ჯგუფური პროექტი

მარიამ დოლიაშვილი

სოფიო ჩიქვინიძე

ხელმძღვანელი: სრული პროფესორი, კობა გელაშვილი

ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი, ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი, კომპიუტერული მეცნიერების მიმართულება

2014-2015 სასწავლი წელი

სარჩევი

ანოტაცია

პროექტის მიზანია პარალელური ალგორითმების იმპლემენტაცია სხვადასხვა ტექნოლოგიების გამოყენებით და მიღებული შედეგების ანალიზი.

ნაშრომში განხილულია პარალელურ გამოთვლებთან დაკავშირებული ძირითადი თეორიული საკითხები, მოყვანილია მატრიცების გადამრავლებისა და სორტირების პარალელური ალგორითმები, მათი ანალიზი იდეალური პარალელური კომპიუტერისათვის, იმპლემენტაცია სხვადასხვა ბიბლიოთეკების გამოყენებით და შედარებულია მიღებული შედეგები. ასევე, განხილულია პარალელური გამოთვლებისათვის უსაფრთხო მონაცემთა სტრუქტურების დიზაინთან დაკავშირებული პრობლემები stack მონაცემთა სტრუქტურის მაგალითზე.

შესავალი

პარალელური კომპიუტერები - კომპიუტერები რამდენიმე პროცესორით - უფრო და უფრო ხშირად გვხვდება და სხვადასხვა ფასისა და წარმადობის არსებობს. შედარებით იაფ პარალელურ პერსონალურ კომპიუტერებში გამოიყენება მრავალბირთვიანი პროცესორები. თითოეული ბირთვი სრულუფლებიანი პროცესორია, რომელსაც აქვს წვდომა საზიარო მეხსიერებაზე. საშუალო დონეზე არის კლასტერები, რომლებიც აგებულია ინდივიდუალური კომპიუტერებისგან, ხშირად ჩვეულებრივი პერსონალური კომპიუტერების კლასის მანქანებისგან, რომლებიც ერთმანეთს ქსელით უკავშირდება. ყველაზე ძვირადღირებულია სუპერკომპიუტერები, რომლებშიც ხშირად გამოიყენება სპეციალურად შედგენილი არქიტექტურა და ქსელები იმისათვის, რომ მაქსიმალურად მაღალი იყოს წარმადობა, რომელიც იზომება ერთ წამში შესრულებული ინსტრუქციების რაოდენობით.

მრავალპროცესორიანი კომპიუტერები, ამა თუ იმ ფორმით, უკვე ათწლეულებია არსებობს. მიუხედავად ამისა, არ არსებობს პარალელური არქიტექტურის მოდელი, რომელიც სტანდარტად არის მიჩნეული. ზოგი პარალელური კომპიუტერი იყენებს საზიარო მეხსიერებას. მეხსიერების ამ მოდელის გამოყენებისას თითოეულ პროცესორს აქვს პირდაპირი წვდომა მეხსიერების ნებისმიერ ნაწილზე. სხვა პარალელურ კომპიუტერებში განაწილებული (დისტრიბუციული) მეხსიერება გამოიყენება, სადაც თითოეულ პროცესორს საკუთარი მეხსიერება აქვს და იმისათვის, რომ ერთმა პროცესორმა მოიპოვოს წვდომა მეორე პროცესორის მეხსიერებაზე, პროცესორებს შორის შესაბამისი შეტყობინება უნდა გაიგზავნოს. თანამედროვე პერსონალური კომპიუტერები საზიარო მეხსიერების მოდელს იყენებენ და ზოგადი ტენდენციაც საზიარო მეხსიერების მოდელისკენაა.

საზიარო მეხსიერების მქონე პარალელური კომპიუტერების დაპროგრამების გავრცელებული მეთოდია სტატიკური thread-ების გამოყენება. thread-ები, რომლებიც იყენებს საზიარო მეხსიერებას, წარმოადგენს ვირტუალური პროცესორების პროგრამულ აბსტრაქციას. თითოეულ thread-ს აქვს საკუთარი პროგრამული მთვლელი და შეუძლია შეასრულოს კოდი სხვა thread-ებისგან დამოუკიდებლად. ოპერაციული სისტემა ტვირთავს thread-ს პროცესორში შესრულებისათვის და წყვეტს მის შესრულებას როდესაც სხვა thread-ის შესრულების დრო დგება. მიუხედავად იმისა, რომ ოპერაციული სისტემა აძლევს პროგრამისტებს thread-ების შექმნისა და განადგურების უფლებას, ეს ოპერაციები საკმაოდ ნელია. სწორედ ამიტომ, ბევრ აპლიკაციაში thread-ები მთელი გამოთვლის განმავლობაში არსებობენ. ამის გამო ვუწოდებთ მათ სტატიკურს.

საზიარო მეხსიერების მქონე პარალელური კომპიუტერის დაპროგრამება სტატიკური thread-ების გამოყენებით საკმაოდ რთულია და მიდრეკილია შეცდომებისაკენ. ერთ-ერთი მთავარი სირთულეა thread-ებს შორის სამუშაოს თანაბარი გადანაწილება დინამიურად. ნებისმიერი პროგრამული უზრუნველყოფისთვის, გარდა, შესაძლებელია, უმარტივესი აპლიკაციებისა, პროგრამისტმა უნდა გამოიყენოს რთული საკომუნიკაციო პროტოკოლები იმისათვის, რომ იმპლემენტაცია გაუკეთოს დამგეგმავს, რომელიც გაანაწილებს სამუშაოს. ამ

სირთულეებისთვის თავის ასარიდებლად შეიქმნა პლატფორმები (concurrency platforms), რომლებიც მომხმარებელს სთავაზობს პროგრამული უზრუნველყოფის ფენას, რომელიც გეგმავს, აკონტროლებს და მართავს პარალელური გამოთვლების რესურსებს. ასეთი პლატფორმები არსებობს როგორც პარალელური პროგრამირების ენების, ასევე ბიბლიოთეკების სახით.

დინამიური multithreaded პროგრამირება

დინამიური multithreaded პროგრამირება საშუალებას გვაძლევს გამოვიყენოთ პარალელიზმი საკომუნიკაციო პროტოკოლებსა და სამუშაოს პროცესორებს შორის განაწილებაზე ფიქრის გარეშე. ამ საკითხებს აგვარებს დამგეგმავი(scheduler), რომელსაც შეიცავს პროგრამული პლატფორმა(concurrency platform). დინამიური multithreaded პროგრამირების გარემოების უმეტესობა მხარს უჭერს ჩადგმულ პარალელიზმს(nested parallelism) და პარალელურ for ციკლს. პარალელურ for ციკლში იტერაციები ერთდროულად სრულდება სხვადასხვა პროცესორებზე, ხოლო ჩადგმული პარალელიზმი საშუალებას იძლევა ბრძანებების რაღაც ნაკრები გადავცეთ შესასრულებლად სხვა thread-ს და ისინი ძირითადი thread-ის პარალელურად შესრულდება (spawn). ამის დახმარებით პროგრამისტს ძირითადად პარალელიზმის ლოგიკაზე უწევს ფიქრი.
multithreaded ალგორითმის ფსევდოკოდი შეიძლება მივიღოთ ჩვეულებრივ, სერიული ალგორითმის ფსევდოკოდში შემდეგი საკვანძო სიტყვების დამატებით:

- spawn – ფსევდოკოდში ამ საკვანძო სიტყვას მოსდევს იმ პროცედურის გამოძახება, რომელიც შესრულდება შვილ thread-ში მშობელი thread-ის პარალელურად. თუმცა, შესაძლებელია გამოძახებული პროცედურა არ შესრულდეს პარალელურად დამგეგმავის გადაწყვეტილების მიხედვით, რომელიც დამოკიდებულია აპარატურულ უზრუნველყოფაზე (პროცესორების რაოდენობაზე).
- sync - აღნიშნავს, რომ მშობელი პროცედურა უნდა დაელოდოს ყველა შვილის დასრულებას სანამ შეასრულებს იმ ბრძანებებს, რომლებიც sync-ის შემდეგ არის დაწერილი. მაშინაც კი, როცა sync ფსევდოკოდში ცხადად არ არის მითითებული, ნებისმიერი პროცედურა ასრულებს sync-ს return ბრძანების შესრულებამდე იმაში დასარწმუნებლად, რომ მისი ყველა შვილის მუშაობა დამთავრებულია.
- parallel - ამ სიტყვის მიწერით for ციკლის წინ ვიღებთ პარალელურ for-ს.

ფსევდოკოდიდან ამ საკვანძო სიტყვების წაშლის შედეგად მივიღებთ სერიულ ალგორითმს, რასაც ალგორითის სერიალიზაცია ეწოდება. ანუ, სერიული ალგორითმი არის ჩვეულებრივი ალგორითმი, რომელიც ხსნის იმავე ამოცანას, რომელსაც პარალელური ალგორითმი.

პარალელური ალგორითმების მუშაობა იდეალურ პარალელურ კომპიუტერზე

ამ თავში პარალელური ალგორითმების მუშაობას შევისწავლით იდეალურ პარალელურ კომპიუტერზე, რომელიც შედგება პროცესორების სიმრავლისა და საზიარო მეხსიერებისაგან, რომელიც მიმდევრობით მდგრადია (sequentially consistent). მიმდევრობით მდგრადობაში იგულისხმება, რომ საზიარო მეხსიერება, რომელიც სინამდვილეში ასრულებს არაერთ ოპერაციას პარალელურად, იძლევა იმავე შედეგს, რასაც მივიღებდით, თუ ყოველ ჯერზე მხოლოდ ერთი ბრძანება შესრულდებოდა. იდეალურ პარალელურ კომპიუტერში ყველა პროცესორს ერთნაირი სიმძლავრე აქვს და უგულებელყოფილია დამგეგმავის მუშაობის დრო, რასაც პრაქტიკაში მართლაც მცირე ღირებულება აქვს თუ ალგორითმი არის საკმარისად პარალელური (პარალელურობის ცნებას შემდეგ განვმარტავთ).

შესრულების დროის გამოთვლა

პარალელური ალგორითმების ასიმპტოდური შეფასებისთვის იდეალურ პარალელურ კომპიუტერზე გამოვიყენებთ ორ საზომს:

1. დრო, რომელიც საჭიროა ამ ალგორითმის ერთ პროცესორზე შესასრულებლად. აღინიშნება T_1-ით.
2. დრო, რომელიც საჭიროა ალგორითმის შესასრულებლად პროცესორების შეუზღუდავი რაოდენობით არსებობის შემთხვევაში. აღინიშნება T_∞-ით.

რეალური მუშაობის დრო დამოკიდებულია არამარტო ამ ორ საზომზე, არამედ იმაზეც, თუ რამდენი პროცესორი გვაქვს და როგორ მუშაობს დამგეგმავი. დროს, რომელიც საჭიროა ალგორითმის P პროცესორზე შესასრულებლად აღვნიშნავთ T_p-თი.

T_1 და T_∞ გვაძლევს შემდეგ ქვედა ზღვრებს T_p-სთვის:

- P პროცესორიან იდეალურ პარალელურ კომპიუტერს ერთ ბიჯზე არაუმეტეს P ერთეული სამუშაოს შესრულება შეუძლია. შესაბამისად, T_p დროში - არაუმეტეს PT_p ერთეულის. რადგანაც სულ T_1 ერთეული სამუშაოა შესასრულებელი, გვაქვს შემდეგი: $PT_p \geq T_1$. უტოლობის ორივე მხარის P-ზე გაყოფის შედეგად მივიღებთ:
$$T_p \geq T_1 / P \qquad (1)$$
- P პროცესორიანი იდეალური პარალელური კომპიუტერი ვერ იმუშავებს იმ მანქანაზე სწრაფად, რომელსაც პროცესორები შეუზღუდავი რაოდენობით აქვს:
$$T_p \geq T_\infty \qquad (2)$$

ჩვენ განვსაზღვრავთ გამოთვლის P პროცესორზე შესრულების აჩქარებას (speedup), როგორც შეფარდებას T_1 / T_p, რომელიც გვიჩვენებს, თუ რამდენჯერ სწრაფად სრულდება გამოთვლები P რაოდენობის პროცესორზე ერთ პროცესორთან შედარებით. (1)-დან გამომდინარე, $T_1 / T_p \leq P$, ანუ აჩქარება P პროცესორის გამოყენებისას შესაძლებელია იყოს მაქსიმუმ P-ს ტოლი. როდესაც აჩქარება არის წრფივი პროცესორების რაოდენობის მიმართ, ანუ როდესაც $T_1 / T_p = \Theta(P)$, ვიტყვით, რომ გვაქვს წრფივი აჩქარება(linear speedup), ხოლო როცა $T_1 / T_p = P$ – იდეალური წრფივი აჩქარება (perfect linear speedup).

T_1-ის შეფარდება T_∞-სთან გვაძლევს multithreaded გამოთვლების პარალელურობას (parallelism). როგორც შეფარდება, პარალელურობა განსაზღვრავს სამუშაოს საშუალო რაოდენობას, რომელიც შეიძლება შესრულდეს პარალელურად პარალელური ალგორითმის მუშაობის ყოველ საფეხურზე. როგორც ზედა ზღვარი, პარალელურობა განსაზღვრავს მაქსიმალურ შესაძლო აჩქარებას, რასაც შეიძლება მივაღწიოთ პროცესორების შეუზღუდავი რაოდენობით არსებობის შემთხვევაში. საბოლოოდ, პარალელურობა შემოსაზღვრავს იდეალური წრფივი აჩქარების მიღწევის შესაძლებლობას. უფრო კონკრეტულად, თუ პროცესორების რიცხვი მეტია პარალელურობის მნიშვნელობაზე, გამოთვლით პროცესს არ შეუძლია მიაღწიოს იდეალურ წრფივ აჩქარებას. ამის დასანახად დავუშვათ, რომ $P > T_1 / T_\infty$, რა შემთხვევაშიც (2)-დან გამომდინარეობს, რომ აჩქარება აკმაყოფილებს შემდეგ პირობას: $T_1 / T_p \leq T_1 / T_\infty < P$. უფრო მეტიც - თუ იდეალურ პარალელურ კომპიუტერში პროცესორების რაოდენობა P ბევრად აღემატება პარალელურობის მნიშვნელობას, ანუ თუ $P \gg T_1 / T_\infty$, მაშინ $T_1 / T_p \ll P$, ანუ აჩქარება პროცესორების რაოდენობას ბევრად ჩამოუვარდება. სხვა სიტყვებით რომ ვთქვათ, რაც უფრო მეტად გადააჭარბებს პროცესორების რაოდენობა პარალელურობის მნიშვნელობას, მით ნაკლებად იდეალურ აჩქარებას მივიღებთ.

მრავალნაკადიანი გამოთვლის პარალელურ სიჭარბეს(slackness) იდეალურ პარალელურ კომპიუტერზე P პროცესორით განვსაზღვრავთ, როგორც შეფარდებას $(T_1 / T_\infty) / P = T_1 / (T_\infty P)$. პარალელური სიჭარბე გვიჩვენებს, თუ რამდენად აღემატება პარალელურობა პროცესორების რაოდენობას. თუ სიჭარბე ნაკლებია ერთზე, არ შეგვიძლია მივაღწიოთ იდეალურ წრფივ აჩქარებას, რადგან რაც მეტად მცირდება სიჭარბე 1-დან 0-მდე ალგორითმის აჩქარება შორდება იდეალურ მაჩვენებელს. თუმცა, თუ სიჭარბე მეტია 1-ზე, შემზღუდველი ფაქტორი არის ის, თუ რა სამუშაოს შესრულება შეუძლია ერთ პროცესორს დროის ერთეულში. რაც უფრო მეტად აღემატება სიჭარბე ერთს, მით უფრო ახლოსაა აჩქარება იდეალურ წრფივთან კარგი დამგეგმავის არსებობის შემთხვევაში.

დაგეგმვა (Scheduling)

შესრულების მაღალი ხარისხის მისაღწევად დიდი მნიშვნელობა აქვს concurrency platform-ის დამგეგმავს. პრაქტიკულად, დამგეგმავი ანაწილებს გამოთვლებს სტატიკურ thread-ებში და ოპერაციული სისტემა ანაწილებს thread-ებს პროცესორებზე. რადგანაც ეს ბოლო ფენა არაა

ჩვენთვის მნიშვნელოვანი, ვთვლით, რომ გადანაწილების პროცესი დამოკიდებულია მთლიანად დამგეგმავზე.

multithreaded დამგეგმავს უნდა შეეძლოს გამოთვლების დაგეგმვა thread-ების მუშაობის დაწყების და დასრულების დროების წინასწარი ცოდნის გარეშე - ის უნდა მუშაობდეს ონ-ლაინ რეჟიმში.

სიმარტივისათვის, შევისწავლოთ ონ-ლაინ ცენტრალიზირებული დამგეგმავი, რომელსაც აქვს ინფორმაცია გამოთვლების გლობალური მდგომარეობის შესახებ დროის ყოველ მომენტში. უფრო კონკრეტულად, ვაანალიზებთ ხარბ დამგეგმავებს, რომლებიც მაქსიმალურად ტვირთავენ პროცესორებს დროის ყოველ მომენტში. როდესაც შესასრულებელია სულ მცირე P გამოთვლა, ვიტყვით, რომ გვაქვს სრული ბიჯი (complete step). წინააღმდეგ შემთხვევაში გვაქვს არასრული ბიჯი.

(1)-დან გამომდინარე, საუკეთესო მუშაობის დრო, რაც შეიძლება ვიმედოვნოთ P პროცესორისთვის არის $T_p = T_1 / P$, (2)-დან გამომდინარე კი - $T_p = T_\infty$. შემდეგი თეორემა ამტკიცებს, რომ ხარბი დამგეგმავი კარგია იმ მხრივ, რომ მისი ზედა ზღვარი არის ამ ორი ქვედა ზღვრის ჯამი.

თეორემა 1:

იდეალურ პარალელურ კომპიუტერზე P პროცესორით ხარბი დამგეგმავი ასრულებს multithreaded გამოთვლას შემდეგ დროში:

$T_p \leq T_1 / P + T_\infty$.

დამტკიცება:

პირველ რიგში, განვიხილოთ სრული ბიჯი. ყოველ სრულ ბიჯზე P პროცესორი ერთად ასრულებს P სამუშაოს (ვთვლით, რომ ბრძანებების მიმდევრობა, რომელიც არ შეიცავს პარალელური ფუნქციის გამოძახებას, სრულდება ერთეულოვან დროში). დავუშვათ საწინააღმდეგო, რომ სრული ბიჯების რაოდენობა მკაცრად მეტია T_1 / P - ის ქვედა მიახლოებაზე. მაშინ, სრული ბიჯების საერთო სამუშაოს მნიშვნელობა გამოდის სულ მცირე

$P \cdot (T_1 / P + 1) = P \cdot T_1 / P + P = T_1 - (T_1 \bmod P) + P > T_1$

ანუ, მივიღეთ წინააღმდეგობა, რომ P პროცესორი ასრულებს იმაზე მეტ სამუშაოს, ვიდრე საჭიროა გამოთვლის შესასრულებლად, საიდანაც შეგვიძლია დავასკვნათ, რომ სრული ბიჯების რაოდენობა არ აღემატება T_1 / P.

ახლა განვიხილოთ არასრული ბიჯი. ალგორითმის მუშაობის პროცესი წარმოვიდგინოთ, როგორც გრაფი G, და ვთქვათ, რომ G' არის ქვეგრაფი, რომელიც უნდა შესრულდეს არასრული ბიჯის დაწყებისას. G''-ით აღვნიშნოთ ქვეგრაფი, რომელიც შესასრულებელი დარჩება არასრული ბიჯის დასრულების შემდეგ. ყველაზე გრძელი გზა გრაფში აუცილებლად დაიწყება წვეროდან, რომლის შემავალი ხარისხია 0.

შედეგი 1:

ხარბი დამგეგმავის მიერ შესრულებული მრავალნაკადიანი ალგორითმის მუშაობის დრო T_P P პროცესორიან იდეალურ კომპიუტერზე ოპტიმალურ დროსთან (T'_P) შედეგ დამოკიდებულებაშია:

$T_P \leq T'_P$

შედეგი 2:

ზემოთ მოცემული თეორემების პირობებში, თუ $P \ll T_1 / T_\infty$, მაშინ $T_P \approx T_1 / P$, ანუ აჩქარება არის დაახლოებით P.

შეჯიბრის მდგომარეობა

multithreaded ალგორითმი არის დეტერმინისტული, თუ ის აბრუნებს ერთსა და იმავე შედეგს ერთსა და იმავე შემავალ მონაცემებზე მიუხედავად იმისა, თუ როგორ მუშაობს დამგეგმავი. ის არის არადეტერმინისტული, თუ მისი მოქმედება შეიძლება განსხვავდებოდეს სხვადასხვა გაშვებებისას. ხშირად, multithreaded ალგორითმი, რომელიც, ერთი შეხედვით, დეტერმინისტული უნდა იყოს, არ მუშაობს დეტერმინისტულად, რადგან შეიცავს შეჯიბრის მდგომარეობას.

შეჯიბრის მდგომარეობა პარალელურობის ყველაზე დიდი პრობლემაა. შეჯიბრის მდგომარეობით გამწვეული შეცდომების აღმოჩენა ძალიან რთულია.

შეჯიბრის მდგომარეობა წარმოიქმნება, როცა ორ ლოგიკურად პარალელურ ინსტრუქციას წვდომა აქვს ერთსა და იმავე მეხსიერებაზე და ერთი ბრძანება მაინც ცვლის მონაცემებს.

შეჯიბრის მდგომარეობასთან გამკლავების არაერთი მეთოდი არსებობს, მათ შორის mutex-ები და სინქრონიზაცია. ჩვენი მიზნებიდან გამომდინარე, აჯობებს დავრწმუნდეთ, რომ ყველა გამოთვლა, რომელიც სრულდება პარალელურად, არის დამოუკიდებელი, და, შესაბამისად, არ გვაქვს შეჯიბრის მდგომარეობა. პარალელურ for ციკლში ყველა იტერაცია უნდა იყოს დამოუკიდებელი. ასევე, ჩადგმული პარალელიზმის გამოყენებისას, მშობლისა და შვილების მიერ შესასრულებელი კოდი უნდა იყოს დამოუკიდებელი ერთმანეთისაგან. აღსანიშნავია, რომ ალგორითმმა, რომელიც შეჯიბრის მდგომარეობას შეიცავს, შეიძლება სწორი შედეგიც დააბრუნოს. მაგალითად, იმ შემთხვევაში, თუ ორი thread საზიარო მეხსიერებაში ერთსა და იმავე მნიშვნელობას წერს.

multithreaded მატრიცების გადამრავლება

პირველი ალგორითმი, რომლის multithreaded ვარიანტსაც განვიხილავთ, არის მატრიცების გადამრავლება. კერძოდ, განვიხილოთ მატრიცების გადამრავლების ალგორითმი, რომელიც ეფუძნება ციკლების გაპარალელებას შემდეგ ალგორითმში:

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

სერიული for ციკლების პარალელური ვერსიებით ჩავაცვლებით მივიღებთ შემდეგ ალგორითმს:

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  parallel for i = 1 to n
4      parallel for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

შევნიშნოთ, რომ მესამე for ციკლი არ იცვლება პარალელური ვარიანტით. მისი გაპარალელება გამოიწვევდა შეჯიბრის მდგომარეობას და ალგორითმი არასწორ შედეგს დააბრუნებდა.

რადგანაც მოცემული ალგორითმის სერიალიზაციით ვიღებთ ჩვეულებრივ მატრიცების გადამრავლების ალგორითმს, რომლის მუშაობის დროა Θ(n3), P-s-m-m-ის work იქნება Θ(n3), ხოლო span Θ(n). მივიღებთ პარალელურობის მნიშვნელობას, რომელიც უდრის Θ(n3) / Θ(n) = Θ(n2).

დალაგება შერწყმით (merge sort)

შერწყმით სორტირების ალგორითმი იყენებს „გაყავი და იბატონე“ პარადიგმას, რის გამოც მისი გაპარალელება ჩადგმული პარალელიზმის გამოყენებით ბუნებრივია. პარალელური merge sort ალგორითმის ფსევდოკოდი, რომელიც ალაგებს A მასივის [p … r] ქვემასივს ($p \le r$), შემდეგია:

```
MERGE-SORT′(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT′(A, p, q)
4      MERGE-SORT′(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

MERGE პროცედურის ფსევდოკოდი შემდეგია:

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

MERGE-SORT`-ის მუშაობის დროის ანალიზისთვის გავიხსენოთ, რომ MERGE პროცედურას n ელემენტის შერწყმისთვის $\Theta(n)$ დრო სჭირდება. რადგანაც MERGE სერიულია, მისი T_1 და

T_∞ Θ (n)-ის ტოლია. $MS_1`(n)$-ით აღვნიშნოთ MERGE-SORT` -ის $T_1(n)$, რომელიც შემდეგი რეკურენტული ფორმულით გამოითვლება:

$MS_1`(n) = 2 \cdot MS_1`(n / 2) + \Theta(n) = \Theta(n \cdot \lg n)$

რაც ემთხვევა სერიული ალგორითმის მუშაობის დროს. რადგანაც MERGE-SORT` -ის ორი რეკურსიული გამოძახება შეიძლება ერთმანეთის პარალელურად შესრულდეს, $T_\infty$, რომელსაც $MS_\infty`(n)$ - ით აღვნიშნავთ, შემდეგი რეკურსიული ფორმულით გამოითვლება:

$MS_\infty`(n) = MS_\infty`(n / 2) + \Theta(n) = \Theta(n)$.

გამოდის, რომ MERGE-SORT` -ის პარალელურობის მნიშვნელობაა $MS_1`(n) / MS_\infty`(n) = \Theta(\lg n)$, რაც არ არის დიდად შთამბეჭდავი მაჩვენებელი. მაგალითად, 10 მილიონი ელემენტის დასალაგებლად ამ ალგორითმმა შეიძლება მიაღწიოს წრფივ აჩქარებას რამდენიმე პროცესორზე, მაგრამ ასეულობით პროცესორზე მისი მუშაობა არ იქნება ეფექტური.

ადვილი მისახვედრია, რომ ამ ალგორითმის სუსტი მხარე სერიული MERGE პროცედურაა. არსებობს MERGE პროცედურის ვარიანტი, რომელიც იყენებს „გაყავი და იბატონე“ პარადიგმას, და რომლის გაპარალელება ადვილად შეიძლება ჩადგმული პარალელიზმის გამოყენებით.

მრავალნაკადიანი MERGE ალგორითმი T მასივის ქვემასივებზე ოპერირებს. წარმოვიდგინოთ, რომ T მასივის ორი დალაგებული ქვემასივი: $T[p_1 \dots r_1]$, რომლის სიგრძეა $n_1 = r_1 - p_1 + 1$, და $T[p_2 \dots r_2]$, რომლის სიგრძეა $n_2 = r_2 - p_2 + 1$, უნდა გავაერთიანოთ $A[p_3 \dots r_3]$ ქვემასივში, რომლის სიგრძეა $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. სიმარტივისათვის, ზოგადობის შეუზღუდავად დავუშვათ, რომ $n_1 \geq n_2$.

თავდაპირველად, ვპოულობთ $T[p_1 \dots r_1]$ ქვემასივის შუა ელემენტს $x = T[q_1]$, სადაც $q_1 = \lfloor (P_1 + r_1) / 2 \rfloor$. რადგანაც ქვემასივი დალაგებულია, x არის $T[p_1 \dots r_1]$-ის მედიანა: ყოველი ელემენტი $T[p_1 \dots q_1 - 1]$-ში არ აღემატება x-ს და ყოველი ელემენტი $T[q_1 + 1 \dots r_1]$-ში არის x-ზე მეტი ან მისი ტოლი. შემდეგ, ორობითი ძებნის გამოყენებით, ვპოულობთ $T[p_2 \dots r_2]$ დალაგებულ ქვემასივში ისეთ q_2 ინდექსს, რომ x-ის $T[q_2 - 1]$-სა და $T[q_2]$-ს შორის ჩასმის შემდეგ $T[p_2 \dots r_2]$ ისევ დალაგებული იქნება. ამის შემდეგ, საწყის $T[p_1 \dots r_1]$ და $T[p_2 \dots r_2]$ ქვემასივებს ვაერთიანებთ $A[p_3 \dots r_3]$-ში შემდეგნაირად:

1. $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.

2. ჩავწეროთ x-ის მნიშვნელობა $A[q_3]$-ში.

3. რეკურსიულად გავაერთიანოთ $T[p_1 \dots q_1 - 1]$ და $T[p_2 \dots q_2 - 1]$ და შედეგი ჩავწეროთ $A[p_3 \dots q_3 - 1]$-ში.

4. რეკურსიულად გავაერთიანოთ $T[q_1 + 1 \dots r_1]$ და $T[q_2 \dots r_2]$ და შედეგი ჩავწეროთ $A[q_3 + 1 \dots r_3]$-ში.

q_3-ის გამოთვლისას, $q_1 - p_1$ არის $T[p_1 \ldots q_1 - 1]$ ქვემასივის ელემენტების რაოდენობა, ხოლო $q_2 - p_2$ არის $T[p_2 \ldots q_2 - 1]$ ქვემასივის ელემენტების რაოდენობა. მათი ჯამი არის $A[p_3 \ldots r_3]$ ქვემასივის იმ ელემენტების რაოდენობა, რომლებიც ქვემასივში x-მდე უნდა იყოს განთავსებული.

საბაზისო შემთხვევა ხდება, როცა $n_1 = n_2 = 0$. ამ დროს არანაირი სამუშაო არ არის ჩასატარებელი ორი ცარიელი ქვემასივის გასაერთიანებლად. რადგანაც დავუშვით, რომ ქვემასივი $T[p_1 \ldots r_1]$ არის სულ მცირე იმავე სიგრძის, რა სიგრძისაც $T[p_2 \ldots r_2]$, ანუ $n_1 \geq n_2$, იმის შესამოწმებლად, გვაქვს თუ არა საბაზისო შემთხვევა, შეგვიძლია შევამოწმოთ სრულდება თუ არა ტოლობა $n_1 = 0$. ასევე, უნდა დავრწმუნდეთ, რომ ალგორითმი სწორად მუშაობს იმ შემთხვევაში, როდესაც $T[p_2 \ldots r_2]$ ცარიელია.

ზემოთ აღწერილი იდეები გადავიტანოთ ფსევდოკოდში. დავიწყოთ ორობითი ძებნით, რომელიც $T[p \ldots r]$ დალაგებულ ქვემასივში ეძებს იმ ინდექსს, რომელზეც შეგვიძლია ჩავწეროთ x ელემენტი ისე, რომ ქვემასივი ისევ დალაგებული იყოს:

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  return high
```

ორობითი ძებნის ზედა ზღვარია $O(\lg n)$, სადაც $n = r - p + 1$ არის იმ მასივის სიგრძე, რომელზეც ის მუშაობს. რადგანაც BINARY-SEARCH სერიული ალგორითმია, $T_1 = T_\infty = \Theta(\lg n)$.

მოვიყვანოთ მრავალნაკადიანი შერწყმის ალგორითმის ფსევდოკოდი. P-MERGE $(T, p_1, r_1, p_2, r_2, A, p_3)$ აერთიანებს ორ დალაგებულ ქვემასივს: $T[p_1 \ldots r_1]$-სა და $T[p_2 \ldots r_2]$-ს $A[p_3 \ldots r_3]$-ში, სადაც $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 1  n1 = r1 − p1 + 1
 2  n2 = r2 − p2 + 1
 3  if n1 < n2
 4       exchange p1 with p2
 5       exchange r1 with r2
 6       exchange n1 with n2
 7  if n1 == 0
 8       return
 9  else q1 = ⌊(p1 + r1)/2⌋
10       q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11       q3 = p3 + (q1 − p1) + (q2 − p2)
12       A[q3] = T[q1]
13       spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14       P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15       sync
```

ამ ფსევდოკოდში 3-6 ხაზების დანიშნულებაა $n_1 \geq n_2$ უტოლობის სისწორის უზრუნველყოფა.

P-MERGE პროცედურის მუშაობის დროის ანალიზი

პირველ რიგში დავითვალოთ T_∞ P-MERGE-სთვის, რომელსაც აღვნიშნავთ $PM_\infty(n)$-ით, სადაც ორი ქვემასივის ელემენტების საერთო რაოდენობაა $n = n_1 + n_2$. რადგანაც პროცედურის 2 გამოძახება მე-13 და მე-14 ხაზებზე ლოგიკურად ერთმანეთის პარალელურად სრულდება, მხოლოდ იმის შეფასებაა საჭირო, რომელიც დროის მეტ რესურსს მოითხოვს. მთავარი იმის გააზრებაა, რომ ყველაზე უარეს შემთხვევაში, ელემენტების მაქსიმალური რაოდენობა რეკურსიულ გამოძახებებს შორის შეიძლება იყოს მაქსიმუმ $3n / 4$, რასაც აქვე დავამტკიცებთ. $n_2 \leq n_1$ უტოლობიდან გამომდინარეობს, რომ $n_2 = 2n_2 / 2 \leq (n_1 + n_2) / 2 = n / 2$. უარეს შემთხვევაში, ერთ-ერთი რეკურსიული გამოძახება გააერთიანებს $T[p_1 \ldots r_1]$ ქვემასივის $\lfloor n_1 / 2 \rfloor$ ელემენტს $T[p_2 \ldots r_2]$ ქვემასივის n_2 ელემენტთან, და იმ ელემენტების საერთო რაოდენობა, რომლებიც გამოძახებაში იღებენ მონაწილეობას არის

$$\lfloor n_1 / 2 \rfloor + n_2 \leq n_1 / 2 + n_2 / 2 + n_2 / 2 = (n_1 + n_2) / 2 + n_2 / 2 \leq n / 2 + n / 4 = 3n / 4.$$

ორობითი ძებნის მუშაობის დროის გათვალისწინებით ვიღებთ შემდეგ რეკურენტულ ფორმულას:

$$PM_\infty(n) = PM_\infty(3n / 4) + \Theta(\lg n). \qquad (27.8)$$

$$PM_\infty(n) = \Theta(\lg^2 n).$$

ახლა გამოვითვალოთ T_1 P-MERGE-სთვის, რომელსაც $PM_1(n)$-ით აღვნიშნავთ და რომელიც უდრის $\Theta(n)$-ს. რადგანაც n ელემენტიდან თითოეული უნდა გადაიწეროს T მასივიდან A მასივში, გვაქვს $PM_1(n) = \Omega(n)$. დასამტკიცებელი გვრჩება მხოლოდ ის, რომ $PM_1(n) = O(n)$.

თავდაპირველად დავითვალოთ $PM_1(n)$ უარესი შემთხვევისთვის. ოროზითი ძებნა მოითხოვს $\Theta(\lg n)$ დროს უარეს შემთხვევაში, რაც მეტია იმ დროზე, რასაც დანარჩენი კოდი მოიხმარს გარდა რეკურსიული გამოძახებებისა. რაც შეეხება რეკურსიულ გამოძახებებს - მიუხედავად იმისა, რომ შესაძლებელია ორი გამოძახება სხვადასხვა რაოდენობის ელემენტების გასაერთიანებლად მუშაობდეს, ერთად ისინი არაუმეტეს n ელემენტზე მუშაობენ. უფრო მეტიც: როგორც T_∞-ის ანალიზისას ვნახეთ, რეკურსიული გამოძახება მუშაობს არაუმეტეს $3n / 4$ ელემენტზე. ამ ყველაფრიდან გამომდინარე ვიღებთ რეკურსიას:

$$PM_1(n) = PM_1(\propto n) + PM_1((1 - \propto)n) + O(\lg n), \qquad (27.9)$$

სადაც $1 / 4 \leq \propto \leq 3 / 4$, და სადაც $\propto$-ს მნიშვნელობა შეიძლება განსხვავებული იყოს რეკურსიის ყოველ დონეზე.

იმას, რომ (27.9) რეკურსიის ამონახსნია $PM_1 = O(n)$, ვამტკიცებთ ჩანაცვლების მეთოდით. დავუშვათ, რომ $PM_1(n) \leq c_1 n - c_2 \lg n$ რაიმე დადებითი c_1-სა და c_1-სთვის. ჩანაცვლების მეთოდით ვიღებთ:

$$\begin{aligned} PM_1(n) &\leq (c_1\alpha n - c_2\lg(\alpha n)) + (c_1(1-\alpha)n - c_2\lg((1-\alpha)n)) + \Theta(\lg n) \\ &= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\ &= c_1 n - c_2(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\ &= c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\ &\leq c_1 n - c_2\lg n\ , \end{aligned}$$

რადგან შეგვიძლია ავირჩიოთ საკმარისად დიდი c_2 იმისათვის, რომ $c_2(\lg n + \lg(\propto(1 - \propto)))$ აჭარბებს $\Theta(\lg n)$-ს. ასევე შეგვიძლია ავირჩიოთ საკმარისად დიდი c_1 იმისათვის, რომ რეკურსიის საბაზისო შემთხვევის პირობები დაკმაყოფილდეს. იქიდან გამომდინარე, რომ $PM_1(n) = \Omega(n)$ და $PM_1(n) = O(n)$, გვაქვს $PM_1(n) = \Theta(n)$.

P-MERGE-ის პარალელურობის მნიშვნელობაა $PM_1(n) / PM_\infty(n) = \Theta(n / \lg^2 n)$.

პარალელური merge sort

MERGE-SORT` ალგორითმისაგან განსხვავებით, multithreaded ალგორითმი, რომელიც იყენებს P-MERGE ალგორითმს, არგუმენტად იღებს B მასივს, რომელშიც ალგორითმის მუშაობის შედეგი ჩაიწერება. უფრო კონკრეტულად, P-MERGE-SORT(A, p, r, B, s) ასორტირებს A[p … r]-ში და ინახავს მათ B[s … s + r - p]-ში.

```
P-MERGE-SORT(A, p, r, B, s)
 1  n = r − p + 1
 2  if n == 1
 3       B[s] = A[p]
 4  else let T[1..n] be a new array
 5       q = ⌊(p + r)/2⌋
 6       q' = q − p + 1
 7       spawn P-MERGE-SORT(A, p, q, T, 1)
 8       P-MERGE-SORT(A, q + 1, r, T, q' + 1)
 9       sync
10       P-MERGE(T, 1, q', q' + 1, n, B, s)
```

n ელემენტების რაოდენობაა A[p … r]-ში. საბაზისო შემთხვევა გვაქვს, როცა n = 1. q არის ინდექსი, რომელიც A[p … r]-ს ორ ნაწილად ყოფს. q` არის ელემენტების რაოდენობა A[p…q]-ში და გამოიყენება იმის განსასაზღვრად, თუ რა ინდექსიდან უნდა დაიწყოს A[q+1…r]-ის დალაგებული ელემენტების შენახვა T-ში.

პარალელური merge sort-ის ანალიზი

დავიწყოთ P-MERGE-SORT-ის T_1-ის ანალიზით, რომელსაც $PMS_1(n)$-ით აღვნიშნავთ. $PMS_1(n)$ მოიცემა რეკურსიით:

$PMS_1(n) = 2PMS_1(n / 2) + PM_1(n) = 2PMS_1(n / 2) + \Theta(n)$.

ისევე როგორც სერიული დალაგებისთვის შერწყმით, ამ რეკურსიის ამონახსნია $PMS_1(n) = \Theta(n \lg n)$.

ახლა გამოვითვალოთ T_∞ უარესი შემთხვევისთვის - $PMS_\infty(n)$. რადგანაც ორი რეკურსიული გამოძახება ლოგიკურად ერთმანეთის პარალელურად სრულდება, შეგვიძლია მხოლოდ ერთი გავითვალისწინოთ, რის შედეგადაც მივიღებთ რეკურენტულ ფორმულას:

$PMS_\infty(n) = PMS_\infty(n / 2) + PMS_\infty(n) = PMS_\infty(n / 2) + \Theta(\lg^2 n)$. (27.10)

რეკურსიის გახსნის შედეგად მივიღებთ: $PMS_\infty(n) = \Theta(\lg^3 n)$.

P-MERGE-SORT-ის პარალელურობის მნიშვნელობაა $PMS_1(n) / PMS_\infty(n) = \Theta(n \lg n) / \Theta(\lg^3 n) = \Theta(n / \lg^2 n)$, რაც ბევრად უკეთესია, ვიდრე იმ ალგორითმის პარალელურობის მნიშვნელობა, რომელიც სერიულ შერწყმას იყენებს. პრაქტიკაში კარგი შედეგის მისაღებად მიზანშეწონილია საბაზისო შემთხვევის „გაუხეშება“: პატარა ზომის მასივებისთვის სერიული სორტირების ალგორითმის გამოძახება.

პარალელური ალგორითმების იმპლემენტაცია C++-ის გამოყენებით

პარალელიზმის სტანდარტული მხარდაჭერა C++-ში არც ისე დიდი ხანია, რაც არსებობს. მხოლოდ C++11-ითაა შესაძლებელი პარალელიზმის გამოყენება დამატებითი ბიბლიოთეკების გარეშე, თუმცა აქტიურად გამოიყენება ისეთი ბიბლიოთეკები, როგორიცაა, მაგალთად Microsoft ppl, Intel tbb, pthread და სხვა. C++11-ის გაუმჯობესება პარალელიზმის მხრივ გამოიხატება არამხოლოდ ახალ thread-aware მეხსიერების მოდელში, არამედ ახალ კლასებშიც, რომლებიც გვაძლევს thread- ების მართვის, საზიარო მონაცემების დაცვის, thread-ებს შორის ოპერაციების სინქრონიზაციისა და ატომური ოპერაციების განხორციელების საშუალებას.

C++11 შეიცავს სხვადასხვა დონის აბსტრაქციებს პარალელიზმისთვის. დაბალი დონის, როგორიცაა <thread> ბიბლიოთეკა, და უფრო მაღალი დონის, როგორიცაა <future> ბიბლიოთეკა და რომელიც საშუალებას გვაძლევს უფრო მარტივი და შეცდომებისაგან უფრო დაცული კოდი შევქმნათ. აღსანიშნავია, რომ მაღალი დონის აბსტრაქციების გამოყენებამ შეიძლება წარმადობაზე უარყოფითად იმოქმედოს, რადგან შესასრულებელი კოდის მოცულობა გაიზრდება.

იშვიათ შემთხვევებში, თუ ამოცანის გადასაჭრელად საჭირო ფუნქციონალი არ მოიძებნება C++11-ში, შეგვიძლია გამოვიყენოთ საშუალებები, რომლებიც კონკრეტული პლატფორმისთვისაა განკუთვნილი.

ზემოთ აღწერილი ალგორითმების იმპლემენტაციისათვის გამოვიყენეთ სხვადასხვა შესაძლებლობები და მიღებული შედეგები გავაანალიზეთ.

Microsoft PPL (Parallel Patterns Library)

PPL-ში არსებობს ისეთი ცნებების აბსტრაქციები, როგორებიცაა პარალელური for ციკლი და ჩადგმული პარალელიზმი. PPL მოყვება Visual Studio-ს, სტილით მსგავსია C++-ის სტანდარტული ბიბლიოთეკის და კარგად მუშაობს C++11-ის ფუნქციებთან ერთად.

მატრიცების გადამრავლების ამოცანაში გამოვიყენეთ PPL-ის პარალელური for ციკლი, ხოლო merge sort-ის იმპლემენტაციისთვის - PPL-ის ბრძანება parallel_invoke. აღსანიშნავია, რომ განსხვავებით იდეალური პარალელური კომპიუტერისაგან, პრაქტიკაში მატრიცების გადამრავლების ალგორითმის მხოლოდ გარე for ციკლის გაპარალელება გვაძლევს მნიშვნელოვან გაუმჯობესებას სისწრაფის მხრივ.

უნდა აღინიშნოს, რომ C++-ის სტანდარტული ბიბლიოთეკისაგან განსხვავებით, PPL-ს აქვს ის უპირატესობა, რომ პროგრამისტს არ უწევს კოდში აპარატურული

უზრუნველყოფის შესაძლებლობების გათვალისწინება. კონკრეტულად, PPL არ მისცემს პროგრამას საშუალებას შექმნას იმაზე მეტი thread ვიდრე აპარატურულ უზრუნველყოფას აქვს შესაძლებლობა რომ გაუშვას პარალელურად, ხოლო C++-ის სტანდარტული ბიბლიოთეკის გამოყენებისას გვიწევს კოდში გავითვალისწინოთ thread-ების მაქსიმალური რაოდენობა.

ცხრილ 1-ში წარმოდგენილია მატრიცების გადამრავლების სერიული და PPL-ით რეალიზებული პარალელური ალგორითმების მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე, რომელსაც აქვს 2 ფიზიკური ბირთვი, რომელთაგან თითოეული შეიცავს 2 ვირტუალურ ბირთვს. აღსანიშნავია, რომ ვირტუალური ბირთვების რაოდენობის გავლენა პროგრამის მუშაობის დროზე ბევრად უმნიშვნელოა ფიზიკური ბირთვების გავლენასთან შედარებით. ალგორითმების მუშაობის დროები მილიწამებშია ნაჩვენები 100x100, 500x500 და 1000x1000 განზომილების მქონე მატრიცებისათვის. მუშაობის დროს ვითვლით <chrono> ბიბლიოთეკის გამოყენებით.

	100 x 100	500 x 500	1000 x 1000
Serial	336.226 ms	41939 ms	340508 ms
Parallel	157.104 ms	19280.9 ms	153041 ms

ცხრილი 1.

ცხრილ 2-ში წარმოდგენილია merge sort-ის სერიული და PPL-ით რეალიზებული პარალელური ალგორითმების მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე 1000, 1000000 და 10000000 ელემენტიანი მასივებისთვის.

	1,000	1,000,000	10,000,000
Serial	2.0018 ms	4585.06 ms	52088.8 ms
Parallel	4.0022 ms	2696.05 ms	28983 ms

ცხრილი 2.

როგორც ცხრილი 2-დან ჩანს, პატარა მონაცემებისათვის პარალელური merge sort არც ისე ეფექტურია. ამისი მთავარი მიზეზი პარალელური merge ალგორითმია. რადგანაც პარალელური merge პატარა მონაცემებზე არ მუშაობს ეფექტურად, საკმარისად პატარა მონაცემებისათვის რეკურსიულ გამოძახებას აღარ ვახორციელებთ და სერიულ merge ალგორითმს ვიძახებთ. მონაცემების მაქსიმალურ რაოდენობას, რომლისთვისაც არ ღირს პარალელური merge ფუქციის რეკურსიული გამოძახება ეწოდება grain value. ექსპერიმენტების შედეგად დავადგინეთ, რომ grain value-ს ოპტიმალური მნიშვნელობა

ჩვენს შემთხვევაში არის 1024. ეს ნიშნავს, რომ თუ პარალელური merge ალგორითმი 1024 ან ნაკლები რაოდენობის მონაცემებისთვის გამოიძახება, იგი სერიულ merge ალგორითმს გამოიძახებს.

C++11-ის ბიბლიოთეკა <thread>

C++11-ის ბიბლიოთეკა <thread>-ში არ არის პარალელური for ციკლი. ამიტომაც მოვახდინეთ მისი იმპლემენტაცია შემდეგნაირად: პარალელური for ციკლი აღვწერეთ, როგორც ფუნქცია, რომელიც პარამეტრებად იღებს ორ მთელ რიცხვს: მასივის ინტერვალის საწყის და საბოლოო ინდექსებს, და ფუნქციის პოინტერს. როგორც ზემოთ აღვნიშნეთ, C++11-ის გამოყენებისას კოდში უნდა გავითვალისწინოთ აპარატურული უზრუნველყოფის შესაძლებლობები. ამისათვის ვიყენებთ ბრძანებას std::thread::hardware_concurrency(), რომელიც აბრუნებს არსებული პროცესორების რაოდენობას. მისი შედეგის გათვალისწინებით ვყოფთ დასამუშავებელ მონაცემებს ბლოკებად და თითოეული ბლოკისთვის ვქმნით ცალკე thread-ს. თითოეულ thread-ში ეშვება სერიული for ციკლი, რომელიც მოცემული ბლოკის თითოეული ელემენტისათვის გამოიძახებს ფუნქციას, რომელსაც გადავაწოდებთ.

ცხრილ 3-ში წარმოდგენილია მატრიცების გადამრავლების std::thread ბიბლიოთეკის გამოყენებით რეალიზებული პარალელური ალგორითმის მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე, 100x100, 500x500 და 1000x1000 განზომილების მქონე მატრიცებისათვის. იმისათვის, რომ შედარების შესაძლებლობა გვქონდეს, აქვეა მოყვანილი ppl-ით რეალიზებული ფუნქციის მუშაობის დროები.

	100 x 100	500 x 500	1000 x 1000
Serial	336.226 ms	41939 ms	340508 ms
Parallel ppl	157.104 ms	19280.9 ms	153041 ms
Parallel std::thread	161.107 ms	19235.8 ms	154487 ms

ცხრილი 3.

როგორც ცხრილიდან ჩანს, std::thread-ის გამოყენებით რეალიზებული პარალელური for ციკლის მუშაობა მნიშვნელოვნად არ განსხვავდება ppl-ის parallel_for ციკლისაგან.

რაც შეეხება merge sort-ის იმპლემენტაციას, თუ PPL-ის გამოყენებისას შეგვეძლო ყოველი რეკურსიული გამოძახება parallel_invoke() ბრძანებით განგვეხორციელებინა, thread ბიბლიოთეკის გამოყენებისას უნდა უზრუნველვყოთ, რომ არ შეიქმნას პროცესორების რაოდენობაზე მეტი thread. ამის მისაღწევად merge და merge sort

ფუნქციებს პარამეტრად გადავცემთ თავისუფალი პროცესორების რაოდენობას. თუ ეს პარამეტრი 0-ის ტოლია, ფუნქციის რეკურსიული გამოძახება მიმდინარე thread-ში ხდება და არ იქმნება ახალი. ამით თავიდან ვიცილებთ ზედმეტი thread-ების არსებობას, რომელსაც thread overhead ეწოდება და, როგორც წესი, პროგრამის მუშაობის დროს მნიშვნელოვნად ზრდის.

ცხრილ 4-ში წარმოდგენილია merge sort-ის სერიული და PPL-ითა და std::thread ბიბლიოთეკით რეალიზებული პარალელური ალგორითმების მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე 1000, 1000000 და 10000000 ელემენტიანი მასივებისთვის.

	1,000	1,000,000	10,000,000
Serial	2.0018 ms	4585.06 ms	52088.8 ms
Parallel ppl	4.0022 ms	2696.05 ms	28983 ms
Parallel std::thread	4.0014 ms	2456.64 ms	26681.8 ms

ცხრილი 4.

ცხრილის მონაცემების მიხედვით შეიძლება ითქვას, რომ thread ბიბლიოთეკის გამოყენებამ უკეთესი შედეგი მოგვცა ppl-თან შედარებით. ეს მოსალოდნელიც იყო, რადგან ppl უფრო მაღალი დონის ბიბლიოთეკაა.

C++11-ის ბიბლიოთეკა <future>

<future> ბიბლიოთეკა არის thread-ების მაღალი დონის აბსტრაქცია. პარალელური for ციკლისა და merge sort-ის იმპლემენტაცია future ბიბლიოთეკის გამოყენებით არ განსხვავდება thread ბიბლიოთეკის გამოყენებით მათი იმპლემენტაციისაგან. <future> ბიბლიოთეკა უფრო მაღალი დონის აბსტრაქციაა, ვიდრე <thread> ბიბლიოთეკა. მას <thread>-თან შედარებით უფრო მეტი ფუნქციონალი აქვს. მაგალითად, future-ს შეუძლია დააბრუნოს რაიმე გამოთვლის შედეგი.

ცხრილ 5-ში ნაჩვენებია მატრიცების გადამრავლების სერიული და ppl, std::thread და std::future ბიბლიოთეკის გამოყენებით რეალიზებული პარალელური ალგორითმის მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე, 100x100, 500x500 და 1000x1000 განზომილების მქონე მატრიცებისათვის.

	100 x 100	500 x 500	1000 x 1000
Serial	336.226 ms	41939 ms	340508 ms
Parallel ppl	157.104 ms	19280.9 ms	153041 ms

Parallel std::thread	161.107 ms	19235.8 ms	154487 ms
Parallel std::future	161.108 ms	19168.8 ms	153557 ms

ცხრილი 5.

მატრიცების გადამრავლების რეალიზაცია future ბიბლიოთეკის გამოყენებით წინა რეალიზაციებთან ძალიან ახლოს მყოფ შედეგებს გვაძლევს.

ცხრილ 6-ში წარმოდგენილია merge sort-ის სერიული და PPL-ით, std::thread ბიბლიოთეკითა და std::future ბიბლიოთეკით რეალიზებული პარალელური ალგორითმების მუშაობის დროები Intel® Core™ i7-3537U პროცესორზე 1000, 1000000 და 10000000 ელემენტიანი მასივებისთვის.

	1,000	1,000,000	10,000,000
Serial	2.0018 ms	4585.06 ms	52088.8 ms
Parallel ppl	4.0022 ms	2696.05 ms	28983 ms
Parallel std::thread	4.0014 ms	2456.64 ms	26681.8 ms
Parallel std::future	4.0011 ms	2405.61 ms	26550.8 ms

ცხრილი 6.

მიუხედავად იმისა, რომ std::future მაღალი დონის ბიბლიოთეკად ითვლება std::thread-თან შედარებით, მისი გამოყენება std::thread-ის მსგავს შედეგებს გვაძლევს. ეს შემდეგნაირად შეიძლება აიხსნას: მიუხედავად იმისა, რომ future ბიბლიოთეკა უფრო ფუნქციონალურია ვიდრე thread, იგი არ აკონტროლებს thread-ების რაოდენობას ppl-სგან განსხვავებით.

Thread-safe მონაცემთა სტრუქტურების დიზაინი

ისეთი მონაცემთა სტრუქტურების დიზაინისას, რომლებზეც ერთდროულად რამდენიმე thread ატარებს ერთსა და იმავე ან სხვადასხვა ოპერაციებს, აუცილებელია დაცვის მექანიზმების გამოყენება, რომლებიც თავიდან აგვაცილებს შეჯიბრის მდგომარეობას. C++11-ში არსებობს დაცვის ასეთი მექანიზმი სინქრონიზაციის პრიმიტივის - mutex-ის (mutual exclusion) სახით. როდესაც მონაცემები, რომლებზეც ერთდროულად რამდენიმე thread მუშაობს დაცულია mutex-ით, მათზე წვდომისთვის ნებისმიერმა thread-მა უნდა ჩაკეტოს (lock) mutex რომელიც ამ მონაცემებთანაა ასოცირებული, მუშაობის დასრულებისას კი უნდა

გახსნას. როდესაც mutex ჩაკეტილია რომელიმე thread-ის მიერ, სხვა thread-ებს არ ექნებათ წვდომა მონაცემებზე, რომლებსაც mutex იცავს.

C++-ში std::mutex-ის ეგზემპლიარს ვკეტავთ std::lock() ბრძანებით და ვხსნით std::unlock() ბრძანებით. პრაქტიკაში ამ ბრძანებების ხელით წერაზე მეტად მიღებულია std::lock_guard კლასის გამოყენება, რომელიც იყენებს RAII (Resourse Aqcuisition Is Initialization) იდიომას std::mutex-სთვის: გადაწოდებული mutex-ისთვის lock() და unlock() ფუნქციებს იძახებს, შესაბამისად, კონსტრუქტორში და დესტრუქტორში.

აღსანიშნავია, რომ mutex-ის გამოყენებამ შეიძლება თავის მხრივ გამოიწვიოს პრობლემები, რომლებიც აუცილებლად უნდა გავითვალისწინოთ thread-safe მონაცემთა სტრუქტურების დიზაინისას. კერძოდ, წარმოვიდგინოთ, რომ ორ thread-ს სჭირდება mutex-ების ერთი და იგივე წყვილის ჩაკეტვა მუშაობის გასაგრძელებლად. ორივე thread-ს ჩაკეტილი აქვს ერთ-ერთი mutex-ების ამ წყვილიდან და ელოდება მეორეს გათავისუფლებას. ასეთ შემთხვევაში thread-ებს უსასრულოდ მოუწევთ ლოდინი, რადგან ისინი ელოდებიან ერთმანეთს. ასეთ სიტუაციას ურთიერთბლოკირების მდგომარეობა (deadlock) ჰქვია და ყველაზე დიდი პრობლემაა, რაც შეიძლება წარმოიშვას mutex-ებთან მუშაობისას. აქვე უნდა აღინიშნოს, რომ ურთიერთბლოკირების მდგომარეობა შეიძლება წარმოიშვას mutex-ების გარეშეც, როცა ორი thread ელოდება ერთმანეთის დასრულებას მუშაობის გასაგრძელებლად.

განვიხილოთ thread-safe მონაცემთა სტრუქტურის დიზაინის კონკრეტული ვარიანტი. კერძოდ, std::stack-ის უსაფრთხო ვერსია.

Thread-Safe Stack

std::stack-ს აქვს შემდეგი ფუნქციები:

1. push() - ახალი ელემენტის ჩამატება.
2. pop() - ბოლოს ჩამატებული ელემენტის ამოღება.
3. top() - ბოლოს ჩამატებული ელემენტის მნიშვნელობის წაკითხვა.
4. empty() - შემოწმება, არის თუ არა stack ცარიელი.
5. size() – ელემენტების რაოდენობა stack-ში.

იმისათვის, რომ stack multithreaded გარემოში უსაფრთხო იყოს, პირველ რიგში stack-ის მონაცემები უნდა დავიცვათ std::mutex-ის გამოყენებით. გარდა ამისა, მნიშვნელოვანია, რომ top() ფუნქციამ დააბრუნოს ბოლოს ჩამატებული ელემენტის ასლი და არა მიმთითებელი ამ ელემენტზე, რადგან ნებისმიერ კოდს, რომელსაც ექნება წვდომა მიმთითებელზე, ექნება წვდომა საზიარო მონაცემებზეც mutex-ების გვერდის ავლით. Thread-safe მონაცემთა სტრუქტურების დიზაინისას ერთ-ერთი ძირითადი რეკომენდაცია სწორედ ისაა, რომ ფუნქციები არ უნდა აბრუნებდნენ მიმთითებელს საზიარო მონაცემებზე.

მიუხედავად იმისა, რომ თითოეული ფუნქცია შესრულდება მხოლოდ მას შემდეგ, რაც მისი გამომძახებელი thread ჩაკეტავს stack-ის შესაბამის mutex-ს, და top() დააბრუნებს ბოლოს ჩამატებული ელემენტის ასლს, მიღებული მონაცემთა სტრუქტურა ჯერ კიდევ არ არის დაცული შეჯიბრის მდგომარეობისაგან. კერძოდ, empty() და size() ფუნქციების შედეგები არ არის სანდო ასეთ stack-ში მიუხედავად იმისა, რომ ისინი გამოძახების მომენტისათვის სწორად იმუშავებენ. მას შემდეგ, რაც ისინი დაასრულებენ მუშაობას, სხვა thread-ებმა შეიძლება ჩაამატონ ან წაშალონ ელემენტები stack-დან მანამ, სანამ empty() ან size() ფუნქციის გამომძახებელი thread გამოიყენებს მიღებულ შედეგებს. შესაბამისად, empty()-ს და size()-ის დაბრუნებული მნიშვნელობების გამოყენების მომენტში ისინი შეიძლება არასწორი იყოს.

განვიხილოთ კონკრეტული მაგალითი:

```
stack <int> s;

if (!s.empty()) {

        int const value = s.top();

        s.pop();

        do_something(value);

}
```

multithreaded გარემოში ეს კოდი არ არის უსაფრთხო, რადგან s.empty() და s.top() გამოძახებებს შორის შეიძლება სხვა thread-მა წაშალოს s-ის ბოლო ელემენტი, top()-ის გამოძახება ცარიელ stack-ზე კი გამოიწვევს შეცდომას.

ამ კოდთან დაკავშირებული კიდევ ერთი პრობლემა შეიძლება შეიქმნას, თუ ზემოთ განხილული კოდი ორ სხვადასხვა thread-ში შესრულდება ქვემოთ ნაჩვენები მიმდევრობით (დავუშვათ, რომ თავიდან s-ში 2 ელემენტია):

Thread A	Thread B
If (!s.empty())	
	If (!s.empty())
Int const value = s.top();	
	Int const value = s.top();
s.pop();	
do_somthing(value);	s.pop();
	do_something(value);

თუ ბრძანებები ამ მიმდევრობით განხორციელდება, ორივე thread ერთსა და იმავე მნიშვნელობას წაიკითხავს s-დან. ამასთან, s-ის ერთი ელემენტი ისე წაიშლება, რომ მის მნიშვნელობას არც ერთი thread არ წაიკითხავს.

ამ პრობლემების თავიდან ასარიდებლად საჭიროა std::stack-ის დიზაინის შეცვლა. პირველი, რაც შეიძლება მოგვაფიქრდეს ამ პრობლემების გადასაჭრელად არის top() და pop() ფუნქციების გამოძახებების გაერთიანება mutex-ით დაცვის ქვეშ, თუმცა top() და pop() ფუნქციების ცალ-ცალკე არსებობას აქვს თავისი მიზეზი და მათი გაერთიანებით შესაძლოა ახალ პრობლემებს გადავაწყდეთ. კერძოდ, განვიხილოთ stack <vector <int> >. ვექტორი დინამიური კონტეინერია. მისი კოპირებისას vector ბიბლიოთეკა heap-ში გამოყოფს დამატებით მეხსიერებას. თუ სისტემა ძალიან დატვირთლია ან ძალიან შეზღუდული რესურსები აქვს, შეიძლება მეხსიერება ვერ გამოიყოს და შედეგად vector-ის კოპირების კონსტრუქტორში მოხდეს std::bad_alloc exception. ამის მოხდენის შანსი განსკუთრებით დიდია, თუ vector-ში ბევრი ელემენტია. pop() ფუნქცია ისე რომ ყოფილიყო განსაზღვრული, რომ ჯერ დაებრუნებინა ბოლოს ჩამატებული ელემენტის მნიშვნელობა და შემდეგ წაეშალა ის stack-დან, შემდეგი პრობლემის წინაშე აღმოვჩნდებოდით: წაკითხული მნიშვნელობა ბრუნდება მხოლოდ მას შემდეგ, რაც stack-იდან წაიშლება ბოლოს ჩამატებული ელემენტი, წაკითხული ელემენტის (ვექტორის) გამომძახებლისთვის დაბრუნებისთვის კი უნდა შესრულდეს კოპირება, რამაც შესაძლოა შეცდომა გამოიწვიო. ასეთ შემთხვევაში ელემენტი, რომლის მნიშვნელობის გაგებაც გამომძახებელს უნდოდა, იკარგება: ის წაიშალა stack-დან, მაგრამ მისი გადაკოპირება ვერ მოხერხდა. სწორედ ამ პრობლემის თავიდან ასაცილებლად არის განცალკევებული top() და pop() ფუნქციები. საბედნიეროდ, thread-safe stack-ის დასაწერად სხვა გზები შეიძლება გამოვიყენოთ:

1. pop() ფუნქციას არგუმენტად გადავცეთ ცვლადი, რომელშიც შეინახება stack-ში ბოლოს ჩაწერილი ელემენტი წაშლამდე:
   vector <int> result;
   some_stack.pop(result);
   ამ მეთოდის ნაკლოვანება ისაა, რომ საჭიროებს stack-ის მნიშვნელობის ტიპის ეგზემპლიარის შექმნას pop()-ის გამოძახებამდე. ზოგიერთი ტიპისთვის ეს არაა პრაქტიკული, რადგან დიდ დროს ან რესურსებს მოითხოვს. ზოგიერთი ტიპისთვის ეს შეუძლებელიც კია, რადგან კონსტრუქტორი შეიძლება მოითხოვდეს არგუმენტებს, რომლემთა მნიშვნელობებიც არ არის ცნობილი pop()-ის გამოძახებამდე. ასევე, stack-ის ტიპისთვის გადატვირთული უნდა იყოს მინიჭების ოპერატორი, რაც მომხმარებლის მიერ განსაზღვრული ბევრი ტიპისთვის არაა გაკეთებული.
2. pop() ფუნქციას დავაბრუნებინოთ პოინტერი stack-ში ბოლოს ჩაწერილ ელემენტზე და შემდეგ წავშალოთ ის. პოინტერის გადაკოპირება თავისუფლად შეიძლება და ის შეცდომას არ გამოიწვევს. ამ მეთოდის ნაკლოვანება ისაა, რომ პოინტერის დაბრუნება მოითხოვს პოინტერის შესაბამისი ობიექტისთვის გამოყოფილი მეხსიერების დამატებით მართვას, რამაც მარტივი ტიპებისთვის შეიძლება უფრო ძვირადღირებული ოპერაცია იყოს მნიშვნელობის დაბრუნებასთან შედარებით.

3. საუკეთესო ვარიანტი იქნება, თუ stack-ს ექნება ზემოთ განხილული ორივე მეთოდის მხარდაჭერა და მომხმარებელს ექნება საშუალება მისი ამოცანისათვის შესაფერისი მეთოდი გამოიყენოს.

გამოყენებული ლიტერატურა

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – Introduction to Algorithms, 3rd Edition.
2. Anthony Williams – C++ Concurrency In Action, Practical Multithreading.