|  |  |
| --- | --- |
|  | **ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**  **ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ** |

**КУРСОВА РАБОТА**

**Дисциплина: „Паралелно програмиране”**

**тема: „Сравнение на производителността на алгоритми за паралелно търсене”**

**Изготвил**: Иван Николов

Фак. № 121221010

Група: 43

IV курс

**Заверяващ**: доц. д-р инж. Ива Николова

02.11.2024 г.

**София, 2024**

# **Резюме**

Проектът цели да изследва и сравни производителността на различни алгоритми за паралелно търсене чрез прилагане на техники за паралелизация към известни и познати алгоритми за линейно, двоично и хеш-базирано търсене, както и търсене в дървовидни структури. Основната цел е да се анализира поведението на тези алгоритми при различни набори от данни върху многоядрен процесор, като се оценяват параметри като време за изпълнение, мащабируемост и ефективност. В рамките на проекта е разработен бенчмаркинг фреймуърк, който позволява автоматично генериране на различни тестови данни (сортирани, обратни и случайни последователности) и прецизно измерване на производителността на всеки алгоритъм при различни условия. Този фреймуърк осигурява платформа за сравнение на ключови показатели за всеки алгоритъм и неговото поведение при различни входни данни. Разработен е удобен потребителски интерфейс, който улеснява потребителите при избора на алгоритми и преглед на техните резултати. Използваните алгоритми включват паралелно линейно търсене, което позволява паралелно претърсване на големи обеми от несортирани данни, паралелно двоично търсене, подходящо за предварително сортирани масиви, хеш-базирано търсене за бързо намиране на елементи чрез хеш-таблици и паралелно търсене в дървовидни структури, което улеснява претърсването на големи йерархични структури. След провеждането на експериментите, резултатите са анализирани и визуализирани, като е направен извод за ефективността и подходящите сценарии за всеки алгоритъм. Извършен е сравнителен анализ на алгоритмите, като се разглеждат възможности за оптимизация на по-слабо представящите се алгоритми чрез допълнителни техники за паралелизация и алгоритмични подобрения. Проектът подчертава значението на избора на подходящ паралелен алгоритъм за търсене според спецификата на данните и наличните хардуерни ресурси..

# **Съдържание**

[**1.** **Резюме** 1](#_Toc181456880)

[**2.** **Съдържание** 2](#_Toc181456881)

[**3.** **Въведение** 3](#_Toc181456882)

[**4.** **Теоретична основа** 4](#_Toc181456883)

[**4.1.** **Преглед на литературата** 4](#_Toc181456884)

[**4.2.** **Алгоритми за паралелно програмиране** 5](#_Toc181456885)

[**5.** **Методология** 6](#_Toc181456886)

[**5.1.** **Избор на технологии** 6](#_Toc181456887)

[**5.2.** **Процес на разработка** 6](#_Toc181456888)

[**5.3.** **Паралелизация на алгоритми** 6](#_Toc181456889)

[**6.** **Реализация** 8](#_Toc181456890)

[**6.1.** **Описание на архитектурата** 8](#_Toc181456891)

[**6.2.** **Код и примери** 9](#_Toc181456892)

[**6.3.** **Инсталация и настройка** 10](#_Toc181456893)

[**7.** **Резултати** 11](#_Toc181456894)

[**7.1.** **Тестови данни** 11](#_Toc181456895)

[**7.2.** **Анализ на производителността** 11](#_Toc181456896)

[**7.2.1.** **Опитна постановка 1 – Windows based** 11](#_Toc181456897)

[**7.2.2.** **Опитна постановка 2 – Apple MacOS based** 16](#_Toc181456898)

[**7.2.3.** **Опитна постановка 3 – малък обем от данни** 18](#_Toc181456899)

[**7.3.** **Визуализация на данни** 19](#_Toc181456900)

[**8.** **Обсъждане** 22](#_Toc181456901)

[**8.1.** **Интерпретация на резултатите** 22](#_Toc181456902)

[**8.2.** **Ограничения на проекта** 25](#_Toc181456903)

[**9.** **Заключение** 26](#_Toc181456904)

[**10.** **Библиография** 27](#_Toc181456905)

[**Приложение № 1: Графичен интерактивен интерфейс** 28](#_Toc181456906)

[**Приложение № 2: Запазване на тестовите данни в JSON файл** 32](#_Toc181456907)

# **Въведение**

Разглежда се основната идея зад паралелизацията за ускоряване на алгоритмите за търсене и предизвикателствата, свързани с изпълнението на такива алгоритми върху съвременните многоядрени процесори и графични процесори.

* 1. **Обща информация**

В условията на нарастващи обеми данни и необходимост от бърз достъп до информация, ефективните алгоритми за търсене заемат важна роля в много области — от бази данни и обработка на изображения до търсачки и анализ на големи данни. Паралелните алгоритми за търсене предлагат възможност за значително намаляване на времето за обработка чрез разпределение на натоварването между процесорни ядра. Същевременно, те поставят нови изисквания към проектирането и оптимизацията на алгоритмите, като разкриват нови възможности за подобряване на производителността и ефективността.

* 1. **Цели на проекта**

Цел 1: Да се изследва ефективността на различни алгоритми за паралелно търсене, включително линейно, двоично, хеш-базирано търсене и търсене в дървовидни структури.

Цел 2: Да се приложат и тестват различни техники за паралелизация на процесорни ядра с цел определяне на подходите, които водят до най-добри резултати при различни условия.

Цел 3: Да се разработи бенчмаркинг фреймуърк за оценка на производителността, включващ показатели като време за изпълнение, мащабируемост и ефективност, с цел съпоставка на различните алгоритми.

Цел 4: Да се анализират и визуализират резултатите и да се направят препоръки за оптимизация на алгоритмите, както и избор на подходящи алгоритми за различни сценарии.

* 1. **Обхват на работата**

Разглеждани аспекти: Изследването обхваща алгоритмите за паралелно търсене върху различни набори от данни (сортирани, обратно сортирани и случайни) и върху различни хардуерни конфигурации, предимно многоядрен процесор. Данните за производителността на алгоритмите ще бъдат събрани и анализирани въз основа на ключовите показатели, а ще бъдат приложени и техники за оптимизация за подобряване на ефективността на по-слабо представящите се алгоритми.

Неразглеждани аспекти: Изследването не обхваща алгоритми, изискващи специфични или по-редки техники за паралелизация извън стандартната CPU/GPU архитектура. Също така, не се акцентира върху разработването на нови алгоритми за търсене, а се поставя фокус върху подобряване на съществуващите методи за паралелно изпълнение и оценка на тяхната ефективност.

# **Теоретична основа**

С увеличаването на обема от данни и необходимостта от бърз достъп до тях, паралелното търсене придобива нарастваща значимост. Чрез разпределение на задачите между няколко процесорни ядра или дори графични процесори, тези алгоритми осигуряват значително ускорение спрямо класическите последователни методи. Теоретичната основа включва преглед на публикации и статии, представящи различни подходи, които са доказали своята ефективност в различни приложения и сценарии.

* 1. **Преглед на литературата**

За целите на анализа са избрани класически и съвременни подходи за паралелизация на алгоритмите за търсене, прилагани в различни контексти и с разнообразни техники.

Паралелизация на линейното търсене: Линейното търсене е базисен алгоритъм за търсене в несортирани данни, при който последователно се проверява всеки елемент. При паралелизацията, задачата се разделя на подмножества от елементи, които се разпределят на различни процесорни ядра. Според публикацията на GeeksforGeeks за мултитрединг в линейно търсене, този подход намалява времето за търсене при големи обеми данни, като всяка нишка се насочва към определен сегмент. Изследванията потвърждават, че този метод е особено ефективен при търсене в несортирани масиви и позволява значително ускорение в условия с голям брой елементи. Линейното търсене с паралелизация има ограничения при малки обеми данни, където времето за създаване и управление на нишките може да надвиши времето за самото търсене.

Паралелизация на двоичното търсене: Двоичното търсене е по-бърз алгоритъм, подходящ за сортирани масиви, който оперира чрез рекурсивно разделяне на масива на две части, докато намери търсения елемент. При паралелизацията, масивът се разпределя на сегменти, като всяка нишка изпълнява бинарно търсене на различна част. Според изследвания публикувани в „GitHub“ за паралелизация на алгоритъма за двоично търсене се разглежда ефективността на този алгоритъм при сортирани данни, като се отбелязва, че паралелизацията е особено подходяща за големи обеми данни. Изследванията показват, че при достатъчно нишки и правилно разпределение на задачите, двоичното търсене може да постигне почти линейно ускорение в зависимост от броя на нишките. Алгоритъмът изисква предварително сортирани данни и не е подходящ за динамични или често променящи се структури, тъй като всяка промяна изисква ново сортиране.

Хеш-базирано паралелно търсене: Търсенето чрез хеш-таблици е ефективно, когато елементите могат да бъдат хеширани по начин, който позволява бързо извличане. Паралелизацията при хеш-базираните алгоритми включва разпределяне на нишките така, че всяка нишка обработва различен сегмент от хеш-таблицата. В публикация във Fiveable, хеш-базираните методи предоставят отлична производителност при големи обеми данни, особено когато хеш-таблицата е разделена на сегменти. Те предлагат логаритмично време за търсене и минимизират риска от колизии, като изолират задачите между нишките. Хеш-базираните алгоритми зависят от доброто разпределение на хеш-функцията. При големи колизии или неравномерно разпределение, ефективността им може значително да намалее. Също така, при големи хеш-таблици използването на памет може да стане предизвикателство.

Паралелно търсене в дървовидни структури: Паралелното търсене в дървовидни структури разчита на рекурсивно претърсване, като всеки клон или ниво от дървото може да бъде обработено от отделна нишка. Тази техника е полезна за претърсване на йерархични структури, като например B-деревата, които често се използват в бази от данни. В публикацията на Арян Джиндал „Parallel Tree Level Finder“ в сайта „medium.com” се изследва подходът за паралелно търсене чрез разпределяне на нивата на дървото към различни нишки. Този метод демонстрира висока ефективност при големи дървовидни структури, като всяко ниво може да бъде обработвано независимо, което намалява времето за достъп до дълбоко разположени елементи. Паралелното търсене в дървета се усложнява при силно небалансирани структури, където някои нишки обработват повече възли от други. Това води до неравномерно разпределение на натоварването и може да намали цялостната производителност.

* 1. **Алгоритми за паралелно програмиране**

Основните разглеждани алгоритми включват паралелно линейно търсене, паралелно двоично търсене, хеш-базирано паралелно търсене и търсене в дървовидни структури. Всяка техника има свои специфични предимства и ограничения, които влияят на изборът на подход за различните типове данни и хардуерни конфигурации.

Паралелно линейно търсене: При паралелното линейно търсене, масивът се разделя на равни части, като всяка част се обработва от отделна нишка. Например, ако има масив от 1 милион елемента и 4 нишки, масивът ще се раздели на 4 сегмента, всеки с по 250,000 елемента.

* Пример: За търсене на елемент в масив [3, 5, 9, 10, 12, 15], всяка нишка ще получи по два елемента. Ако елементът е намерен, останалите нишки спират.

Паралелно двоично търсене: Двоичното търсене изисква сортиран масив. Техниката за паралелизация тук включва разделяне на масива и извършване на бинарно търсене във всеки сегмент едновременно.

* Пример: При сортиран масив [1, 3, 7, 9, 12, 15, 20, 25] и търсенето на 9, масивът може да се раздели между 4 нишки. Всяка нишка започва бинарното търсене в своя сегмент. Нишката, която намери елемента, сигнализира на останалите да спрат.

Хеш-базирано паралелно търсене: Хеш-базираните структури позволяват бързо търсене чрез уникални ключове. В паралелния модел, всяка нишка работи върху подмножество от хеш-таблицата.

* Пример: При хеш-таблица {10: true, 20: true, 30: true}, където се търси ключ 20, нишките се разпределят по сегменти, всяка обработвайки част от хеш-таблицата. Ако ключът бъде намерен, всички нишки се уведомяват да спрат.

Търсене в дървовидни структури: Търсенето в дървовидна структура изисква едновременно претърсване на различните нива на дървото, при което всяка нишка работи върху отделен клон.

* Пример: При двоично дърво с възли [15, 10, 20, 8, 12, 17, 25], всяка нишка започва търсене от коренния възел и навлиза в своето поддърво. При достигане на елемент, другите нишки спират работа.

# **Методология**

* 1. **Избор на технологии**

Проектът е реализиран с помощта на езика Go заради неговите вградени възможности за управление на конкурентност и ефективна паралелизация. Избрани са следните библиотеки и инструменти:

* *sync*: Използвана за координиране на горутини с wait groups и синхронизиране на достъпа до споделени ресурси. Това позволява ефективно управление на нишките и гарантира точност при обработката на данните.
* *net/http*: Използвана за създаване на HTTP сървър, който обработва заявки за изпълнение на алгоритми за търсене, като същевременно предоставя резултатите на потребителски интерфейс.
* *encoding/json*: Използвана за сериализация и десериализация на JSON данни, което улеснява съхранението и споделянето на резултати.

Go е избран също заради лесното управление на горутини (concurrent routines), което го прави подходящ за приложения с високи изисквания към паралелната обработка.

* 1. **Процес на разработка**
* Избор и имплементация на алгоритмите за търсене: Първоначално са реализирани четири алгоритма за паралелно търсене – линейно, двоично, хеш-базирано и търсене в дървовидни структури. Всеки алгоритъм е разработен и тестван за коректност, като е използвано паралелно изпълнение на задачи чрез горутини.
* Генериране на тестови данни: Създаден е модул за генериране на различни видове входни данни – сортирани, обратно сортирани и случайни. Данните се генерират с функции като GenerateOrderedData, GenerateReverseOrderedData и GenerateRandomData, което позволява да се тества ефективността на алгоритмите при различни сценарии.
* Изпълнение и измерване на алгоритмите за търсене: Модулът ExecSearch обработва различни видове заявки за търсене, като използва предоставения тип на търсене и ключ. Времето за изпълнение на всеки алгоритъм се измерва с помощта на time.Now() и time.Since() за по-добра оценка на производителността.
* Съхранение и визуализация на резултатите: Използвана е функционалността за сериализация на JSON данни, за да се запазват резултатите от търсенето във файлове, които лесно могат да бъдат визуализирани и анализирани.
  1. **Паралелизация на алгоритми**

Основният подход за паралелизация в проекта се базира на използването на горутини и wait groups за управление на конкурентността. Всеки алгоритъм прилага паралелизация по специфичен начин:

* Паралелно линейно търсене: Масивът се разделя на равни части според броя на нишките. Всяка нишка обработва своя сегмент и изпраща резултата, когато намери търсения елемент. Горутини се синхронизират с wait group, а когато една от тях намери резултата, останалите се прекратяват.
* Паралелно двоично търсене: Масивът се разпределя на сегменти, в които всяка нишка изпълнява двоично търсене. Резултатите се проверяват за наличие на ключ, след което wait group прекратява останалите горутини.
* Хеш-базирано паралелно търсене: При хеш-базираното търсене всеки елемент се добавя към хеш-таблица от нишките, а след това се извършва паралелно търсене на конкретния ключ. Каналите (channels) се използват за комуникация между нишките и за сигнализация при откриване на резултата.
* Търсене в дървовидни структури: Дървото се претърсва, като всяка нишка обработва отделен клон или ниво. Търсенето продължава до достигане на търсения елемент, като wait group се използва за координация на горутините и завършване на процеса.

Подходът за паралелизация осигурява оптимално разпределение на задачите между процесорните ядра и намалява времето за изпълнение, особено при големи обеми от данни.

# **Реализация**

* 1. **Описание на архитектурата**

На „Фигура 1“ е представена компонентна диаграма на проекта. Основни компоненти биват:

* Сървър за обработка на заявки: Основният сървър, реализиран с помощта на библиотеката net/http на Go, обслужва входящите заявки за търсене и връща резултати на потребителите. SearchHandler обработва GET и POST заявки, като позволява зареждане на уеб интерфейс и изпращане на заявки за търсене.
* Модул за паралелно търсене: Модулът parallel\_search съдържа основните алгоритми за търсене (линейно, двоично, хеш-базирано и търсене в дървовидни структури). Всеки алгоритъм е разработен с подходи за паралелизация, които използват горутини за разпределяне на задачите.
* Генератор на данни: Модулът data\_generator осигурява автоматично създаване на различни типове данни за тестване (сортирани, обратно сортирани и случайни). Данните се използват за измерване на ефективността на всеки алгоритъм при различни условия.
* Модул за обработка на резултати: След приключване на търсенето, резултатите се съхраняват в JSON формат чрез SaveHandler. Това позволява последващ анализ и визуализация.

Компонентите взаимодействат, като сървърът приема входни данни от потребителите, изпълнява съответния алгоритъм за търсене и връща резултата. Всеки алгоритъм използва паралелизация за оптимално разпределение на натоварването върху многоядрен процесор.

A diagram of a computer

Description automatically generated

Фигура компонентна диаграма

* 1. **Код и примери**

ExecSearch избира и изпълнява алгоритъм за търсене според типа, зададен от потребителя, и записва времето за изпълнение. Това осигурява гъвкавост и улеснява съпоставянето на резултатите.

|  |
| --- |
| func ExecSearch(w http.ResponseWriter, ctx \*Context, threadCount int) {  var startTime time.Time  var result bool  switch ctx.SearchType {  case LINEAR:  startTime = time.Now()  result = sreacher.ParallelLinearSearch(ctx.Data, ctx.Key, threadCount)  case BINARY:  startTime = time.Now()  result = sreacher.ParallelBinarySearch(ctx.Data, ctx.Key, threadCount)  // Останалите алгоритми също се изпълняват на база на типа търсене  }  searchTime := time.Since(startTime).Seconds()  response := SearchResult{  Result: result,  SearchType: ctx.SearchType,  SearchTime: fmt.Sprintf("%.0f", searchTime \* 1 000 000 000), // измерено в наносекунди  }  \_ = json.NewEncoder(w).Encode(response)  } |

Функцията GenerateRandomData създава масив със случайни стойности, което позволява тестване на алгоритмите с различни типове входни данни.

|  |
| --- |
| func GenerateRandomData(size int, maxElement int) []int {  r := rand.New(rand.NewSource(time.Now().UnixNano()))  data := make([]int, size)  for i := range data {  data[i] = r.Intn(maxElement)  }  return data  } |

SaveHandler записва детайлите от търсенето във файл, което позволява лесно съхранение на резултатите за анализ.

|  |
| --- |
| func SaveHandler(w http.ResponseWriter, r \*http.Request) {  var searchDetails map[string]interface{}  err := json.NewDecoder(r.Body).Decode(&searchDetails)  filePath := fmt.Sprintf("resources/results/search\_results\_%d.json", time.Now().UnixNano())  err = saveToFile(filePath, searchDetails)  if err != nil {  http.Error(w, "Error writing to file", http.StatusInternalServerError)  return  }  \_ = json.NewEncoder(w).Encode(map[string]string{"message": "Search details saved successfully!"})  } |

* 1. **Инсталация и настройка**

Инсталацията на проекта изисква предварително инсталиране на Go и настройка на необходимите директории и файлове.

1. Изтегляне и инсталация на Go от golang.org.
2. Потвърждение на инсталацията с командата `go version`.
3. Навигирайте до проекта с `cd project\_directory`.
4. Стартиране на сървъра: `go run main.go` за стартиране на HTTP сървъра.
5. Достъпване на потребителския интерфейс чрез браузър на http://localhost:8888.

# **Резултати**

* 1. **Тестови данни**

Долуописаните тестови данни са създадени, за да оценят производителността на четирите паралелни алгоритми за търсене при различни условия, включващи случайни, възходящо и низходящо подредени набори от данни. Търсените стойности са избрани стратегически, за да осигурят проверка на ефективността на алгоритмите в голям обхват от стойности.

* Произволен набор от данни:
  + Брой елементи: 10,000,000
  + Диапазон на стойностите: [0 – 10,000,000]
  + Използвани брой нишки: {2, 8}
  + Търсени стойности: {43, 121,221, 6,515,010}
* Възходящо подреден набор от данни:
  + Брой елементи: 10,000,000
  + Диапазон на стойностите: [0 – 10,000,000]
  + Използвани брой нишки: {2, 8}
  + Търсени стойности: {43, 121,221, 6,515,010}
* Низходящо подреден набор от данни:
  + Брой елементи: 10,000,000
  + Диапазон на стойностите: [0 – 10,000,000]
  + Използвани брой нишки: {2, 8}
  + Търсени стойности: {43, 121,221, 6,515,010}
  1. **Анализ на производителността**
     1. **Опитна постановка 1 – Windows based**
* Хардуерна база
* Процесор: 12th Gen Intel(R) Core(TM) i5-12450H, 2.00 GHz
* RAM: 16 GB
* Архитектура: Intel x86-64

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Ключ на търсене | Тип данни | Алгоритъм | Брой нишки | Време за обработка (s) | Време за търсене (ns) | Обработен брой параметри | Резултат |
| 43 | Произволни | Паралелно линейно търсене | 8 | 0,193218 | 1004300 | 10000000 | наличен |
| 43 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 43 | Произволни | Паралелно хеш търсене | 8 | 6,336704 | 72199700 | 6322042 | наличен |
| 43 | Произволни | Паралелно дървовидно търсене | 8 | 1,215478 | 349946800 | 10000000 | наличен |
| 43 | Произволни | Паралелно линейно търсене | 2 | 0,055028 | 2086300 | 10000000 | наличен |
| 43 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 43 | Произволни | Паралелно хеш търсене | 2 | 6,878468 | 66384300 | 6322042 | наличен |
| 43 | Произволни | Паралелно дървовидно търсене | 2 | 1,958235 | 852624500 | 10000000 | наличен |
| 43 | Подредени | Паралелно линейно търсене | 8 | 0,954187 | 0 | 10000000 | наличен |
| 43 | Подредени | Паралелно бинарно търсене | 8 | 0,894470 | 0 | 10000000 | наличен |
| 43 | Подредени | Паралелно хеш търсене | 8 | 4,745985 | 72611700 | 6322042 | наличен |
| 43 | Подредени | Паралелно дървовидно търсене | 8 | 1,143710 | 570754200 | 10000000 | наличен |
| 43 | Подредени | Паралелно линейно търсене | 2 | 0,924283 | 0 | 10000000 | наличен |
| 43 | Подредени | Паралелно бинарно търсене | 2 | 0,909149 | 0 | 10000000 | наличен |
| 43 | Подредени | Паралелно хеш търсене | 2 | 11,060889 | 169023400 | 6322042 | наличен |
| 43 | Подредени | Паралелно дървовидно търсене | 2 | 3,051387 | 2090913700 | 10000000 | наличен |
| 43 | Подредени в низходящ ред | Паралелно линейно търсене | 8 | 0,965690 | 2514500 | 10000000 | наличен |
| 43 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 43 | Подредени в низходящ ред | Паралелно хеш търсене | 8 | 7,118706 | 79895600 | 6322042 | наличен |
| 43 | Подредени в низходящ ред | Паралелно дървовидно търсене | 8 | 1,176532 | 1529608400 | 10000000 | наличен |
| 43 | Подредени в низходящ ред | Паралелно линейно търсене | 2 | 0,956214 | 3556000 | 10000000 | наличен |
| 43 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 43 | Подредени в низходящ ред | Паралелно хеш търсене | 2 | 5,364134 | 128415900 | 6322042 | наличен |
| 43 | Подредени в низходящ ред | Паралелно дървовидно търсене | 2 | 1,198332 | 1486383200 | 10000000 | наличен |
| 121221 | Произволни | Паралелно линейно търсене | 8 | 0,261881 | 999700 | 10000000 | наличен |
| 121221 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 121221 | Произволни | Паралелно хеш търсене | 8 | 11,263932 | 321484300 | 6322042 | наличен |
| 121221 | Произволни | Паралелно дървовидно търсене | 8 | 1,177095 | 1369217500 | 10000000 | наличен |
| 121221 | Произволни | Паралелно линейно търсене | 2 | 0,084890 | 993700 | 10000000 | наличен |
| 121221 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 121221 | Произволни | Паралелно хеш търсене | 2 | 12,068426 | 141151900 | 6322042 | наличен |
| 121221 | Произволни | Паралелно дървовидно търсене | 2 | 1,249314 | 1172170300 | 10000000 | наличен |
| 121221 | Подредени | Паралелно линейно търсене | 8 | 0,922988 | 0 | 10000000 | наличен |
| 121221 | Подредени | Паралелно бинарно търсене | 8 | 0,917912 | 0 | 10000000 | наличен |
| 121221 | Подредени | Паралелно хеш търсене | 8 | 4,803543 | 70220900 | 6322042 | наличен |
| 121221 | Подредени | Паралелно дървовидно търсене | 8 | 1,316296 | 1499736600 | 10000000 | наличен |
| 121221 | Подредени | Паралелно линейно търсене | 2 | 3,011142 | 0 | 10000000 | наличен |
| 121221 | Подредени | Паралелно бинарно търсене | 2 | 3,091135 | 0 | 10000000 | наличен |
| 121221 | Подредени | Паралелно хеш търсене | 2 | 8,041741 | 138449600 | 6322042 | наличен |
| 121221 | Подредени | Паралелно дървовидно търсене | 2 | 1,237025 | 1063572500 | 10000000 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно линейно търсене | 8 | 2,500928 | 7110200 | 10000000 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 121221 | Подредени в низходящ ред | Паралелно хеш търсене | 8 | 5,081093 | 196370800 | 6322042 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно дървовидно търсене | 8 | 1,631323 | 2615151400 | 10000000 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно линейно търсене | 2 | 0,943114 | 4614300 | 10000000 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 121221 | Подредени в низходящ ред | Паралелно хеш търсене | 2 | 5,700611 | 66154500 | 6322042 | наличен |
| 121221 | Подредени в низходящ ред | Паралелно дървовидно търсене | 2 | 1,133887 | 635856800 | 10000000 | наличен |
| 6515010 | Произволни | Паралелно линейно търсене | 8 | 0,897276 | 999900 | 10000000 | наличен |
| 6515010 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 6515010 | Произволни | Паралелно хеш търсене | 8 | 6,050142 | 181213600 | 6322042 | наличен |
| 6515010 | Произволни | Паралелно дървовидно търсене | 8 | 1,198921 | 1919437300 | 10000000 | наличен |
| 6515010 | Произволни | Паралелно линейно търсене | 2 | 0,136754 | 1021300 | 10000000 | наличен |
| 6515010 | Произволни | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 6515010 | Произволни | Паралелно хеш търсене | 2 | 13,554340 | 344871500 | 6322042 | наличен |
| 6515010 | Произволни | Паралелно дървовидно търсене | 2 | 1,198891 | 1791164600 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно линейно търсене | 8 | 0,953888 | 2000500 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно бинарно търсене | 8 | 0,950472 | 0 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно хеш търсене | 8 | 3,401917 | 194561500 | 6322042 | наличен |
| 6515010 | Подредени | Паралелно дървовидно търсене | 8 | 1,182748 | 1479999000 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно линейно търсене | 2 | 3,106303 | 10106800 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно бинарно търсене | 2 | 2,487075 | 0 | 10000000 | наличен |
| 6515010 | Подредени | Паралелно хеш търсене | 2 | 12,585925 | 295668300 | 6322042 | наличен |
| 6515010 | Подредени | Паралелно дървовидно търсене | 2 | 1,116692 | 1373032700 | 10000000 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно линейно търсене | 8 | 0,916164 | 2524000 | 10000000 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 6515010 | Подредени в низходящ ред | Паралелно хеш търсене | 8 | 10,754364 | 613464200 | 6322042 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно дървовидно търсене | 8 | 1,153739 | 1403298200 | 10000000 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно линейно търсене | 2 | 0,921849 | 3043700 | 10000000 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно бинарно търсене | — | — | — | 10000000 | — |
| 6515010 | Подредени в низходящ ред | Паралелно хеш търсене | 2 | 5,136761 | 64178400 | 6322042 | наличен |
| 6515010 | Подредени в низходящ ред | Паралелно дървовидно търсене | 2 | 1,174638 | 1214482000 | 10000000 | наличен |

* + 1. **Опитна постановка 2 – Apple MacOS based**
* Хардуерна база
* Процесор: Apple M3 Pro 12-core CPU 4.05 GHz
* RAM: 36 GB
* Архитектура: ARMv8.6-A

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Ключ на търсене | Тип данни | Алгоритъм | Брой нишки | Време за обработка (s) | Време за търсене (ns) | Обработен брой параметри | Резултат |
| 121221 | Произволни | Паралелно линейно търсене | 8 | 0,077424542 | 789375 | 10000000 | наличен |
| 121221 | Произволни | Паралелно бинарно търсене | — | — | — | — | — |
| 121221 | Произволни | Паралелно хеш търсене | 8 | 4,882374542 | 142230792 | 6322452 | наличен |
| 121221 | Произволни | Паралелно дървовидно търсене | 8 | 0,213486209 | 1269867250 | 10000000 | наличен |
| 121221 | Произволни | Паралелно линейно търсене | 2 | 0,078447458 | 2517334 | 10000000 | наличен |
| 121221 | Произволни | Паралелно бинарно търсене | — | — | — | — | — |
| 121221 | Произволни | Паралелно хеш търсене | 2 | 4,7911945 | 100498250 | 6322452 | наличен |
| 121221 | Произволни | Паралелно дървовидно търсене | 2 | 0,220475791 | 1331113167 | 10000000 | наличен |
| 121221 | Подредени | Паралелно линейно търсене | 8 | 0,676532292 | 96084 | 10000000 | наличен |
| 121221 | Подредени | Паралелно бинарно търсене | 8 | 0,690775875 | 27584 | 10000000 | наличен |
| 121221 | Подредени | Паралелно хеш търсене | 8 | 3,830728708 | 58404084 | 6322452 | наличен |
| 121221 | Подредени | Паралелно дървовидно търсене | 8 | 0,21203925 | 1140437500 | 10000000 | наличен |
| 121221 | Подредени | Паралелно линейно търсене | 2 | 0,686572125 | 102000 | 10000000 | наличен |
| 121221 | Подредени | Паралелно бинарно търсене | 2 | 0,693518792 | 18542 | 10000000 | наличен |
| 121221 | Подредени | Паралелно хеш търсене | 2 | 3,844497875 | 101957667 | 6322452 | наличен |
| 121221 | Подредени | Паралелно дървовидно търсене | 2 | 0,2138335 | 1007655709 | 10000000 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно линейно търсене | 8 | 0,750327542 | 1075417 | 10000000 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно бинарно търсене | — | — | — | — | — |
| 121221 | Подредение в низходящ ред | Паралелно хеш търсене | 8 | 3,806608833 | 131508708 | 6322452 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно дървовидно търсене | 8 | 0,222129584 | 29240083 | 10000000 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно линейно търсене | 2 | 0,688444708 | 2125000 | 10000000 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно бинарно търсене | — | — | — | — | — |
| 121221 | Подредение в низходящ ред | Паралелно хеш търсене | 2 | 3,862794958 | 64008333 | 6322452 | наличен |
| 121221 | Подредение в низходящ ред | Паралелно дървовидно търсене | 2 | 0,193558958 | 98129792 | 10000000 | наличен |

* + 1. **Опитна постановка 3 – малък обем от данни**

Експеримента е проведен върху конфигурацията от „Опитна постановка 1“

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Ключ на търсене | Тип данни | Алгоритъм | Брой нишки | Време за обработка (s) | Време за търсене (ns) | Обработен брой параметри | Резултат |
| 43 | Подредени | Паралелно линейно търсене | 8 | 0,000073375 | 62500 | 100 | наличен |
| 43 | Подредени | Паралелно бинарно търсене | 8 | 0,000194959 | 54000 | 100 | наличен |
| 43 | Подредени | Паралелно хеш търсене | 8 | 0,000194959 | 21042 | 61 | наличен |
| 43 | Подредени | Паралелно дървовидно търсене | 8 | 0,000084083 | 387458 | 100 | наличен |
| 43 | Подредени | Паралелно линейно търсене | 2 | 0,000158875 | 39834 | 100 | наличен |
| 43 | Подредени | Паралелно бинарно търсене | 2 | 0,000063833 | 35792 | 100 | наличен |
| 43 | Подредени | Паралелно хеш търсене | 2 | 0,000475959 | 79125 | 61 | наличен |
| 43 | Подредени | Паралелно дървовидно търсене | 2 | 0,000138833 | 146334 | 100 | наличен |

* 1. **Визуализация на данни**

Въз основа на получените резултати могат да се направят следните графични заключения:

* Фигура 2: Представя сравнението на паралелните алгоритми за търсене при 2 и 8 нишки и времето в наносекунди за намиране на елемент в структурата, при случайно генерирани данни.

A comparison of a graph

Description automatically generated with medium confidence

Фигура Анализ на ефективност при произволни данни

* Фигура 3: Показва сравнението на паралелните алгоритми за търсене при 2 и 8 нишки и времето в наносекунди за намиране на елемент в структурата, при данни, генерирани в нарастващ ред.

A graph of different types of data

Description automatically generated with medium confidence

Фигура Анализ на ефективност при подредени данни

* Фигура 4: Илюстрира сравнението на паралелните алгоритми за търсене при 2 и 8 нишки и времето в наносекунди за намиране на елемент в структурата, при данни, генерирани в намаляващ ред.

A screenshot of a graph

Description automatically generated

Фигура Анализ на ефективност при обратно подредени данни

* Фигура 5: Представя сравнението на паралелните алгоритми за търсене при 2 и 8 нишки и времето в секунди за обработка на елементите и генериране на структурата, при данни, генерирани в нарастващ ред.

A comparison of a graph

Description automatically generated with medium confidence

Фигура Анализ на време на обработка на данните преди изпълнение на алгоритъма за търсене

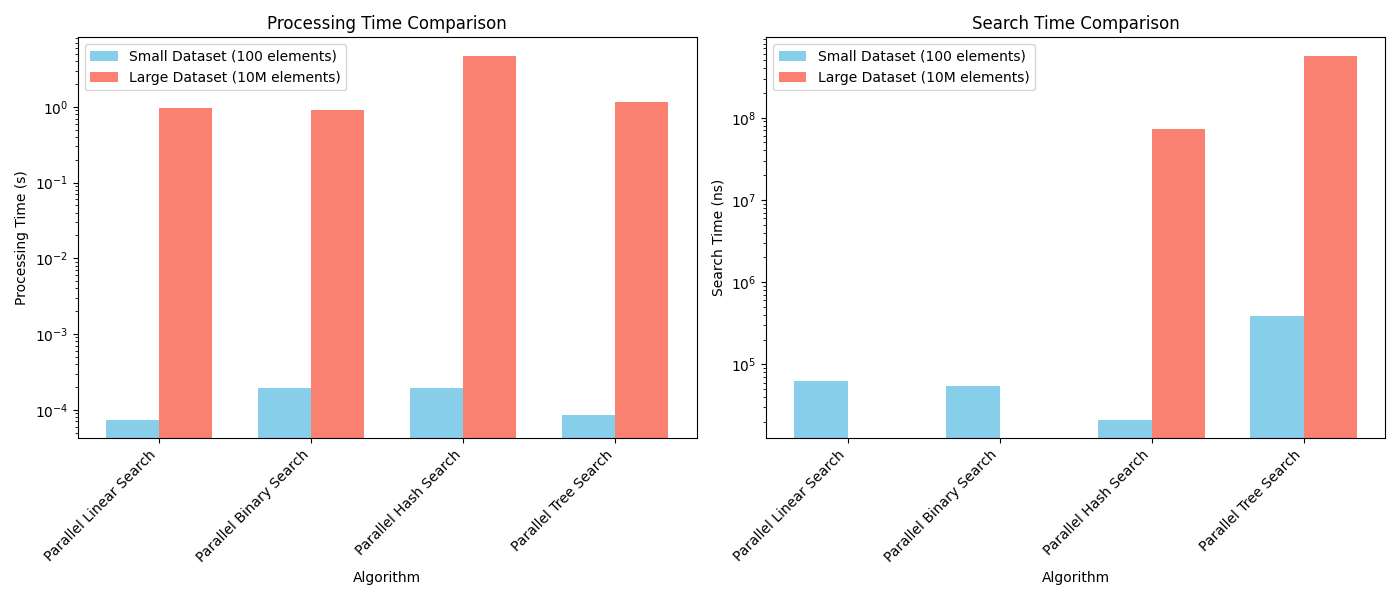
* Фигура 6: Показва сравнението на паралелните алгоритми за търсене при 8 нишки и времето в секунди за обработка на елементите и генериране на структурата, както и времето за намиране на елемент в нея в наносекунди, при различни подредби на данните и изпълнено върху „Опитна постановка 2“.

A screenshot of a graph

Description automatically generated

Фигура Анализ на ефективността на алгоритмите на "Опитна постановка 2"

* Фигура 7: Показва сравнението на паралелните алгоритми за търсене при 8 нишки и времето в секунди за обработка на елементите и генериране на структурата, както и времето за намиране на елемент в нея в наносекунди, при подредени данни от два сета от по съответно 100 и 10 000 000 елемента, скалирани логаритмично по оста „y”.

****

Фигура Анализ при различни сетове от данни

# **Обсъждане**

* 1. **Интерпретация на резултатите**

Анализът на времената за търсене в различни структури от данни (произволни, подредени и обратно подредени) показва специфични модели на производителност за всеки тип алгоритъм при изпълнение с различен брой нишки (2 и 8). Наблюденията по-долу обобщават последиците за производителността и ефективността на паралелните алгоритми в различни конфигурации.

* Основни наблюдения при произволно генерирани данни (Фигура 2)
  + Паралелното дървовидно търсене показва най-дългото време за търсене в наносекунди, особено при изпълнение с 2 нишки. Въпреки това, при 8 нишки времето за търсене значително намалява, което показва, че този алгоритъм се възползва от увеличената паралелност.
  + Паралелното хеш търсене демонстрира по-кратко време за търсене в сравнение с дървовидното търсене, но също така значително подобрява производителността при увеличаване на броя нишки от 2 на 8.
  + Паралелното линейно търсене показва най-кратко време за търсене, особено при изпълнение с 8 нишки, където разликата във времето между 2 и 8 нишки е минимална. Това може да предполага, че този алгоритми е по-малко чувствителен към подобрения в паралелността при случайно генерирани данни.
* Основни наблюдения при подредени данни (Фигура 3)
  + Паралелното бинарно търсене е най-ефективният алгоритъм за подредени данни, както се очаква, тъй като бинарното търсене се възползва от подредени структури. То показва минимално или нулево време за търсене и при 2, и при 8 нишки, което потвърждава пригодността му за сортирани структури от данни.
  + Паралелното линейно търсене също показва добра производителност при подредени данни, особено при 8 нишки. Въпреки това, то не е толкова ефективно, колкото бинарното търсене, тъй като линейното търсене не използва подредбата на данните за по-бързо намиране.
  + Паралелното дървовидно търсене показва намалено време за търсене в сравнение с произволно подредените данни, като се възползва частично от подредената структура, но остава по-бавно от бинарното и линейното търсене.
  + Паралелното хеш търсене не показва значително подобрение при подредени данни, което показва, че неговата производителност е слабо зависима от подредбата, тъй като хеш търсенето работи независимо от реда на данните.
* Основни наблюдения при обратно подредени данни (Фигура 4)
  + Паралелното линейно търсене все още поддържа по-кратко време за търсене в сравнение с дървовидното и хеш търсене, като разликите във времето между 2 и 8 нишки са минимални.
  + Паралелното дървовидно търсене показва високо време за търсене при 2 нишки, но значително се подобрява при 8 нишки, въпреки че остава най-бавният алгоритъм за обратно подредени данни.
  + Паралелното хеш търсене показва средно време за търсене между бинарното/линейното и дървовидното търсене. То се възползва от 8 нишки, но, както при случайни и подредени данни, неговата производителност остава сравнително независима от подредбата на данните.

В заключение:

* Паралелното бинарно търсене изпъква като най-ефективният алгоритъм за подредени данни, като се възползва от подредбата на структурата, но не е приложим при останалите типове подредба на данните. Това е предпочитаният избор в ситуации, където подредбата на данните е гарантирана, особено при 8 нишки, тъй като постига най-кратко време за търсене.
* Паралелното линейно търсене се представя стабилно във всички видове подредба на данните, особено при 8 нишки. Въпреки че не е толкова ефективно, колкото бинарното търсене при подредени данни, остава надеждно за различни структури от данни.
* Паралелното хеш търсене е със средна ефективност, но е относително нечувствително към подредбата на данните. Неговата производителност се подобрява с увеличаване на нишките, но не е оптималният избор за подредени или обратно подредени данни.
* Паралелното дървовидно търсене показва най-дълги времена за търсене във всички структури от данни, особено при по-малко нишки. То се възползва от увеличените нишки, но производителността му остава значително по-бавна от останалите алгоритми. Това предполага, че е по-малко подходящо за директни задачи за търсене в големи структури от данни, но може да има приложение при по-сложни структури от данни, където йерархичните връзки са важни.

За високопроизводителни задачи за търсене бинарното търсене е препоръчително за подредени данни поради минималното време за търсене. Линейното търсене може да служи като гъвкава алтернатива за различни структури от данни, като показва стабилни подобрения при повече нишки. Хеш и дървовидните търсения са по-подходящи за специализирани случаи и може да не предоставят оптимална производителност в големи, подредени структури от данни.

Сравнявайки резултатите за изпълнението на паралелните алгоритми на двете различни конфигурации, могат да се изведат следните изводи относно времето за търсене и времето за обработка на база „Опитна постановка 1“ и „2“:

* Общи наблюдения върху постановките
  + Опитна постановка 1:
    - Процесор: 12th Gen Intel Core i5-12450H, 2.00 GHz
    - RAM: 16 GB
    - Архитектура: Intel x86-64
  + Опитна постановка 2:
    - Процесор: Apple M3 Pro, 12-core CPU, 4.05 GHz
    - RAM: 36 GB
    - Архитектура: ARMv8.6-A

„Опитна постановка 1“ използва стандартна x86-64 архитектура, което може да се отрази върху съвместимостта с някои оптимизации за паралелизация. „Опитна постановка 2“ разполага с по-бърз процесор и повече RAM, което предполага по-добра производителност при многопоточни операции. ARM архитектурата може да е по-ефективна при паралелизация на специфични алгоритми.

#### **Сравнение на Времето за обработка (Processing Time)**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритъм | Брой нишки | Време за обработка (s) - Apple M3 Pro | Време за обработка (s) - Intel Core i5 |
| Паралелно линейно търсене | 8 | 0.6765 | 0.9229 |
| Паралелно бинарно търсене | 8 | 0.6908 | 0.9179 |
| Паралелно хеш търсене | 8 | 3.8307 | 4.8035 |
| Паралелно дървовидно търсене | 8 | 0.2120 | 1.3163 |

При паралелното линейно и бинарно търсене, конфигурацията с Apple M3 Pro показва по-ниско време за обработка, особено при бинарното търсене. ARM архитектурата извършва по-ефективни операции по достъп до паметта.

При паралелното хеш търсене конфигурацията с Apple M3 Pro също показва по-добри резултати, но разликата не е толкова значителна, което предполага, че хеш-търсенето е по-зависимо от RAM и управлението на паметта.

При паралелното дървовидно търсене разликата в производителността е значителна. Конфигурацията с Apple M3 Pro се справя по-добре, което показва, че многоядреният процесор и ARM архитектурата могат да оптимизират дървовидните структури по-ефективно.

#### **Сравнение на Времето за търсене (Search Time)**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритъм | Брой нишки | Време за търсене (ns) - Apple M3 Pro | Време за търсене (ns) - Intel Core i5 |
| Паралелно линейно търсене | 8 | 96084 | 0 |
| Паралелно бинарно търсене | 8 | 27584 | 0 |
| Паралелно хеш търсене | 8 | 58404084 | 70220900 |
| Паралелно дървовидно търсене | 8 | 1140437500 | 1499736600 |

При паралелното линейно и бинарно търсене конфигурацията с Apple M3 Pro показва минимално време за търсене, което демонстрира ефективността на ARM архитектурата при по-прости операции за търсене.

При паралелното хеш и дървовидно търсене Apple M3 Pro е по-бърз в хеш търсенето, но при дървовидното търсене Intel Core i5 показва значително по-дълго време. ARM архитектурата вероятно е по-оптимизирана за работа със структури като дървета.

* 1. **Ограничения на проекта**

Основните ограничения на този проект произтичат от липсата на тестове върху GPU и специфичните характеристики на използваните данни. Поради фокуса върху изпълнението на алгоритмите върху CPU, ефективността на тези паралелни алгоритми в GPU среда остава неоценена. Това ограничава анализа на производителността, тъй като много паралелни алгоритми могат да демонстрират значително по-добра ефективност при използване на GPU, където е налице по-голям брой ядра за едновременно обработване на данни.

Освен това, данните, използвани в тестовете, са ограничени до произволно генерирани, възходящо подредени и низходящо подредени набори от 10 милиона елемента. Въпреки че тези тестови набори отразяват типични структури, реалните сценарии често включват по-сложни структури от данни с различна плътност и хетерогенност на стойностите. Липсата на разнообразие в тестовите данни ограничава обобщаването на резултатите и приложимостта на изводите върху различни типове данни и контексти на използване.

# **Заключение**

Проведените тестове и анализи предоставят информация за практическото приложение на алгоритми за паралелно търсене – линейно, бинарно, хеш-базирано и дървовидно – в зависимост от различни условия, като специфика на данните и хардуерни конфигурации.

Бинарното търсене показва най-добра ефективност при подредени. Това е логично, тъй като алгоритъмът използва разпределението на данните за бързо елиминиране на половината от тях при всяка итерация, което го прави изключително подходящ за сортирани структури. Неговата производителност значително се подобрява при по-висок брой нишки, особено на хардуери с многоядрени процесори. Линейното търсене показва стабилна производителност при всички типове данни, което го прави универсален, но не винаги най-ефективен избор. То се представя добре в среди с малко на брой елементи или когато не се изисква специфично подреждане на данните. При паралелно изпълнение линейното търсене показва подобрение, но не толкова значително, колкото бинарното търсене. Хеш-базираното търсене демонстрира добри резултати при произволни данни, но неговата ефективност намалява при подредени и обратно подредени структури, където хеш-алгоритмите не могат да се възползват от подредбата на данните. То обаче се представя добре в случаи с бърз достъп до данни и показва подобрение при по-висок брой нишки. Дървовидното търсене показва най-високо време за търсене, особено при малък брой нишки. Неговото представяне се подобрява значително с увеличаването на нишките, тъй като дървовидните структури могат да бъдат лесно паралелизирани. Въпреки това, този тип алгоритъм е по-подходящ за случаи, където се търси хиерархична организация на данните.

Анализът показва, че за всички алгоритми увеличаването на броя нишки води до подобрение на производителността, особено при по-сложни алгоритми като дървовидното и хеш-базираните търсения. Паралелизацията позволява на алгоритмите да обработват големи обеми от данни по-ефективно, намалявайки времето за търсене и обработка. Увеличаването на нишките от 2 на 8, например, води до значително подобрение при дървовидното търсене, което се възползва от едновременното обработване на множество клонове в структурата.

Резултатите от тестовете на Apple M3 Pro и Intel Core i5 показват различия в производителността в зависимост от архитектурата на процесорите. Apple M3 Pro, който разполага с ARM архитектура и повече RAM, показва предимства при изпълнението на задачи, свързани с бърз достъп до паметта, като линейното и бинарното търсене. ARM архитектурата е известна с енергийната си ефективност и с оптимизация за многоядрена работа, което допринася за по-висока производителност при алгоритми, зависещи от честотата на паметта.

Intel Core i5, базиран на x86-64 архитектурата, показва стабилност и висока съвместимост с различни приложения, но се представя по-слабо при операции, които се възползват от паралелизацията, като дървовидното търсене. Това може да се дължи на по-малкия брой ядра и по-ниската RAM памет, което ограничава неговата способност да изпълнява едновременно множество задачи.

Един от основните ограничения на проекта е липсата на тестове върху графични процесори (GPU). GPU предоставят значителни предимства за паралелизация, особено за алгоритми, които изискват едновременно обработване на големи обеми от данни. Те биха позволили по-бързо изпълнение на алгоритмите и по-добра производителност, особено при хеш-базираните и дървовидните търсения.

# **Библиография**

1. https://robert1003.github.io/2020/02/05/parallel-binary-search.html
2. https://www.geeksforgeeks.org/linear-search-using-multi-threading/
3. https://library.fiveable.me/key-terms/exascale-computing/parallel-hash-based-search
4. https://aryanjindal20.medium.com/parallel-tree-level-finder-18f3458b8d3a
5. https://github.com/mtz1024/parallel-tree-search/blob/master/src/parallel\_search.cu

# **Приложение № 1: Графичен интерактивен интерфейс**

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Parallel Searching Algorithm</title>  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"> </head> <body> <div class="container mt-5">  <h1 class="text-center">Parallel Searching Algorithm</h1>  <div class="row">  *<!-- Column 1: Form -->* <div class="col-md-6">  <h3>Search settings:</h3>  <form id="searchForm" class="mt-4">  <div class="mb-3">  <label for="algorithm" class="form-label">Select Algorithm</label>  <select class="form-select" id="algorithm" name="algorithm" required>  <option value="" disabled selected>Select an algorithm</option>  <option value="linear">Parallel Linear Search</option>  <option value="binary">Parallel Binary Search</option>  <option value="hash">Parallel Hash-based Search</option>  <option value="tree">Parallel Tree Search</option>  </select>  </div>   <div class="mb-3">  <label for="dataType" class="form-label">Data Type</label>  <select class="form-select" id="dataType" name="dataType" required onchange="*toggleManualInput*()">  <option value="" disabled selected>Select data type</option>  <option value="ordered">Ordered</option>  <option value="reverse">Reverse Ordered</option>  <option value="random">Random</option>  <option value="manual">Manual</option>  </select>  </div>   <div id="manualDataSection" class="mb-3" style="display: none;">  <label for="manualData" class="form-label">Manual Data (comma-separated values)</label>  <input type="text" class="form-control" id="manualData" name="manualData">  </div>   <div id="generatedDataSection">  <div class="mb-3">  <label for="threadCount" class="form-label">Thread Count</label>  <input type="number" class="form-control" id="threadCount" name="threadCount"  placeholder="Enter thread count" value=1>  </div>  <div class="mb-3">  <label for="size" class="form-label">Data Size</label>  <input type="number" class="form-control" id="size" name="size"  placeholder="Enter size of data">  </div>  <div class="mb-3">  <label for="maxElement" class="form-label">Max Element Value</label>  <input type="number" class="form-control" id="maxElement" name="maxElement"  placeholder="Enter max element value">  </div>  </div>   <div class="mb-3">  <label for="key" class="form-label">Key to Search</label>  <input type="number" class="form-control" id="key" name="key" required  placeholder="Enter the key to search">  </div>   <button type="button" class="btn btn-primary" id="submitBtn">Search</button>  <button type="button" class="btn btn-secondary" id="saveBtn">Save Details</button>  </form>  </div>   *<!-- Column 2: Results -->* <div class="col-md-6">  <h3>Result:</h3>  <div class="mt-5">  <div id="result"></div>  </div>  </div>  </div> </div>  <script>  function *toggleManualInput*() {  const dataType = document.getElementById('dataType').value;  const manualDataSection = document.getElementById('manualDataSection');  const generatedDataSection = document.getElementById('generatedDataSection');   if (dataType === 'manual') {  manualDataSection.style.display = 'block';  generatedDataSection.style.display = 'none';  } else {  manualDataSection.style.display = 'none';  generatedDataSection.style.display = 'block';  }  }   document.getElementById('submitBtn').addEventListener('click', async function () {  const form = document.getElementById('searchForm');  const formData = new *FormData*(form);   const alertMessage = document.createElement('div');  *initStyle*(alertMessage);  document.body.appendChild(alertMessage);   try {  const response = await *fetch*('/', {  method: 'POST',  body: new *URLSearchParams*(formData),  });   alertMessage.remove();   if (response.ok) {  const result = await response.json();  window.latestSearchResult = result;  *updateResult*(result);  } else {  document.getElementById('result').innerHTML = 'An error occurred while processing your request.';  }  } catch (error) {  alertMessage.remove();  document.getElementById('result').innerHTML = 'A network error occurred: ' + error.message;  }  });   document.getElementById('saveBtn').addEventListener('click', async function () {  const searchResult = window.latestSearchResult;  if (!searchResult) {  *alert*("No results to save!");  return;  }   try {  const response = await *fetch*('/save', {  method: 'POST',  headers: {  'Content-Type': 'application/json'  },  body: JSON.stringify(searchResult)  });   const result = await response.json();  *alert*(result.message);  } catch (error) {  *alert*('Error saving results: ' + error.message);  }  });   function *updateResult*(data) {  const resultDiv = document.getElementById('result');  resultDiv.innerHTML = `  <p><strong>Found:</strong> ${data.result}</p>  <p><strong>Searched Element:</strong> ${data.searched\_key}</p>  <p><strong>List Size:</strong> ${data.list\_size}</p>  <p><strong>Range:</strong> ${data.range}</p>  <p><strong>Search Time (seconds):</strong> ${data.search\_time}</p>  `;  }   function *initStyle*(alertMessage) {  alertMessage.id = 'alertMessage';  alertMessage.style.position = 'fixed';  alertMessage.style.top = '20px';  alertMessage.style.left = '50%';  alertMessage.style.transform = 'translateX(-50%)';  alertMessage.style.padding = '10px 20px';  alertMessage.style.backgroundColor = '#FFFFFF';  alertMessage.style.color = '#000000';  alertMessage.style.border = '5px solid #3B0BD7FF';  alertMessage.style.borderRadius = '10px';  alertMessage.innerText = 'This could take a while!';  } </script>  </body> </html> |

# **Приложение № 2: Запазване на тестовите данни в JSON файл**

|  |
| --- |
| package handlers  import (  "encoding/json"  "fmt"  "net/http"  "os"  "time" )  *// SaveHandler handles the saving of search details to a file.* func SaveHandler(w http.ResponseWriter, r \*http.Request) {  if r.Method != http.MethodPost {  http.Error(w, "Invalid request method", http.StatusMethodNotAllowed)  return  }   var searchDetails map[string]interface{}  err := json.NewDecoder(r.Body).Decode(&searchDetails)  if err != nil {  http.Error(w, "Invalid JSON input", http.StatusBadRequest)  return  }   filePath := fmt.Sprintf("resources/results/search\_results\_%d.json", time.Now().UnixNano())  err = saveToFile(filePath, searchDetails)  if err != nil {  http.Error(w, "Error writing to file", http.StatusInternalServerError)  return  }   w.Header().Set("Content-Type", "application/json")  \_ = json.NewEncoder(w).Encode(map[string]string{"message": "Search details saved successfully!"}) }  func saveToFile(filePath string, data interface{}) error {  file, err := os.OpenFile(filePath, os.O\_CREATE|os.O\_WRONLY, 0644)  if err != nil {  return err  }  defer func(file \*os.File) {  \_ = file.Close()  }(file)   encoder := json.NewEncoder(file)  encoder.SetIndent("", " ")  return encoder.Encode(data) } |