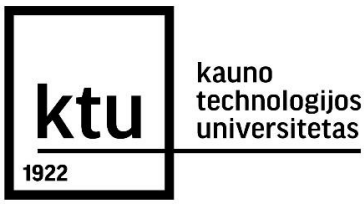


KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Inžinerinis projektas

P170B328 Lygiagretusis programavimas

Darbą atliko:
Karolis Samuolis IFF-7/13 gr.

Darbą priėmė:
lekt. BARISAS Dominykas
lekt. VASILJEVAS Mindaugas

KAUNAS, 2019

Turniys

1 Užduotis 3
2 Programos aprašymas 3
3 Programos kodas 4
4 Vykdymo laiko kitimo tyrimas 8
4.1 10000 operacijos 8
4.2 100000 operacijos 9
4.3 1000000 operacijos 10
4.4 10000000 operacijos 11
5 Išvados 12

1 Užduotis

Užduotis. Surikiuoti sugeneruotą masyvą naudojant „merge sort“ metodą.

Parašyti programą, kuri, naudojant lygiagretų programavimą, surikiuotų sugeneruotą n elementų masyvą, naudojant „merge sort“ rikiavimo metodą. Palyginti rezultatus su nelygiagretinto programavimo rikiavimu naudojant skirtingą kiekį operacijų semafore

2 Programos aprašymas

Programos eiga

1. Pagrindinė gija sugeneruoja masyvą s.
2. Pagrindinė gija paleidžia nelygiagretų rikiavimą.
3. Pagrindinė gija paleidžia funkciją, kuri paleidžia duomenų semaforą, kuris perduoda i kiekį duomenų į rikiavimo metodą.
4. Rikiavimo metode rekursiškai kviečiamas rikiavimo metodas su kaire ir dešine puse duomenų
5. Semaforas laukia kol visos operacijos yra atliktos, kol perduoda kitus i duomenų.
6. Pagrindinė gija skaičiuoja kiek laiko visi procesai užtruko.
7. Pagrindinė gija spausdina rezultatus.

Programa naudoja semaforus duomenų dalinimui

3 Programos kodas

```
func main() {
    // Array size
    var size int
    var threadCount int
    var err error
    // Check if there is an argument in main call
    if len(os.Args) < 3 {
        // set size if not to 10000
        size = defaultSize
        threadCount = defaultThreads
    } else {
        // set size to the argument
        size, err = strconv.Atoi(os.Args[1])
        threadCount, err = strconv.Atoi(os.Args[4])
    }
    // Check if number conversion didnt fail
    if err != nil {
        fmt.Println("Convert number incorrectly")
        os.Exit(1)
    }
    // Generate new array
    s := generate(size)
    // Array for single threaded sequential sort
    seq := s
    // Array for parallel sort
    par := s
    fmt.Println("----------------------------------------")
    // Single threaded merge sort
    fmt.Println("Sequential")
    // Start a timer for benchmarking
    start := time.Now()
    // Call sort func
    SingleMergeSort(seq)
    // Find how much time has passed
    fmt.Println(time.Since(start))
    fmt.Println("/// /// ///")
    // Parallel merge sort
    fmt.Println("Parallel with ", threadCount, " max operations per semaphore")
    // Start a timer for benchmarking
    start := time.Now()
    // Call sort manager func
    RunMultiMergesortWithSem(par, threadCount)
    // Find how much time has passed
    elapsed := time.Since(start)
    fmt.Println(elapsed.Seconds())
    fmt.Println("----------------------------------------")
}
```

```go
// Parallel merge sort manager
func RunMultiMergesortWithSem(data []float64, threadCount int) {
    // Make {threadCount} semaphores
    // This will let only up to {threadCount} concurrent operations
    // other data will have to wait until the others are done
    sem := make(chan struct{}, threadCount)
    // Call main merge func with semaphores
    MultiMergeSortWithSem(data, sem)
}
```

```go
// Main parallel merge sort func
func MultiMergeSortWithSem(data []float64, sem chan struct{}) []float64 {
    // If data has less than 2 elements
    if len(data) < 2 {
        return data
    }
    //Find middle lenght
    middle := len(data) / 2
    //Assign a wait group
    wg := sync.WaitGroup{}
    // Adds delta for wg
    // If wg falls below negative, it panics
    // If wg equals to 0, all goroutines are released
    wg.Add(2)
    // left and right data arrays
    var ldata []float64
    var rdata []float64
    select {
    // Run parallel merge sort with left data
    case sem <- struct{}{}:
        // Parallel func call
        go func() {
            // Merge sort with only left half of data (recursive)
            ldata = MultiMergeSortWithSem(data[:middle], sem)
            // Release the semaphore
            <-sem
            // Finish the job
            wg.Done()
        }()
    default:
        // No semaphore = no parallel sort
        ldata = SingleMergeSort(data[:middle])
        // Finish work
        wg.Done()
    }
```

```
    select {
    // Run merge sort with right data
    case sem <- struct{}{}:
        // Parallel func call
        go func() {
            // Merge sort with only right half of data (recursive)
            rdata = MultiMergeSortWithSem(data[middle:], sem)
            // Release the semaphore
            <-sem
            // Finish the job
            wg.Done()
        }()
    default:
        // No semaphore = no parallel sort
        rdata = SingleMergeSort(data[middle:])
        // Finish work
        wg.Done()
    }
    // Wait until wg hits 0, then release all goroutines
    wg.Wait()
    // Merge sorted data
    return Merge(ldata, rdata)
}
```

```
func Merge(ldata []float64, rdata []float64) (result []float64) {
    // Results output
    result = make([]float64, len(ldata)+len(rdata))
    // Left and right indexes
    leftId, rightId := 0, 0
    //Find which side is higher to switch
    for i := 0; i < cap(result); i++ {
        switch {
        // If left index is higher or eq to len of left data
        case leftId >= len(ldata):
            // i'th result = rightId'th right element
            result[i] = rdata[rightId]
            rightId++
            // If right index is higher or eq to len of right data
        case rightId >= len(rdata):
            // i'th result = leftId'th left element
            result[i] = ldata[leftId]
            leftId++
            // If leftId'th left element is lower than rightId'th right element
        case ldata[leftId] < rdata[rightId]:
            // i'th result = leftId'th left element
            result[i] = ldata[leftId]
            leftId++
```

```
            // If rightId'th right element is lower than leftId'th left element
        default:
            // i'th result = rightId'th right element
            result[i] = rdata[rightId]
            rightId++
        }
    }
    return
}
```

```
func SingleMergeSort(data []float64) []float64 {
    // If data has less than 2 elements
    if len(data) < 2 {
        return data
    }
    // Find the middle counter
    middle := len(data) / 2
    // Merge left and right arrays
    return Merge(SingleMergeSort(data[:middle]), SingleMergeSort(data[middle:]))
}
```

```
func generate(count int) []float64 {
    s := make([]float64, count)
    for i := 0; i < cap(s); i++ {
        s[i] = rand.Float64() * float64(count)
    }
    return s
}
```

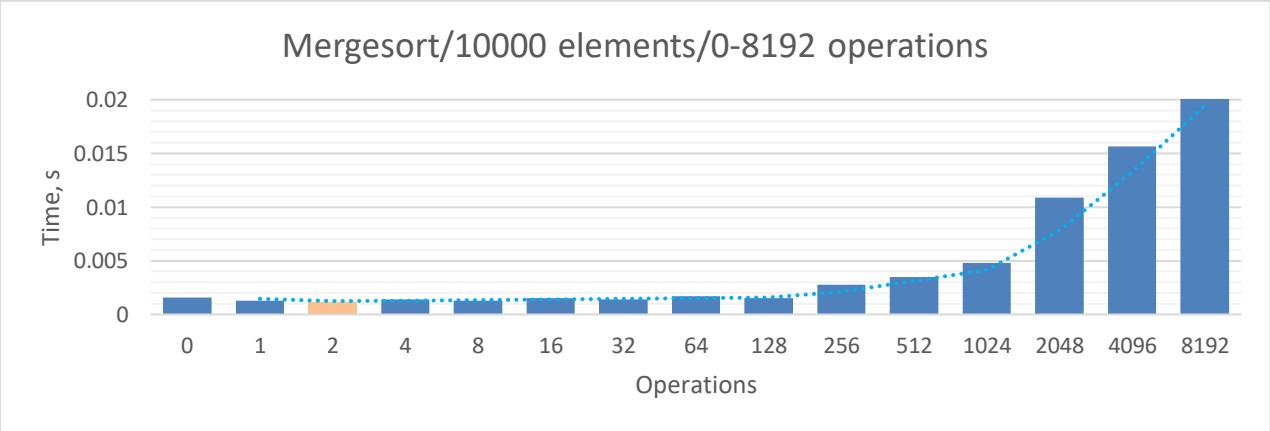
```
const defaultThreads = 4
const defaultSize = 10000
const runCounter = 1
```

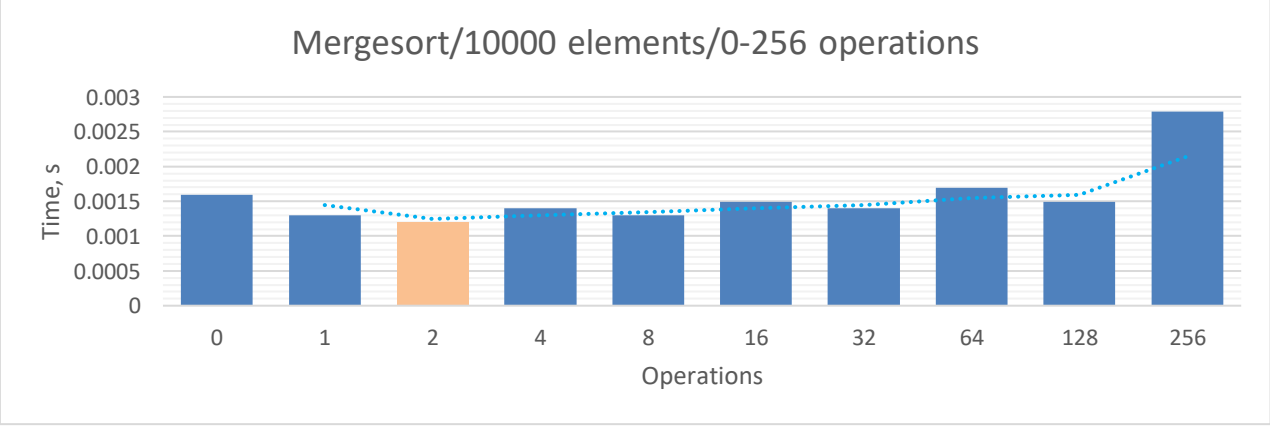
4 Vykdymo laiko kitimo tyrimas

Vykdymo laiko kitimo tyrimas darytas po 10 kartų su kiekvienu operacijų kiekiu. Lentelėse yra pateikti rezultatų vidurkiai. Tyrimo metu pastebėta, kad optimaliausias operacijų kiekis priklauso nuo duomenų kiekio. Didelis operacijų kiekis labai sulėtina rikiavimo greitį. Grafikuose ir lentelėse yra pažymėti optimaliausi operacijų kiekiai.

4.1 10000 operacijos

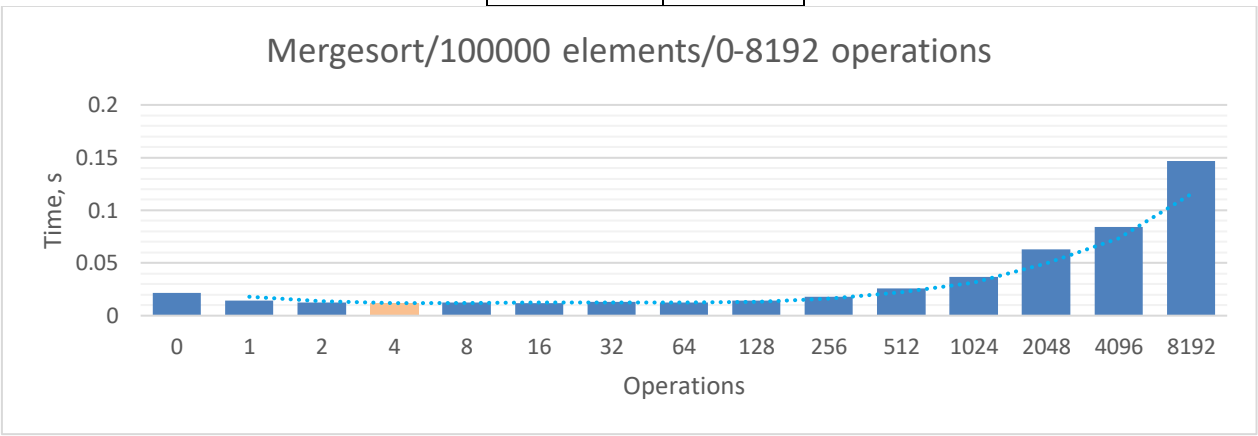
Operacijos	Laikas, s
0	0.001596
1	0.001297
2	**0.001197**
4	0.001396
8	0.001297
16	0.001495
32	0.001396
64	0.001695
128	0.001496
256	0.002793
512	0.00349
1024	0.004787
2048	0.010871
4096	0.015657
8192	0.02324

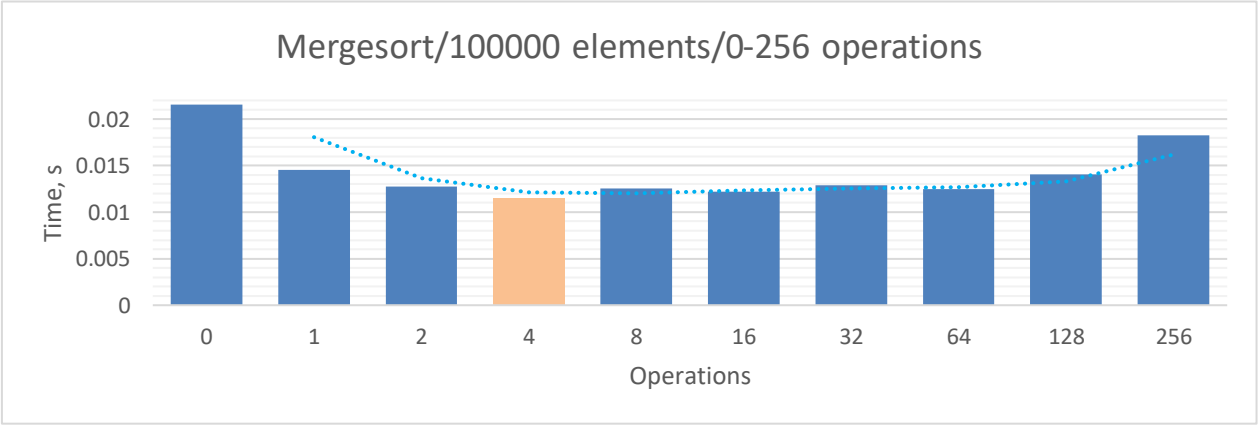




4.2 100000 operacijos

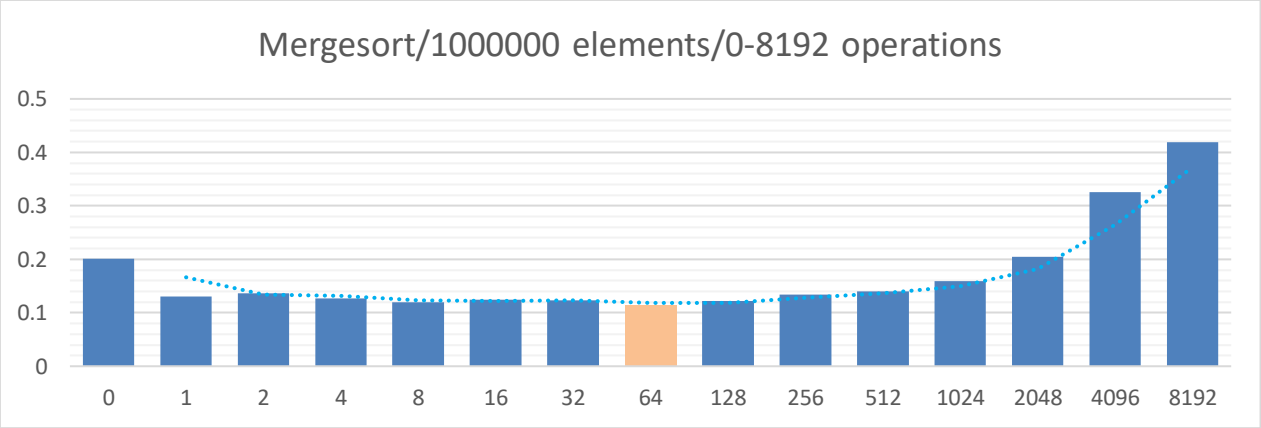
Operacijos	Laikas, s
0	0.021543
1	0.014562
2	0.012767
4	**0.011469**
8	0.012567
16	0.012167
32	0.012865
64	0.012468
128	0.014062
256	0.018252
512	0.026031
1024	0.0367
2048	0.062832
4096	0.084076
8192	0.146509

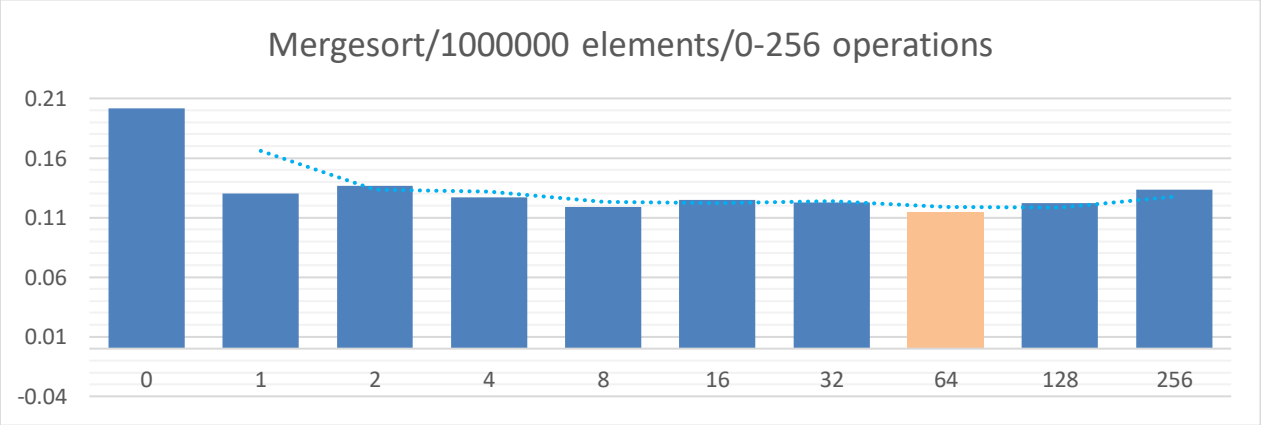




4.3 1000000 operacijos

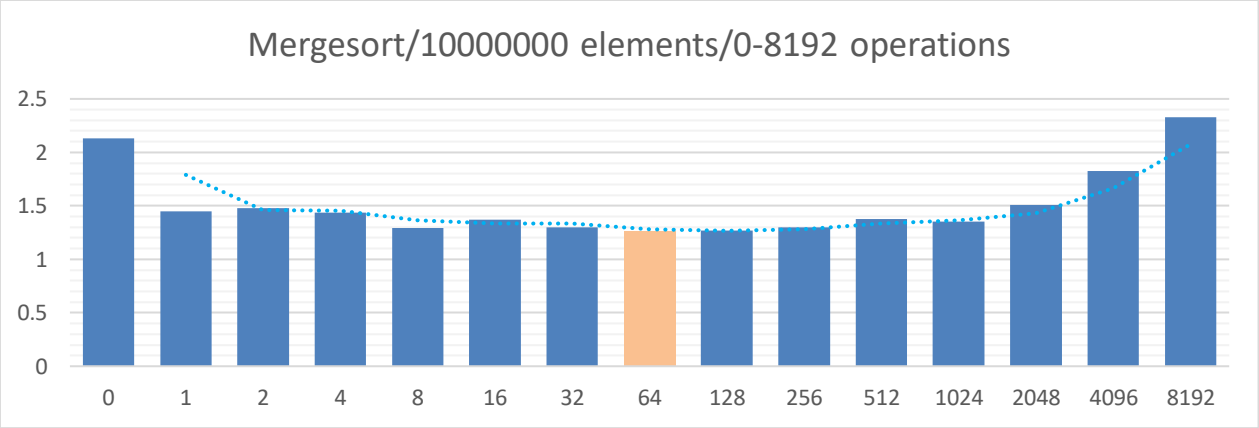
Operacijos	Laikas, s
0	0.201645
1	0.130351
2	0.136634
4	0.126861
8	0.119182
16	0.124663
32	0.122671
64	**0.114789**
128	0.121974
256	0.133443
512	0.139427
1024	0.158675
2048	0.204254
4096	0.325529
8192	0.41858

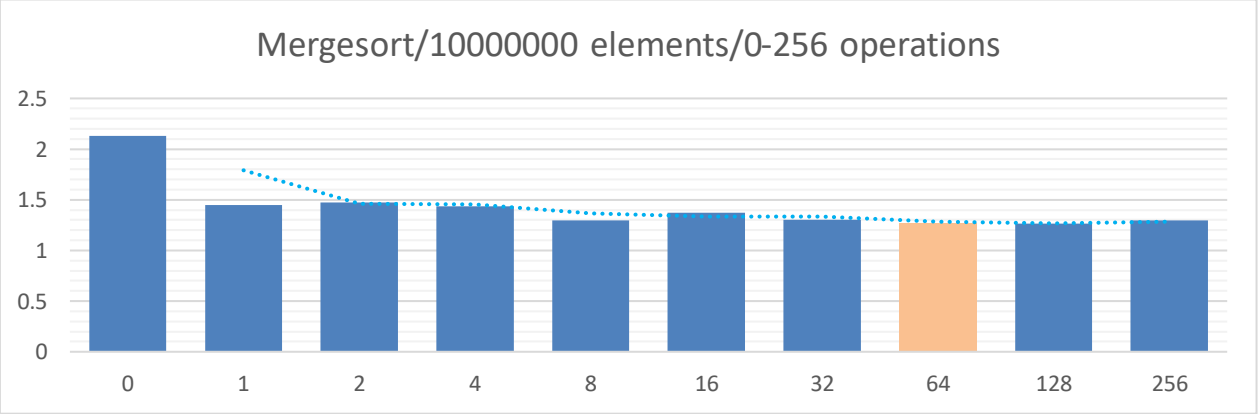




4.4 10000000 operacijos

Operacijos	Laikas, s
0	2.131092
1	1.450719
2	1.475935
4	1.437251
8	1.293724
16	1.373416
32	1.301206
64	**1.265794**
128	1.26779
256	1.29601
512	1.373921
1024	1.355261
2048	1.505953
4096	1.824213
8192	2.327555





5 Išvados

Parašytoje programoje lygiagretumas suteikė apytiksliai 1,5 našumo palyginus su programos veikimu be semaforų, kai operacijų kiekis nėra labai dielis, tačiau su labai dideliu operacijų kiekiu lygiagretumas veikia lėčiau nei viena gija. Taip yra dėl to, kad vienu metu yra atliekama labai daug operacijų, jų našumas sparčiai mažėja.