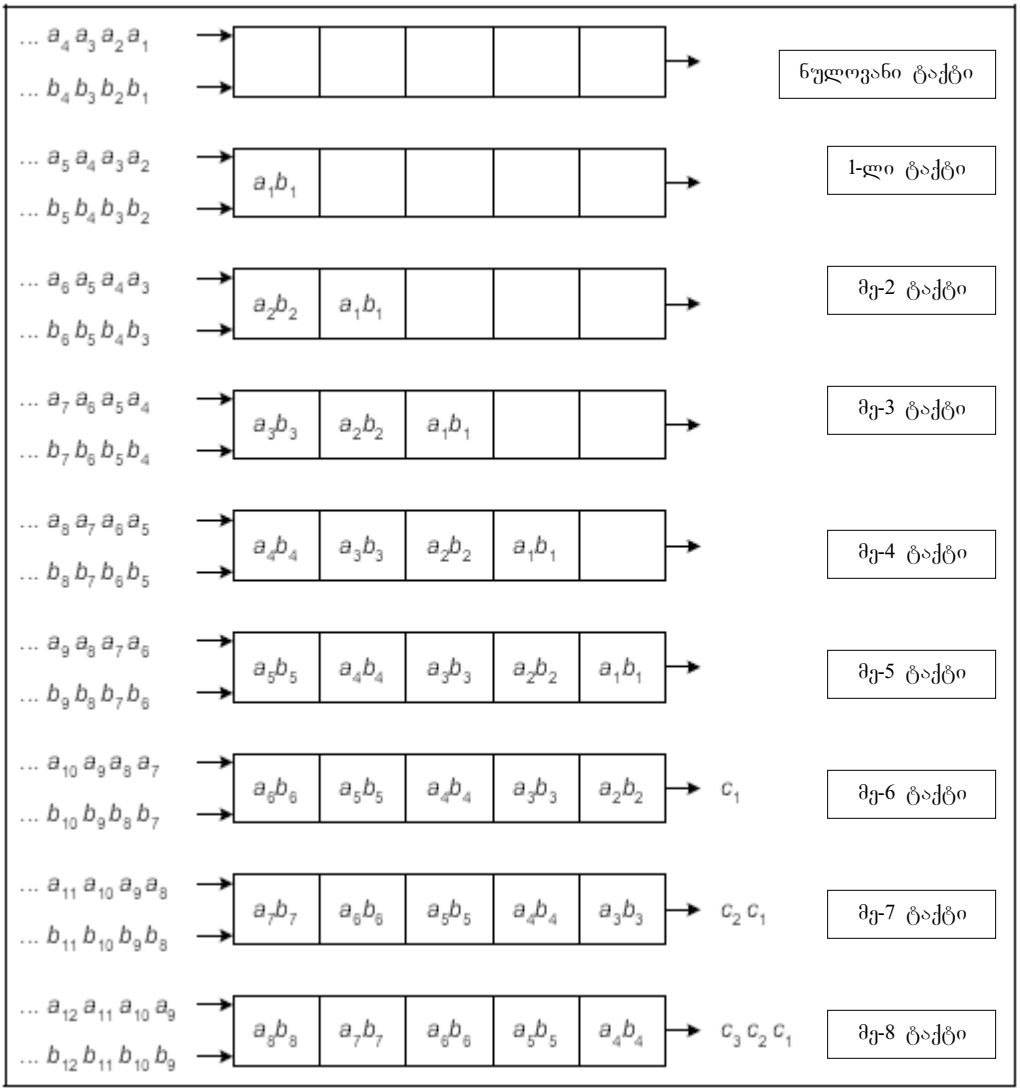
აქდან გამომდინარე ავაგოთ შეკრების მოწყობილობა შემდეგნაირად. თითოეული მიკროოპერაცია გამოვყოთ მოწყობილობის ცალკე ნაწილად და განვათავსოთ ისინი შესრულების რიგის მიხედვით. დასაწყისში მონაცემები ხვდებიან პირველ ნაწილში. პირველი მიკროოპერაციის შესრულების შემდეგ მონაცემები გადადიან მეორე ნაწილში და პირველ ნაწილში შემოდის ახალი წყვილი. როდესაც შემავალი არგუმენტები გაივლიან დამუშავების ყველა ეტაპს, გამოსავალზე იქნება შედეგი.

მონაცემების ასეთ დამუშავებას ეწოდება კონვეიერული დამუშავება. მოწყობილობის თითოეულ ნაწილს ეწოდება კონვეიერის საფეხური, ხოლო საფეხურების რაოდენობას – კოენვეიერის სიგრძე. დავუშვათ, შეკრების ოპერაციის შესასრულებლად დამზადებულია ხუთსაფხურიანი კონვეიერული მოწყობილობა და თითოეული საფეხურო მუშაობს ერთი ტაქტის განმავლობაში. ეს ნიშნავს რომ შეკრების ერთი ოპერაცია შესრულდება 5 ტაქტში, ე.ი. იმდენივე დროში რამდენიც საჭირო იყო მიმდევრობით მოწყობილობაზე წინა შემთხვევაში.

ახლა განვიხილოთ ორი მასივის შეკრების შემთხვევა. როგორც წინა შემთხვევაში პირველი წყვილის შეკრების შედეგი მიღებული იქნება 5 ტაქტის შემდეგ. მაგრამ შევნიშნოთ, რომ პირველი წყვილის შეკრებასთან ერთად ნაწილობრივ დამუშავებული იქნება მეორე წყვილიც, მესამე წყვლიც, მეოთხე წყვილიც და მეხუთე წყვილიც. ოღონდ ისინი იქნებიან დამუშავების სხვადასხვა სტადიაზე. ყოველ მომდევნო ტაქტზე კი მიღებული იქნება მორიგი ელემენტების წყვილის შედეგი. ასე, რომ 100 ელემენტის შეკრებას დასჭირდება 104 ტაქტი. მიმდევრობითი მოწყობილობის შემთხვევაში კი საჭირო იქნებოდა 500 ტაქტი. თითქმის 5-ჯერ ნაკლები დრო დასჭირდა კონვეიერულ დამუშავებას.

ახლა განვიხილოთ ზოგადი შემთხვევა. დავუშვათ, კონვეიერის სიგრძე არის l, ხოლო თითოეული საფეხური მუშაობს ერთი ტაქტის განმავლობაში. ამ შემთხვევაში n სიგრძის მასივის დამუშავებას დასჭირდება $n+l-1$ ტაქტი. ეს

მოწყობილობა გამოყენებული რომ ყოფილიყო მიმდევრობით რეჟიმში n სიგრძის მასივის დამუშავებას დასჭირდებოდა $n \times l$ ტაქტი. n-ის დიდი მნიშვნელობებისთვის კონვიერული დამუშავებისას ოპერაცია შესრულდება თითქმის l-ჯერ სწრაფად.



როგორც ვხედავთ, კონვეიერული მოწყობილობის გამოყენება ეფექტს იძლევა ვექტორების დამუშავებისას.

შემოვიტანოთ ვექტორული ოპერაციის ცნება. ეს არის ისეთი ოპერაცია, რომლის ერთი არგუმენტი მაინც არის ვექტორი. წინააღმდეგ შემთხვევაში ოპერაცია არის სკალარული. განვიხილოთ ორი ვექტორის შეკერების ოპერაცია. იგი შეიძლება შესრულდეს ერთი ვექტორული ოპერაციით ან n სკალარული ოპერაციით. თუ n საკალარული ოპერაცია თანმიმდევრობით შესრულდება კონვეიერულ მოწყობილობაზე, ჩვენს მიერ მიღებული შეფასების თანახმად მის შესრულებას დასჭირდება $n+l-1$ ტაქტი. მაგრამ პრაქტიკაში ასეთი შედეგის მიღწევა შეუძლებელია. ამ შედეგის მისაღწევად საჭიროა, რომ ყოველ ტაქტზე კონვეიერულ მოწყობილოდას მიეწოდოს მონაცემთა ახალი წყვილი. მაგრამ ეს შეუძლებელია, რადგანაც გარდა ამისა, ოპერანდების ყოველი წყვილისათვის უზრუნველყოფილი

უნდა იყოს შედეგის შენახვა, შემოწმებული უნდა იყოს საჭიროა თუ არა მომდევნო იტერაციის შესრულება, ციკლის მთვლელი უნდა გაიზარდოს ერთით და ა.შ. ეს ნიშნავს, რომ სკალარული ოპერაციებით ვექტორების დამუშავებისთვის საჭიროა დამატებით დამხმარე ოპერაციების შესრულება, რის გამოც შეუძლებელი ხდება ყოველ ტაქტზე კონვეიერულ მოწყობილობას მიეწოდოს მონაცამთა ახალი წყვილი. ამის გამო, შედეგსაც ვეღარ მივიღებთ ყოველ ტაქტზე და ჩვენს მიერ მიღებული შეფასება მიუღწეველი იქნება.

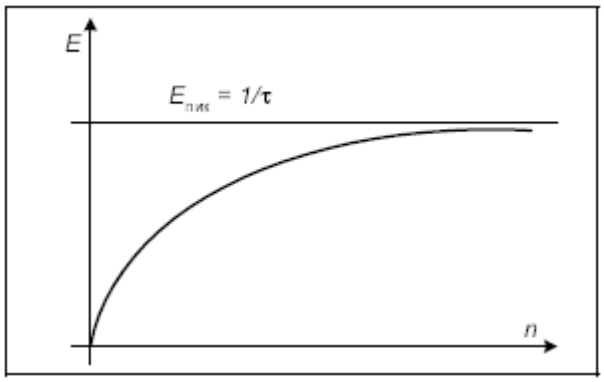
სხვანაირად არის საქმე ვექტორული ოპერაციების შემთხვევაში. ვექტორული ოპერაცია არ ითხოვს რაიმე დამატებითი მოქმედებების შესრულებას გარდა თვით ამ ოპერაციის ინიცალიზაციისა. დავუშვათ, ვექტორული ოპერაციის ინიცალიზაციისათვის საჭიროა σ ტაქტი. მაშინ n სიგრძის ვექტორის დამუშავებას დასჭირდება $\sigma+n+l-1$ ტაქტი. ამ ფორმულაში σ-ს შემოტანა ნიშნავს, რომ n-ის მცირე მნიშვნელობებისათვის უმჯობესია გამოვიყენოთ სკალარული ბრძანებები, და არა ვექტორული.

რადგანაც σ-ს და l-ის მნიშვნელობები არანაირად არ არის დამოკიდებული n-ზე, ამიტომ ვექტორის სიგრძის ზრდასთან ერთად კონვეიერული მოწყობილობის ეფექტურობა იზრდება. თუ ეფექტურობის ქვეშ გავიდებთ კონვეიერული მოწყობილობის მწარმოებლობას (E), რომელიც ტოლია ერთ წამში შესრულებული ოპერაციების რაოდენობისა (შესრულებული ოპერაციების რაოდენობა შეფარდებული შესრულებაზე დახარჯულ დროსთან), მაშინ ეფექტურობის ვექტორის სიგრძეზე დამოკიდებულიების ფორმულას ექნება სახე:

$$E=\frac{n}{t}=\frac{n}{(\sigma+n+l-1)\tau}=\frac{1}{\tau+(\sigma+l-1)\dfrac{\tau}{n}}$$

სადაც τ არის კომპიუტერის ტაქტის ხანრგრძლივობა.

ნახაზზე მოყვანილია ამ დამიკიდებულების გრაფიკი. შემავალი მონაცემების ზრდასთან ერთად კონვეიერული მოწყობილობის მწარმოებლობა სულ უფრო უახლოვდება პიკურ მწარმოებლობას. მაგრამ ნებისმიერი კონვეიერული მოწყობილობის პიკური მწარმოებლობა ვერასოდეს ვერ იქნება მიღწეული.



1.3. პარალელური გამოთვლითი სისტემების კლასიფიკაცია

თანამედროვე პარალელური გამოთვლითი სისტემები შეიძლება დაიყოს სამ ძირითად კლასად:

1) მულტიპროცესორები საერთო მეხსიერებით (ან უბრალოდ მულტიპროცესორები). ეს არის კომპიუტერული სისტემა, რომლეშიც ორი ან მეტი ცენტრალურ პროცესორს აქვს სრული და თანაბარი წვდომა საერთო მეხსიერებასთან. ასეთი სისტემის მაგალითია თანამედროვე მრავალპროცესორიანი (მრავალბირთვიანი) კომპიუტერები. პროგრამა, რომელიც მუშაობს ნებიმიერ პროცესორზე (ბირთვზე) ხედავს მთელ ვირტუალურ სამისამართო სივრცეს. ერთიდაერთი "უცნაურობა" რაც გააჩნია ასეთ სისტემას არის ის, რომ შეიძლება ერთმა პროცესორმა ჩაწეროს რაომე მონაცემი მეხსიერების ველში, შემდეგ წაიკითხოს იგივე ველიდან მონაცემი და იგი აღმოჩნდეს განსხვავებული ჩაწერილისგან. გამოთვლების სწორი ორგანიზაციის შემთხვევაში სწორედ ეს თვისება არის საფუძველი პროცესორებს შორის ინფორმაციის გაცვლისა: ერთი პროცესორი ჩაწერს მეხსიერებაში მონეცემს, მეორე წაიკითხავს მას.
2) გამოთვლითი სისტემები განაწილებული მეხსიერებით. ამ კლასის სისტემის აგების იდეა ძალიან მარტივია. გამოთვლითი კვანძების გარკვეული რაოდენობა გაერთიანებულია რაიმე კომუნიკაციური ქსელის საშუალებით. ყოველ გამოთვლით კვანძს აქვს ერთი ან რამდენიმე პროცესორი და საკუთარი მეხსიერება, რომლებთანაც აქვთ წვდომა ამ პროცესორებს. მეხსიერების განაწილება ნიშნავს, რომ თითოეულ პროცესორს აქვს უშუალო წვდომა მხოლოდ თავის კვანძებში განთავსებულ მეხსიერებასთან. სხვა კვნაძებში განთავსებულ მონაცემებთან წვდომა ხდება სპეციალური, უფრო რთულ საშუალებებით (როგორც წესი, ეს არის ინფორმაციის მიღება/გაგზავნაზე დაფუძნებული საშუალებები). კომუნიკაციური ქსელი შეიძლება იყოს სპეციალურად ამ გამოთვლითი სისტემისათვის დაპროექტებული ან გამოყენებული იყოს ბაზარზე არსებული სტანტარტული საქსელო ტექნოლოგიები. უკანასკნელ პერიოდში, კვანძების სახით ძირითადად გამოიყენება სრულფასოვანი კომპიუტერები, რომლებსაც შეიძლება ჰქონდეთ თავიანთი პერიფერიული მოწყობილობებიც კი. ასეთ სისტემებს გამოთვლითი კლასტერები ეწოდებათ. ისინი დღეისათვის შეადგენენ პარალელური გამოთვითი სისტემების უდუდეს ნაწილს.
3) მეტაკომპიტინგი (სხვაგვარად მას GRID-ტექნოლოგიას უწოდებენ). ნებისმიერი გამოთვილითი მოწყობილობები შეიძლება ჩაითვალოს პარალელურ გამოთვლით სისტემად თუ ისინი მუშაობენ ერთდროულად და მათი გამოყენება შესაძლებელია ერთი ამოცანის ამოსახსნელად. ამ მხრივ უნიკალურ შესაძლებლობას იძლევა ინტერნეტი, რომელიც შეიძლება განვიხილოთ როგორც მსოფლიოში ყველაზე დიდი კომპიუტერი. ეს არის კომპიუტერი, რომელიც შესდგება კომპიუტერებისაგან, ანუ მეტაკომპიუტერი. ერთმანეთისგან დაშორებული (მთელ დედამიწაზე გაფანტული) კომიუტერების გლოებალური ქსელის საშუალებით ერთ გამოთვლით სისტემაში გაერთიანებით შესაძლებელია სპეციალური გამოთვლითი გარემოს ფორმირება. ზოგი კომპიუტერი შეიძლება ჩაირთოს და გამოირთოს ამ გარემოში, მაგრამ მომხმარებლის თვალსაზრისით ეს ვირტუალური გარემო წარმოადგენს ერთიან მეტაკომპიუტერს. ასეთ გარემოში მუშაობისას მომხმარებელი მხოლოდ აგზავნის

დავალებას ამოცანის ამოსახსნელად, დანარჩენს კი მეტაკომპიუტერი აკეთებს თვითონ: ეძებს შესაბამის გამოთვლით რესურსებს, თვალყურს ადევნებს მათი მუშაობის პროცესს (მაგალითად, ამოცანის ამოხსნის პროცესში ეს კომპიუტერი შეიძლება გამოირთოს, ამიტომ მეტაკომპიუტერს შესაძლებლობა უნდა ჰქონდეს გადაამისამართოს დავალება სხვა კომპიუტერზე), საჭიროების შემთხვევაში ახორციელებს მონაცემთა გარდაქმნას კონკრეტული კომიუტერისათვის საჭირო ფორმატში და ა.შ. მომხმარებელმა შეიძლება ვერც კი გაიგოს თუ რომელ კომიუტერზე იქნა ამოხსნილი მისი ამოცანა. პრინციპში, ეს მას არაფერში არ ჭირდება.

1.4. ამდალის კანონი

ახლა გამოვთვალოთ თუ რამდენჯერადი აჩქარება შეიძლება მოგვცეს პროგრამის შესრულებისას მულტიპროცესორის გამოყენებამ. ჯერ განვმარტოთ აჩქარების ცნება. აჩქარება – ეს არის ერთ პროცესორზე პროგრამის შესრულებისათვის საჭირო დროის შეფარდება ზუსტად ასეთივე p პროცესორზე იგივე პროგრამის შესრულებისთვაის საჭირო დროსთან. თუ 1 პროცესორზე პროგრამა სრულდება T_1 დროში, ხოლო p პროცესორზე - T_p დროში, მაშინ აჩქარება κ იქნება:

$$\kappa = \frac{T_1}{T_p}$$

ყველა პროგრამის შესრულება მრავალპროცესორული სისტემაზე არ შეიძლება. უფრო სწორედ, პროგრამა კი შესრულდება მრავალპროცესორულ სისტემაზე, მაგრამ გამოიყენებს მხოლოდ ერთ პროცესორს. ასეთ შემთხვევაში ამბობენ, რომ პროგრამა არ პარალელდება ან პროგრამის გაპარალელება შეუძლებელია. მაგალითისთვის განვიხილოთ მასივის ელემენტების ჯამის პოვნის ტრადიციული ალგორითმი: $S = S + a_i$. ამ გამოსახულების გამოთვლა ხდება ციკლში n-ჯერ. ასეთ პროგრამას პარალელურად ვერ შევასრულებთ, რადგან S-ს ვერ დავუმატებთ a_2-ს სანამ S-ში არ იქნება a_1, ვერ დავუმატებთ a_3-ს, სანამ S-ში არ იქნება a_1-ისა და a_2-ის ჯამი და ა.შ. თუ ამ პროგრამას შევასრუ;ლებთ, მაგალითად, 4 ბირთვიან კომპიუტერზე, იმუშავებს მხოლოდ ერთი ბირთვი. რა უნდა ვქნათ ასეთ შემთხვევაში? უნდა ვეაცდოთ როგორღაც შევცვალოთ ალგორითმი ისე, რომ შესაძლებელი იყოს ბევრი პროცესორის გამოყენება. ამ ალგორითმის გარდაქმნის ერთ-ერთი ვარიანტი შეიძლება იყოს შემდეგი. მასივი დავყოთ 4 შეძლებლისდაგვარად ტოლ ნაწილად (თუ ვიყენებთ 4 ბირთვიან კოპიუტერს, თუ გვაქვს 8 ბირთვიანი კომპიუტერი, მაშინ დავყოთ 8 ნაწილად). თითოეულ ჯგუფში დავითვალოთ ცალ-ცალკე ჯამი და შემდეგ ეს ჯამები შევკრიბოთ. მაგალითად, თუ გვქონდა 1000-ელემენტიანი მასივი, პირველ ჯგუფში იქნება პირველი 250 ელემენტი, მეორეში – შემდეგი 250 და ა.შ. ასეთი ალგორითმის გაპარალელება ადვილია, თითოეულ ჯგუფი მივაკუთვნოთ ცალკე პროცესორს და მათ გამოთვალონ ჯგუფის ელემენების ჯამი. ანუ, ჩვენ შემთხვევაში ოთხივე ჯგუფში ჯამი გამოითვლება პარალელურად 4 ბირთვის მიერ – თითოეული ბირთვი გამოთვლის მისთვის მიკუთვნებული ჯგუფის ჯამს. ამის შემდეგ ეს ჯამები შევკრიბოთ ტრადიციული ალგორითმით. ცხადია, ეს ნაწილი არ გაპარალელდება, მაგრამ გაუპარალელებელი დაგვრჩება 4 რიცხვის ჯამის პოვნა და არა 1000-ის, როგორც თავდაპირველ ალგორითმის შემთხვევაში იყო. მაგრამ ეს უკვე სხვა ალგორითმია. მცოცავმძიმიან მონაცემებზე

მოქმედებისას ამ ორმა ალგორითმმა შეიძლება სხვადასხვა შედეგი მოგვცენ, რაც განპირობებულია იმით, რომ მცოცავ მძიმიან მონაცემებზე ოპერირებისას თითქმის ყოველთვის ხდება შედეგის დამრგვალება და ოპერციების სხვდასხვა თანმიმდევრობით ჩატარებისას დამრგვალებით შემოტანილი ცდომილება სხვადასხვა იქნება.

როგორც განხილული მაგალითიდან ჩანს, პროგრამაში არსებობს ოპერაციები, რომლებიც შეიძლება შესრულდნენ პარალელურად და არსებობს ოპერაციები, რომლებიც აუცილებლად მიმდევრობით უნდა შესრულდნენ. პროგრამაში არსებული ყველა ოპერაცია გავყოთ ორ ნაწილად – ის ოპერაციები, რომლებიც შეიძლება შესრულდნენ პარალელურად და ის ოპერაციები, რომლებიც უნდა შესრულდნენ მიმდევრობით. დავუშვათ, მიმდევრობითი ოპერაციების წილი არის f. ცხადია, $0 \le f \le 1$. თუ $f = 1$, ეს ნიშნავს, რომ პროგრამაში ყველა ოპერაცია უნდა შესრულდეს მიმდევრობით. ამის მაგალითია, უკვე განხილული მასივის ელემენტების ჯამის პოვნა ტრადიციული ალგორითმით. თუ $f = 0$, ეს ნიშნავს, რომ მთელი ალგორითმის გაპარალელება არის შესაძლებელი. ამის მაგალითად შეიძლება მივიყვანოთ ორი ვექტორის შეკრების ალგორითმის (ანუ, მოცემული ორი მასივიდან მივიღოთ მესამე მასივი, რომელშიც თითოეული ელემენტი არის პირველი ორი მასივის შესაბამისი ელემენტების ჯამის ტოლი. ცხადია, ამ ალგორითმში ყველა ელემენტი შეიძლება გამოითვალოს პარალელურად). ეს არის ზღვრული შემთხვევები. ალგორითმების უმრავლესობაში f მოთავსებულია 0-სა და 1-ს შორის. ზოგ შემთხვევაში, 0–თან უფრო ახლოსაა, ზოგ შემთხვევაში 1-თან ან სადღაც შუაშია. ცხადია, რაც უფრო ახლოს არის 0-ტან მით უკეთესია. ანუ, რაც უფრო ნაკლებია მიმდევრობით შესასრულებელი ოპერაციების წილი, მით უფრო უკეთეს შედეგს მივიღებთ აჩქარების თვალსაზრისით.

დავუშვათ, ერთ პროცესორზე პროგრამა სრულდება t დროში, ე.ი. $T_1 = t$. გამოვთვალოთ რა დრო დაჭირდება ამ პროგრამას p პროცესორიან გამოთვლით სისტემაზე შესრულებისთვის, თუ მიმდევრობითი ოპერაციების წილი არის f. T_1 წარმოვადგინოთ ასეთი სახით:

$$T_1 = ft + (1 - f)t$$

პირველი შესაკრები არის იმ ოპერაციების შესასრულებლად საჭირო დრო, რომლებიც უნდა შესრულდნენ მიმდევრობით და ეს დრო არ შეიცვლება იმ შემთხვევაშიც თუ გამოვიყენებთ p პროცესორიან გამოთვლით სისტემას. მეორე შესაკრები არის იმ ოპერციებისთვის საჭირო დრო, რომლებიც შეიძლება შესრულდეს პარალელურად. p პროცესორიან გამოთვლით სისტემაზე ეს დრო იქნება p-ჯერ ნაკლები (თუ უგულვებელყოფთ იმ დროს, რომელიც საჭიროა პარალელური გამოთვლების ორგანიზებისათვის, მონაცემთა პროცესორებს შორის განაწილებისთვის, ინფორმაციის გაცვლისთვის). ამრიგად, p პროცესორიან გამოთვლით სისტემაზე პროგრამის შესრულებისთვის საჭირო დრო იქნება

$$T_p = ft + \frac{(1 - f)t}{p}$$

აჩქარება კი ტოლი იქნება

$$\kappa = \frac{T_1}{T_p} = \frac{t}{ft + \frac{(1-f)t}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

თუ გავითვალისწინებთ პარალელური გამოთვლების ორგანიზებისათვის და მონაცემთა გადაცემისათვის საჭირო დროს, მაშინ რეალური აჩქარება ნაკლები იქნება მიღებულ შეფასებაზე:

$$\kappa < \frac{1}{f + \frac{1-f}{p}}$$

ეს არის **ამდალის კანონი**. იგი გვაძლევს პარალელური გამოთვლითი სისტემის გამოყენების შედეგად მიღებული მოსალოდნელი აჩქარების ზედა ზღვრის შეფასებას. კერძოდ, თუ პროგრამაში მიმდევრობით შესასრულებელი ოპერაციების წილი არის f $(0 \le f \le 1)$, მაშინ p პროცესორიან გამოთვლით სისტემაზე მისი შესრულებისას მოსალოდნელი აჩქარება ნაკლები იქნება $\frac{1}{f + \frac{1-f}{p}}$ -ზე.

განვიხილოთ მაგალითი. ვთქვათ, $f = 0{,}05$, ე.ი. მთელი პროგრამის მხოლოდ 5% არის მიმდევრობით შესასრულებელი, 95% შეიძლება შესრულდეს პარალელურად, რაც არცთუ ისე ურიგო მაჩვენებელი უნდა იყოს. ვნახოთ, რა მოსალოდნელი აჩქარება შეიძლება მივიღოთ 100 პროცესორიან გამოთვლით სისტემაზე შესრულებისას. ჩავსვათ ეს მნიშვნელობები $f = 0{,}05$ და $p = 1000$ ფორმულაში. მივიღებთ:

$$\kappa < \frac{1}{0{,}05 + 0.0095} = \frac{1}{0{,}0595} \approx 16{,}8$$

ე.ი. 100 პროცესორიან სისტემაზე პროგრამა შესრულდება მაქსიმუმ 16,8-ჯერ სწრაფად. თუ პროცესორების რაოდენობას 10-ჯერ გავზრდით შედეგი დიდად არ გაუმჯობესდება, ამ შემთხვევაში $\kappa = \frac{1}{0{,}05095} \approx 19{,}6$, ანუ 1000 პროცესორის გამოყენება გვაძლევს 16,8-ჯერ აჩქარებას. 10 პროცესორის შემთხვევაში კი გვექნებოდა $\kappa \approx 6{,}9$. როგორც ვხედავთ პროცესორების რაოდენობის მნიშვნელოვანი ზრდა ყოველთვის არ იწვევს აჩქარების მნიშვნელოვან ზრდას. მაგალითად, 100-დან 1000 პროცესორამდე (ანუ 10-ჯერ) გაზრდას მოჰყვა აჩქარების უმნიშვნელო გაზრდა, თუმცა 10-დან 100-მდე პროცესორების რაოდენობის გაზრდისის აჩქარება შედარებით საგრძნობლად გაიზარდა. თუ გავითვალისწინებთ დამატებით დროით დანახარჯებს, რომელიც საჭიროა პარალელური გამოთვლების ორგანიზებისთვის, მაშინ ზოგიერთ სიტუაციაში პროცესორების რაოდენობის ზრდამ შეძლება გამოიწვიოს არა პროგრამის აჩქარება, არამედ შენელება. ე.ი. ყოველი პროგრამისათვის არსებობს პროცესორების რაოდენობა, რომლის ზემოთაც პროცესორების რაოდენობის გაზრდა არაგონივრული იქნება (თუნდაც, ამის შედეგად არ მოხდეს პროგრამის შენელება. ჩვენს შემთხვევაში 100 პროცესორის მაგივრად 1000-ის გამოყენება ნამდვილად არაგონივრული იქნება).

ახლა წარმოვიდგიოთ, რომ გვაქვს პროცესორების უსასრულო რაოდენობა და ვნახოთ რა მაქსიმალური აჩქარება შეიძლება მივიღოთ ამ შემთხვევაში. ეს ნიშნავს, რომ უნდა ვიპოვით ამდალის კანონის შეფასების ზღვარი, როცა p მიისწრაფვის უსასრულობისკენ.

$$\kappa = \frac{1}{f + \frac{1-f}{p}} \xrightarrow[p\to\infty]{} \frac{1}{f}$$

რადგანაც p-ს მიმართ ფუნქცია არის ზრდადი, გვექნება

$$\kappa \le \frac{1}{f}$$

ანუ, პარალელური *პროგრამის აჩქარება ვერავითარ შემთხვევაში ვერ იქნება მეტი პროგრამაში მიმდევრობით შესასრულებელი ოპერაციების წილის შებრუნებულ რიცხვზე.* ეს არის **ამდალის კანონის შედეგი.**

1.5. ალგორითმის გაპარალელება

პარალელური გამოთვილითი სისტემის გამოყენების ძირითადი მიზანი არის პროგრამის უფრო სწრაფად შესრულება. რაც უფრო მეტ პროცესორს ვიყენებთ მით უფრო სწარფად შესრულების მოლოდინი უნდა გვქონდეს. მაგრამ ამდალის კანონი და მისი შედეგი გვიჩვენებს, რომ ყოველთვის ეს ასე არ არის. არსებობს ზღვარი, რომლის იქეთაც პროგრამის აჩქარება შეუძლებელია რაც არ უნდა ბევრი პროცესორი გამოვიყენოთ. ეს ზღვარი კი დამოკიდებულია მიმდევრობით შესასრულებელი ოპერაციების წილზე. რაც უფრო ნაკლებია ეს წილი, მით უფრო დიდია ეს ზღვარი ანუ მით უფრო მეტჯერ არის შესძლებელი პროგრამის აჩქარება პროცესორების საჭირო რაოდენობის არსებობის შემთხვევაში. სხვა სიტყვებით რომ ვთქვათ, ყველაფერი დამოკიდებულია ალგორითმის თვისებაზე – რამდენად გაპარალელებადია იგი. ტრადიციული ალგორითმების დიდი ნაწილი თავისი ბუნებით პარალელურია, ანუ ოპერაციების უდიდესი ნაწილი შეიძლება შესრულდეს პარალელურად. მაგრამ არსებობს არცთუ მცირე ნაწილი ალგორითმებისა, რომლებიც ცუდად პარალელდება ან საერთოდ არ პარალელდება. მაგალითად, თუ ალგორითმის ოპერაციების მხოლოდ ნახევარი შეიძლება შესრულდეს პარალელურად, იგი 2-ჯერ მეტადაც კი ვერ აჩქარდება. თუ ალგორითმის მხოლოდ 1% უნდა შესრულდეს მიმდევრობით მაშინ შეიძლება მივუახლოვდეთ 100-ჯერად აჩქარებას, თუ რა თქმა უნდა გვექნება შესაბამისი კომპიუტერული რესურსები. მაგრამ ეს შედეგიც შეიძლება არ იყოს დამაკმაყოფილებელი თუ გავითვალისწინებთ თანამედროვე მძლავრი სისტემების შესაძლებლობებს, რომლეთაც ათასობით პროცესორი გააჩნიათ. 100-ჯერადი აჩქარება, მაგალითად, 5000-პროცესორიან სისტემაზე გარკვეულ დაუკმაყოფილების გრძნობას აუცილებლად გამოიწვევს. როგორ უნდა მოვიქცეთ ასეთ შემთხვევაში? უნდა ვეცადოთ მიმდევრობით შესასრულებელი ოპერაციების რაოდენობა შევამციროთ. ამისთვის შეიძლება დაგვჭირდეს ალგორითმის მოდიფიცირება ან ახალი ალგორითმის მოფიქრება. შეიძლება აღმოჩნდეს ისეც, რომ ვერ შევძლოთ დახახული მიზნის მიღწევა.

დავუბრუნდეთ ჩვენს მიერ განხილულ მასივის ელემენტთა ჯამის პოვნის ალგორითმს. როგორც უკვე აღვნიშნეთ, მისი ტრადიციული ვარიანტი აბსოლუტურად მიმდევრობითია და მისი პარალელურ პროცესორებიან

სისტემაზე შესრულება ოდნავაც კი ვერ შეამცირებს შესრულების დროს. ქვემოთ მოყვანილია ამ ალგორითმის რეალიზაციის ფრაგმენტი:

```
S=0
for(i=0;i<n;i++)
      S+=a[i];
```

იმისათვის, რომ შევძლოთ მასივის ელემენტების ჯამი გამოვთვალოთ პარალელურ რეჟიმში, ალგორითმი უნდა შევცვალოთ – მასივი დავყოთ იმდენ ნაწილად რამდენი პროცესორიც გვაქვს, თითოეულმა პროცესორმა გამოთვალოს შესაბამისი ნაწილის ჯამი და შემდეგ ვიპოვოთ ამ ჯამების ჯამი. ქვემოთ მოყვანილია ფრაგმენტი, რომელიც რეალიზაციას უკეთებს ამ ალგორითმს. მასივი იყოფა p ნაწილად, თითიეულ ნაწილში გამოითვლება ელემენტების ჯამი და გროვდება locS[p] მასივში. შემდეგ კი გამოითვლება ამ მასივის ელემენტების ჯამი:

```
l=n/p;
for (j=0; j<p; j++){
      locS[j]=0;
      i_st=j*l;
      i_end=(j+1)*l;
      if(j==p-1) i_end=n;
      for(i=i_st;i<i_end;i++)
            locS[j]+=a[i];
}
S=0;
for(j=0;j<p;j++)
      S+=locS[j];
```

ცოტაოდენი განმარტება ამ ფრაგმენტის შესახებ.

ცვლადი i_st მიუთითებს j-ური ნაწილის დასაწყისის ინდექსს, - j-ური ნაწილის უკანასკნელ ინდექსს დამატებულ 1-ს. უკანასკნელი ნაწილისთვის ეს მნიშვნელობა იქნება n-ის ტოლი.

ასეთი პროგრამის გაპარალელების იდეა არცთუ ისე ძნელი მასახვედრია. გარე ციკლის j-ური იტერაცია უნდა შესრულდეს j-ურ პროცესორზე.

შემდეგი ნაბიჯები დამოკიდებულია იმაზე თუ რა ტიპის პარალელურ სისტემაზე ვაწარმოებთ გამოთვლებს.

1.6. გაპარალელება საერთო მეხსიერების მქონე პარალელური სისტემისთვის

შეგახსენებთ, რომ ასეთ სისტემებში მეხსიერება არის საერთო და ყველა პროცესორს აქვს წვდომა ნებისმიერ ცვლადზე. ამიტომ უნდა ვიზრუნოთ იმაზე, რომ პროცესორებმა ერთმანეთს არ გაუფუჭონ ცვლადის მნიშვნელობები. ამისათვის შემოვიღოთ საერთო და ლოკალური (კერძო) ცვლადის ცნება. საერთო არის ის ცვლადი, რომელთანაც წვდომა აქვს ყველა პროცესს, ლოკალური არის ის ცვლადი, რომელიც არსებობს იმდენ ეგზემპლარად, რამდენ პროცესსაც ვიყენებთ მოცემულ მომენტში. დავუბრუნდეთ ზემოთმოყვანილ ფრაგმენტს. j-ური იტერაცია ცვლის locS[j]–ის მნიშვნელობას, რომელიც სხვადასხვაა სხვადასხვა j-თვის. ამიტომ locS

შეიძლება იყოს საერთო ყველასთვის. სხვადასხვა პროცესორი მიმართავს ამ მასივის სხვადასხვა ელემენტს და ერთმანეთს ხელს ვერ შეუშლიან.

განსხვავებული სიტუაცია გვაქვს j, i_st, i_end, i ცვლადების შემთხვევაში. ყველა მათგანს j-ურ იტერაციაზე ენიჭება სხვადასხვა მნიშვნელობები. რაც ნიშნავს, რომ სხვადასხვა პროცესორები მათ მიანიჭებენ სხვადასხვა მნიშვნელობებს და თუ ისინი იქნებიან საერთო ცვლადები მათი მნიშვნელობები გაფუჭდება. ამიტომ ეს ცვლადები უნდა გამოვაცხადოთ კერძო ცვლადებად. ანუ თითოეულ პროცესორს ექნება თავისი j, i_st, i_end, i და იმუშავებენ მათ მნიშვნელობებთან.

რაც შეეხება l, n, p, a ცვლადებს, მათი მნიშვნელობები იმ ფრაგმენტში, რომლის გაპარალელებასაც ჩვენ ვაპირებთ, არ იცვლება. ამიტომ ისინი უნდა გამოვაცხადოთ საერთო ცვლადებად.

ზოგადი წესი შეიძლება ჩამოვაყალიბოთ შემდწგნაირად: საერთო ცვლადებად ვაცხადებთ იმ ცვლადებს, რომელთა მნიშვნელობა გასაპარალელებელ ფრაგმენტში არ იცვლება. თუ მასივის ელემენტის მნიშვნელობებს ცვლიან სხვადასხვა პროცესორები და ეს მასივი საჭიროა მომდევნო გამოთვლებში (გასაპარალელებელი უბნის შემდეგ) ეს მასივიც უნდა გამოცხადდეს საერთოდ. წინააღმდეგ შემთხვევაში ყველა პროცესორს ექნება მასივის თავის ეგზემპლარი და პარალელური უბნის დამთავრების შემდეგ მათი მნიშვნელობები დაიკარგება

დანარჩენი ცვლადები, რომელთა მნიშვნელობა იცველება სხვადასხვა პროცესორების მიერ, უნდა გამოცხადდნენ კერძო (ლოკალურ) ცვლადებად. ჯერჯერობით ჩვენ ვგულისხმობთ, რომ პროსცესორები ასეთ ცვლადებს ანიჭებენ მნიშვნელობებს ერთმანეთისგან დამოუკიდებლად, ანუ ერთი პროცესორის მიერ გამოთვლილი მნიშვნელობა არ გამოიყენება მეორე პროცესორის გამოთვლებში. თუ ეს ასე არ არის, მაშინ უნდა გამოვიყენოთ სხვა მექანიზმები, რომლებზეც მიმავალში გვექნება საუბარი.

2. მატრიცული გამოთვლების ზოგიერთი პარალელური ალგორითმი

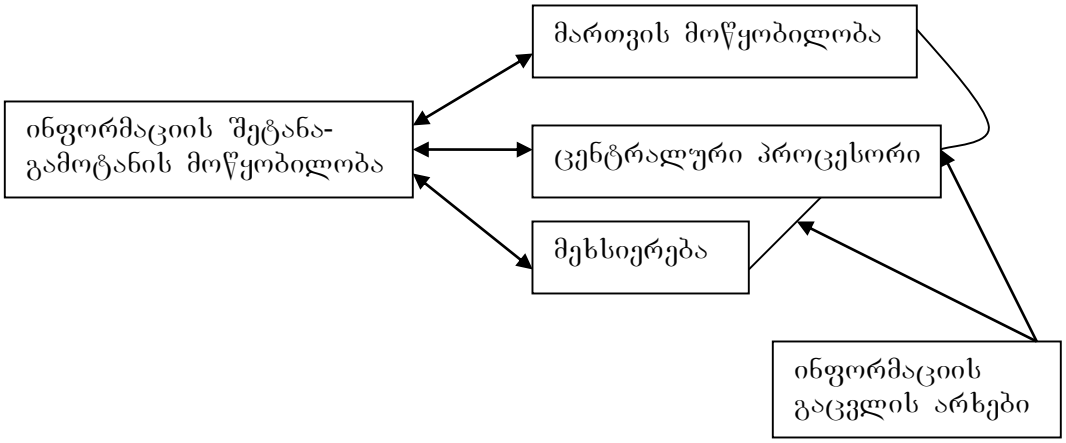
2.1. გამოთვლითი სისტემების არქიტექტურის შესახებ

1^0. ერთპროცესორიანი სისტემები.

ერთპროცესორიანი გამოთვლითი სისტემების არქიტექტურაში განიხილება შემდეგი მოწყობილობები:

- მართვის მოწყობილობა
- ცენტრალური პროცესორი
- მეხსიერება
- ინფორმაციის შეტანა-გამოტანის მოწყობილობა
- ინფორმაციის გაცვლის არხები

ამ მოწყობილობების ურთიერთქმედების მექანიზმი ნაჩვენებია შემდეგ ნახაზზე:



ერთპროცესორიანი გამოთვლითი სისტემის ფუნქციონირების პრინციპია ბრძანებების მიმდევრობითი შესრულება. ამოხსნის ალგორითმის აგების ძირითადი ამოცანაა ალგორითმის წარმოდგენა მიმდევრობითი ბრძანებების საშუალებით. ამ შემთხვევაში ოპტიმიზაციის მთავარი პრობლემა ოპერატორების რაოდენობისა და მეხსიერების საჭირო მოცულობის მინიმიზაციაა.

2^0. მრავალპროცესორიანი გამოთვლითი სისტემები.

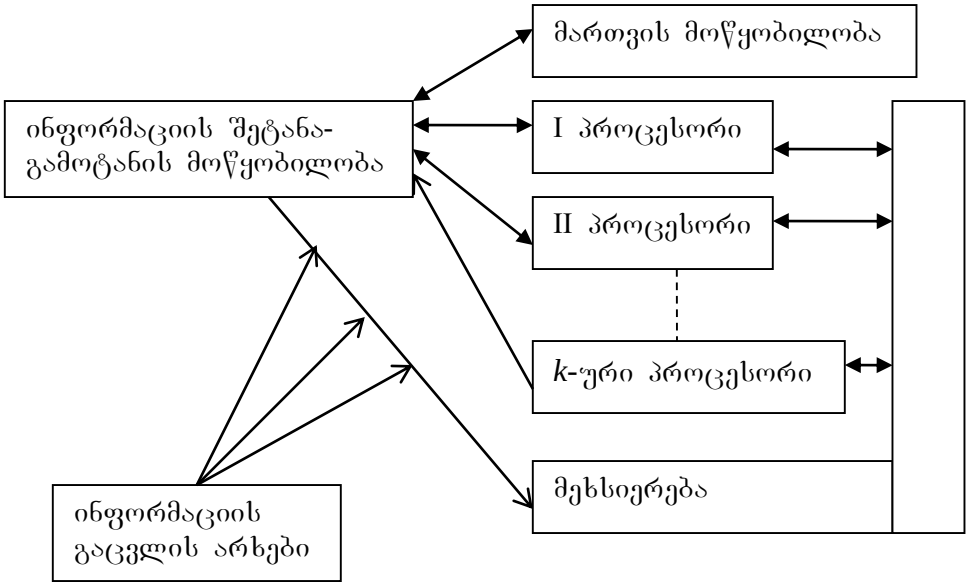
მრავალპროცესორიან გამოთვლით სისტემებს ფორმალურად აქვთ ერთპროცესორიანი სისტემების მსგავსი არქიტექტურა:

- მართვის მოწყობილობა
- პირველი პროცესორი
- მეორე პროცესორი

.....

- *k*-ური პროცესორი
- მეხსიერება
- ინფორმაციის შეტანა-გამოტანის მოწყობილობა
- ინფორმაციის გაცვლის არხები

ამ მოწყობილობების ურთიერთქმედების მექანიზმი ნაჩვენებია შემდეგ ნახაზზე:



ჩვენი სისტემის სუსტი ადგილი საკომუნიკაციო ქსელია (ინფორმაციის გაცვლის არხები). ქსელის სირთულე ჩვეულებრივ სწრაფად იზრდება პროცესორების ზრდასთან ერთად. ამჟამად მნიშვნელოვანი პრობლემაა მრავალპროცესორიანი გამოთვლითი სისტემების მოწყობილობებს შორის ეფექტური კავშირის დამყარება.

3^0. მრავალპროცესორიანი გამოთვლითი სისტემების გამოყენების სირთულეები.

ძირითადი სირთულე არის ის, რომ მრავალპროცესორიანი გამოთვლითი სისტემების მაღალი ეფექტურობა (რაც ახლოსაა მაქსიმალურთან) შეიძლება მიღწეული იქნას მხოლოდ მაშინ, როდესაც ყველა პროცესორი (ან მათი უდიდესი ნაწილი) მთლიანად დატვირთულია. შევნიშნოთ, რომ ერთპროცესორიან სისტემას (შესაბამისი პროგრამის შემთხვევაში) აქვს მაქსიმალური ეფექტურობა.

როდესაც ამოცანას მრავალპროცესორიან გამოთვლით სისტემაზე ვხსნით, ჩვეულებრივ საქმე გვაქვს შემდეგ პრობლემებთან:

– რომელი ალგორითმების რეალიზაციაა ეფექტურად შესაძლებელი მოცემულ მრავალპროცესორიან სისტემაზე.

– როგორი მრავალპროცესორიანი გამოთვლითი სისტემები უნდა შეიქმნას მოცემული კლასის ამოცანების ამოსახსნელად.

– როგორ ავაგოთ ისეთი ისეთი რიცხვითი მეთოდები, რომელთა რეალიზაციაც მოხერხებული იქნება მრავალპროცესორიან გამოთვლით სისტემაზე.

უნდა აღინიშნოს ისიც, რომ ყოველი ახალი სუპერკომპიუტერის შექმნის შემდეგ ამ სისტემის ეფექტური გამოყენებისთვის აუცილებელი ხდება უმოკლეს დროში ამ სუპერკომპიუტერისათვის მათემატიკური და პროგრამული უზრუნველყოფისა და შესაბამისი რიცხვითი ალგორითმების შექმნა, რათა შევძლოთ სულ უფრო და უფრო მაღალი სირთულის ამოცანების ამოხსნა.

4^0. ალგორითმის პარალელური ფორმა.

პარალელურ გამოთვლით მანქანაზე ალგორითმის რეალიზაციისთვის საჭიროა, ეს ალგორითმი წარმოვადგინოთ არა ცალკეული ოპერაციების, არამედ ოპერაციების ჯგუფების მიმდევრობის სახით. თითოეულ ჯგუფში ცალკეულ ოპერაციებს უნდა ჰქონდეთ შემდეგი თვისება: მათი შესრულება შესაძლებელი

უნდა იყოს ერთდროულად პარალელურ გამოთვლით სისტემაში არსებული ფუნქციონალური მოწყობილობების საშუალებით.

ამრიგად, ვთქვათ, ალგორითმში შემავალი ოპერაციები დაყოფილია ჯგუფებად, ხოლო ჯგუფების სიმრავლე ისეა დალაგებული. რომ თითოელი ჯგუფის ყოველი ოპერაცია დამოკიდებულია ან ამოცანის საწყის მონაცემებზე, ან ამ ჯგუფის წინა ჯგუფებში ჩატარებული ოპერაციების საშუალებით მიღებული შედეგებზე.

ასეთი სახით ალგორითმის წარმოდგენას **ალგორითმის პარალელურ ფორმას** უწოდებენ. ოპერაციების თითოეულ ჯგუფს უწოდებენ იარუსს, ხოლო ასეთი იარუსების რაოდენობას – პარალელური ფორმის სიმაღლეს. იარუსებში ოპერაციების მაქსიმალურ რაოდენობას უწოდებენ პარალელური ფორმის სიგანეს. შევნიშნოთ, რომ ერთსა და იმავე ალგორითმს შეიძლება ჰქონდეს რამდენიმე ფორმა. უპირატესობა ენიჭება მინიმალური სიმაღლის ფორმებს.

განვიხილოთ რამდენიმე მაგალითი.

მაგალითი 1. ვთქვათ, უნდა გამოვთვალოთ შემდეგი

$$(a_1a_2 + a_3a_4)(a_5a_6 + a_7a_8)$$

გამოსახულების მნიშვნელობა. ამასთან უნდა შევინარჩუნოთ მოქმედებების ჩატარების ის რიგი, რომელიც მოცემულია ჩანაწერში.

მოვიყვანოთ ამ გამოსახულების მნიშვნელობის გამოთვლის ალგორითმის რამდენიმე პარალელურ ფორმა.

1.1. პირველი ფორმა:

მონაცემები $a_1, a_2,\ a_3, a_4,\ a_5, a_6,\ a_7, a_8$

I იარუსი $a_1a_2,\ a_3a_4\quad a_5a_6,\ a_7a_8$

II იარუსი $a_1a_2 + a_3a_4\quad a_5a_6 + a_7a_8$

III იარუსი $(a_1a_2 + a_3a_4)(a_5a_6 + a_7a_8)$

ამ პარალელური ფორმის სიმაღლეა 3, ხოლო სიგანე 4.

ამ პარალელურ ფორმას ახასიათებს ის, რომ 4 პროცესორი დატვირთულია მხოლოდ პირველ ეტაპზე, ხოლო უკანასკნელ ეტაპზე დატვირთულია მხოლოდ ერთი პროცესორი.

1.2. მეორე პარალელურ ფორმას აქვს შემდეგი სახე:

მონაცემები $a_1, a_2,\ a_3, a_4,\ a_5, a_6,\ a_7, a_8$

I იარუსი $a_1a_2\quad a_3a_4$

II იარუსი $a_5a_6\quad a_7a_8$

III იარუსი $a_1a_2 + a_3a_4\quad a_5a_6 + a_7a_8$

IV იარუსი $(a_1a_2 + a_3a_4)(a_5a_6 + a_7a_8)$

1.3. მესამე პარალელურ ფორმას აქვს სახე:

მონაცემები $a_1, a_2,\ a_3, a_4,\ a_5, a_6,\ a_7, a_8$

I იარუსი $a_1 \cdot a_2\quad a_3 \cdot a_4$

II იარუსი $a_1 \cdot a_2 + a_3a_4\quad a_5 \cdot a_6$

III იარუსი a_7a_8

IV იარუსი $a_5a_6 + a_7 \cdot a_8$.

V იარუსი $(a_1a_2 + a_3a_4)(a_5a_6 + a_7a_8)$

ცხადია, რომ მეორე პარალელური ფორმის სიმაღლეა 4, მესამე პარალელური ფორმის კი 5, ორივე პარალელური ფორმის სიგანეა 2.

მაგალითი 2. გამოვთვალოთ n რიცხვის ნამრავლი. ვთქვათ, $n=8$. უნდა გადავამრავლოთ რიცხვები a_1, a_2, a_3, …, a_8

2.1. მიმდევრობითი გამრავლების ჩვეულებრივ სქემას აქვს სახე:

მონაცემები a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8

I იარუსი $a_1 \cdot a_2$

II იარუსი $(a_1 a_2)\cdot a_3$

……

VII იარუსი $(a_1 a_2 a_3 \cdots a_7)\cdot a_8$

ცხადია, რომ ამ ფორმის სიმაღლეა 7, ხოლო სიგანე კი 1.

2.2. იარუსების რაოდენობის შემცირება შესაძლებელია **დაწყვილების ალგორითმის** საშუალებით, რაც შემდეგში მდგომარეობს:

მონაცემები a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8

I იარუსი $a_1 \cdot a_2$, $a_3 \cdot a_4$, $a_5 \cdot a_6$, $a_7 \cdot a_8$

II იარუსი $(a_1 a_2)(a_3 a_4)$, $(a_5 a_6)(a_7 a_8)$

III იარუსი $(a_1 a_2 a_3 a_4)\cdot(a_5 a_6 a_7 a_8)$

ამ პარალელური ფორმის სიმაღლეა 3, ხოლო სიგანე კი 4. ასეთი ტიპის პროცესს **დაწყვილების სქემას** უწოდებენ. მისი რეალიზაციისთვის საჭიროა ყოველ იარუსზე განვახორციელოთ იმ რიცხვების წყვილების მაქსიმალურად შესაძლებელი გამრავლება, რომელიც მიღებულია წინა იარუსებზე ან მოცემულია საწყისი მონაცემების სახით.

მაგალითი 3. ვთქვათ, უნდა გამოვთვალოთ $I=\int_a^b f(x)dx$ ინტეგრალის მიახლოებითი მნიშვნელობა

$$I(N)=\sum_{i=1}^{N} c_i f(x_i)$$

კვადრატურული ფორმულის საშუალებით, სადაც x_1, x_2, …, x_N კვანძითი წერტილებია, $x_i \in [a,b]$, $i=1,2,\ldots,N$, ხოლო c_i კვადრატურული ფორმულის კოეფიციენტებია. ჩავთვალოთ, რომ კვადრატურული ფორმულის კოეფიციენტები ცნობილია.

ვთქვათ, $N=8$.

I იარუსი $f(x_1)$, $f(x_2)$, $f(x_3)$, $f(x_4)$, $f(x_5)$,
$f(x_6)$, $f(x_7)$, $f(x_8)$,

(ფუნქციის მნიშვნელობების გამოთვლა კვანძის წერტილებში)

II იარუსი $c_1 f(x_1)$, $c_2 f(x_2)$, $c_3 f(x_3)$, $c_4 f(x_4)$, $c_5 f(x_5)$,
$c_6 f(x_6)$, $c_7 f(x_7)$, $c_8 f(x_8)$,

III იარუსი $c_1 f(x_1)+c_2 f(x_2)$, $c_3 f(x_3)+c_4 f(x_4)$, $c_5 f(x_5)+c_6 f(x_6)$,
$c_7 f(x_7)+c_8 f(x_8)$,

IV იარუსი $\sum_{i=1}^{4} c_i f(x_i)$ $\sum_{i=5}^{8} c_i f(x_i)$

V იარუსი $\sum_{i=1}^{8} c_i f(x_i)$

ცხადია, რომ ამ ალგორითმის პარალელური ფორმის სიმაღლეა 5, ხოლო სიგანე 8. როგორც ვნახეთ, ეს პარალელური ალგორითმი დაწყვილების სქემას იყენებს.

5^0. შემოუსაზღვრელი პარალელიზმის კონცეფციის შესახებ.

შეუზღუდავი პარალელიზმის კონცეფცია გულისხმობს შემოუსაზღვრელი პარალელური გამოთვლითი სისტემის გამოყენებას.

სისტემას ეწოდება შემოუსაზღვრელი პარალელური გამოთვლითი სისტემა, თუ მას გააჩნია შემდეგი თვისებები:

1) სისტემა შეიცავს იდენტური პროცესორების ნებისმიერ რაოდენობას.

2) სისტემას აქვს ნებისმიერად დიდი მეხსიერება, რომელთანაც ერთდროულად აქვს წვდომა ყველა პროცესორს.

3) ოპერაციების წინასწარ მოცემული სიმრავლიდან ყოველ პროცესორს დროის დისკრეტულ ერთეულში შეუძლია შეასრულოს ნებისმიერი ოპერაცია. ამ სიმრავლის ოპერაციებს ძირითად ოპერაციებს ვუწოდებთ.

4) უგულებელყოფილია დამხმარე ოპერაციების (ე. ი. იმ ოპერაციების, რომლებიც ძირითადნი არ არიან) გამოთვლით სისტემაზე შესრულების დრო.

5) პროცესორების მეხსიერებასთან მიმართვის დროს არ წარმოიქმნება კომპლიქტება

6) სისტემაზე მუშაობის დაწყების წინ ყველა საჭირო მონაცემი ჩაწერილია მეხსიერებაში.

7) გამოთვლითი პროცესის დამთავრების შემდეგ ყველა მიღებული შედეგი მეხსიერებაში რჩება.

5^0. შეწყვილების სქემის შესახებ.

შემდგომში $[a]$ სიმრავლით აღვნიშნავთ a რიცხვის მთელ ნაწილს.

თეორემა 1. n რიცხვის მამრავლების შეწყვილების სქემის შესაბამისი პარალელური ფორმის სიმაღლეა $h = \left[\log_2^n\right] + 1$.

დამტკიცება. იმ შემთხვევაში, როდესაც n არ არის 2-ის ხარისხი, მოიძებნება k-ს ისეთი მნიშვნელობა, რომ სამართლიანი იქნება $2^{k-1} < n < 2^k$ უტოლობები. მოცემული რიცხვების ნამრავლს თანამამრავლებად დავუმატოთ $2^k - n$ რაოდენობის რიცხვი, რომელთაგან თითოეული 1-ის ტოლია. შემდეგ ავაგებთ შეწყვილების სქემის შესაბამის პარალელურ ფორმას და ამ ფორმით გამოვთვლით სიმაღლეს. შევნიშნოთ, რომ შემოუსაზღვრელი გამოთვლითი სისტემის შემთხვევაში ფორმის სიგანეს არსებითი მნიშვნელობა არ აქვს.

თეორემა დამტკიცებულია.

ანალოგიურად შეიძლება გამოვიყენოთ შეწყვილების სქემა n რიცხვის ჯამის პარალელური ფორმის აგებისას. მაშინ სამართლიანი იქნება შემდეგი თეორემა:

თეორემა 2. შეწყვილების სქემით n რიცხვის შეკრების პარალელური ფორმის სიმაღლეა $h = \left[\log_2^n\right] + 1$.

ამ თეორემის დამტკიცების მეთოდი ანალოგიურია თეორემა 1-ის დამტკიცების მეთოდისა (ამ შემთხვევაში, თუ n არ არის 2-ის ხარისხი, მაშინ ჯამს უნდა დავუმატოთ შესაბამისი რაოდენობის ნულოვანი შესაკრებები).

მაგალითი 4. ვთქვათ, უნდა გამოვთვალოთ n რიცხვის a_1, a_2, a_3, …, a_n ნამრავლი და აგრეთვე ყოველი მომდევნო

$$a_1,\ a_1 \cdot a_2,\ a_1 \cdot a_2 a_3,\ a_1 a_2 a_3 \cdots a_{n-1}$$

ნამრავლი.

შეწყვილების სქემის გამოყენებით ეს ამოცანა შეიძლება ძალიან ეფექტურად ამოიხსნას. მოვახდინოთ ამ ამოცანის ამოხსნის პოვნის დემონსტრაცია $n=8$ შემთხვევაში. ქვემოთ მოყვანილ სქემაში ყველა საძიებელი ნამრავლი აღნიშნულია ხაზის გასმით.

მონაცემები a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8

I იარუსი $\underline{a_1 \cdot a_2}$, $a_3 \cdot a_4$, $a_5 \cdot a_6$, $a_7 \cdot a_8$

II იარუსი $(a_1 a_2) a_3$, $\underline{(a_1 a_2)(a_3 a_4)}$, $(a_5 a_6) a_7$, $a_5 a_6 a_7 a_8$

III იარუსი $\underline{(a_1 a_2 a_3 a_4) a_5}$ $\underline{(a_1 a_2 a_3 a_4)(a_5 a_6)}$

$\underline{(a_1 a_2 a_3 a_4)(a_5 a_6 a_7)}$ $\underline{(a_1 a_2 a_3 a_4)(a_5 a_6 a_7 a_8)}$

ცხადია, რომ ამ პარალელური გამოთვლითი ფორმის სიმაღლე $h=5$, ხოლო სიგანე $w=4$. ამ ალგორითმის რეალიზაციისას გვაქვს ყველა პროცესორის სრული დატვირთვა. მაგრამ უნდა შევნიშნოთ, რომ ზოგიერთი შედეგი "ზედმეტია" იმ აზრით, რომ საბოლოო პასუხისთვის ისინი არ გვჭირდება (ეს სიტუაცია საკმარისად ტიპური სიტუაციაა პარალელური ალგორითმების რეალიზაციისას).

ცხადია, რომ ზემოაღნიშნული სქემის გამოყენება შეიძლება იმ შემთხვევაშიც, როდესაც n არ არის 2-ის ხარისხი. ამ შემთხვევაში მონაცემები უნდა შევცვალოთ საჭირო რაოდენობის 1-ით.

აღვნიშნოთ პარალელური ალგორითმების რამდენიმე მნიშვნელოვანი თვისება:

1. ბევრი პარალელური ალგორითმის რეალიზაციისას სრულდება დიდი რაოდენობის ზედმეტი ოპერაცია.

2. ერთი და იმავე ამოცანისთვის სხვადასხვა პარალელური ალგორითმების გამოყენებისას მიღებული შედეგები შეიძლება განსხვავებული იყოს ერთმანეთისგან ("მანქანური არითმეტიკისა" და მასთან დაკავშირებული დამრგვალებების ცდომილების გამო).

3. პროცესორების დიდი რაოდენობის შემთხვევაში პარალელური ალგორითმები უფრო ნაკლებად მდგრადია მიმდევრობით ალგორითმებთან შედარებით.

2.2. მატრიცული გამოთვლების ზოგიერთი პარალელური ალგორითმი შემოუსაზღვრელი პარალელიზმის შემთხვევაში

1^0. მატრიცის გამრავლება ვექტორზე

ვთქვათ, მოცემულია n-ური რიგის

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

კვადრატული მატრიცა და $\vec{x}=(x_1,x_2,\ldots,x_n)$ ვექტორი. უნდა გამოვთვალოთ $\vec{y}=A\vec{x}$ ვექტორის $\vec{y}=(y_1,y_2,\ldots,y_n)$ კოორდინატები:

$$y_i=\sum_{i=1}^{n} a_{ij}x_j\ ,\ \ i=1,2,\ldots,n\,.$$

n^2 რაოდენობის პროცესორის შემთხვევაში გამოვთვლით $a_{ij}x_j$ $(i=1,2,\ldots,n,\ j=1,2,\ldots,n)$ ნამრავლებს:

I იარუსი $a_{11}x_1$, $a_{12}x_2$, …, $a_{1n}x_n$, $a_{21}x_1$, $a_{22}x_2$, …, $a_{2n}x_n$,

…, $a_{n1}x_1$, $a_{n2}x_2$, …, $a_{nn}x_n$

შემდეგ გამოვთვლით $\sum_{j=1}^{n} a_{1j}x_j$, $\sum_{j=1}^{n} a_{2j}x_j$, …, $\sum_{j=1}^{n} a_{nj}x_j$ ჯამებს.

თითოეული ჯამისთვის გამოვიყენებთ შეწყვილების სქემას. ამრიგად, ამ ჯამების გამოსათვლელად საჭირო იქნება $\left[\log_2^n\right]+1$ იარუსი. ამრიგად, ამ ამოცანის ამოსახსნელად n^2 პროცესორის შემთხვევაში დაგვჭირდება $\left[\log_2^n\right]+2$ იარუსი.

თეორემა 1. შეუზღუდავი პარალელიზმის კონცეფციის შემთხვევაში $A(n\times n)$ მატრიცისა და $\vec{x}\in R^n$ ვექტორის გამრავლებისთვის შეიძლება აიგოს პარალელური ფორმა, რომლის სიმაღლეა $h=\left[\log_2^n\right]+2$, ხოლო სიგანე $w=n^2$.

2^0. მატრიცების გამრავლება

განვიხილოთ ორი

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix},\quad B=\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}.$$

n-განზომილებიანი კვადრატული მატრიცის გამრავლების ალგორითმი. B მატრიცი წარმოვადგინოთ შემდეგი სახით:

$$B=\left(\vec{b}_1,\vec{b}_2,\ldots,\vec{b}_n\right),$$

სადაც

$$\vec{b}_1=\begin{pmatrix} b_{11} \\ b_{21} \\ \cdots \\ b_{n1} \end{pmatrix},\ \vec{b}_2=\begin{pmatrix} b_{12} \\ b_{22} \\ \cdots \\ b_{n2} \end{pmatrix},\ \ldots,\ \vec{b}_n=\begin{pmatrix} b_{1n} \\ b_{2n} \\ \cdots \\ b_{nn} \end{pmatrix}$$

მაშინ მატრიცების AB ნამრავლი შეიძლება წარმოვადგინოთ შემდეგი სახით:

$$AB=\left(A\vec{b}_1,A\vec{b}_2,\ldots,A\vec{b}_n\right).$$

ამრიგად, მატრიცების ნამრავლი შეიძლება ჩაიწეროს როგორც პირველი მატრიცის ნამრავლი მეორე მატრიცის სვეტებზე (ასეთი სვეტების რაოდენობაა n). A

მატრიცის გამრავლება თითოეულ სვეტზე შეიძლება განვიხილოთ პარალელური გამოთვლითი ფორმის საშუალებით, რომლის სიმაღლეა $\left[\log_2^n\right]+2$, ხოლო სიგანეა n^2.

თუ გვექნება n^3 რაოდენობის პროცესორი, მაშინ A მატრიცის გამრავლება თითოეულ სვეტზე შეიძლება განვახორციელოთ პარალელურად. ამრიგად მივიღებთ მატრიცების გამრავლების პარალელურ ფორმას, რომლის სიმაღლეა $\left[\log_2^n\right]+2$, ხოლო სიგანეა n^3.

თეორემა 2. n-ური რიგის ორი კვადრატული მატრიცის გამრავლებისთვის შეიძლება აიგოს პარალელური გამოთვლითი ფორმა, რომლის სიმაღლე $h=\left[\log_2^n\right]+2$, ხოლო სიგანეა $w=n^3$.

3^0. ერთი რეკურსიული პროცესის პარალელური ფორმა.

განვიხილავთ ერთ რეკურსიულ პროცესს, რომელიც ხშირად გვხვდება სხვადასხვა ამოცანების განხილვისას.

ვთქვათ, r და S ფიქსირებული ნატურალური რიცხვებია, ხოლო

$$\vec{X}_0,\ \vec{X}_{-1},\ \ldots,\ \vec{X}_{-r+1} \tag{1}$$

მოცემული n-განზომილებიანი კვადრატული ვექტორებია. უნდა გამოვთვალოთ

$$\vec{X}_1,\ \vec{X}_2,\ \ldots,\ \vec{X}_S \tag{2}$$

ვექტორები შემდეგი

$$\vec{X}_i = A_{i1}\vec{X}_{i-1} + A_{2i}\vec{X}_2 + \cdots + A_{ir}\vec{X}_{i-r} + \vec{B}_i \tag{3}$$

რეკურენტული დამოკიდებულების საშუალებით, სადაც A_{ij} $(j=1,2,\ldots,r,\ i=1,2,\ldots,s)$ მოცემული n-ური რიგის კვადრატული მატრიცებია, ხოლო $\vec{B}_i$ – მოცემული ვექტორები.

ამრიგად, (2) ვექტორები გამოითვლებიან შემდეგი

$$\begin{aligned}
\vec{X}_1 &= A_{11}\vec{X}_0 + A_{12}\vec{X}_2 + \cdots + A_{1r}\vec{X}_{-r+1} + \vec{B}_1\\
\vec{X}_2 &= A_{21}\vec{X}_1 + A_{22}\vec{X}_0 + \cdots + A_{2r}\vec{X}_{-r+2} + \vec{B}_2\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
\vec{X}_s &= A_{S1}\vec{X}_{S-1} + A_{S2}\vec{X}_{r-2} + \cdots + A_{Sr}\vec{X}_{S-r} + \vec{B}_S
\end{aligned}$$

ტოლობების საშუალებით. შევნიშნოთ, რომ $A_{SS}\vec{X}_0$ შესაკრები გვხვდება იმ შემთხვევაში, თუ $S \le r$.

თეორემა 3. (2) ვექტორების კომპონენტების გამოსათვლელად არსებობს პარალელური ფორმა, რომლის სიმაღლეა $h=S\left\{\left[\log_2^n\right]+\left[\log_2(r+1)\right]+3\right\}$, ხოლო სიგანეა $w=n^2r$.

დამტკიცება. ჯერ დავაფიქსიროთ $i\in\{1,2,\ldots,S\}$ და განვიხილოთ

$$\vec{X}_i = A_{i1}\vec{X}_{i-1} + A_2\vec{X}_{i-2} + \cdots + A_{ir}\vec{X}_{i-r} + \vec{B}_i \tag{4}$$

ტოლობა, სადაც $\vec{X}_{i-1}$, $\vec{X}_{i-2}$, $\vec{X}_{i-r}$ უკვე გამოთვლილი და ამდენად ცნობილი ვექტორებია.

თითოეული $A_{ij}\vec{X}_{i-j}$ გამოსათვლელად თეორემა 1-ის თანახმად შეიძლება ავაგოთ პარალელური ფორმა, რომლის სიმაღლეა $\left[\log_2^n\right]+2$, ხოლო სიგანეა n^2.

რადგან ყველა ეს თანამამრავლი შეიძლება გამოვთვალოთ პარალელურად (ეს თანამამრავლები ერთმანეთისგან დამოუკიდებელია), ამიტომ საკმარისია პარალელური ფორმის სიგანე გავზარდოთ r-ჯერ, მაშინ ფორმის სიმაღლე უცვლელი დარჩება. (4) გამოსახულების გამოსათვლელად საკმარისია, შევკრიბოთ ვექტორები, რომელთა რაოდენობაა $(r+1)$, რაც მოითხოვს პარალელურ ფორმას, რომლის სიმაღლეა $[\log_2(r+1)]+1$, ხოლო სიგანეა $n\left\{\left[\frac{r+1}{2}\right]+1\right\}$. გამოთვლების ეს ეტაპი მხოლოდ დაამატებს პარალელური ფორმის იარუსებს, არ მოითხოვს სიგანის გაზრდას, რადგან $n\left\{\left[\frac{r+1}{2}\right]+1\right\}\le n^2 r$. ამრიგად, i-ურ ბიჯზე ($\vec{X}_i$ ვექტორის გამოსათვლელად) მივიღებთ პარალელური ფორმას, რომლის სიმაღლეა

$$h=\left\{\left[\log_2^n\right]+2+\left[\log_2(r+1)\right]+1\right\},$$

ხოლო სიგანეა $w=n^2 r$.

(4) ფორმულის საშუალებით გამოთვლები უნდა ჩატარდეს S-ჯერ. ყოველი $\vec{X}_i$ ვექტორი დამოკიდებულია ადრე გამოთვლილ ვექტორებზე (ან საწყის მონაცემებზე), ამიტომ იარუსების რაოდენობა n-ჯერ უნდა გავზარდოთ.

ამრიგად, (3) რეკურენტული სქემის რეალიზაციისთვის შეიძლება ავაგოთ პარალელური ფორმა, რომლის სიმაღლეა

$$h=S\left\{\left[\log_2^n\right]+\left[\log_2(r+1)\right]+3\right\},$$

ხოლო სიგანეა $w=n^2 r$.

თეორემა დამტკიცებულია.

შევნიშნოთ, რომ ამ ამოცანისთვის შეიძლება აიგოს პარალელური ფორმა, რომელსაც ექნება უფრო ნაკლები სიმაღლე და ამას შეიძლება მივაღწიოთ პარალელური ფორმის სიგანის გაზრდის ხარჯზე.

თეორემა 4. (2) ვექტორების კომპონენტების გამოსათვლელად არსებობს პარალელური ფორმა, რომლის სიმაღლეა

$$h=S\left\{\left[\log_2(S+1)\right]+1\right\}\cdot\left\{\left[\log_2(rn+1)\right]+2\right\}, \tag{5}$$

ხოლო სიგანეა

$$w=\left\{\left[\frac{S+1}{2}\right]+1\right\}(nr+1)^3. \tag{6}$$

დამტკიცება. (3) რეკურენტული დამოკიდებულებები წარმოვადგინოთ ექვივალენტური ფორმით, რისთვისაც გამოვიყენებთ

$$\begin{pmatrix}\vec{X}_i\\ \vec{X}_{i-1}\\ \vec{X}_{i-2}\\ \vdots\\ \vec{X}_{i-r+2}\\ \vec{X}_{i-r+1}\\ 1\end{pmatrix}=\begin{pmatrix}A_{i1} & A_{i2} & \cdots & A_{i\,r-2} & A_{i\,r-1} & A_{ir} & B_i\\ E & 0 & \cdots & 0 & 0 & 0 & 0\\ 0 & E & \cdots & 0 & 0 & 0 & 0\\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & E & 0 & 0 & 0\\ 0 & 0 & \cdots & 0 & E & 0 & 0\\ 0 & 0 & \cdots & 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\vec{X}_{i-1}\\ \vec{X}_{i-2}\\ \vec{X}_{i-3}\\ \vdots\\ \vec{X}_{i-r+1}\\ \vec{X}_{i-r}\\ 1\end{pmatrix} \tag{7}$$

ბლოკურ მატრიცებსა და ვექტორებს. იმისთვის, რომ დავრწმუნდეთ (3) და (7) ტოლობების ექვივალენტობაში, (7) ტოლობის მარჯვენა მხარეში ბლოკური მატ-

რიცა გავამრავლოთ ბლოკურ ვექტორზე, რის შედეგადაც მივიღებთ r ვექტორულ და ერთ (უკანასკნელ)

$$\vec{X}_i = A_{i1}\vec{X}_{i-1} + A_{i2}\vec{X}_{i-2} + \cdots + A_{i\,r-2}\vec{X}_{i-r+2} + A_{i\,r-1}\vec{X}_{i-r+2} + A_{ir}\vec{X}_{i-r} + B_i$$

$$\vec{X}_{i-1} = E\vec{X}_{i-1}$$

$$\vec{X}_{i-2} = E\vec{X}_{i-2}$$

$$\ldots\ldots\ldots\ldots$$

$$\vec{X}_{i-r+1} = E\vec{X}_{i-1+1}$$

$$1 = 1$$

სკალარულ ტოლობას.

ამ ტოლობებიდან ჩანს, რომ (7) ტოლობები (3) რეკურენტული დამოკიდებულებების ექვივალენტურია. შემოვიღოთ აღნიშვნა

$$Q_i = \begin{pmatrix} A_{i1} & A_{i2} & \cdots & A_{i\,r-2} & A_{i\,r-1} & A_{ir} & B_i \\ E & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & E & \cdots & 0 & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & E & 0 & 0 & 0 \\ 0 & 0 & \cdots & 0 & E & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \vec{y}^{\,i} = \begin{pmatrix} \vec{X}_i \\ \vec{X}_{i-1} \\ \vec{X}_{i-2} \\ \vdots \\ \vec{X}_{i-r+2} \\ \vec{X}_{i-r+1} \\ 1 \end{pmatrix}.$$

ცხადია, რომ Q არის კვადრატული მატრიცა, რომლის განზომილებაა $nr+1$.

ამ აღნიშვნის შემდეგ (7) ტოლობა მიიღებს სახეს:

$$\vec{y}_i = Q_i \vec{y}_{i1} \tag{8}$$

თუ გამოვიყენებთ (8) ტოლობას, მივიღებთ

$$\vec{y}_i = (Q_i Q_{i-1} \cdots Q_1)\vec{y}_0, \quad i = 1,2,\ldots,S.$$

კერძოდ,

$$\vec{y}_1 = Q_1\vec{y}_0$$

$$\vec{y}_2 = Q_2 Q_1 \vec{y}_0$$

$$\vec{y}_S = (Q_S Q_{S-1} \cdots Q_1)\vec{y}_0$$

გამოვიყენოთ შეწყვილების ალგორითმი, რომელიც გამოთვლის არა მარტო ყველა $(S+1)$ რაოდენობის მატრიცის ნამრავლს, არამედ ყოველი მომდევნო თანამამრავლების ნამრავლსაც (იხ. §1, მაგალითი 4). განვიხილოთ $S = 7$ შემთხვევა:

I მაკროიარუსი [ორი $(n \in nL)$ რიგის მატრიცის გამრავლება]

$$\underline{Q_1\vec{y}_0} \qquad Q_3Q_2 \qquad Q_5Q_4 \qquad Q_7Q_6$$

II მაკროიარუსი

$$Q_2(Q_1\vec{y}_0) \qquad \underline{Q_3Q_2(Q_1\vec{y}_0)} \qquad Q_6(Q_5Q_4) \qquad Q_7Q_6(Q_5Q_4)$$

III მაკროიარუსი

$$\underline{Q_4(Q_3Q_2Q_1\vec{y}_0)} \quad \underline{(Q_5Q_4)(Q_3Q_2Q_1\vec{y}_0)} \quad \underline{(Q_6Q_5Q_4)(Q_3Q_2Q_1\vec{y}_0)}$$

$$\underline{Q_7Q_6(Q_5Q_4Q_3Q_2Q_1\vec{y}_0)}$$

ცხადია, რომ ზემოთ აღწერილი სქემის გამოყენება შეიძლება S-ის ნებისმიერი მნიშვნელობის შემთხვევაში. ამ შემთხვევაში საჭირო იქნება $[\log_2(S+1)]+1$

მაკროკონტაქტი (მაკროიარუსი), თუ გამოვიყენებთ $\left[\frac{S+1}{2}\right]+1$ მაკროპროცესორს, რომელიც მაკროოპერაციის საშუალებით გადაამრავლებს ორ $(nr+1)$ რიგის მატრიცას, ე. ი. მაკროოპერაციას ვუწოდებთ ორი მატრიცის გამრავლების ალგორითმს.

თეორემა 2-ის თანახმად $(nr+1)$ რიგის ორი კვადრატული მატრიცის გადასამრავლებლად (ერთი მაკროოპერაციის განსახორციელებლად) შეიძლება ავაგოთ პარალელური ფორმა, რომლის სიმაღლეა $[\log_2(nr+1)]+2$, ხოლო სიგანეა $(nr+1)^3$. ამრიგად, მთლიანი ალგორითმის განსახორციელებლად აგებული იქნება პარალელური ფორმა, რომლის სიმაღლე იქნება

$$h=\{[\log_2(S+1)]+1\}\cdot\{[\log(nr+1)]+2\},$$

ხოლო სიგანეა

$$w=\left\{\left[\frac{S+1}{2}\right]+1\right\}(nr+1)^3.$$

თეორემა დამტკიცებულია.

შენიშვნა. (3) რეკურენტული დამოკიდებულებანი ფართოდ გამოიყენება რიცხვით ანალიზში იტერაციული პროცესების განხილვისას. მათთვის პარალელური ფორმა მიიღება (7) – (8) ტოლობების საშუალებით, რომლებიც (3) რეკურენტული დამოკიდებულებების ექვივალენტურია მხოლოდ ზუსტი გამოთვლების შემთხვევაში. მაგრამ გამოთვლების პროცესში წარმოშობილმა გამოთვლების ცდომილებებმა შეიძლება მნიშვნელოვნად იმოქმედონ საბოლოო შედეგზე გამოთვლითი პროცესის არამდგრადობის გამო. ამიტომ პარალელური ფორმების გამოყენებისას ამ გარემოებას განსაკუთრებული ყურადღება უნდა მიექცეს.

4^0. პარალელური ალგორითმები განტოლებათა სისტემებისთვის სამდიაგონალური მატრიცით.

ჩვენს მიერ განხილული იყო A მატრიცის LU დაშლის ალგორითმი (თავი IV, §).

ახლა განვიხილოთ სამდიაგონალური მატრიცა და წარმოვადგინოთ ის $A=LU$ ნამრავლის სახით:

$$A\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & \cdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & & a_{k-1,kr} & a_{k-1,knL} & a_{k-1,k} \\ & & & & & 0 & a_{k\,k-1} & a_{kk} \end{pmatrix}$$

ვგულისხმობთ, რომ A მატრიცის ყველა მთავარი მინორი განსხვავებულია 0-ისგან. A მატრიცის LU სახით გაშლის შედეგად მივიღებთ

$$A=BC,$$

სადაც

$$B=\begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ b_{21} & 1 & 0 & \dots & 0 & 0 \\ 0 & b_{32} & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & b_{k\,k-1} & 1 \end{pmatrix}$$

$$C=\begin{pmatrix} c_{11} & a_{12} & 0 & \dots & 0 & 0 \\ 0 & c_{12} & c_{23} & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & c_{k-1,\,k-1} & c_{\,k-1,k} \\ 0 & 0 & 0 & \dots & 0 & c_{k12} \end{pmatrix}$$

გამოვიყვანოთ ფორმულები B და C მატრიცების ელემენტებისთვის. გადმოცემის სიმარტივისთვის განვიხილოთ შემთხვევა, როდესაც $k=4$.

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$C=\begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ 0 & b_{32} & 1 & 0 \\ 0 & 0 & b_{43} & 1 \end{pmatrix} \quad C=\begin{pmatrix} c_{11} & c_{12} & 0 & 0 \\ 0 & c_{22} & c_{23} & 0 \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix}$$

$$AB=\begin{pmatrix} c_{11} & c_{12} & 0 & 0 \\ b_{21}c_{11} & b_{21}c_{12}+c_{22} & c_{23} & 0 \\ 0 & b_{32}c_{22} & b_{32}c_{23}+c_{33} & c_{34} \\ 0 & 0 & b_{43}c_{33} & b_{43}c_{34}+c_{44} \end{pmatrix}$$

$A=BC$ ტოლობის გათვალისწინებით მივიღებთ:

$$c_{11}=a_{11},\ \ c_{12}=a_{12},$$

$$b_{21}=\frac{a_{11}}{c_{11}},\ \ c_{22}=a_{22}-b_{21}c_{12},\ \ c_{23}=a_{23},$$

$$b_{32}=\frac{a_{32}}{c_{22}},\ \ c_{22}=a_{33}-b_{32}c_{23},\ \ c_{34}=a_{34},$$

$$b_{43}=\frac{a_{43}}{c_{33}},\ \ b_{44}=a_{44}-b_{43}c_{34}.$$

შეიძლება დამტკიცდეს, რომ ზოგად შემთხვევაში (A მატრიცის რიგია k) სამართლიანია შემდეგი ფორმულები:

$$c_{11}=a_{11}$$

$$c_{i-1,i}=a_{i-1,i},\ \ i=2,3,\dots,k, \tag{9}$$

$$b_{i-1,i}=\frac{a_{i-1,i}}{c_{i-1,i}},\ \ i=2,3,\dots,k, \tag{10}$$

$$c_{i,i} = a_{i,i} - b_{i,i-1}a_{i-1,i}\,,\ \ i = 2,3,\ldots,k\,, \tag{11}$$

(11) ტოლობის ორივე მხარე გავამრავლოთ $c_{i-1,i-1}$-ზე, მივიღებთ

$$c_{i,i}c_{i-1,i-1} = a_{i,i}c_{i-1,i-1} - b_{i,i-1}a_{i-1,i}c_{i-1,i-1}\,.$$

ამ უკანასკნელ ტოლობაში ვისარგებლოთ (10) ტოლობით:

$$c_{i,i}c_{i-1,i-1} = a_{i,i}c_{i-1,i-1} - a_{i,i-1}a_{i-1,i}\,.$$

შემოვიღოთ აღნიშვნა:

$$q_i = c_{i,i}c_{i-1,i-1}\cdots c_{2,2}c_{1,1}\,,\ \ i = 2,3,\ldots,k\,,$$

(12) ტოლობის ორივე მხარე გავამრავლოთ $q_{i-2} = c_{i,i}c_{i-1,i-1}\cdots c_{2,2}c_{1,1}$-ზე, როდესაც $i \geq 3$, მივიღებთ

$$c_{i,i}c_{i-1,i-1}q_{i-2} = a_{i,i}c_{i-1,i-1}q_{i-2} - a_{i,i-1}a_{i-1,i}q_{i-2}\,,\ \ i = 2,3,\ldots,k\,,$$

ე. ი.

$$q_i = a_{ii}q_{i-1} - a_{i-1,i}a_{i,i-1}q_{i-2}\,,\ \ i = 3,4,\ldots,k\,.$$

აღვნიშნოთ $i = j+2$, მაშინ მივიღებთ

$$q_{j+2} = a_{j+2,j+2}q_{j+1} - a_{j+1,j+2}a_{j+2,j+1}q_j\,,\ \ j = 1,2,\ldots,k-2\,. \tag{12}$$

თუ გავიხსენებთ (3) რეკურენტულ დამოკიდებულებას

$$\vec{X}_j = A_{j1}\vec{X}_{j-1}\cdots + A_{jr}\vec{X}_{j-r} + \vec{B}_j\,,\ \ j = 1,2,\ldots,S\,, \tag{3}$$

მაშინ (12) ტოლობა შეიძლება გადავწეროთ (3) სახით, თუ დავუშვებთ, რომ

$n = 1$, ე. ი. X_j, A_{Sj}, $\vec{B}_j$ რაიმე რიცხვებია

$$\vec{X}_j = q_{j+2}\,,\ \ A_{j1} = a_{j+1,j+2}\,,\ \ A_{j2} = -a_{j+1,j+2}\cdot a_{j+2,j+1}\,,\ \ \vec{B}_j = 0\,,$$

$$S = k-2\,,\ \ r = 2\,.$$

ჩვენს მიერ (3) რეკურენტული დამოკიდებულებისთვის აგებული აგებული იქნება პარალელური ფორმა, რომლის სიმაღლეა

$$h = \{[\log_2(S+1)]+1\}\{[\log_2(rn+1)]+2\}\,,$$

ხოლო სიგანეა

$$w = \left\{\left[\frac{S+1}{2}\right]+1\right\}(nr+1)^3\,.$$

თუ ამ გამოსახულებაში შევიტანთ $n = 1$, $S = k-2$, $r = 2$ და გავითვალისწინებთ, რომ

$$[\log_2(rn+1)]+2 = [\log_2 3]+1 = 3\,,$$

მაშინ დავინახავთ, რომ სამართლიანია შემდეგი თეორემა:

თეორემა 5. k-ური რიგის სამდიაგონალური მატრიცის LU დაშლისთვის არსებობს კვადრატული ფორმა, რომლის სიმაღლეა

$$h = 3\left\{\left[\log\frac{k-1}{2})\right]+1\right\}\,,$$

ხოლო სიგანეა

$$w = 27\left\{\left[\frac{k-1}{2}\right]+1\right\}\,.$$

ახლა განვიხილოთ პარალელური ალგორითმი განტოლებათა სისტემისთვის სამდიაგონალური მატრიცით.

თეორემა 6. $A\vec{x}=\vec{y}$ განტოლებათა სისტემის ამოხსნისთვის, თუ A k-ური რიგის სამდიაგონალური მატრიცაა, შეიძლება აიგოს პარალელური ფორმა, რომლის სიმაღლეა $0(\log_2 k)$, ხოლო სიგანეა $0(k)$.

დამტკიცება. ვთქვათ, აგებულია A მატრიცის LV დაშლა $A=BC$, მაშინ $A\vec{x}=\vec{y}$ განტოლებათა სისტემის ამონახსნი შეიძლება მიღებული იქნას ორი

$$B\vec{z}=\vec{y}\,,\ \ C\vec{x}=\vec{Z}$$

განტოლებათა სისტემის მიმდევრობითი ამოხსნით. შევნიშნოთ, რომ B და C ორდიაგონალური მატრიცებია.

ამ განტოლებათა სისტემების ამოხსნა დაიყვანება (3) რეკურენტულ დამოკიდებულებებამდე. მართლაც, პირველი სისტემიდან გვექნება

$$\begin{cases} Z_1=y_1 \\ b_{21}Z_1+Z_2=y_2 \\ b_{21}Z_2+Z_3=y_3 \\ \dots\dots\dots\dots\dots\dots \\ b_{ii-1}Z_{i-1}+Z_i=y_i \\ \dots\dots\dots\dots\dots\dots \\ b_{k\,k-1}Z_{k-1}+Z_k=y_k \end{cases}$$

საიდანაც მივიღებთ

$$\begin{cases} Z_1=y_1 \\ Z_2=y_2-b_{21}Z_1 \\ Z_3=y_3-b_{32}Z_2 \\ \dots\dots\dots\dots\dots\dots \\ Z_i=y_i-b_{i\,i-1}Z_{i-1} \\ \dots\dots\dots\dots\dots\dots \\ Z_k=y_k-b_{k,k-1}Z_{k-1} \end{cases}$$

გამოვიყენოთ (3) რეკურენტული პროცესი, სადაც $n=1$, $\vec{X}_0=0$, $\vec{X}_j=Z_j$, $B_j=y_j$, $j=1,2,\dots,k$, $S=k$, $r=1$, $A_{11}=0$, $A_{i1}=-b_{i,r-1}$, $i=2,3,\dots,k$.

თუ გამოვიყენებთ თეორემა 4-ს, მაშინ მივიღებთ, რომ Z_i $(i=1,2,\dots,k)$ რიცხვების მოსაძებნად შეიძლება ავაგოთ პარალელური ფორმა, რომლის სიმაღლეა $h=3\{[\log(r+1)]+1\}$, ხოლო სიგანეა $w=8\left\{\left[\dfrac{k+1}{2}\right]+1\right\}$.

ანალოგიური შედეგი მიიღება მეორე განტოლებისთვის. თუ გავითვალისწინებთ თეორემა 5-ს, მივიღებთ, რომ k-ური რიგის განტოლებათა სისტემის, რომლის მატრიცა სამდიაგონალურია, ამოხსნისთვის შეიძლება აიგოს პარალელური ფორმა, რომლის სიმაღლეა $0(\log_2 k)$, ხოლო სიგანეა $0(k)$.

თეორემა დამტკიცებულია.

3. MPI ტექნოლოგია

3.1. ძირითადი ცნებები

დღეისათვის ყველაზე უფრო გავრცელებულ პარალელური დაპროგრამების ტექნოლოგიას განაწილებულ მეხსიერებიანი გამოთვლითი სისტემებისთვის წარმოადგენს MPI. ასეთ ციცტემებში ძირითადი საშუალება პროცესებს შორის ურთიერთქმედებისა არის შეტყობინების გადაცემა. ეს ასახულია ტექნოლოგიის დასახელებაშიც - Message Passing Interface (შტყობინების გადაცემის ინტერფეისი) MPI-ის მხარდაჭერა აქვს დაპროგრამების ენა Fortran და C/C++-ის მისი სრული ვერსია შეიცავს 100-ზე მეტ ფუნქციას და პროცედურას. MPI უზრუნველყოფს პარალელური პროგრამის შექმნას MIMD (Multiple Instruction Multiple Data) სტილში, რაც ნიშნავს სხვადსხვა კოდის მქონე პროცესების გაერთიანებას. მაგრამ ასეთ სტილში პროგრამების წერა და გამართვა ძალიან ძნელია. ამიტომ პრაქტიკაში გამოიყენება SPMD მოდელი (Single Program Multiple Data), რომელშიც ყველა პარალელური პროცესი იყენებს ერთი და იმავე კოდს.

MPI წრმოადგენს ბიბლიოთეკას, ამიტომ კომპილირების დროს აუცილებელია მოერთებული იყოს შასაბამისი მოდულები. ეს შეიძლება გაკეთდეს სხვდასხვანაირად: ბრძანებათა ველში მითითებით, პროგრამულ გარსში შესაბამისი ოპციის განსაზღვრით ან გამოყენებული იქნეს სპეციალური სკრიპტი, მაგალითად, mpicc – C-ზე დაწერილი პროგრამის კომპილაციისათვის.

შემსრულებელი ფაილის მიღების შემდეგ იგი გაშვებული უნდა იყოს საჭირო რაოდენობის პროცესორებზე. ამისთვის, როგორც წესი, გამოიყენება MPI-პროგრამის გაშვებიც სკრიპტი mpirun, მაგალითად,

mpirun -np N <პროგრამა თავისი არგუმენტებით>

სადაც N არის პროცესების რაოდენობა. გაშვების შემდეგ ერთი და იგივე პროგრამა შესრულდება ყველა გაშვებული პროცესების მიერ და შედეგი გამოიტანება ეკრანზე ან ჩაიწერება აილში წინასწარ განსაზღვრული სახელით (ეს დამოკიდებულია სისტემაზე). ის, თუ რომელ კომპიუტერებზე უნდა იყოს გაშვებული პროგრამები მიეთითება ან წინასწარგანსაზღვრული ფაილიდან (მისი სახელი და მდებარეობა დამოკიდებულია სისტემაზე), ან ჩვენს მიერ შექმნილი ფაილიდან, რომელიც უნდა მიეთითოს ოპციაში -machinefile, მაგალითად,

mpirun -np 8 –machinefile hosts <პროგრამა თავისი არგუმენტებით>

სადაც hosts არის ფაილის სახელი, რომელშიც ჩამოთვლილი უნდა იყოს იმ კომპიუტერების IP მისამართები ან სახელები, რომელებზეც უნდა გაეშვას პროგრამა. (ცხადია, არ არის აუცილებელი ამ ფაილს ერქვას hosts, შეიძლება ნებისმიერი სხვა სახელი. შესაძლებელია აგრეთვე მითითებული იყოს გზა ამ ფაილამდე, თუ იგი არ არის მოთავსებული მიმდინარე დირექტორიაში). hosts-ფაილის მაგალითია:

```
10.0.0.1
10.0.0.2
10.0.0.3
```

ეს ნიშნავს, რომ 8 პროცესის შემთხვევაში პირველი გაუშვება კომიუტერზე ლოკალური IP–მისამართით 10.0.0.1, მეორე – 10.0.0.2–ზე, მესამე – 10.0.0.3–ზე, მე–4 – ისევ 10.0.0.1–ზე და ა.შ. ციკლურად, მე–8 პროცესი გაეშვება 10.0.0.2–ზე
hosts-ფაილს შეიძლება ჰქონდეს ასეთი სახეც:

node01
node02
node03

სადაც node01, node02 და node03 არის კომპიუტერების ქსელური სახელები.
MPI–ში შემოღებულ ყველა ობიექტს – პროცედურის სახელს, კონსტანტებს, წინასწარგანსაზღვრულ მონაცემთა ტიპებს და ა.შ. აქვთ პრეფიქსი MPI_. გარდა ამისა, C/C++-ში ფუნქციების სახელში პირველი სიმბოლო პრეფიქსის შემდეგ იწრება დიდი ასოთი (ზედა რეგისტრში), დანარჩენი პატარა ასოებით. ონსტანტებში ყველა ასო იწრება ზედა რეგისტრში. (ფორტრანში დიდ და პატარა ასოებს შორის განსხვავება არ არის).
MPI–ის გამოყენეის დროს პროგრამაში ჩართული უნდა იყოს ფაილი mpi.h (ფორტრანში – mpif.h).
პარალელური პროცესების ურთიერთქმედებისთვის შესაძლებელია შეიქმნას პროცესორთა ჯგუები, რომელიც შეთავაზებს მათ ურთიერთობის სპეციალურ გარემოს – კომუნიკატორს. პროცესება ურთიერთობა შეუძლიათ მხოლოდ რაიმე კომუნიკატორის შიგნით. ისინი შედიან ერთ ან რამდენიმე კომუნიკატორში. კომუნიკატორები შიძლება შედიოდნენ ერთი მეორეში, გადაიკვეტებოდნენ ან გადაიკვეთებოდნენ. შეტყობინებები, რომლებიც იგზავნებიან სხვადსხვა კომუნიკატორებში არ გადაიკვეთებიან და ხელს არ უშლიან ერთმანეთს. კომნიკატორის ტიპი ფორტრანში არის INTEGER, ხოლო C/C++-ში MPI_Comm.
პროგრამის მუშაობის დაწყებისას იგულისხმება, რომ ყველა შექმნილი პროცესი მუშაობს ერთ ყოვლისმომცველ კომუნიკატორში, რომელსაც ჰქვია MPI_COMMM_WORLD. ეს კომუნიკატორი არსებობს ყოველთვის და გამოიყენება MPI-პროგრამის ყველა პროცესის ურთიერთობისთვის. გარდა ამის, პროგრამის დაწყებისას იქმნება კომუნიკატორი MPI_COMM_SELF, რომელშიც შედის მხოლოდ ერთი მიმდინარე პროცესი, და კომუნიკატორი MPI_COMM_NULL, რომლეშიც არცერთი პროცესი არ შედის. პროცესებს შორის ურთიერთობა მიმდინარეობს ერთი კონკრეტული კომუნიკატორის შიგნით. შეტყობინებები, გადაცემული სხვდასხვა კომუნიკატორებში, არ უშლიან ერთმანეთს ხელს. მომხმარებელს აქვს შესაძლებლობა საჭიროებისამებრ შექმნას თავისი კომუნიკატორები. ამისთვის MPI-ში არსებობს სპეციალური ფუნქციები და პროცედურები.
MPI–პროგრამის თითოეულ პროცესს ყველა კომუნიკატორში, რომელშიც ის შედის აქვს უნიკალური ატრიბუტი – პროცესის ნომერი, რომელიც არის მთელი არაუარყოფითი რიცხვი. ამ ატრიბუტის საშუალებით ხორციელდება პროცესთა ურთიერთობის უდიდესი ნაწილი. ერთი კომუმიაკტორის შიგნით ყველა პროცესს აქვს განსხვავებული ნომერი, ნუმერიაცია იწყება 0–დან და მთავრდება p-1–ით, სადაც p არის კომუნკატორში პროცესების რაოდენობა. რადგანაც პროცესი ერთდროულად შეიძლება შედიოდეს სხვადასხვა კომუნიკატორში მისი ნომერი ერთ კომუნიკატორში შეიძლება განსხვავდებოდეს ნომრისგან მეორე კომუნიკატორში და ორ სხვადსხვა პროცესს

სხვადასხვა კომუნიატორში შეიძლება ჰქონდეს ერთიდა იგივე ნომერი. აქრდან გამომდინარე, პროცესის იდენტიფიცირებიათვის საჭიროა მისი ორი ძირითად ატრიბუტი – კომუმიკატორი და ნომერი კომუნიკატორში.
პროცესებს შორის ურთიერთობის ძირითადი მეთოდი არის მათ შორის შეტყობინებების ცხადი გაგზავნა. შეტყობინება – ეს არის გარკვეული ტიპის მონაცემთა ერთობლიობა. ოველ შეტყობინებას აქვს რამდენიმე ატრიბუტი, კერძოდ გამგზავი პროცესის ნომერი, მიმღები პროცესის ნომერი, შეტყობინების იდენტიფიკატორი (ტეგი) და სხვა. ერთ–ერთი მნიშვნელოვანი ატრიბუტია ტეგი. მისი საშუალებით მიმღებ პროცესს შეუძლია გაარჩიოს ორი შეტყობინაბე, რომელიც მიუვიდა მას ერთიდაიმავე პროცესიდან. თვითონ ტეგი არის მთელი არაუარყოფითი რიცხვი, რომელიც მდებარეობს დიაპაზონში 0–დან MPI_TAG_UP. ამასთან, გარანტირებულია, რომ MPI_TAG_UP არის არანაკლებ 32767–ისა. შეტყბინების ატრიბუტებთან სამუშაოდ C/C++–ში შემოღებულია სპეციალური სტრუქტურა (ფორტრანში – მასივი), რომლის ველებში (ელემენტებში) ჩაწერილია ატრიბუტების მნიშვნელობები.
C/C++–ზე MPI–ის ფუნქციების უმრავლესიბა აბრუნების წარმატებით შესრულების კოდს (ფორტრანში ეს ხდება პროცესურის ბოლო პარამეტრის საშუალებით). წარმატებული დამთავრების შემთხვევაში ბრუნდება მნიშვნელობა MPI_SUCCESS, წინააღმდეგ შემთხვევაში შეცდობის კოდი.

3.2. MPI–ის ძირითადი პროცედურები

ეს პროცესურები არ არიან დაკავშირებული მონაცემთა გადაგზავნასთან. პრაქტიკულად თითქმის ყველა ეს პროცედურა საჭიროა ყველა პარალელურ პროგრამაში

```
int MPI_Init( int *argc, char ***argv )
```

argc – მიმთითებელი არგუმენტების რაოდენობაზე
argv – მიმთითებელი არგუმენტების ვექტორზე.

argc და **argv** არის პროგრამის ბრძანებათა ველის არგუმენტები, რომლის საშუალებითაც პარალელურ პროცესებს გადაეცემა პროგრამის გაშვების პარამეტრები. ამ ფუნქციით ხდება პროგრამის პარალელური ნაწილის ინიციალიზაცია. MPI-ის ყველა სხვა პროცედურა გამოზახებული უნდა იყოს **MPI_Init**–ის შესრულების შემდეგ. პარალელური ნაწილის ინიციალიზაცია შეიძლება მოხდეს მხოლო ერთხელ.

```
int MPI_Finalize( void )
```

პროგრამის პარალელური ნაწილი დამთავრება. ამ ფუნქციის გამოძახების შემდეგ აკრძალულია მიმართვია რაიმე MPI–ის ფუნქციაზე, მათ შორის, **MPI_Init**-ზე. პროცესის ამ ფუნქციის გამოზახების მომენტისთვის ყველა მოქმედება, რომელიც დაკავშირებული პროცესის მონაწილებიასთან შეტყობინებების გაცვლაში უკვე დასრულებული უნდა იყოს.

უმარტივესი MPI-პროგრამა C-ზე გამოიყურება შემდეგნაირად

```
#include "mpi.h"
main(int argc, char **argv)
{
…
printf("Before MPI_INIT\n");
MPI_Init(&argc, &argv);
printf("Parallel section\n");

…
MPI_Finalize();
printf("After MPI_FINALIZE\n");
…
}
```

სტრიქონები **Before MPI_INIT** და **After MPI_FINALIZE** გამოიტანება ერთი პროცესის მიერ ან ყველა პროცესის მიერ. ეს დამოკიდებულია სისტემის რეალიზაციაზე. სტრიქონი **Parallel section** ყველა გაშვებული პროცესის მიერ. სხვდასხვა პროცესიდან სტრიქონის გამოტანის თანმიმდევრობა შეიძლება იყოს სხვადასხვა.

```
int MPI_Initialized( int *flag )
```

ეს ფუნქცია აბრუნებს არუმენტში flag მნიშვნელობას true, თუ იგი გამოძახებულია პარალელურ ნაწილში, წინააღმდეგ შემთხვევაში აბრუებს – false-ს. ეს არის ერთადერთი MPI-ის ფუნქცია, რომელიც შეიძლება გამოძახებული იყოს MPI_init–მდე.

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

ეს ფუნქცია პარამეტრ size–ში აბრუნებს comm კომუნიკატორში პროცესების რაოდენობას.

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

ეს ფუნქცია პარამეტრ `rank`–ში აბრუნებს `comm` კომუნიკატორში პროცესის ნომერს.

ვემოთმოყვანილ მაგალითში თითოეული გაშვებული პროცესი ბეჭდავს მის უნიკალურ ნომერს კომუნიკატორ **MPI_COMM_WORLD**–ში

```
#include "mpi.h"
main(int argc, char **argv)
{
Int size,rank;
…
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("process=%d  size=%d\n",rank,size);
…
MPI_Finalize();
```

…
}

სტრიქონი, რომელიც შეესაბამება **printf-**ს გამოტანება იმდენჯერ, რამდენი პროცესიც იყო შექმნილი პროგრამის გაშვებისას. თითოეული პროცესი გამოიტანს თავის შესაბამის სტრიქონს. სტრიქონების გამოტანის თანმიმდევრობა შეიძლება იყოს ნებისმიერი.

```
double MPI_Wtime( void )
```

ეს ფუნქცია აბრუნებს გამომზახებელ პროცესში ასტრონომიულ დროს წამებში (ორმაგი სიზუტის ნამდვილი რიცხვი), რომელიც გასულია წარსულში გარკეული მომენტიდან. პროგრამის რაიმე მონაკვეთი რომ მოვაქციოთ ორი ასეთი გამოძახებას შორის, დაბრუნებული მნიშვნელობების სხვაობა მოგვცემს მოცემული ფრაგმენტის მუშაობის დროს.

```
int MPI_Get_processor_name( char *name, int *resultlen )
```

ეს ფუნქცია არგუმენტში `name` აბრუნებს იმ კვანძის (კომპიუტერის) სახელს, რომელზეც გაშვებულია გამოძახებელი პროცესი. ცვლადში `resultlen` იწერება სახელის სიგრძე (მასში სიმბოლოების რაოდენობა). იგი არ არის კონსტანტა **MPI_MAX_PROCESSOR_NAME**–ზე მეტი. ამ ფუნქციის საშუალებით შეგვიძლია დავადგინოთ კონკრეტულად რომელ ფიზიკურ პროცესორებზე გაეშვა MPI-პროგრამის პროცესები.

ვემოთმოყვანილ პროგრამაში ნაჩვენებია `MPI_Get_processor_name`–ის გამოყენება. აგრეთვე ხდება განსაზღვრა იმისა თუ რა დრო სჭირდება `MPI_Wtime` ფუნქციის გამოძახებას.

```
#include "mpi.h"
main(int argc, char **argv)
{
int rank, i, len, NTIMES=100;
char name[MPI_MAX_PROCESSOR_NAME+1];
double time_start, time_finish;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(name, &len);
time_start = MPI_Wtime();
for(i=1, i<=NTIMES, i++)
     time_finish = Wtime();
printf("processor %s,  process=%d, rank=%d  time=%lf\n"
     ,name,rank,(time_finish-time_start)/NTIMES);
MPI_Finalize();
}
```

3.3. პროცესებს შორის შეტყობინებების მიღება/გადაცემა

პრაქტიკულად ყველა პროგრამა, რომელიც დაწრილია MPI ტექნოლოგიის გამოყენებით უნდა შეიცავდეს გაშვებულ პროცესებს შორის ურთიერთქმედების საშუალებებს. ასეთი ურთიერთქმდება MPI–ში ხდება შეტყობინებების გაგზავნით.
MPI-ში შეტყობინებების გაგზავნის ფუნქციები იყოფა ორ ჯგუფად. ერთ ჯგუფში შედიან ფუნქციები, რომლებიც განკუთვნილია ორ პროცეს შორის ურთერთქმედებისთვის. ასეთი ოპერაციებს ეწოდება წერტილოვანი (point-to-point) ოპერაციები. მეორე ჯგუში კი შედიან ფუნქციები, რომლებიც ითხოვენ რომ ოპერაციის შესრულებაში მონაწილებდეს კონმუნიკატორში შემავალი ყველა პროცესი. ასეთ ოპერაციებს ჰქვია კოლექტიური ოპერაციები.
განიხილოთ point-to-point ოპერაციები. მასში მონაწილებს ორი პროცესი. ერთი არის გამგზავნი, მეორე – მიმღები. გამგზავნმა პროცესმა უნდა გამოიძახოს მონაცემთა გადაცემის ერთ–ერთ ფუნქცია და მოუთითოს მიმღები პროცესის ნომერი რომელიმე კომუნიკატორში. მიმღებმა პროცესმა უნდა გამოიძახოს მონაცემთა მიღების ერთ–ერთ ფუნქცია და მიუთითოს გამომგზავნი პროცესის ნომერი იმავე კომუნიკატორში. ზოგიერთ შემთხვევაში დასაშვებია რომ მიმღებმა არ იცოდეს გამომგზავნის ნომერი და მოითხოვოს ნებისმიერი პროცესიდან გამოგზავნილი ინფორმაციის მიღება.
ამ ჯგუის ფუნქციაები ასევე იყოფიან ორ კლასად: ფუნქციები ბლოკირებით (სინქრონიზაციით) და ბლოკირების გარეშე (ასინქრონული). ბლოკირებადი ფუნქციები ბლოკავენ პროცესის მუშაობას გარკვეული პორობის შესრულებამდე. ასინქრონული ფუნქციიდან გამოსვლა კი ხდება მყისიერად, შესაბამისი კომუნიკაციური ოპერაციი ინიცალიზაციისთანავე. ბლოკირებადი ფუნქციები არაკორექტულმა გამოყენებამ შეიძლება მიგვიყვანოს ჩიხურ სიტუაციამდე, ამოტომ საჭიროა განსაკუთრებული ყურადღება. ასინქრონული ოპერაციების გამოყენეს ციხურ სიტუაციამდე არ მივყევართ, ითხოვს განსაკუთრებულ ყურადღება ს მასივებთან მუშაობის დროს.

3.3.1. ბლოკირებადი მიღება/გაგზავნის ფუნქციები

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
dest,
             int tag, MPI_Comm comm)
```

ეს არის ბლოკირებადი გაგზავნის ფუნქცია. მონაცემები (მასივი) buf, რომელიც შედგება count ცალი datatype ტიპის ელემენტისგან გაეგზავნება პროცესს, რომლის ნომერი comm კომუნიკატორში არის dest. გაგზავნილი შეტყობინების იდენტიფიკატორი არის tag. დასაგზავნი მონაცემები უნდა იყვენენ თანმიმდევრობით ჩალაგებულნი მასივში buf. გაგზავნის ოპერაცია იწყება იმისგან დამოუკიდებლად იყო თუ არა ინიცირებული მიმღებ პროცესზე მიღები ოპერაცია. ამასთან, შეტყობინება შეიძლება უშუალოდ კოპირებული იქნას მიღების ბუფერში, ასევე შესაძლებელია იგი მოთავსებული იყოს სისტემურ ბუფერში. Count–ის მნიშვნელობა შეიძლება იყოს 0–ის ტოლი. MPI–ის ზოგიერთ რეალიზაციაში დასაშვებია, რომ შეტყობინება გაეგზანოს საკუთარ თავს., მაგრამ ეს უსაფრთხო არ არის. ზოგიერთი რეალიზაცია კრძალავს საკუთარი თავისთვის

შეტყობინების გაგზავნას. პარამეტრი datatype–ის მნიშვნელობა უნდა მიეთითოს წინასწარგანსაზღვრული კონსტანტების საშუალებით. C-ის საბაზო ტიპებისთვის ამ კონტანტების მნიშვნელობა მიიღება შემდეგნაირად: საბაზო ტიპის დასახელებას წინ ეწერაბა პრფოქსი MPI_ და შემდეგ იწერება ტიპის დასახელება დიდი ასიებით. მაგალითად, თუ იგზავნება int ტიპის მონაცემები მას შეესაბამება MPI_INT, float–ს შეესაბამება MPI_FLOAT, char – MPI_CHAR, double – MPI_DOUBLE და ასე შემდეგ. კონსტანტა MPI_BYTE გამოიყენება არატიპიზირებული მონაცემთა გადაცემისთვის. შეტყობინების გაგზავნისას არარსებული პროცესისთვის შეიძლება გამოვიყენოთ მნიშვნელობა **MPI_PROC_NULL.** ოპერაცია ასეთი პროცესით მთავრდება მომენტალურად დასრულების კოდით **MPI_SUCCESS**. მაგალითად, ერთით მეტი ნომრის მქონე პროცესისთვის შეტყობინების გასაგზავნად შეიძლება ვისარგებლოთ შემდეგი ფრაგმენტით:

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
next = rank+1
if(next == size) next = MPI_PROC_NULL;
MPI_Send(&n, 1, MPI_FLOAT, next, 5, MPI_COMM_WORLD);
```

ამ მაგალითში უკანასკნელი ნომრის მქონე პროცესი არავითარ რეალურ გაგზავნას არ განახორციელებს, რადგანაც იგი უგზავნის არარსებულ პროცესს.
ბლოკირება იძლევა იმის გარანტიას, რომ ფუნქციის პარამეტრები შეიძლება უპრობლემოდ გამოყენებული იქნას ფუნქციიდან გამოსვლის შემდეგ, ყოველგვარი შიშის გარეშე იმისა, რომ შეიძლება გავაფუჭოთ გასაგზავნი ინფორმაცია.. ეს ნიშნავს, რომ გასაგზავნი შეტყობინება უკვე გაგზავნილია, თუმცა გარანტია იმისა, რომ მიღებულია მიმღები პროცესის მიერ არ გვაქვს. არც იმის გარანტია გვაქვს, რომ შეტყობინაბამ დატოვა ის პროცესორული ელემენტი, რომლეზეც იყო პროცესი. ასეთი გაურკვევლობა შეიძლება ყოველთვის არ აწყობდეს მომხმარებელს. ამიტომ MPI-ში შემოღებულია დამატებით კიდევ სამი ფუნქცია. მათ ზუსტად ისეთივე პარამეტრები აქვთ როგორც MPI_Send–ს.

MPI-ში არის შემდეგი 3 მოდიფიკაცია ბლოკირებადი გაგზავნის ფუნქციისა:

- MPI_Bsend - მონაცემების გადაცემა ბუფერიზაციით. თუ შეტყობინების მიღება არ იყო ინიცირებული მიმღებ პროცესზე, შეტყობინე ჩაიწერება სპეციალურ ბუფერში და მოხდეა დაუყონებლივი გამოსვლა ფუნქციიდან. ამ ფუნქციის შესრულება არანაირად არ არის დამოკიდებული შესაბამისი მიღების ფუნქციის გამოძახებისგან. ფუნქციამ შეიძლება დააბრუნოს შეცდომის კოდი, თუ ადგილი ბუფერისთვის არ არის საკმარისი. ბუფერიზაციისთვის მასივის გამოყოფაზე უნდა იზრუნოს მომხმარებელმა.
- MPI_Ssend - შეტყობინების გადაცემა სინქრონიზაციით. ამ ფუნქციიდან დაბრუნება ხდბა მხოლოდ იმ შემთხვევაში, როცა მიმღები პროცესის მიერ იქნება ინიცირებული მირები ფუნქცია. ამ ფუნქციის დასრულება მეტყველებს არამარტო იმაზე, რომ გაგზავნის ბუფერის ხელმეორედ გამოყენება შესაძლებელია, არამედ

იმაზე, რომ მიმღებმა პროცესმა გარანტირებულად მიაღწია იმ ადგილს პროგრამაში, სადაც უნდა მოხდეს ამ შეტყობინაბის მიღება. სინქრონიზაციით მონაცემების გადაცემამ შეიძლება შეანელოს პროგრამის შესრულება, მაგრამ შეიძლება თავიდან აგვაცილოს ბევრი არმიღებული ბუფერიზებული შეტყობინების მიღება.

- MPI_Rsend – მონაცემების გადაცემა მზადყოფნით. ამ ფუნქციით სარგებლობა შეიძლება მხოლოდ იმ შემთხვევაში, როდესაც მიმღებმა პროცესმა უკვა მოახდინე მიღების ინიციალიზაცია. წინააღმდეგ შემთხვევაში ამ ფუნქციის გამოყენება მცდარია და მისი შედეგი განუსაზღვრელია. მიღების ინიციალიზაციის გარატირება MPI_Rsend–ის გამოზაების წინ შესაძლებელია იმ ფუნქციების გამოყენებით, რომლებიც ახდენენ ცხად ან არაცხად სინქრონიზაციას (მაგალიად, MPI_Barrier ან MPI_Ssend). MPI-ის ბევრ რეალიზაციებში MPI_Rsend ამცირებს გამგზავნსა და მიმღებს შორის პროტოკოლს და ამით ამცირებს მონაცემთა გადაცემის ორგანიზაციისთვის საჭირო დამატებთ ხარჯებს

თუ მომხამრებელი იყენებს ფუნქციას MPI_Bsend, მან უნდა შექმნას გამგზავნ პროცესზე სპეციალური მასივი და გამოზაცხადოს იგი ბუფერიზაციის მასივად. გაოცხადება ხდება ფუნქციით

```
int MPI_Buffer_attach(void *buffer, int size)
```

`buffer` – არის მასივის მისამართი

`size` – მასივის ზომა (ბაიტებში)

თითოეულ პროცესში ასეთი ბუფერი შეიძლება იყოს მხოლოდ ერთი. ბუფერთან ასოცირებული ეს მასივი არ უნდა იყოს გამოყენებული სხვა მიზანებისთვის. ბუფერიზაციისთვის გამოყოფილი მასივის ზომა უნდა იყოს გასაგზავნი შეტყობინების მაქსიმალურ ზომაზე სულ მცირე MPI_BSEND_OVERHEAD კონსტანტით განსაზღვრული სიდიდით.

ბუფერის გათავისუფლება ხდება ფუნქციით

```
int MPI_Buffer_detach(void *buffer, int *size)
```

buffer და size, ამ შემთხვევაში არიან გამომავალი პარამეტრები. Buffer–ში ბრუნდება გათავისებული ბუფერის მისამართი, size–ში იწერება გათავისუფლებული ბუფერის სიგრძე (ბაიტებში). ფუნქციის გამომძახებელი პროცესი იბლოკება მანამ, სანამ მასში მოთავსებული ყველა შეტყობინება არ წავა.
როგორც წესი, MPI-ში გამოიყოფა მეხსიერების გარკვეული ოდენობა გასაგზავნი შეტყობინებების ბუფერიზაციისთვის. ზოგიერთ შემთხვევაში ეს ბუფერი შეიძლება არ იყოს საკმარისი გადასაგზავნი ინფორმაციისთვის. ამიტომ, რომ არ ვიყოთ დამოკიდებული ბიბლიოთეკის კონკრეტული რეალიზაციის პარამეტრებზე შესაძლებელია პროგრამაში გამოიყოს გადაგზავნებისათვის საკმარისი სიდიდის ბუფერი.

მომდევნო მაგალითში გამოიყენება გაგზავნა ბუფერიზაციით. ბუფერიზაციისთვის იქმნება მასივი **buf**, გაგზანის დასრილების შემდეგ იგი თავისუფლდება. ბუფერის ზომა განისაზღვრება გადასაცემი შეტყობინების ზომას (აერთი მთელი რიცხვი – 4 ბაიტი) პლუს **MPI_BSEND_OVERHEAD.**

```
#include <mpi.h>
#include <malloc.h>

int main (int argc, char **argv) {
const int BUFSIZE = 4 + MPI_BSEND_OVERHEAD;
char *buf;
int rank, ierr, ibufsize, rbuf;
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0) {
buf = (unsigned char *)malloc(BUFSIZE);
MPI_Buffer_attach(buf, BUFSIZE);
MPI_Bsend(&rank, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
     MPI_Buffer_detach(buf, &ibufsize);
free(buf);
}
if(rank == 1) {
MPI_Recv(&rbuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD,     &status);
printf("Process 1 received %d from process %d\n",
           rbuf, status.MPI_SOURCE);
}
MPI_Finalize();
return 0;
}
```

შერტყობინების მიღების ბლოკირებული ფუნქცია არის:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
           source, int     tag, MPI_Comm comm, MPI_Status *status)
```

ეს ფუნქცია მასივი buf–ში ჩაწერს datatype ტიპის არაუმეტეს count ელემენტს გამოგზავნილი შეტყობინებისა, რომლის ნომერია tag და გამოგზავნილია comm კომუნიკატორში შემავალი პროცესიდან ნომრით source. ამასთან შეივსება MPI_Status ტიპის სტრუქტურა მისამრთით status. თუ რეალურად მიღებული ელემენტების რაოდენობა ნაკლებია count–ზე ველში buf შეიცველება მხოლოდ ის ელემენტები, რომლებიც შეესაბამებიან მიღებული შეტყობინების ელემენტებს. თუ რეალურად მიღებული ელემენტების რაოდენობა მეტია count–ზე, მაშინ წარმოიშობა გადავსების შეცდომა. გადავსება თავი რომ ავარიდოთ შეიძლება ჯერ განვსაზღვროთ გამოგზავნილი შეტყობინების სტრუქტურა ფუნქციით MPI_Probe (ან MPI_Iprobe), თუ გვინდა გავიდოთ გამოგზავნილი ელემენტების ზუსტი რაოდენობა უნდა ვისარგებლოთ ფუნქციით

MPI_Get_count. ფუნქციის ბლოკირებულობა ნიშნავს, რომ ფუნქციიდან გამოცვლის შემდეგ მისაღები შეტყობინების ყველა ელემენტი მიღებული იქნება მისაღებ ველში buf. ქვემოთ მოყვანილ პროგრამაში ნულოვანი პროცესი უგზავნის შეტყობინებას პროცესს ნომრით 1 და ელოდება მისგან პასუხს. თუ პროგრამა გაშვებული იქნება 2–ზე მეტი პროცესით, რეალურად გადაგზავნებს მაინც განახორციელებებ მხოლოდ ნულივანი და პეორველი პროცესები. დანარჩენი პროცესები გამოიტანენ **a** და **b** ცვლადების საწყის მნიშვნელობებს.

```
#include <mpi.h>
#include <malloc.h>

int main (int argc, char **argv) {
int ierr, size, rank;
float a, b;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
a = 0.0; b = 0.0;
if(rank == 0) {
b = 1.0;
MPI_Send(&b, 1, MPI_FLOAT, 1, 5,MPI_COMM_WORLD);
     MPI_Recv(&a, 1, MPI_FLOAT, 1, 5,MPI_COMM_WORLD, &status);
}
else {
if(rank == 1) {
a = 2.0;
MPI_Recv(&b,1, MPI_FLOAT,0,5,MPI_COMM_WORLD, &status);
MPI_Send(&a, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
}
}
printf("process  %d   a = %f   b = %f\n", rank, a, b);
MPI_Finalize();
return 0;
}
```

მომდევნო მაგალითში ლუწნომრიანი პროცესები მომდევნო ნომრის მქონე პროცესს უგზავნიან შეტყობინებას. დამატებითად მოწმდება უმაღლესი ნომრის მქონე პროცესის შემთხვევა, რომ მან არ გაუგზავნოს შეტყობინაბე არ არსებულ პროცესს. **b** ცვლადის მნშვნელობა იცვლება მხოლოდ კენტნომრიან პროცესებზე.

```
#include <mpi.h>

int main1 (int argc, char **argv) {
int ierr, size, rank, a, b;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
a = rank;
b = -1;
if(rank%2 == 0) {
      if(rank+1 < size) {
            MPI_Send(&a,1,MPI_INT,rank+1,5,MPI_COMM_WORLD);
```

```
        }
}
else {
        MPI_Recv(&b, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD, &status);
}

printf("process  %d   a = %f   b = %f\n", rank, a, b);
MPI_Finalize();
return 0;
}
```

შეტყობინების მიღების დროს არგუმენტების source და tag ადგილას შეიძლება გამოყენებულ იქნას შემდეგი კონსტანტები:

- **MPI_ANY_SOURCE** - თუ გვინდა, რომ ნებისმიერი პროცესიდან გამოგზავნილი შეტყობინება მივიღოთ.
- **MPI_ANY_TAG** – თუ გვინდა, რომ ნებისმიერი ნომრით (ტეგით) გამოგზავნილი შეტყობინება მივიღოთ.

ორივე ამ კონსტანტის ერთდროული გამოყენებისას მიღებული იქნება ნებისმიერი პროცესიდან ნებისმიერი ნომრით გამოგზავნილი შეტყობინება. მიღებული შეტყობინების რეალური ატრიბუტები ყოველთვის შეიძლება განსაზღვრული იქნეს MPI_Status ტიპის სტრუქტურა status–დან. მას აქვს ველები **MPI_SOURCE**, **MPI_TAG** და **MPI_ERROR**, რომლებიც შესაბამისად განსაზღვრავენ გამგზავნი პროცესის ნომერს, შეტყობინების ნომერს (ტეგს) და შეცდომის კოდს.
თუ მისაღები შეტყობინების ატრიბუტებს (გამომგზავნი პროცესის ნომერი, შეტყობინების ნომერი) შეესაბამება რამდენიმე გამოგზავნილი შეტყობინება ერთი მიღების ფუნქციით ერთ–ერთი მათგანის მიღება მოხდება.
ყურადღება მივაქციოთ მიღების და გაზავნის ოპერაციების არასიმეტრიულობას. **MPI_ANY_SOURCE** კონსტანტის საშუალებით შეიძლება მივიღოთ შეტყობინება ნებისმიერი პროცესიდან, ხოლო გაგზავნის შემთხვევაში უნდა მივუთითოთ კონკრეტულად მიმღები პროცესის ნომერი.
MPI-ის სტანტარტში ნათქვამია, რომ თუ ერთი პროცესი ზედიზედ უგზავნის მეორე პროცესს ორ შეტყობინებას, რომელიც შეესაბამება ერთი და იმავე `MPI_Recv`–ს, პირველად მიღებული იქნება შეტყობინება, რომელიც გაგზავნილი იყო ადრე. ამასთან, თუ ორი შეტყობინება ერთდროულად იყო გაგზავნილი სხვადასხვა პროცესის მიერ, მათი მიღების რიგი წინასწარგანსაზღვრული არ არის.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int
*count )
```

ეს ფუნქცია status არგუმენტის მიხედვით განსაზღვრავს datatype ტიპის მიღებული (MPI_Recv ფუნქციის შემდეგ) ან მისაღები (MPI_Probe ფუნქციის შემდეგ) ელემნტების რაოდენობა და დააბრუნებს მას count არგუმენტის საშუალებით. ეს ფუნქცია, კერძოდ, საჭიროა იმის დასადგენად თუ რა ოდენობის ველი უნდა გამოიყოს მისაღები ინფორმაციის შესანახად.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status
*status)
```

ეს ფუნქცია MPI_Status ტიპის სტრუქტურაში (ცვლადში status) ჩაწერს პროცესი source – დან tag იდენტიფიკატორის (ტეგის) მქონე მისაღები შეტყობინების სტრუქტურის შესახებ ინფორმაციას. ეს ფუნქცია არის ბლოკირეული. ამ უნქციიდან დაბრუნება არ მოხდება მანამ, სანამ შეტყობინება შესაბამის ნომრით და შესაბამის გამომგზავნით არ იქნება ხელმისაწვდომი მიღებისთვის. ურადღება უნდა მივაქციოთ იმ ფაქტს, რომ ეს პროცედურა განსაზღვრავს მხოლოდ შეტყობინების მოსვლის ფაქტს, თავად შეტყობინებას კი ის არ იღებს. თუ ამ ფუნქციის შემდეგ გამოიძახება ფუნქცია `MPI_Recv` იგივე არგუმენტებით მიღებული იქნება იგივე შეტყობიება, რომლის შესახება ინფორმაცია მირებული იქნა MPI_Probe–ის საშუალებით.

შემდეგ მაგალითში ნაჩვენებია MPI_Probe–ის გამოყენების შესაძლებლობა მისაღები ინფორმაციის სტრუქტურის დასადგენად. პროცესი 0 ელოდება შეტყობინებას პროცესებიდან 1 ან 2 ერთიდაიმავე ტეგით. მაგრამ გამოგზავნილ მონაცემები არიან სხვადასხვა ტიპის. იმისათვის რომ განვსაზღვროთ რომელ ცვლადში ჩავწეროთ მისაღები შეტყობინება MPI_Probe–ის საშუალებით ვადგენთ ვიგან მოვიდა შეტყობილება. MPI_Probe–ის მომდევნო MPI_Recv ფუნქცია გარანტირებულად მიიღებს საჭირო შეტყობინებას, რომლის შემდეგაც მირებული იქნება შეტყობინება მეორე პროცესიდანაც.

```
#include <mpi.h>
int main (int argc, char **argv) {
int ierr, rank, ibuf;
float rbuf;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

ibuf = rank;
rbuf = 1.0 * rank;
if(rank == 1)
      MPI_Send(&ibuf, 1, MPI_INT, 0, 5,MPI_COMM_WORLD);
if(rank == 2)
      MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
if(rank == 0) {
      MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
if(status.MPI_SOURCE == 1) {
      MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
      MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
}
else {
      if(status.MPI_SOURCE == 2) {
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
      }
}
printf("Process 0 recv %d from process 1 %f from process 2", ibuf, rbuf);
}
MPI_Finalize();
return 0;
}
```

4. OpenMP ტექნოლოგია

საერთო მეხსიერების მქონე კომპიუტერებისთვის პარალელური დაპროგრამების ერთ–ერთ ყველაზე პოპულარულ საშუალებას წარმოადგენს OpenMP ტექნოლოგია. იგი ეფუძნება დაპროგრამების ტრადიციულ ენებს (C,C++, ფორტრანი) და იყენებს სპეციალურ კომენტარებს და კომპილატორის დირექტივებს. პარალელური პროგრამის საფუძვლად აიღება "ჩვეულებრივი" მიმდევრობითი პროგრამა. პარალელური ვერსიის შესაქმნელად გამოიყენება კომპილატორის დირექტივები, სპეციალური ფუნქციები და გარემოს ცვლადები. ამ ტექნოლოგიის ერთ-ერთ მიზანს წარმოადგენს აგრეთვე ის, რომ მომხამრებელს ჰქონდეს პროგრამის ერთიდაიგივე ვარიანტი მიმდევრობითი და პარალელური შესრულებისთვის. თუმცა, არსებობენ ისეთი პროგრამებიც, რომლებიც კორექტულად მუშაობენ პარალელურ რეჟიმში და მიმდევრობით მუშაობისას შეიძლება მოგვცეს არასწორი შედეგი. უფრო მეტიც, ცდომილების დაგროვების გამო სხვადასხვა რაოდენობის პროცესორების გამოყენების შემთხვევაში გამოთვლების შედეგი შეიძლება იყოს სხვადასხვა.

4.1. ძირითადი ცნებები

4.1.1. პროგრამის კომპილაცია

იმისათვის რომ OpenMP–ის მექანიზმები გამოვიყენოთ კომპილატორს უნდა ჰქონდეს მისი მხარდაჭერა და პროგრამა უნდა დავაკომპილიროთ შესაბამისი ოპციით. Visual C++-ში ეს ოპცია ერის /openmp, მის გასააქტიურებლად Visual Studio-ს პროექტში პარამეტრში **Project – Properties - Configuration Properties - C/C++ - Language - OpenMP Support** უნდა ამოვირჩიოთ Yes(/openmp)
კომპილატორი, რომელსაც აქვს OpenMP-ის მხარდაჭერა განსაზღვრავს მაკროსს _OPENMP. საკმარისია გამოვიყენოთ პირობითი კომპილაციის დირექტივები #ifdef ან #ifndef რომ გავიგოთ აქვს თუ არა კონკრეტულ კომპილატორს OpenMP-ის მხარდაჭერა. ქვემოთ მოყვანილია პირობითი კომპილაციის უმარტივესი მაგალითი.

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
    printf("OpenMP is supported!\n");
#else
   printf("OpenMP is not supported!\n");
#endif
     }
```

4.1.2. პარალელური პროგრამის მოდელი

გაპარალელება OpenMP-ში ხორციელდება უშუალოდ პროგრამის ტექსტში სპეციალური დირექტივების ჩასმით, აგრეთვე დამხმარე ფუნქციების გამოძახებით. OpenMP-ის გამოყენების დროს ყველა პარალელური ნაკადისთვის (ე.ი.ყველა პროცესორისთვის) გამოიყენება ერთი და იგივე პროგრამული კოდი, ანუ ყველა პროცესორი ასრულებს ერთი და იგივე პროგრამას.

OpenMP-ის საშუალებით დაწრილი პროგრამა შესდგება მიმდევრობითი და პარალელური უბნებისგან. პროგრამა იწყება მიმდევრობითი უბნიდან – თავიდან მუშაობს ერთი პროცესორი (ნაკადი). პარალელურ უბანში შესვლისას წარმოიშობა რამდენიმე ნაკადი, რომლებიც ასრულებენ ერთი და იმავე პროგრამულ კოდს და სრულდებიან ცალ-ცალკე პროცესორებზე (ბირთვებზე). აქ ვგულისხმობთ რომ პროცესორების და ნაკადების რაოდენობა ტოლია. თუ ნაკადების რაოდენობა მეტი იქნება პროცესორების რაოდენობაზე, მაშინ ნაკადები რიგ-რიგობით შესრულდებიან პროცესორებზე. ამის კონტროლს ახორციელებს ოპერაციული სისტემა.
პარალელურ უბანში გარკვეული მექანიზმების საშუალებით ხდება სამუშაოს განაწილება პროცესორებს შორის. პარალელური უბნის დასრულების შემდეგ რჩება ერთი ნაკადი. პროგრამაში შეიძლება იყოს რამდენიმე პარალელური და მიმდევრობითი უბანი.
ეფექტური პარალელური პროგრამის დასაწერად აუცილებელია, რომ ყველა ნაკადი თანაბრად იყოს დატვირთული. ამის მიღწევა შესაძლებელია OpenMP-ის სხვადასხვა მექანიზმების გამოყენებით.
რადგანაც OpenMP ტექნოლოგია გამოიყენება საერთო მეხსიერების მქონე კომპიუტერებში, არსებით მომენტს წარმოადგენს საერთო მონაცემებთან წვდომის სინქრონიზაცია. თავისთავად, საერთო მონაცემების არსებობა ერთდროულად არაშეთანხმებული წვდომის შემთხვევაში იწვევს კონფლიქტებს და მცდარი შედეგის მიღებას. ამიტომ OpenMP-ის მთელი რიგი მექნიზმები განკუთვნილია ერთდროულად მომუშავე ნაკადების სინქრონიზაციისთვის.
OpenMP-ში არ არის არავითარი მექანიზმი ნაკადების მიერ ფაილებთან ერთდროული წვდომის სინქრონიზაციისთვის. თუ ეს საჭიროა, მომხმარებელმა ცხადად უნდა გამოიყენოს სინქრონიზაციის დირექტივები და ბიბლითეკური ფუნქციები. თუ ყველა ნაკადი განსხვავებულ ფაილს მიმართავს, მაშინ არავითარი სინქრონიზაცია არ არის საჭირო.

4.1.3. დირექტივები და ფუნქციები

OpenMP-ის ფუნქციონალურობის უდიდესი ნაწილი რეალიზაბულია კომპილატორის დირექტივების საშუალებით. ისინი უშუალოდ უნდა იყვნენ ჩასმული პროგრამის ტექსტში. C/C++-ში ეს დირექტივები იწყება #pragma omp –თი. დიერქტივის ფორმატი შემდეგია:

#pragma omp *directive-name* [ოპცია[[,]ოპცია]...]

სადაც ***directive-name*** არის კონკრეტული დირექტივის სახელი, რომელსაც შემდეგ შეიძლება მოსდევდეს სხვადასხვა ოპციები. დირექტივის მოქმედების ობიექტი არის ოპერატორი ან ოპერატორთა ბლოკი, უშუალოდ რიმლის წინაც არის მოთავსებული დირექტივა. ასეთ ბლოკში შესვლა უნდა ხდებოდეს მხოლოდ დასაწყისში და გამოსვლა უნდა ხდებოდეს დასასრულიდან. რაც ნიშნავს, რომ არ უნდა ხდებოდეს ბლოკში შესვლა-გამოსვლა goto ოპერატორის საშუალებით.
OpenMP-ის დირექტივები შეიძლება დაიყოს 3 ნაწილად: პარალელური ნაწილის განმსაზღვრელი, სამუშაოს გამნაწილებელი და სინქრონიზაციის. დირექტივას შეიძლება ჰქონდეს რამდენიმე დამატებითი ატრიბუტი – ოპციები.
იმისათვის რომ გამოვიყენოთ OpenMP-ის ბიბლიოთეკის ფუნქციები პროგრამაში უნდა ჩავრთოთ omp.h ფაილი. OpenMP-ის ყველა ფუნქცია იწყება პრეფიქსით omp_.

4.2. პარალელური და მიმდევრობითი უბნები.

პროგრამის გაშვების მომენტში იქმნება ერთადერთი “ძირითადი” ნაკადი, რომელიც იწყებს პროგრამის შესრულებას პირველი ოპერატორიდან. მხოლოდ ძირითადი ნაკადი ასრულებს პროგრამის მიმდევრობით უბნებს. პარალელურ უბანში შესვლისას იქმნება დამატებითი ნაკადები.

დირექტივა parallel

პარალელური უბანი განისაზღვრება დირექტივით
#pragma omp parallel [ოპცია[[,]ოპცია]...]
ოპციები შეიძლება იყოს შემდეგი:

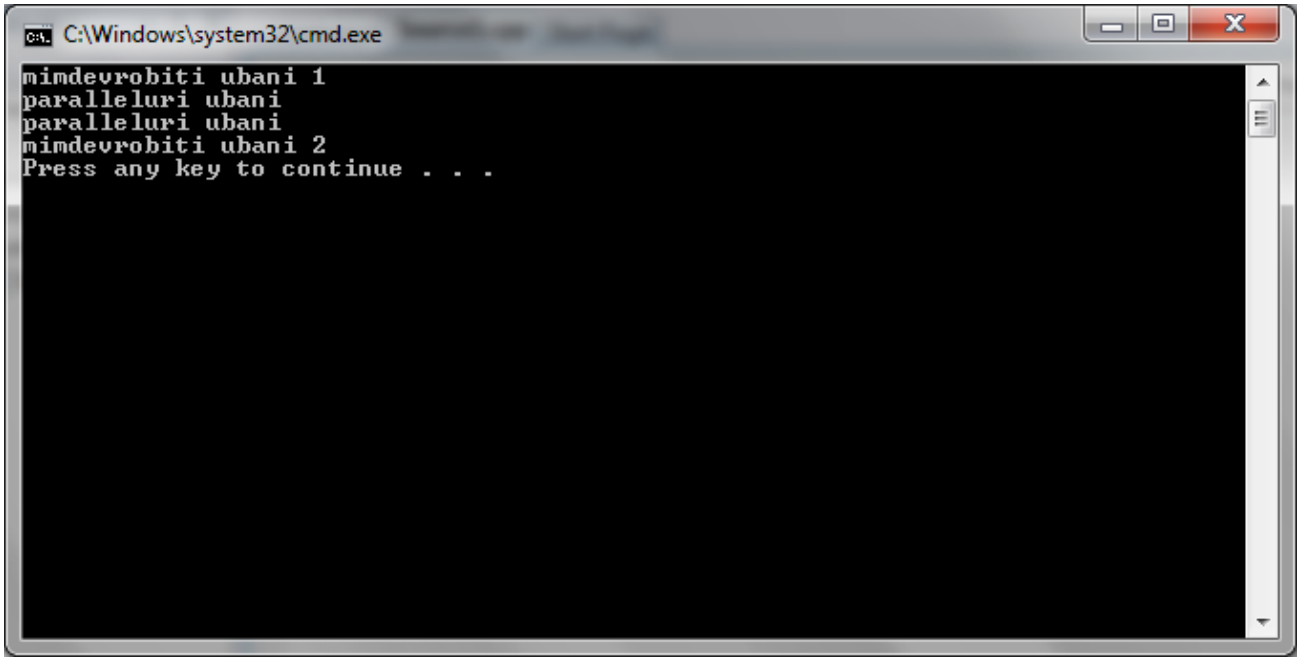
- **num_threads (მთელი გამოსახულება)** – ცხადად მიუთითებს პარალელურ უბანში შესვლისას შესაქმნელი ნაკადების რაოდენობას. თუ ოპცია გამოყენებული არ არის შესაქმნელი ნაკადების რაოდენობა განისაზღვრება **OMP_NUM_THREADS** გარემოს ცვლადის მნიშვნელობით ან ფუნქცია **omp_set_num_threads()-**ის საშუალებით უკანასკნელად მითითებული მნიშვნელობით.
- **private(ცვლადების სია)** – განსაზღვრავს იმ ცვლადებს, როლებისთვისაც თითოეულ ნაკადში შეიქმნება საკუთარი ეგზემპლარი. რაც ნიშნავს, რომ ეს ცვლადები პარალელურ უბანში გამოცხადებული იქნებიან როგორც ლოკალური ცვლადები. ამ ლოკალური ცვლადების საწყისი მნიშვნელობები განსაზღვრული არ არის.
- **shared(ცვლადების სია)** – განსაზღვრავს იმ ცვლადებს, რომლებიც უნდა იყოს საერთო ყველა ნაკადისთვის.

გარდა ამ ოპციებისა კიდევ გამოიყენება სხვა ოპციებიც, რომელთა შესახებაც მოგვიანებით ვისაუბრებთ.

პარალელურ უბანში შესვლისას წარმოიშობა ახალი ნაკადები, რომელთა რაოდენობა ჩავთვალოთ **NP-1** –ის ტოლად (თუ რამდენი ნაკადი წარმოიშობა ამის შესახებ ზემოთ გვქონდა საუბარი). სულ ნაკადების რაოდენობა იქნება **NP.** ყოველ ნაკადს აქვს თავისი უნიკალური ნომერი, ამასთან ის ნაკადი, რომლისგანაც წარმოიშვა დანარჩენი ნაკადები იღებს ნომერს 0, დანარჩენი ნაკადების ნომრები არის 1-დან **NP-1-**მდე. ნაკადების რაოდენობა, რომლებიც ასრულებენ კონკრეტულ პარალელურ უბანს რჩება უცვლელი ამ უბნიდან გამოსვლამდე. პარალელური უბნიდან გამოსვლისას ხდება ნაკადების სინქრონიზაცია, ანუ ყველა ნაკადები ელოდებიან ერთმანეთს და როდესაც ყველა მათგანი მიაღწევს პარალელური უბნის დასასრულს, ხდება ამ უბნიდან გამოსვლა. ამასთან ყველა ახალი ნაკადი ნადგურდება და რჩება მხოლოდ ის, რომლისგანაც წარმოიშვნენ დამატებითი ნაკადები, ანუ ნაკადი ნომრით 0.
ქვემოთ მოყვანილი პროგრამა ახდენს **parallel** დირექტივის მუშაობის დემონსტრაციას. ამ პროგრამის შესრულების შედეგად “მთავარ” ნაკადს (ნაკადს ნომრით 0) გამოაქვს ტექსტი “**mimdevrobiti ubani 1**”. შემდეგ **parallel** დირექტივით იქმნება ახალი ნაკადები, რომლებსაც გამოაქვთ ტექსტი

“**paralleluri ubani**”. ეს ტექსტი გამოიტანება იმდენჯერ რამდენი ნაკადიც არის შექმნილი. ამის შემდეგ ახალი ნაკადები ამთავრებენ მუშაობას და დარჩენილი “მთავარი” ნაკადი ბეჭდავს ტექსტს “**mimdevrobiti ubani 2**”

```
#include <stdio.h>
int main()
{
printf("mimdevrobiti ubani 1\n");
#pragma omp parallel
{
printf("paralleluri ubani \n");
}
printf("mimdevrobiti ubani 2\n");
}
```



4.3. გარემოს ცვლადები და დამხმარე ფუნქციები.

ნაკადების რაოდენობა, რომლებმაც უნდა შეასრულონ პარალელური უბანი შეიძლება მოცემული იყოს გარემოს პარამეტრით **OMP_NUM_THREADS.** მას მნიშვნელობას თავდაპირველად ანიჭებს ოპერაციული სისტემა და, როგორც წესი, იგი ტოლია კომპიუტერში არსებული ბირთვების რაოდენობის. სხვადსხვა საშუალებებით შესაძლებელია ამ ცვლადის მნიშვნელობის შეცვლა. პროგრამიდან მისი შეცვლა შეიძლება ფუნქციით **omp_set_num_threads().** მისი გამოძახების ინტერფეისი არის შემდეგი

void omp_set_num_threads(int num);

სადაც **num** არის ნაკადების რაოდენობა, რომლითაც უნდა შეიცვალოს **OMP_NUM_THREADS** –ის მნიშვნელობა. თუ **parallel** დირექტივაში გამოყენებულია

num_threads ოპცია, მაშინ შეიქმნება იმდენი ნაკადი, რაც ამ ოპციაში არის მითითებული და **omp_set_num_threads()**-ს მნიშვნელობა არა აქვს. ქვემოთ მოყვანილი პროგრამა ახდენს ამ ფუნქციის და **num_threads** ოპციის გამოყენების დემონსტრაციას.

```
#include <stdio.h>
#include <omp.h>
int main()
{
#pragma omp parallel
{
printf("paralleluri ubani 1\n");
}
omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
{
printf("paralleluri ubani 2\n");
}
#pragma omp parallel
{
printf("paralleluri ubani 3\n");
}
}
```

ეს პროგრამა შედგება 3 პარალელური უბნისგან. პირველ პარალელურ უბანში ნაკადების შესახებ არავითარი ინფორმაცია არ არის მითითებული, ამიტომ იგი შესრულდება იმდენი ნაკადის მიერ რისი ტოლოც არის გარემოს ცვლადი **OMP_NUM_THREADS.** თუ პროგრამა გაეშვა 4-ბირთვიან კომპიუტერზე, მაშინ მისი მნიშვნელობა იქნება 4 და ტექსტი **paralleluri ubani 1** გამოიტენება 4-ჯერ. მეორე პარალელური უბნის წინ ფუნქცია **omp_set_num_threads(2)** ნაკადების რაოდენობას გახდის 2-ის ტოლად. მაგრამ ამ უბანში გამოყენებულია ოპცია **num_threads(3),** რაც მიუთითებს იმაზე, რომ მოცემული პარალელური უბანი უნდა შესრულდეს 3 ნაკადის საშუალებით. შესაბამისად, ტექსტი **paralleluri ubani 2** გამოიტენება ეკრანზე 3-ჯერ. მესამე პარალელურ უბანში ოპცია **num_threads** არ გამოიყენება, ამიტომ მოქმედებს მნიშვნელობა, რომელიც განსაზღვრული იყო **omp_set_num_threads(2)**ფუნქციით და ტექსტი **paralleluri ubani 3** გამოიტანება 2-ჯერ, იმისდა მიუხედავად თუ რამდენი ბირთვი აქვს კომპიუტერს.

```
C:\WINDOWS\system32\cmd.exe
paralleluri ubani 1
paralleluri ubani 1
paralleluri ubani 1
paralleluri ubani 1
paralleluri ubani 2
paralleluri ubani 2
paralleluri ubani 2
paralleluri ubani 3
paralleluri ubani 3
Press any key to continue . . .
```

4.4. სამუშაოს განაწილება

როგორც უკვე აღვნიშნეთ, პარალელური უბნის შექმნის შემდეგ ყველა პროცესორი ასრულებს ერთი და იმავე კოდს. ამიტომ მნიშვნელოვანია, რომ ეს კოდი ისე იყოს დაწერილი, რომ სხვდასხვა პროცესორებმა სხვადასხვა სამუშაო შეასრულონ, ანუ მოხდეს სამუშაოს განაწილება. OpenMP-ში არის სამუშაოს განაწილების რამოდენიმე ვარიანტი. აქვე აღვნიშნოთ, რომ სამუშაოს განაწილების კონსტრუქციები არ წარმოშობენ ახალ ნაკადებს. სამუშაოს გამაწილება ხდება უკვე არსებული ნაკადების ფარგლებში.

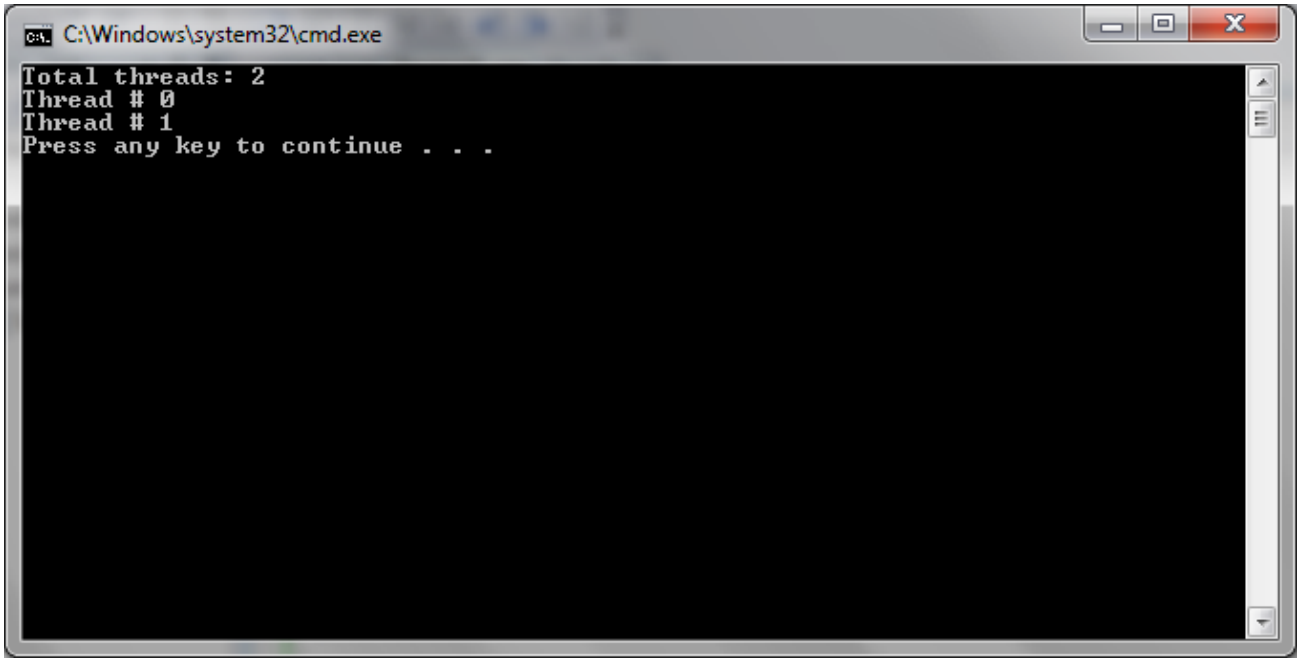
4.4.1. განაწილება დაბალ დონეზე.

დაბალ დონეზე განაწილება ნიშნავს იმას, რომ მომხმარებელი არ იყენებს კომპილატორის შესაძლებლობებს. იგი თვითონ ცხადად მიუთითებს თუ რომელმა პროცესორმა რა სამუშაო უნდა შეასრულოს. ამის მიღწევა შესაძლებელია ნაკადების საერთო რაოდენობისა და თითოეული ნაკადის უნიკალური ნომრის ცოდნის საშუალებით. ყველა ნაკადი პარალელურ უბანში გადანომრილია 0-დან N-1-მდე, სადაც N – არის ნაკედების საერთო რაოდენობა.

დაბალ დონეზე სამუშაოს განაწილება ხდება **omp_get_thread_num()** და **omp_get_num_threads()** ფუნქციების საშუალებით, რომლებიც, შესაბამისად აბრუნებენ ნაკადის ნომერს და ნაკადების საერთო რაოდენობას. ამ ფუნქციებს პარამეტრი არ გააჩნიათ.

ქვემოთ მოყვანილია ამ ფუნქციების მუშაობის საილუსტრაციო პროგრამა. ნაკადებს გამოაქვთ ეკრანზე თავიანთი უნიკალური ნომერი, ხოლო ნაკადს, რომლის ნომერიც არის 0 დამატებით გამოაქვს ნაკადების საერთო რაოდენობა.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int count, num;
#pragma omp parallel shared(count) private(num)
{
count=omp_get_num_threads();
num=omp_get_thread_num();
if (num == 0) printf("Total threads: %d\n", count);
printf("Thread # %d\n", num);
}
}
```



ამ ფუნქციების გამოყენებით შესაძლებელია ყველა ნაკადისთვის განისაზღვროს სამუშაოს ნაწილი, რომელიც მან უნდა შეასრულოს.

დავუბრუნდეთ მასივის ელემენტების ჯამის დათვლის პროგრამას და შევეცადოთ დავწეროთ მის საფუძველზე პარალელური პროგრამა OpenMP-ის საშუალებით. სამუშაოს განაწილება მოვახდინოთ ამ ფუნქცების გამოყენებით.

როგორც უკვე აღვნიშნეთ, გაპარალელებას ექვემდებარება j ციკლი. თუ რომელი ცვლადები უნდა იყოს საერთო და რომელი ლოკალური ამის შესახებ უკვე გვქონდა საუბარი. ამიტომ გაპარალელებისთვის უნდა დავწეროთ დირექტივა

```
#pragma omp parallel default(none) shared(locS,a,p,n,l) \
```

```
      private(i,j,i_st,i_end)
```

თუ დირექტივა ერთ ხაზზე არ გვეტევა მომდევნო სტრიქონზე გადასატანად წინა სტრიქონის ბოლოს უნდა დავწეროთ სიმბოლო \.

იმისათვის, რომ ციკლი გაპარალელდეს, ანუ მისი იტერაციები დაინაწილონ პროცესორებმა, ჩვენს შემთხვევაში საჭიროა მივუთითოთ, რომ j-ური იტერაცია შესრულდეს j-ურ ნაკადზე. ამისათვის დაგვჭირდება გავიგოთ ნაკადის ნომერი და ეს ნომერი გამოვიყენოთ j-ის მნიშვნელობად, ანუ

```
    j=omp_get_thread_num();
```

ხოლო j ციკლი აღარ დაგვჭირდება. პროგრამის ფრაგმენტი მიიღებს სახეს:

```
#pragma omp parallel default(none) shared(locS,a,p,n,l) \
    private(i,j,i_st,i_end)
    {
    j=omp_get_thread_num();
    locS[j]=0;
    i_st=j*l;
    i_end=(j+1)*l;
    if(j==p-1) i_end=n;
    for(i=i_st;i<i_end;i++)
      locS[j]+=a[i];
    }
    S=0;
    for(j=0;j<p;j++)
      S+=locS[j];
```

ეს ფრაგმენტი შედგება ერთი პარალელური უბნისგან და ერთი მიმდევრობითი უბნისგან. მაგრამ ამ სახით პროგრამა კორექტულად არ იმუშავებს, რადგან ნაკადების რაოდენობა p ჩვენ ჩავთვალეთ წინასწარ მოცემულ სიდიდედ (ჩვენ ავიღეთ იგი 4-ის ტოლად). რეალურად ეს ასე არ არის. სხვადასხვა კომპიუტერებზე ბირთვების რაოდენობა იქნება სხვადასხვა. გამოსავალი შეიძლება იყოს ორი. პირველი, გაპარალელების დროს მივუთითოთ, რომ შეიქმნას p ნაკადი, ამის გაკეთება შეიძლება num_threads ოპციით:

```
#pragma omp parallel default(none) shared(locS,a,p,n,l) \
    private(i,j,i_st,i_end) num_threads(p)
```

ამ შემთხვევაში, მიუხედავად იმისა თუ რამდენი ბირთვი აქვს კომპიუტერს, შეიქმნება p ნაკადი. ცხადია, ეს გზა არ არის ოპტიმალური, რადგანაც კომპიუტერს შეიძლება ჰქონდეს p-ზე მეტი ბირთვი და ნაწილი ბირთვებისა არ იქნება გამოყენებული. შეიძლება პირიქითაც მოხდეს, კომპიუტერში იყოს p-ზე ნაკლები ბირთვი, მაშინ p ნაკადის შექმნა არაეფექტური იქნება.

მეორე გზა არის ის, რომ parallel დირექტივა დავწეროთ num_threads-ის გარეშე და პარალელურ უბანში დავადგინოთ თუ რამდენი ნაკადი შეიქმნა. ასეთ შემთხვევაში, როგორც წესი, შეიქმნება იმდენი ნაკადი რამდენი ბირთვიც აქვს

სისტემას, თუ მანამადე არ იყო გამოყენებული omp_set_num_threads ფუნქცია. გავითვალოსწინოთ ისიც, რომ locS მასივი ახლა უნდა აღვწეროთ როგორც მიმთითებელი და new ოპერატორის საშუალებით გამოვყოთ იმდენი ელემენტი რამდენი ნაკადიც იქნება. (აქამდე ვგულისხმობდით რომ გვქონდა ამ მასივის აღწერა, როგორც locS[p]). new ოპერატორი უნდა შეასრულოს მხოლოდ ერთმა ნაკადმა, რადგანაც locS არის საერთო ცვლადი. მჯობესია, თუ ამ ოპერაციას შევასრულებინებთ 0-ოვან ნაკადს, რადგანაც ნაკადი, რომლის ნომერი არის 0 არსებობს ყოველთვის, იმისდა მიუხედავად რამდენი ნაკადიც არ უნდა შეიქმნას parallel დირექტივით. გარდა ამისა, აქვე უნდა გამოითვალოს l-ის მნიშვნელობაც. ასე რომ, ჩვენი ფრაგმენტი მიიღებს სახეს:

```
#pragma omp parallel default(none) shared(locS,a,p,n,l) \
    private(i,j,i_st,i_end)
    {
    j=omp_get_thread_num();
    if(j==0) {
      p=omp_get_num_threads();
      l=n/p;
      locS=new float[p];
    }
    locS[j]=0;
    i_st=j*l;
    i_end=(j+1)*l;
    if(j==p-1) i_end=n;
    for(i=i_st;i<i_end;i++)
      locS[j]+=a[i];
    }
    S=0;
    for(j=0;j<p;j++)
      S+=locS[j];
```

ეს ფრაგმენტიც არ იმუშავებს კორექტულად და შეიძლება ავარიულადაც დამთავრდეს. მიზეზი არის ის, რომ locS მეხსიერების გამოყოფა ხდება 0-ოვანი ნაკადის მიერ, ხოლო შემდეგ მას იყენებს ყველა ნაკადი. შეიძლება მოხდეს ისე, რომ სხვა ნაკადებმა მანამ მიმართონ locS –ს, ვიდრე 0-ვანი მის შექმნას მოასწრებს. ყველაფერი, რომ კორექტული იყოს locS მასივის გამოყენებამდე უნდა ვაიძულოთ ყველა ნაკადი დაელოდოს 0-ვანს. ამის გაკეთება შესაძლებელი ბარიერული სინქრონიზაციის საშუალებით. ეს კეთდება

#pragma omp barrier

დირექტივის საშუალებით. ნაკადები, რომლებიც ასრულებენ პარალელურ უბანს, ამ დირექტივასთან ჩერდებიან და ელოდებიან სანამ ყველა ნაკადი არ მოვა ამ წერტილში. ამის შემდეგ ყველა ნაკადი აგრძელებს მუშაობას. ცხადია, რომ ეს მექნიზმი გადაჭრის ჩვენს პრობლემას, თუ მას დავსვამთ locS[j]=0; ოპერატორის წინ. ამ შემთხვევაში ჩვენ გარანტირებული ვართ, რომ ამ ოპერატორის შესრულებამდე ყვალა ნაკადი მოვიდა ბარიერის წერტილამდე და მათ შორის 0-

ვანიც, რომელიც განახორციელებდა მეხსიერების გამოყოფას. ჩვენი ფრაგმენტი მიიღებს სახეს:

```
#pragma omp parallel default(none) shared(locS,a,p,n,l) \
     private(i,j,i_st,i_end)
     {
     j=omp_get_thread_num();
     if(j==0) {
       p=omp_get_num_threads();
       l=n/p;
       locS=new float[p];
     }
#pragma omp barrier
     locS[j]=0;
     i_st=j*l;
     i_end=(j+1)*l;
     if(j==p-1) i_end=n;
     for(i=i_st;i<i_end;i++)
       locS[j]+=a[i];
     }
     S=0;
     for(j=0;j<p;j++)
       S+=locS[j];
```

ეს ჯერ კიდევ არის არის საბოლოო სახე. ჩვენ კიდევ გარდავქმნით ამ პროგრამას.

4.4.2. პარალელური ციკლები

თუ პარალელურ უბანში არის ციკლი, ზოგადი წესის თანახმად ეს ციკლი შესრულდება ყვალა ნაკადის მიერ. ციკლის გაპარალელებისთვის ციკლის საზღვრები დავწერეთ ისე, რომ ციკლის მთვლელის საწყისი და საბოლოო მნიშვნელობა დამოკიდებული იყო ნაკადის ნომერზე და ამ გზით მივაღწიეთ სხვადასხვა ნაკადების მიერ ციკლის სხვადასხვა იტერაციების შესრულებას. თუმცა, ციკლის იტერაციების ნაკადებს შორის გასანაწილებლად არსებობს დირექტივა for

#pragma omp for [ოპცია [[,]ოპცია]...]

თუ ერცერთი ოპცია არ არის მითითებული, მაშინ იტერაციები მაქსიმალურად თანაბრად ნაწილდება ნაკადებს შორის. ოპციებიდან ჯერ განვიხილოთ ოპცია **schedule,** დანარჩენებს მოგვიანებით განვიხილავთ. ოპცია

schedule(type[, m]) განსაზღვრავს თუ როგორ უნდა იქნენ განაწილებული ციკლის იტერაციები ნაკადებს შორის. ამას განსაზღვრავს პარამეტრი **type.** მისი შესაძლო მნიშვნელობებია:

- **static –** ეს არის იტერაციების ბლოკურ-ციკლური განაწილება. ბლოკის ზომა არის **m.**პირველი **m** იტერაციისაგან შემდგარი ბლოკი სრულდება 0-ვან ნაკადზე, მეორე ასეთი ბლოკი – პირველ ნაკადზე და ა.შ. უკანასკნელ ნაკადამდე. უკანასკნელი ნაკადის შემდეგ მიმდევნო ბლოკი ნაწილდება ისევ 0-ვან ნაკადზე და ყველაფერი თავიდან იწყება. თუ **m-**ის მნიშვნელობა არ არის მითითებული, მთელი იტერაციები იყოფა დაახლოებით თანაბარ უწყვეტ ბლოკებად ისე, რომ ბლოკების რაოდენობა იყოს ნაკადების რაოდენობის ტოლი. მიღებული ბლოკები უნაწილდება ნაკადებს.
- **dynamic –** ფიქსირებული ზომის იტერაციების ბლოკების დინამიური განაწილება. თავიდან ყველა ნაკადი იღებს **m** ზომის ბლოკებს. შემდეგ ის ნაკადი, რომელიც დაამთავრებს მუშაობას, იღებს მომდევნო ბლოკს. გათავისუფლებული ნაკადები იღებენ იტერაციების ახალ ბლოკებს მანამ, სანამ ბლოკები არ ამოიწურება. უკანასკნელ ბლოკში შეიძლება იყოს ნაკლები იტერაცია, ვიდრე სხვა ბლოკებში. თუ **m** მითითებული არ არის, იგულისხმება **m=1.**
- **guided –** იტერაციების დინამიური განაწილება, რომლის დროსაც ბლოკის ზომა მცირდება დაწყებული რაღაც სიდიდიდან **m-**მდე ჯერ კიდევ გაუნაწილებელი იტერაციების რაოდენობის ნაკადების რაოდენობაზე განაყოფის პროპორციულად. პირველი განაწილებული ბლოკის ზომა დამოკიდებულია კომპილატორზე. მთელ რიგ შემთხვევებში განაწილების ასეთი სქემა საშუალებას იძლევა უფრო აკურატულად გავანაწილოთ იტერაციები და დავაბალანსოთ ნაკადების დატვირთვა. ბოლი ბლოკში იტერაციების რაოდენობა შეიძლება იყოს **m-**ზე ნაკლები. თუ **m** მითითებული არ არის, იგულისხმება **m=1.**
- **auto –** იტერაციების განაწილების სქემა აირჩევა კომპილატორის მიერ. ამ შემთხვევაში **m** არ მიეთითება.
- **runtime –** იტერაციების განაწილების სქემა აირჩევა პროგრამის მუშაობის დროს გარემოს ცვლადის **OMP_SCHEDULE** მნიშვნელობის მიხედვით. ამ შემთხვევაშიც **m** არ მიეთითება.

პარალელურ ციკლებს წაეყენება საკმაოდ მკაცრი მოთხოვნები. პროგრამის მუშაობის შედეგი არ უნდა იყოს დამოკიდებული იმაზე, თუ რომელ იტერაციას რომელი ნაკადი შეასრულებს. ციკლიდან დამატებითი გამოსასვლელი არ უნდა იყოს. იტერაციების ბლოკის ზომა, რომლეიც მითითებულია **schedule**–ში არ უნდა იცვლებოდეს ციკლის მუშაობის პროცესში. გარდა ამისა, **for** ოპერატორში მოცემული გამოსახულებების მნიშვნელობა არ უნდა იცვლებოდეს ციკლის ტანში. ეს მოთხოვნა უზრუნველყოფს იმას, რომ ციკლის დაწყებამდე შესაძლებელი იყოს ციკლის იტერაციების რაოდენობის გამოთვლა. ცილის მთვლელი (საიტერაციო ცვლადი) უნდა იყოს ლოკალური, ამიტომ თუ იგი აღწერილია როგორც საერთო, ციკლის დაწყების წინ არაცხადად გარდაიქმნება ლოკალურად. ციკლის დამთავრების შემდეგ ციკლის მთვლელის მნიშვნელობა განსაზღვრული არ არის, თუ იგი არ არის **lastprivate** ოპციის ჩამონათვალში.

ციკლის იტერაციების განაწილების სქემის შერჩევის დროს გასათვალისწინებელია თუ რა დროში სრულდებიან იტერაციები. თუ ისეთ ციკლთან გვაქვს საქმე, სადაც ყველა იტერაცია დაახლოებით თანაბარ დროში სრულდება, მაშინ სტატიკური განაწილება სავსებით მისაღებია. თუ ციკლის იტერაციების შესრულების დროს

მნიშვნელოვნად განსხვავდება ერთმანეთისგან, მაშინ სტატიკური განაწილებისას შეიძლება მოხდეს ისე, რომ ის იტერაციები, რომლებიც დიდ დროს მოითხოვენ მოვდნენ ერთიდაიმავე ნაკადზე, ამიტომ ამ ნაკადს გაცილებით მეტი სამუშაო ექნება. როცა სხვა ნაკადები დაამთავრებენ მუშაობას, მას მოუწევს მუშაობის გაგრძელება. ამიტომ ასეთ დროს დინამიური განაწილება უფრო დააბალანსებს ნაკადებს. თავიდან ყველა მიიღებს თანაბარ პორციებს, ხოლო შემდეგ პორციებს მიიღებს ის ნაკადები, რომლებიც უკვე მიღებულ სამუშაოს შეასრულებენ.

ერთი შეხედვით, შეიძლება შეგვექმნას შთაბეჭდილება, რომ დინამიური განაწილებე ყველა შემთხვევაში მოსახერხებელია, მათ შორის მაშინაც, როცა ციკლის იტერაციები დაახლოებით თანაბარ დროში სრულდება. მაგრამ უნდა გავითვალისწინოთ ისიც, რომ დინამიური განაწილების დროს მის ორგანიზებისთვის გაწეული დამატებითი დროითი დანახარჯები უფრო დიდია ვიდრე სტატიკური განაწილებისას. ამიტომ დინამიური განაწილების უპირატესობა ცხადი აღარ არის. “თანაბარი” იტერაციების შემთხვევაში უმჯობესია სტატიკური განაწილების გამოყენება.

ქვემოთ მოყვანილია პროგრამა, რომლის საშუალებითაც შეგვიძლია დავადგინოთ, იტერაციები როგორ ნაწილდება ნაკადებზე განაწილების სხვადასხვა სქემის შემთხვევაში. არდა ამისა ამ პროგრამაში ვაკეთებთ მთელი ციკლის შესრულებისთვის დახარჯული დროის გაზომვას. დროის გაზომვისათვის ვიყენებთ ფუნქციას **omp_get_wtime(),** რომელიც აბრუნებს წარსულში დროის რაღაც მომენტიდან გასულ დროს წამებში. თუ ამ ფუნქციას გამოვიძახებთ ორჯერ – ერთხელ იმ ფრაგმენტის დასაწყოსში, რომლის შესრულებაზეც დახარჯული დრო გვაინტერესებს და მეორეჯერ ფრაგმენტის დასასრულს, ფუნქციის მიერ დაბრუნებული მნიშვნელობებს შორის სხვაობა მოგვცემს სწორე იმ დროს, რომელიც დაიხარჯე ფრაგმენტის შესრულებაზე.

იმისათვის, რომ ციკლის სხვადასხვა იტერაციებმა იმუშაონ სხვადასხვა დრო გამოვიყენეთ ფუნქცია **Sleep().** ამ ფუნქციას პარამეტრად გადაეცემა მთელი რიცხვი **n.** ფუნქციის მუშაობის შედეგი არის ის, რომ პროგრამის შესრულება იბლოკება **n** მილიწამის განმავლობაში. თუ სხვდასხვა იტერაციებს დავბლოკავთ სხვდასხვა დროით, მივიღებთ, რომ ციკლის იტერაციები მუშაობენ სხვადასხვა დროის განმავლობაში.
ამ პროგრამით დავაკვირდით **(static,3), (static), (dynamic,3)** და **(guided,2)** განაწილების სქემებს. პროგრამაში გამოიყენება 2 ნაკადი.

```
#include <stdio.h>
#include <omp.h>
#include <windows.h>
int main(int argc, char *argv[])
{
int i;
```

```
int m[10]={1,7,2,3,1,4,5,6,2,4};
float t1,t2;

t1=omp_get_wtime();
#pragma omp parallel private(i) num_threads(2)
{
//#pragma omp for schedule (static, 3)
//#pragma omp for schedule (static)
//#pragma omp for schedule (dynamic, 3)
#pragma omp for schedule (guided, 2)
      for (i=0; i<10; i++)
      {
            printf("Thread %d exectes iteration %d\n",
            omp_get_thread_num(), i);
            Sleep(m[i]*1000);
      }
}
t2=omp_get_wtime();
printf("time=%f\n",t2-t1);
return 0;
}
```

ქვემოთმოყვანილ ცხრილში მოცემულია თითოელი იტერაციის მუშაობის დრო და ნაკადის ნომრები, რომლებზეც ისინი მოხვდნენ განაწილების სხვადასხვა სქემის დროს. ბოლო სტრიქონში მოცემულია მთლიანად ციკლის შესრულების დრო სხვადასხვა განაწილების შემთხვევაში.

i	იტერაციის შესრულების დრო, წმ	static	static,3	dynamic,3	guided,2
0	1	0	0	0	0
1	7	0	0	0	0
2	2	0	0	0	0
3	3	0	1	1	0
4	1	0	1	1	0
5	4	1	1	1	1
6	5	1	0	1	1
7	6	1	0	1	1
8	2	1	0	1	0
9	4	1	1	0	0
ციკლის შესრულების დრო, წმ		21	23	21	20

ამ ციკლში კი შეჯამებულია თითოეული ნაკადის მუშაობის დრო. ვიცით რა, რომლეი იტერაცია რომელ ნაკადზე განაწილდა და ამ იტერაციის მუშაობის დრო, ადვილად შეგვიძლია გამოვთვალოთ თუ რა დროს დახარჯავდა ნაკადი მასზე განაწილებული იტერაციების შესასრულებლად.

	0	1	საერთო დრო
Static	14	21	21
Static,3	23	12	23
Dynamic,3	14	21	21
Guided,2	20	15	20

ახლა კი გადავწეროთ ჩვენი პროგრამა (მასივის ელემენტების ჯამის პოვნა) ისე რომ ციკლის იტერაციების განაწილება მივანდოთ კომპილატორს. მ შემთხვევაში აღარ დაგვჭირდება i_st, i_end, და l ცვლადები.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
const int n=100;
float *locS,a[n];
float S;
int i,j,p;
      for (i=0;i<n;i++)
            a[i]=i;
#pragma omp parallel default(none) shared(locS,a,p,n) \
      private(i,j)
      {
      j=omp_get_thread_num();
      if(j==0) {
            p=omp_get_num_threads();
            locS=new float[p];
      }
#pragma omp barrier
      locS[j]=0;
#pragma omp for
      for(i=0;i<n;i++)
            locS[j]+=a[i];
      }
      S=0;
      for(j=0;j<p;j++)
            S+=locS[j];

      printf("S=%f\n",S);
}
```

4.4.3. პარალელური სექციები

პარალელური სექციების დირექტივა გამოიყენება პროგრამაში სასრული (არაიტერაციული) პარალელიზმის მისათითებლად. აქამდე ჩვენ ვიხილავდით ისეთ შემთხვევებს, როდესაც პარალელურად სრულდებოდა ციკლის იტერაციები. პროგრამაში შეიძლება იყოს აგრეთვე რამდენიმე ფრაგმენტი რომლებიც შეიძლება შესრულდნენ პარალელურად. მაგალითად,

```
for(i=1;i<n;i++)
   x[i]=x[i-1]+a[i];
for(i=1;i<m;i++)
   y[i]=y[i-1]+b[i];
```

ამ ფრაგმენტში თითოეული ციკლის ცალ-ცალკე გაპარალელება არ შეიძლება, რადგანაც ყოველ იტერაციაზე საჭიროა წინა იტერაციაზე გამოთვლილი მნიშვნელობა. მაგრამ ეს ორი ციკლი ერთად შეიძლება შესრულდეს პარალელურად. ანუ, ერთმა ბირთვმა შეასრულოს პირველი ციკლი, მეორემ – მეორე. ამას ჰქვია სწორედ სასრული პარალელიზმი. პარალელურად შესასრულებელი ბლოკების რაოდენობა არის კონკრეტულად განსაზღვრული (სასრული), განსხვავებით პარალელური ციკლის იტერაციებისგან, რომელთა რაოდენობა უმრავლეს შემთხვევაში არის დამოკიდებული პარამეტრზე და შეიძლება იცვლებოდეს.

პარალელური სექციები პროგრამაში მოიცემა **sections** დირექტივის საშუალებით.

#pragma omp sections [ოპცია[[,] ოპცია]...]

ეს დირექტივა განსაზღვრავს დამოუკიდებელი უბნების (სექციების) ერთობლიობას, თითოეული სექცია სრულდება ცალკე ნაკადის მიერ. შესაძლებელი ოპციების შესახებ მოგვიანებით გვექნება საუბარი. ჯერჯერობით გამოვიყენოთ ოპციების გარეშე.

სექციები გამოიყოფა დირექტივით

#pragma omp section

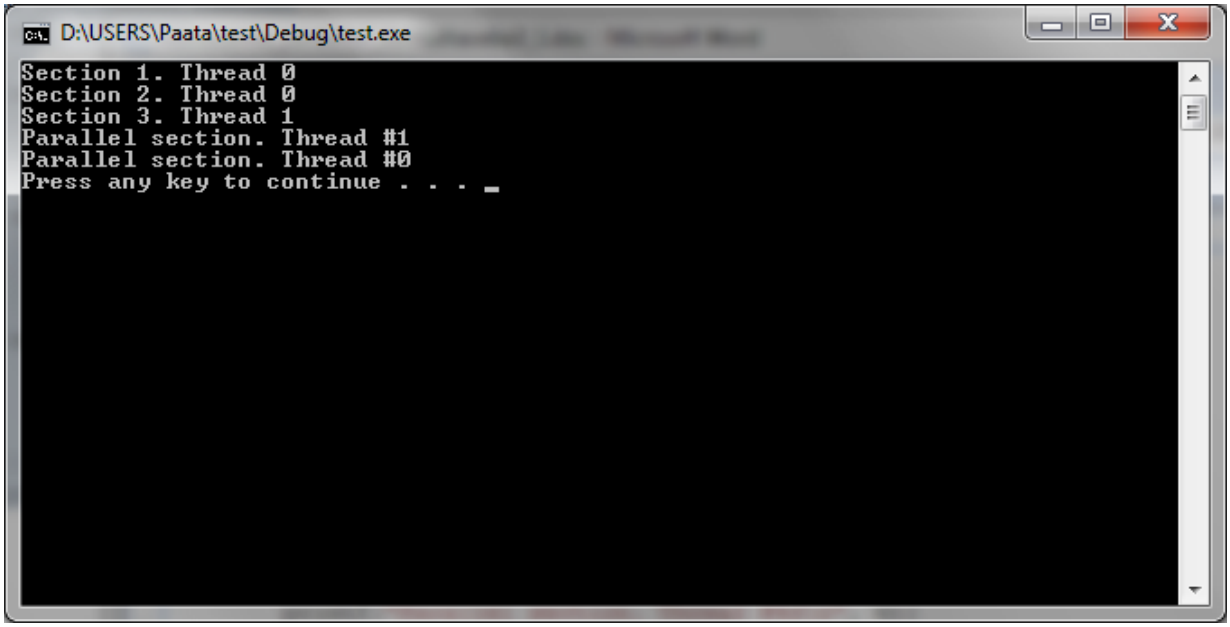
რომელიც უნდა განთავსდეს თითოეული პარალელური სექციის წინ, ხოლო ყველა ეს უბანი უნდა იყოს **sections** დირექტივით განსაზღვრული ბლოკის შიგნით. ბლოკის პირველი სექციის წინ **section** დირექტვა აუცილებელი არ არის. რომელი ნაკადი რომელ სექციას შეასრულებს ეს სტანდარტით წინასწარ განსაზღვრული არ არის, შეიძლება იყოს ნებისმიერი შესაბამისობა. თუ ნაკადების რაოდენობა მეტია სექციების რაოდენობაზე ზოგი ნაკადი სამუშაოს გარეშე დარჩება. თუ ნაკადების რაოდენობა ნაკლებია სეციების რაოდენობაზე, მაშინ ზოგ ნაკადს (შეიძლება ყველას) შეხვდება რამდენიმე სექცია შესასრულებლად.

შემდეგი მაგალითი წარმოადგენს **sections** დირექტივის მუშაობის ილუსტრაციას.

```
#include <stdio.h>
#include <omp.h>
```

```
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel private(n) num_threads(2)
   {
   n=omp_get_thread_num();
#pragma omp sections
      {
#pragma omp section
        {
           printf("Section 1. Thread %d\n", n);
        }
#pragma omp section
        {
           printf("Section 2. Thread %d\n", n);
        }
#pragma omp section
        {
           printf("Section 3. Thread %d\n", n);
        }
     }
     printf("Parallel section. Thread #%d\n", n);
   }
}
```

სამი სექცია ნაწილდება ორ ნაკადზე. თითოეულ სექციას გამოაქვს ეკრანზე სექციის ნომერი და ნაკადის ნომერი. შემდეგ კი თითოეულ პარალელურ ნაკადს გამოაქვს ნაკადის ნომერი:



4.5. სინქრონიზაციის საშუალებები

OpenMP შეიცავს მთელ რიგ საშუალებებს, რომლებიც ახორციელებენ ნაკადების სინქრონიზაციას. ნაკადების სინქრონიზაცია არის ნაკადების შესრულების თანმიმდევრობის გარკვეული პრინციპით მოწესრიგება

4.5.1. ბარიერული სინქრონიზაცია.

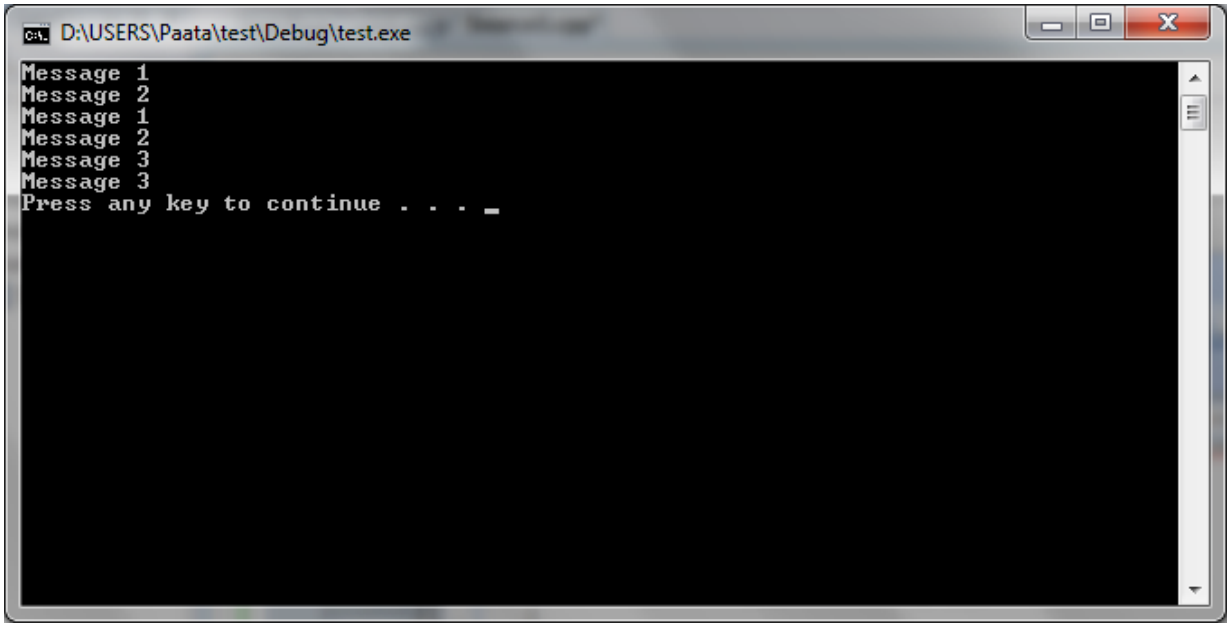
ბარიერული სინქრონიზაციის შესახებ უკვე გვქონდა საუბარი. იგი არის ყველაზე უფრო გავრცელებული სინქრონიზაციის საშუალება. იგი მოიცემა დირექტივით

```
#pragma omp barrier
```

ნაკადები, რომლებიც ასრულებენ პროგრამის პარალელურ ნაწილს, ამ დირექტივასთან მისვლისას ჩერდებიან და ელოდებიან, სანამ ყველა ნაკადი არ მივა პროგრამის ამ წერტილამდე, შემდეგ კი ყველანი ერთად აგრძელებენ მუშაობას. შემდეგი პროგრამა არის ბარიერული სინქრონიზაციის ილუსტრაცია.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
#pragma omp parallel num_threads(2)
      {
      printf("Message 1\n");
      printf("Message 2\n");
#pragma omp barrier
      printf("Message 3\n");
      }
}
```

ამ პროგრამაში პარალელური ნაწილი სრულდება 2 ნაკადის მიერ. თითოელი ნაკადს გაომაქვს სამი სტრიქონი (`Message 1, Message 2, Message 3)`. ამათგან, სხვადასხვა ნაკადების მიერ გამოტანილი პირველი ორი სტრიქონი შეიძლება იყოს ერთმანეთში არეული, ხოლო მესამე სტრიქნი ყოველთვის გამოვა მას შემდეგ რაც წინა ორი სტრიქონი იქნება გამოტანილი ორივე ნაკადის მიერ:



4.5.2. კრიტიკული უბნები

კრიტიკული უბნის მითითება ხდება

#pragma omp critical [(<კრიტიკული უბნის სახელი**>)]**

დირექტივის საშუალებით. დროის ყოველ მომენტში კრიტიკულ უბანში შეიძლება იყოს მხოლოდ ერთი ნაკადი. თუ კრიტიკული უბანი სრულდება უკვე რომელიმე ნაკადის მიერ ამავე სახელის მქონე კრიტიკულ უბანში სხვა ნაკადი ვერ შევა. იგი დაიბლოკება უბნის დასაწყისში და მხოლოდ მას შემდეგ გააგრძელებს მუშაობას როდესაც უბანში შესული ნაკადი გამოვა ამ უბნიდან. თუ კრიტიკულ უბანთან დაბლოკილია რამდენიმე ნაკადი, მაშინ ერთ-ერთ მათგანს მიეცემა კრიტიკულ უბანში შესვლის საშუალება, დანარჩენები დაელოდებიან ამ ნაკადის გამოსვლას და ა.შ. როგორც კი კრიტიკულ უბანში შესული ნაკადი გამოვა ამ უბნიდან, ერთ-ერთი მომლოდინე ნაკადი შევა მასში.

ყურადღება მივაქციოთ იმას, რომ პროგრამაში შეიძლება იყოს რამდენიმე კრიტიკული უბანი. ზემოთ აღწერილი რეგულაცია (ნაკადების რიგ-როგობით შეშვება) ეხება ერთი და იმავე სახელის მქონე უბნებს. თუ ერთი ნაკადი შევიდა რომელიმე კრიტიკულ უბანში და მეორე ნაკადა მიადგა სხვა სახელის მქონე კრიტიკულ უბანს, იგი მასში უპრობლემოდ შევა. სახელის არ მქონე უბნები განიხილება როგორც ერთი და იგივე სახელის მქონე უბნები.

მოვიყვანოთ კრიტიკული უბნის გამოყენების ილუსტრაცია:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel
      {
#pragma omp critical
            {
                  n=omp_get_thread_num();
                  printf("Thread %d\n", n);
            }
      }
}
```

ამ პროგრამაში ცვლადი `n` არის გამოცხადებული როგორც საერთო (იგი არ არის არც `private`, არც `shared` ოპციაში მოცემული, ამიტომ არის `shared`). რომ არ იყოს კრიტიკული სექცია პარალელურ უბანში `n`-ის ასეთი გამოყენება იქნება არამართებული, რადგანაც მას სხვადასხვა ნაკადები მიანიჭებენ სხვადასხვა მნიშვნელობებს და წინა მინიჭებული მნიშვნელობა მის გამოყენებამდე შეიძელბა დაიკარგოს. სე მაგალითად, შეიძლება მოხდეს ისე, რომ სანამ ნაკადი მოასწრებს `n`-ის მნიშვნელობის გამოტანას (`printf("Thread %d\n", n)` ოპერატორით) იგი შეიძლება შეცვალოს სხვა ნაკადმა და შესაბამისად, გამოიტანება არასწორი მნიშვნელობა. მაგარამ თუ ეს ოპერატორები მოქცეული არიან კრიტიკულ უბანში (როგორც ზემოთმოყვანილ პროგრამაშია), მაშინ სანამ ერთი ნაკადი არ დაასრულებს ამ უბნის შესრულებას (მათ შორის `printf("Thread %d\n", n)` ოპერატორს) მეორე ნაკადი ვერ შევა მასში და ვერ შეძლებს `n`-ის მნიშვნელობის შეცვლას. ამიტომ პროგრამა კორექტულად იმუშავებს.

დავუბრუნდეთ ახლა ისევ ჩვენს ამოცანას – მასივის ელემენტების ჯამის პოვნას. პროგრამის ბოლო სახე, რომელზეც ჩვენ მივედით იყო შემდეგი

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
const int n=100;
float *locS,a[n];
float S;
int i,j,p;

      for (i=0;i<n;i++)
            a[i]=i;
#pragma omp parallel default(none) shared(locS,a,p,n) \
      private(i,j)
      {
      j=omp_get_thread_num();
      if(j==0) {
            p=omp_get_num_threads();
            locS=new float[p];
      }
#pragma omp barrier
      locS[j]=0;
#pragma omp for
      for(i=0;i<n;i++)
            locS[j]+=a[i];
      }
      S=0;
      for(j=0;j<p;j++)
            S+=locS[j];

      printf("S=%f\n",S);
}
```

ყურადღება მივაქციოთ `locS` მასივს, რომელიც არის საერთო ცვლადი. იგი შედგება იმდენი ელემენტისგან რამდენი ნაკადიც გამოიყენება გაპარალელების დროს. თითოეული ელემენტი შეესაბამება რომელიმე ნაკადს. ამ მხრივ, იგივე ეფექტს მივაღწევდით თუ `locS` იქნებიდა არა მასივი, არამედ ჩვეულებრივი ცვლადი და მას გამოვაცხადებდით კერძო (ლოკალურ) ცვლადად. ამ ცვლადის იმდენი ეგზემპლარი იქნებოდა რამდენი ნაკადიც გვექნებოდა. მაგრამ მასივი `locS` გამოიყენება პარალელური უბნის გარეთაც, ხოლო კერძო ცვლადად აღწერის შემთხვევაში მისი მნიშვნელობები პარალელური უბნის გარეთ დაკარგული იქნება. მაგრამ ხომ შეიძლება ლოკალური ჯამების ჯამის პოვნა შევიტანოთ პარალელური უბნის შიგნით. მაშინ იქ შეიძლება გამოვიყენოთ კერძო ცვლადად გამოცხადებული ჩვეულებრივი ცვლადი `locS`, ამასთან ცვლადი `S` უნდა გამოვაცხადოთ საერთო ცვლადად. შესაბამისი ფრაგმენტი მიიღებს სახეს:

```
#pragma omp parallel default(none) shared(a,p,n,S) \
      private(i,j,locS)
{

      .  .  .
```

```
      locS=0;
#pragma omp for
      for(i=0;i<n;i++)
            locS+=a[i];
      }
      S+=locS;
}

      printf("S=%f\n",S);
}
```

ლოკალური ჯამების ჯამის საპოვნელად უკვე ციკლი აღარ გვჭირდება, რადგანაც `S+=locS` ოპერატორი შესრულდება ყველა ნაკადის მიერ, ყველა მათგანი დაუმატებს `S`-ს თავის `locS`–ის მნიშვნელობას და მიიღება ყველას ჯამი. მაგრამ სწორედ ეს ოპერატორი არის საფრთხის შემცველი. წარმოვიდგინოთ, რომ ორმა ნაკადმა ერთდროულად მიღწია ამ ოპერატორს. ერთი ნაკადი წაიკითხავს `S`-ის მნიშვნელობას სანამ დაუმატებს მას თავის `locS`–ს, მეორე ნაკადი წაიკითხავს `S`-ის იგივე მნიშვნელობას და დაუმატებს მას თავის `locS`–ს. ორივე მათგანი ჩაწერს `S`-ის განახლებულ მნიშვნელობას. მაგრამ ეს უკვე არ იქნება ორი `locS`–ის ჯამი. სიტუაცია გამოსწორდება თუ ჩვენ ნაკადებს ვაიძულებთ რიგ-რიგობით შეასრულონ ეს ოპერატორი. მაშინ ჯერ ერთი დაუმატებს `S`-ს თავის `locS`–ის მნიშვნელობას, შემდეგ მეორე და ა.შ. ეს ნიშნავს, რომ ოპერტორი უნდა მოვაქციოთ კრიტიკულ სექციაში. პროგრამა ამ გარდაქმნის შედეგად იქნება ასეთი:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
const int n=100;
float locS,a[n];
float S;
int i;

for (i=0;i<n;i++)
   a[i]=i;
S=0;
#pragma omp parallel default(none) shared(a,n,S) private(locS)
   {
      locS=0;
#pragma omp for
      for(i=0;i<n;i++)
         locS+=a[i];
#pragma omp critical
      {
          S+=locS;
      }
   }
   printf("S=%f\n",S);
}
```

4.5.3. დირექტივა atomic

კრიტიკული უბნების გამოყენების კერძო შემთხვევას წარმოადგენს საერთო ცვლადის განახლება. სწორედ ეს სიტუაცია გვქონდა უკანასკნელ მაგალითში. თუ ოპერატორი არის **sum=sum+expr** პარალელურ უბანში და **sum** არის საერთო ცვლადი, ამ ოპერტორის სხვდასხვა ნაკადების მიერ ერთდროულად შესრულებამ შეიძლება მოგვცეს არასწორი შედეგი. ჭეშმარიტი შედეგის მისაღებად შეგვიძლია ვისარგებლოთ კრიტიკული სექციებით, ან სპეციალურად ასეთი სიტუაციებისთვის შექმნილი დირექტივით

#pragma omp atomic

ეს დირექტივა მოქმედებს უშუალოდ მის შემდეგ მდებარე მინიჭების ოპერატორზე და გარანტირებულია მარცხენა მხარეში მდგომი საერთო ცვლადთან კორექტული მუშაობა. ოპერატორის შესრულების პროცესში ამ საერთო ცვლადთან წვდომა ბლოკირებულია ყველა ნაკადისათვის, გარდა იმ ნაკადის, რომელმაც დაიწყო მისი შესრულება. პრაქტუკულად ეს არის კრიტიკული უბნის ის შემთხვევა, როდესაც კრიტიკული უბანი შედგება ერთი მინიჭების ოპერატორისგან, რომლის მარცხენა მხარესაც წარმოადგენს საერთო ცვლადი.

ქვემოთმოყვანილ მაგალითში ცვლადი **count** გამოცხადებულია როგორც საერთო. თავიდან მას ენიჭება 0. შემდეგ ყველა ნაკადი მას ზრდის 1-ით. თუ ამ ოპერაციას ნაკადები შეასრულებენ რიგ-რიგობით **count** -ში იქნება ნაკადების საერთი რაოდენობა. ამიტომ ეს ოპერაცია უნდა გამოვაცხადოთ ატომარუ;ად, ანუ მის წინ მოვათავსოთ დირექტივა **atomic**. ამ დირექტივის გარეშე პროგრამამ შეიძლება მოგვცეს მცდარი შედეგი.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int count = 0;
#pragma omp parallel
{
#pragma omp atomic
count++;
}
printf("Number of threads: %d\n", count);
}
```

4.5.4. დირექტივა master

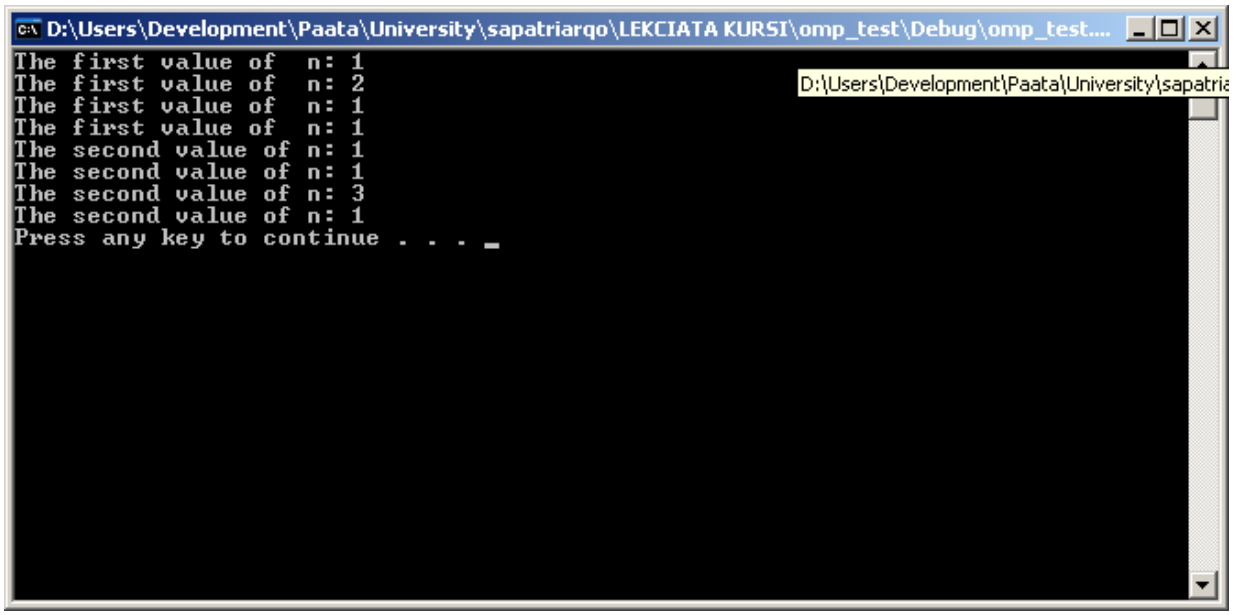
ამ დირექტივის საშუალებით პროგრამული კოდის ის ნაწილი, რომელზეც იგი ვრცელდება შესრულდება მხოლოდ მთავარი ნაკადის (ნაკადის, რომლის ნომერი არის 0) მიერ. დანარჩენი ნაკადები გამოტოვებენ ამ ფრაგმენტს და აგრძელებენ მუშაობას შემდეგი ოპერატორიდან. არაცხადი ბარიერული სინქრონიზაცია არ არის. თუ საჭიროა, ჩვენ თვითონ უნდა დავსვათ სინქრონიზაციის დირექტივა. დირექტივას აქვს სახე:

#pragma omp master

ქვემოთმოყვანილ პროგრამაში ცვლადი n გამოცხადებულია, როგორც კერძო (ლოკალური) ცვლადი. მას თავდაპირველად ენიჭება 1, შემდეგ მთავარი ნაკადი master-ბლოკში ანიჭებს 2-ს და ყველა ნაკადს გამოაქვს მისი მნშვნელობა და ნაკადის ნომერი. შემდეგ მთავარი ნაკადი ანიჭებს 3-ს და ისევ გამოაქვთ n-ის მნიშვნელობა.

```
#include <stdio.h>
#include <omp.h>
int main()
{
int n,id;
#pragma omp parallel private(n,id)
{
   n=1;
   id=omp_get_thread_num();
#pragma omp master
   {
     n=2;
   }
   printf("The first value of  n: %d    thread #%d\n", n,id);
#pragma omp barrier
#pragma omp master
   {
     n=3;
   }
   printf("The second value of n: %d   thread #%d\n", n,id);
}
}
```

შედეგიდან ჩანს, რომ n-ის მნიშვნელობა იცვლება მხოლოდ 0-ვან ნაკადზე:



4.5.5. დირექტივა single

თუ პროგრამის პარალელურ ნაწილში საჭიროა გარკვეული ფრაგმენტის შესრულება ერთი ნაკადის მიერ (მაგალითად, მონაცემების შეტანა, საერთო ცვლადებზე ოპერაციები და ა.შ.) უნდა გამოვიყენოთ დირექტივა single. მას აქვს ასეთი სახე:

#pragma omp single [ოპცია [[,] ოპცია]…]

ის ბლოკი, რომელზეც მოქმედებს ეს დირექტივა შესრულდება ერთი ნაკადის მიერ. რომელი ნაკადის მიერ განსაზღვრული არ არის. დანარჩენი ნაკადები გამოტოვებენ ამ ბლოკს და ელოდებიან ბლოკის მომდევნო ოპერატორთან მის დასრულებას, თუ არ არის მითითებული ოპცია nowait. ანუ არაცხადად არის ბარიერული სინქრონიზაცია.

ოპციის შესაძლო მნიშვნელობებია:

- **copyprivate(ცვლადების სია)-** ამ სიაში ჩამოთვლილი ცვლადების განახლებული მნიშვნელობები ბლოკის დამთავრების შემდეგ გადაიწერება ყველ ნაკადის შესაბამის ცვლადებში.

- **nowait –** ამ ოპციის გამოყენების შემთხვევაში ბლოკის ბოლოს ბარიერული სინქრონიზაცია არ იქნება, ანუ სხვა ნაკადები აღარ დაელოდებიან single-ბლოკის დასრულებას

ქვემოთ მოყვანილია ამ დირექტივის გამოყენების ილუსტრაცია:

```
#include <stdio.h>
#include <omp.h>

int main()
{
int id;
#pragma omp parallel private(id)
{
   id=omp_get_thread_num();
   printf("message 1   therad #%d\n",id);
#pragma omp single nowait
   {
     printf("single    therad #%d\n",id);
   }
   printf("message 2   therad #%d\n",id);
}
}
```

ყველა ნაკადს გამოაქვს ეკრანზე message 1, ერთ რომელიმე ნაკადს გამოაქვს single, შემდეგ ყველას - message 2. თანაც ნაკადები არ ელოდებიან single-ბლოკის დასრულებას. შედეგი ოთხ ნაკადზე გაშვებისას:

```
D:\Users\Development\Paata\University\sapatriarqo\LEKCIATA KURSI\omp_test\Debug\omp_test....
message 1    therad #2
message 1    therad #0
message 2    therad #0
message 1    therad #3
message 2    therad #3
message 1    therad #1
message 2    therad #1
single     therad #2
message 2    therad #2
Press any key to continue . . .
```

თუ nowait ოპციას არ დავწერთ მაშინ შედეგი იქნება ასეთი:

```
D:\Users\Development\Paata\University\sapatriarqo\LEKCIATA KURSI\omp_test\Debug\omp_test....
message 1    therad #1
message 1    therad #2
message 1    therad #0
message 1    therad #3
single     therad #1
message 2    therad #1
message 2    therad #0
message 2    therad #2
message 2    therad #3
Press any key to continue . . .
```

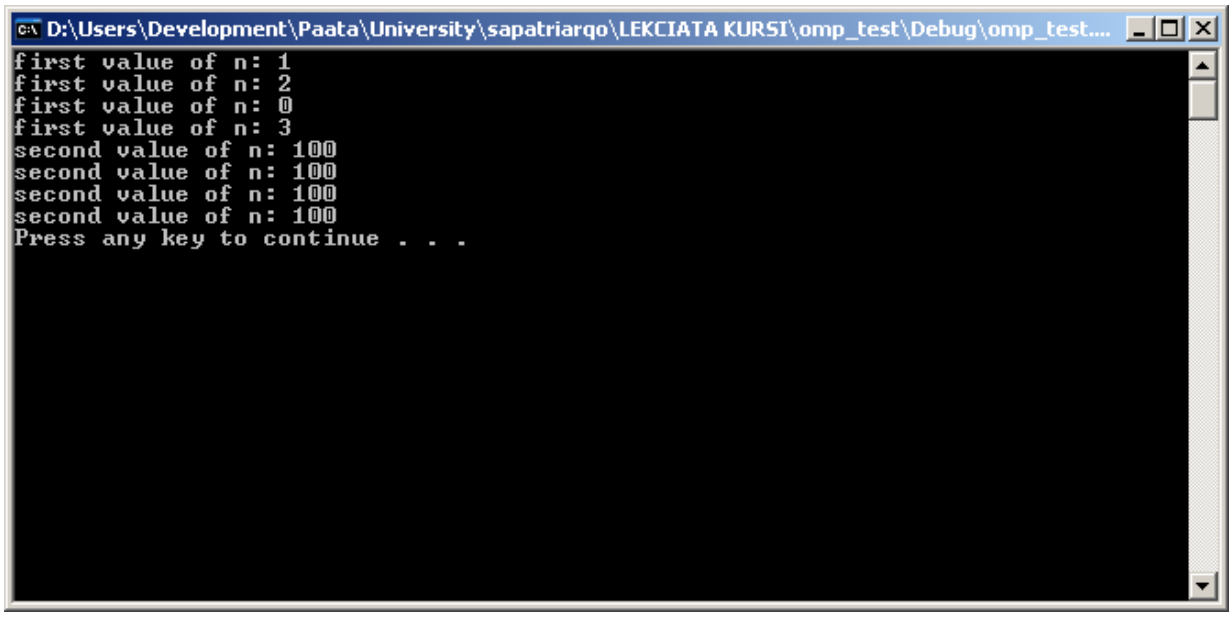
როგორც ვხედავთ message 2 გამოტანილი იქნა მას შემდეგ რაც ერთ-ერთმა ნაკადმა გამოიტანა ტექსტი single, ამ შემთხვევაში ეს იყო ნაკადი ნომრით 1.

ახლა ვნახოთ copyprivate ოპციის გამოყენების მაგალითი

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
```

```
printf("first value of n: %d\n", n);
#pragma omp single copyprivate(n)
{
n=100;
}
printf("second value of n: %d\n", n);
}
}
```

ამ პროგრამაში n გამოცხადებულია კერძო ცვლადად, მას თავიდან ყველა ნაკადი ანიჭებს ნაკადის ნომერს და გამოაქვს ეკრანზე. შემდეგ ერთ-ერთი ნაკადი n-ს ანიჭებს 100-ს. მეორედ ეკრანზე გამოტანისას ყველა ნაკადზე n-ის მნიშვნელობა იქნება 100, რადგანაც გამოყენებული იყო copyprivate ოპცია.



4.6. ოპცია reduction

დირექტივა parallel-ში შესაძლებელია იყოს ოპცია reduction. მას აქვს შემდეგი ფორმა:

reduction(*ოპერატორი* : *ცვლადების სია***)**

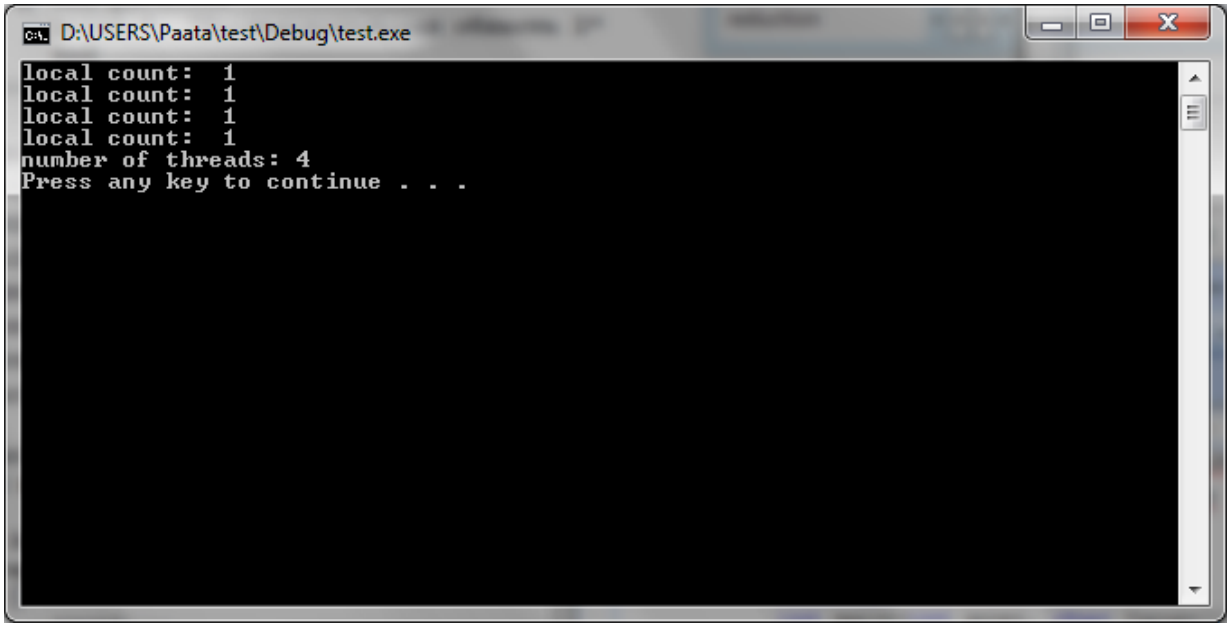
ამ ოპციაში მოცემული ცვლადები ატარებენ როგორც კერძო, ისე საერთო ცვლადების თვისებებს. ისინი ჰგვანან კერძო ცვლადებს იმით, რომ ყოველ ნაკადს აქვს ამ ცვლადის საკუთარი ეგზემპლარი. მაგრამ პარალელური ნაწილის დასრულების შემდეგ მათი მნიშვნელობა კი არ იკარგება, არამედ მათზე ტარდება *ოპერატორით* განსაზღვრული ოპერაცია და მიღებული შედეგი იქნება დაბრუნებული მთავარ ნაკადში პარალელური ნაწილის დასრულების შემდეგ. *ოპერატორი* შეიძლება იყოს: +, *, -, &, |, ^, &&, ||. ცვლადის ლოკალური ეგზემპლარების ინიციალიზაცია ხდება 0-ით ან მისი ანალოგით, თუ ოპერაცია ადიტიურია, და 1-ით ან მისი ანალოგით თუ ოპერაცია მულტიპლექსურია. ოპერაციების შესრულების თანმიმდევრობა განსაზღვრული არ არის. მაგალითად, თუ გვაქვს დირექტივა

#pragma omp parallel reduction (+: S)

ნიშნავს, რომ ყველა ნაკადი გამოთვლის ლოკალურად S-ის მნიშვნელობას და შემდეგ ამ მნიშვნელობების ჯამი დაემეტება S-ის თავდაპირველ მნიშვნელობას, რომელიც ჰქონდა მას ძირითად ნაკადში პარალელურ უბანში შესვლამდე. განვიხილოთ მაგალითი:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int count = 0;
#pragma omp parallel num_threads(4) reduction (+: count)
{
   count++;
   printf("local count:  %d\n", count);
}
printf("number of threads: %d\n", count);
}
```

ამ მაგალითში ხდება შექმნილი ნაკადების რაოდენობის და `count` თვლა. ყველა ნაკადი ინიციალიზაციას უკეთებს ლოკალურ `count`-ს 0-ით. შემდეგ ყველა ნაკადი ზრდის ლოკალური `count`-ის მნიშვნელობას 1-ით, ანუ ყველა ნაკადზე მისი მნიშვნელობა იქნება 1. პარალელური უბნიდან გამოსვლისას კი ლოკალური `count`-ების მნიშვნელობა შეიკრიბება და ჩაიწერება ისევ `count`-ში. ყურადღება მიაქციეთ, რომ უკანასკნელი ოპერატორი, რომელსაც გამოაქვს `count`-ის მნიშვნელობა არ არის პარალელურ ნაწილში.



ახლა გამოვიყენოთ ეს მექნიზმი ჩვენს ამოცანაში – მასივის ელემენტების ჯამის დათვლაში. ჩვენ უკვე აღარ დაგვჭირდება ცვლადი `locS`, ცვლადი `S` ჩავწროთ `reduction` ოპციაში შეკრების ოპერაციით. ამით ჩვენმა პროგრამამ უკვე მიიღო საბოლოო სახე.

```
#include <stdio.h>
#include <omp.h>
```

```
int main(int argc, char *argv[])
{
const int n=100;
float a[n], S;
int i;
      for (i=0;i<n;i++)
            a[i]=i;
      S=0;
#pragma omp parallel default(none) shared(a,n) \
      private(i) reduction(+:S)
      {
#pragma omp for
         for(i=0;i<n;i++)
            S+=a[i];
 }
 printf("S=%f\n",S);
}
```

5. პროგრამების გაპარალელების ხერხები

5.1. პარალელური ციკლები

უმრავლეს შემთხვევაში რეალურ პროგრამებში გაპარალელების რესურსი ჩადებულია ციკლებში, რადგანაც, როგორც წესი, დროის ძირითადი ნაწილი ციკლებით მოცემული პროგრამული კოდის რეალიზაციაზე იხარჯება. ამიტომ, გაპარალელების დროს პირველ რიგში უნდა გამოვიკვლიოთ ციკლები და დავადგინოთ, შესაძლებელია თუ არა ციკლის იტერაციები შესრულდეს ერთმანათისგან დამოუკიდებლად. ეს ნიშნავს, რომ ციკლის ერთ იტერაციაზე გამოთვლილი ცვლადების მნიშვნელობები არ უნდა გამოიყენებოდეს შემდეგ იტერაციაზე. ციკლის იტერაციების პარალელურად შესრულება ნიშნავს, რომ იტერაციების შესრულების თანმიმდევრობა არ იქნება ისეთი, როგორც ციკლის ოპერატორის სემანტიკით არის განსაზღვრული, ანუ იტერაციები შესრულდებიან განსხვავებილი თანმიმდევრობით, ვიდრე მიმდევრობით პროგრამაში შესრულდებოდნენ. თუ ციკლის იტერაციები დამოუკიდებელია, მაშინ არეული თანმიმდევრობით შესრულების შემთხვევაშიც შედეგი არ შეიცვლება. მაგალითად,

```
for (i=0; i<n; i++) {
   a=2*c+u[i];
   if(a!=0)
     b=4./a;
   else
     b=0;
   x[i]=(a+b)/2.*h;
   y[i]= x[i]+(a-b)/2*h;
}
```

ამ ციკლის იტერაციები ერთმანეთისგან დამოუკიდებელია და ის შეიძლება შესრულდეს პარალელურად. მართლაც, ყველა ცვლადის მნიშვნელოდა, რომელიც გამოითვლება ციკლის იტერაციაზე ნებისმიერი i-თვის გამოიყენება იმავე იტერაციაზე. თუ ამ ფრაგმენტში შევიტანთ ოდნავ ცვლილებას, მაგალითად,

```
x[0]=0; y[0]=0;
for (i=1; i<n; i++) {
   a=2*c+u[i];
   if(a!=0)
     b=4./a;
   else
     b=0;
   x[i]=(a+b)/2.*h;
   y[i]= x[i-1]+(a-b)/2*h;
}
```

ციკლის იტერაციები აღარ იქნება დამოუკიდებელი, რადგანაც y[i]-ის გამოსათვლელად საჭიროა x[i-1] ანუ, წინა იტერაციაზე გამოთვლილი x[i]-ის მნიშვნელობა. თუ ციკლის იტერაციებს შევასრულებთ განსხვავებული თანმიმდევრობით y მასივის ელემენტების მნიშვნელობები იქნება მცდარი.

იტერაციების დამოკიდებულება შეიძლება იყოს არამარტო მასივის ელემენტების გამოთვლის დროს, განვიხილოთ ასეთო ფრაგმენტი:

```
c=12.1;
for (i=0; i<n; i++) {
   if(a!=0)
     b=4./a+c;
   else
     b=10;
   c=(a+b)/2.*h;
}
```

აქაც ციკლის იტერაციები დამოუკიდებელი არ არის. ამ დამოკიდებულებას ქმნის ცვლადი c. თავდაპირველად მისი მნიშვნელოდა განისაზღვერაბ ციკლის გარეთ (c=12.1). მაგრამ შემდეგ, ყოველ i-ურ იტერაციაზე გამოიყენება c-ს ის მნიშვნელობა, რომელიც გამოთვლილი იყო i-1 იტერაციაზე. ამიტომ აქაც ციკლის იტერაციებს არეული თანმიმდევრობით ვერ შევასრულებთ.

არსებობს სიტუაციები, როდესაც ციკლის იტერაციები დამოუკიდებელი არ არის, მაგრამ ისინი მაინც შეიძლება შესრულდნენ ნებისმიერი თანმიმდევრობით. ამის მაგალითია ჩვენს მიერ განხილული მასივის ელემენტების ჯამის პოვნის ალგორითმი:

```
s=0;
for(i=0; i< n; i++) {
   s=s+x[i];
}
```

მართალია, ამ ფრაგმენტში i-ურ იტერაციაზე s-ის გამოთვლისთავის გამოიყენება s-ის მნიშვნელობა, რომელიც გამოთვლილი იყო i-1 იტერაციაზე, მაგრამ ოპერაციის (შეკრების ოპერაცია) თვისებიდან გამომდინარე არავითარი მნიშვნელობა არა აქვს რა თანმიმდევრობით შეიკრიბება მასივის ელემენტები, ამიტომ იტერაციები შეიძლება შესრულდეს ნებისმიერი თანმიმდევრობით. იგივე სიტუაცია გვექნებოდა შეკრების მაგივრად რომ გვქონდეს გამრავლების ან გამოკლების ოპერაცია. მაგრამ s-ის გამოთვლის ოპერატორი რომ ყოფილიყო სხვანაირი, მაგალითად ასეთი:

```
s=0;
for(i=0; i< n; i++) {
   if(s+x[i]!=0)
     s=1./(s+x[i]);
   else
     s=0;
}
```

ამ შემთხვევაში იტერაციების შესრულების თანმიმდევრობას აქვს მნიშვნელობა. სხვდასხვა თანმიმდევრობით შესრულების დროს სხვადასხვა შედეგს მივიღებთ.

იტერაციების თანმიმდევრობას ვერ შევცვლით მაშინაც, როდესაც ჯამის გამოთვლის დროს გვჭირდება პირველი 2 ელემენტის ჯამი, 3 ელემენტის ჯამი, 4 ელემენტის ჯამი და ა.შ. მაგალითად

```
s=0;
for(i=0; i<n; i++) {
   s=s+x[i];
   u[i]=s*h;
}
```

თუ ამ ფრაგმენტში ციკლის იტერაციებს შევასრულებთ განსხვავებული თანმიმდევრობით, ვიდრე for ოპერატორით არის განსაზღვრული, შედეგი იქნება მცდარი, რადგანაც ოპერატორის სემანტიკიდან გამომდინარე, i-ურ იტერაციაზე საჭიროა რომ s-ში იყოს x-ის პირველი i ელემენტის ჯამი. იტერაციების თანმიმდევრობის დარღვევის შემთხვევაში კი ეს ასე არ იქნება.

5.2. დამოუკიდებელი იტერაციების მქონე ციკლის გაპარალელება

თუ ციკლის იტერაციებს შორის დამიკიდებულება არ არის მაშინ მისი გაპარალელება ძალიან მარტივია. ეს ციკლი უნდა მოვაქციოთ პარალელურ უბანში (omp parallel დირექტივის მოქმედების ზონაში) და თვითონ ციკლი მოვნიშნოთ პარალელურად (omp for დირექტივით). ყურადღება უნდა მივაქციოთ ცვლადების საერთოდ და ლოკალურად (კერძოდ) გამოცხადებას. უნდა ვიხელმძღვანელოთ ზოგადი პრინციპით: ის ცვლადები, რომელთა მნიშვნელობა ციკლის შიგნით არ იცვლება უნდა გამოვაცხადოთ საერთო (shared) ცვლადად, ის არაინდექსიანი ცვლადები, რომელთა მნიშვნელობები იცვლება ციკლის შიგნით (მათ შორის ციკლის მთვლელი) უნდა გამოვაცხადოთ კერძო (private) ცვლადად, ხოლო ის მასივები, რომელთა ელემენტებიც იცვლებიან, მაგრამ ყველა იტერაცია ცვლის განსხვავებული ინდექსიან ელემენტს (ამ შემთხვევაში ეს ასე იქნება აუცილებლად, რადგან წინააღმდეგ შემთხვევაში ციკლის იტერაციები დამოუკიდებლად ვერ შესრულდება) უნდა გამოცხადდნენ საერთო ცვლადებად. ზემოთ მოყვანილი ფრაგმენტი გაპარალელდეა ასე

```
#pragma omp parallel private(i,a,b) shared(n,c,u,h);
{
  .  .  .
#pragma omp for
     for (i=0; i<n; i++) {
        a=2*c+u[i];
        if(a!=0)
          b=4./a;
        else
          b=0;
        x[i]=(a+b)/2.*h;
        y[i]= x[i]+(a-b)/2*h;
     }
   .   .   .
}
```

რაც შეეხება ციკლის განაწილების სქემას, იგი დამიკიდებულია იმაზე რამდენად თანაბარი დრო სჭირდება იტერაციების შესრულებას. თუ დაახლოებით თანაბარი დროა საჭიროა უმჯობესია static განაწილება, თუ არა – dynamic ან guided.

5.3. რედუქციის ოპერაციის შემცველი ციკლის გაპარალელება.

ზოგად შემთხვევაში რედუქციული ოპერაცია არის ისეთი ოპერაცია, რომლის არგუმენტი არის ვექტორი (მასივი), ხოლო შედეგი – სკალარული რიცხვი. ასეთი ოპერაციის მაგალითებია მასივის ელემენტების ჯამი, მაქსიმუმი, მინიმუმ და ა.შ.

რედუქცული ოპერაციის შემთხვევა ჩვენ განხილული გვაქვს (მასივის ელემენტების ჯამი), ამიტომ დაწვრილიებით არ შევჩერედებით. თუ მოცემული გვაქვს ასეთი ტიპის ფრაგმენტი:

```
s=0;
for(i=0; i< n; i++) {
   s=s+x[i];
}
```

მისი გაპარელელებული სახე იქნება შემდეგი

```
s=0
#pragma omp parallel private(i)reduction(+:s)
{
#pragma omp for
    for (i=0;i<n; i++)
       s+=x[i];
 .   .   .
}
```

თუ გამოიყენება ისეთი ოპერაცია, რომელიც არ მოიცემა reduction ოპციაში, მაგალითად, ვეძებთ მასივში მაქსიმუმს:

```
t=x[0];
for(i=1; i< n; i++) {
   if(t<x[i]) t=x[i];
}
```

მის გასაპარალელებლად უნდა გამოვიყენ ლოკალური მაქსიმუმის ცვლადი და კრიტიკული უბანი

```
t=x[0];
#pragma omp parallel private(i,loc_t) {
loc_t=x[0];
#pragma omp for
    for (i=0;i<n; i++)
       if(loc_t<x[i]) loc_t=x[i];
 #pragma critical
     if(t<loc_t) t=loc_t;
}
```

5.4. დამოკიდებული იტარციების მქონე ციკლის გაპარალელების ზოგიერთი შემთხვევა.

თუ ციკლის იტერაციები დამოკიდებულია, ანუ მომდევნო იტერციაზე გამოიყენება წინა იტერაციაზე გამოთვლილი მნიშვნელობა, პირველი რიგში უნდა დავადგინოთ რომელი ცვლადი (ან ცვლადები) იწვევს ამ დამოკიდებულიებას და ვეცადოთ გამოსახულება გარდავქმნათ ისე, რომ ეს დამოკიდებულება მოისპოს. შეიძლება ეს ყოველთვის არ მოხერხდეს, მაგრამ არსებობს რამდენიმე სტანდარტული სიტუაცია, როდესაც ასეთი გარდაქმნა შესაძლებელია.

განვიხილოთ ასეთი ფრაგმენტი:

```
s=0; x=a;
for(i=0; i<n; i++) {
   s=s+(f(x)+f(x+h))/2*h;
   x=x+h;
}
```

სადაც f არის რაიიმე ფუნქცია. ამ ფრაგმენტში იტერაციების დამოკდებულებას იწვევს ცვლადი x. ეს გამოსახულება (x=x+h) თითქოს ჰგავს ჯამის პოვნის გამოსახულებას (s+=x[i]), მაგრამ არის პრინციპული განსხვავება. ჯამის პოვნის დროს გვჭირდება, მხოლოდ ბოლო იტერაციით მიღებული შედეგი, ახლა კი გვჭირდება ყოველ იტერაციაზე იღებული შედეგები. ამიტომ რედუქციული ოპერაციის გამოყენება აქ არ გამოგვადგება. ერთადერთი გამოსავალი არის ის, რომ როგორმე ისე გარდავქმნათ ეს ფრაგმენტი, რომ იტერაციული დამოკიდებულება მოისპოს. აქ უნიკალური რეცეპტია არ არსებობს. ყოველ კონკრეტულ შემთხვევაში ალგორითმის ანალიზის საფუძველზე კონკრეტული გადაწყვეტილების მიღებაა საჭირო.

თუ გავაანალიზებთ ამ ფრაგმენტს, ადვილი მისახვედრია, რომ i=0-თვის x-ის მნიშვნელობა არის a, x=1-თვის – a+h, i=2–თვის – a+2*h და ა.შ., ზოგადად, i-ურ იტერაციაზე x-ის მნიშვნელიბა არის a+i*h. ეს ფრაგმენიი გადავწეროთ ასე:

```
s=0;
for(i=0; i<n; i++) {
   x=x+i*h;
   s=s+(f(x)+f(x+h))/2*h;
}
```

იტერაციებში x-ით დამოკიდებულება მოისპო. მართალია დარჩა s-ით დამოკიდებულება, მაგრამ ეს არის რედუქციული შეკრების ოპერაცია (მართალია, აქ არის არის ცხადად მასივი, მაგრამ (f(x)+f(x+h))/2*h შეგვიძლია განვიხილოთ როგორც მასივის ელემენტები, რადგანაც x არის დამოკიდებული i – ზე და ამიტომ თვითონ ეს გამოსახულებაც დამოკიდებული იქნება i-ზე, ანუ შეგვიძლია განვიხილოთ მასივის ელემენტებად). ასეთი ფრაგმენტის გარდაქმნა კი უკვე ვიცით,

```
s=0;
```

```
#pragma omp parallel private(i,x)reduction(+:s)
{
#pragma omp for
    for (i=0;i<n; i++) {
       x=x+i*h;
       s+=(f(x)+f(x+h))/2*h;
    }
}
```

ამ ფრაგმენტის გარდაქმნაში ჩვენ დაგვეხმარა ის, რომ x ყოველ იტერაციაზე იცვლებოდა ერთი და იმავე ბიჯით (h-ით). წინააღმდეგ შემთხვევაში ასეთ გარდაქმნას ვერ ჩავატარებდით. განვიხილოთ ახლა სწორედ ასეთ შემთხვევა:

```
     s=0; x=a;
     for(i=0; i<n; i++) {
        s=s+(f(x)+f(x+h[i]))/2*h;
        x=x+h[i];
     }
```

აქ h არის უკვე მასივი და ყოველ იტერაციაზე x იზრდება h-ის შესაბამისი ელემენტით. აქ შეგვიძლია მივმართოთ ასეთ ხერხს. x წარმოვადგინოთ როგორც მასივი, რომლის i-ური ელემენტი იქნება მისი ის მნიშვნელობა, რომელიც გამოიყენება i-ურ იტერაციაზე. ციკლი გავყოთ ორ ციკლად. პირველ ციკლში გამოვთვალოთ x მასივი. მეორე ციკლში s.

```
     s=0; x[0]=a;
     for(i=1; i<n; i++) {
       x[i]=x[i-1]+h[i-1];
     }

     for(i=1; i<n; i++) {
        s=s+(f(x[i])+f(x[i]+h[i]))/2*h;
     }
```

ამ ორი ციკლიდან პირველ არ პარალელდება, ხოლო მეორის გაპარალელება ვიცით.

```
s=0; x[0]=a;
for(i=1; i<n; i++) {
   x[i]=x[i-1]+h[i-1];
}

#pragma omp parallel private(I,x)reduction(+:s)
{
#pragma omp for
    for (i=0;i<n; i++) {
       s+=(f(x[i])+f(x[i]+h[i]))/2*h;
    }
}
```

ეს ხერხი არის საკმაოდ მიღებული საშუალება ციკლების გაპარალელებისთვის. ციკლში უნდა განვაცალკევოთ ის ოპერატორები, რომლებიც იწვევენ დამოკიდებულებას და რომლებიც არ იწვევენ იტერაციების დამოკიდებულებას. განვათავსოთ ისინი ცალ-ცალკე ციკლებში. ამ დროს შეიძლება სკალარული ცვლადების შეცვლა დაგვჭირდეს მასივებით. ციკლი, რომელშიც განვათავსეთ ის ოპერატორები, რომლებიც არ იწვევენ დამოკიდებულებას იქნება პარალელური და შეგვიძლია გავაპარალელოთ. განვიხილოთ მაგალითები.

გვაქვს ზემოთმოყვანილი ფრაგმენტი

```
x[0]=0; y[0]=0;
for (i=1; i<n; i++) {
   a=2*c+u[i];
   if(a!=0)
     b=4./a;
   else
     b=0;
   x[i]=(a+b)/2.*h;
   y[i]= x[i-1]+(a-b)/2*h;
}
```

აქ დამოკიდებულებას იწვევს x მასივი. y-ის გამოთვლას შევძლებთ პარალელურად თუ წინასწარ გვეცოდინება x მასივის ელემენტები. გავყოთ ეს ციკლი ორ ნაწილად. პირველში გამოვთავლი მასივი, მეორეში – y.

```
x[0]=0; y[0]=0;
for (i=1; i<n; i++) {
   a=2*c+u[i];
   if(a!=0)
     b=4./a;
   else
     b=0;
   x[i]=(a+b)/2.*h;
}

for (i=1; i<n; i++) {
   a=2*c+u[i];
   if(a!=0)
     b=4./a;
   else
     b=0;
   y[i]= x[i-1]+(a-b)/2*h;
}
```

როგორც ვხედავთ, ორივე ციკლში მოგვიხდა a და b ცვლადების გამოთვლასთან დაკავშირებული ფრაგმენტის გამეორება. მეორე ციკლში დამოკიდებულება აღარ არის ამიტომ ის შეიძლება გავაპარალელოთ. მაგრამ დამოკიდებულება აღარ არის პირველ ციკლშიც და მისი გაპარალელებაც შესძლებელია.

```
x[0]=0; y[0]=0;
#pragma omp parallel private(i,a,b)
{
#pragma omp for
     for (i=1; i<n; i++) {
        a=2*c+u[i];
        if(a!=0)
          b=4./a;
        else
          b=0;
        x[i]=(a+b)/2.*h;
     }
#pragma omp for
     for (i=1; i<n; i++) {
        a=2*c+u[i];
        if(a!=0)
          b=4./a;
        else
          b=0;
        y[i]= x[i-1]+(a-b)/2*h;
     }
}
```

როგორც ვხედავთ, ციკლის გაწყვეტით მივიღეთ ორი ციკლი, რომლებიც ცალ-ცალკე პარალელდებიან. აქ სკალარული ცვლადის მასივით შეცვლა არ დაგვჭირდა. თუმცა შეგვეძლო a და b ცვლადების ნაცვლად შემოგვეღო მასივები და დაგვეთვალა ისინი მხოლოდ პირველ ციკლში

```
     x[0]=0; y[0]=0;
#pragma omp parallel private(i,a,b)
{
#pragma omp for
     for (i=1; i<n; i++) {
        a[i]=2*c+u[i];
        if(a!=0)
          b[i]=4./a;
        else
          b[i]=0;
        x[i]=(a[i]+b[i])/2.*h;
     }
#pragma omp for
     for (i=1; i<n; i++) {
        y[i]= x[i-1]+(a[i]-b[i])/2*h;
     }
}
```

კიდევ ერთი ფრაგმენტი:

```
t1=a; t2=b;
for (i=0;i<n;i++) {
```

```
     x[i]=(t1+t2)/2;
     y[i]= (t1+t2)/2;
     t1=t1+w[i];
     t2=t2+w[i];
}
```

ამ ფრაგმენტის გაპარალელებაც არ უნდა იყოს რთული. უნდა გავყოთ ორ ციკლად და დაგვჭირდება t1 ფა t2 მასივების შემოღება

```
t1[0]=a; t2[0]=b;
for (i=1;i<n;i++) {
     t1[i]=t1[i-1]+w[i-1];
     t2[i]=t2[i-1]+w[i-1];
}

#pragma omp parallel private(i)
{
#pragma omp for
  for (i=0;i<n;i++) {
     x[i]=(t1[i]+t2[i])/2;
     y[i]= (t1[i]+t2[i])/2;
  }
}
```

თუ გვაქვს ასეთი ფრაგმენტი

```
     x[0]=0; y[0]=0;
     for (i=1; i<n; i++) {
        a=2*c+u[i];
        if(a!=0)
          b=4./a;
        else
          b=0;
        x[i]=y[i-1]+(a+b)/2.*h;
        y[i]= x[i-1]+(a-b)/2*h;
}
```

ამ ფრაგმენტში არის დამიკიდებულება x და y მასივებს შორის. x-ის გამოსათვლელად საჭიროა y-ის მნიშვნელობა, რომელიც გამოთვლილია წინა იტერაციაზე, ხოლო y-ის გამოსათვლელად საჭიროა წინდა იტერაციაზე გამოთვლილი x-ის მნიშვნელობა. ერთადერთ აქ ჩვენ შეგვიძლია a და b-ს გამოთვლა გავაპარალელოთ.

```
x[0]=0; y[0]=0;
#pragma omp parallel private(i)
     for (i=1; i<n; i++) {
        a[i]=2*c+u[i];
        if(a!=0)
          b[i]=4./a;
        else
          b[i]=0;
```

```
      }
}

for (i=1; i<n; i++) {
      x[i]= y[i-1]+ (a[i]+b[i])/2.*h;
      y[i]= x[i-1]+(a[i]-b[i])/2*h;

}
```

ხოლო თუ გვექნებოდა ასეთი ფრაგმენტი

```
      x[0]=0; y[0]=0;
      for (i=1; i<n; i++) {
         x[i]=y[i-1]+(a+b)/2.*h;
         y[i]= x[i-1]+(a-b)/2*h;
      }
```

იგი არ გაპარალელდება არცერთი მეთოდით. აქ a და b წინასწარ არის გამოთვლილი, მათი მნიშვნელობა არ იცვლება და ციკლში აღარ გამოითვლება.