

Проект по Системи за Паралелна Обработка на тема Изобразяване на фрактал

Хакан Сунай Халил, Компютърни науки,
3^{ти} курс, 2^{ра} група,
ФН: 81406

Софийски университет “Св. Климент Охридски”
Факултет по математика и информатика

Проверил:
/ас. Христо Христов/

Цел

Целта на този проект е да визуализира множеството на Манделброт, определено от следната формула:

$$F(Z) = C * \cos(Z)$$

Програмата трябва да използва паралелни процеси (нишки) за да разпредели работата по търсенето на точките от множеството на Манделброт на повече от един процесор. Език за програмиране: **GoLang**.

Изисквания

- Генериране на изображение в png формат, показващо намереното множество;
- Команден параметър, който задава големината на генерираното изображение, като широчина и височина в брой пиксели във вида:
-s 640x480
При невъведен параметър, програмата подразбира - width 640px и height 480px;
- Команден параметър, който да задава частта от комплексната равнина, в която ще търсим визуализация на множеството на Манделброт в следния вид:
-r -2.0:2.0:-1.0:1.0
Стойността на параметъра се интерпретира както следва:
 $a \in [-2.0, 2.0], b \in [-1.0, 1.0]$. При невъведен параметър програмата приема че е зададена стойност по подразбиране: -2.0:2.0:-1.0:1.0.
- Команден параметър, който задава максималния брой нишки (паралелни процеси), на които разделяме работата по генерирането на изображението:
-t 3
При невъведен параметър програмата подразбира 1 нишка;
- Команден параметър указващ името на генерираното изображение:
-o zad18.png
При невъведен параметър програмата подразбира zad18.png.

Допълнителни параметри

- Команден параметър регулиращ "сложността" на фрактала, т.е абсолютната стойност , която не трябва да се надвиши, за да се остане в множеството на Манделброт.
-c 3 (default 8)
- Команден параметър указващ максималният брой итерация на точка за принадлежност в множеството на Манделброт
-i 100 (default 50)

Алгоритми

1. Манделброт алгоритъм

Програмата използва всеизвестния алгоритъм за намиране на множеството на Манделброт. Той може да се опише по следния начин:

1. Взима се нова точка от комплексната равнина := **num**.
2. Използвайки тази точка се смята $Z_n = \cos(Z_{n-1}) * \text{num}$, като се има предвид, че $Z_0 = 0 + 0i = 0$
3. Този изчислителен процес продължава, докато се достигнат максималния брой итерации (указан от параметър) или абсолютната стойност на някое Z_i надвиши параметъра за сложност, в такъв случай казваме, че точката не принадлежи на множеството.
! За целите на проекта, пазим броя на извършените итерации, тъй като с тяхна помощ прецизно визуализираме множеството на Манделброт.
4. Обратно към стъпка 1.

```
func (c *Converter) mandelbrot(num complex128) uint8 {
    currentIterations := uint8(0)
    for z := num; cmplx.Abs(z) <= c.Complexity && currentIterations <
c.MaxIterations; currentIterations++ {
        z = cmplx.Cos(z) * num
    }
    return currentIterations
}
```

Тъй като разполагаме само с размерите на изображението, което трябва да генерираме, преобразуваме пикселите на изображението в точки от комплексната равнина. Имайки предвид, че всеки пиксел се определя еднозначно от две стойности: ширина и височина, транслираме пикселите използвайки входните параметри с долния алгоритъм.

```
func (c *Converter) pixelToComplex(x, y int) complex128 {  
    return  
    complex(c.RealMin+(float64(x)/float64(c.Width))*(c.RealMax-c.RealMin),  
           c.ImagMin+(float64(y)/float64(c.Height))*(c.ImagMax-c.ImagMin))  
}
```

2. Въведение в GoLang

Какво представляват Горутините?

Goroutines са функции, които се изпълняват едновременно с други функции. За тях може да си мислим като много леки нишки. Цената на създаването на горутина е малка в сравнение с OS нишка. Оттук следва и възможността на Go да поддържа хиляди горутини едновременно.

Goroutines са изключително евтини в сравнение с threads. Те са само няколко kb в размер на стека и той може да расте и да се свива според нуждите на приложението, докато в случаите на нишки размерът на стека трябва да бъде фиксиран.

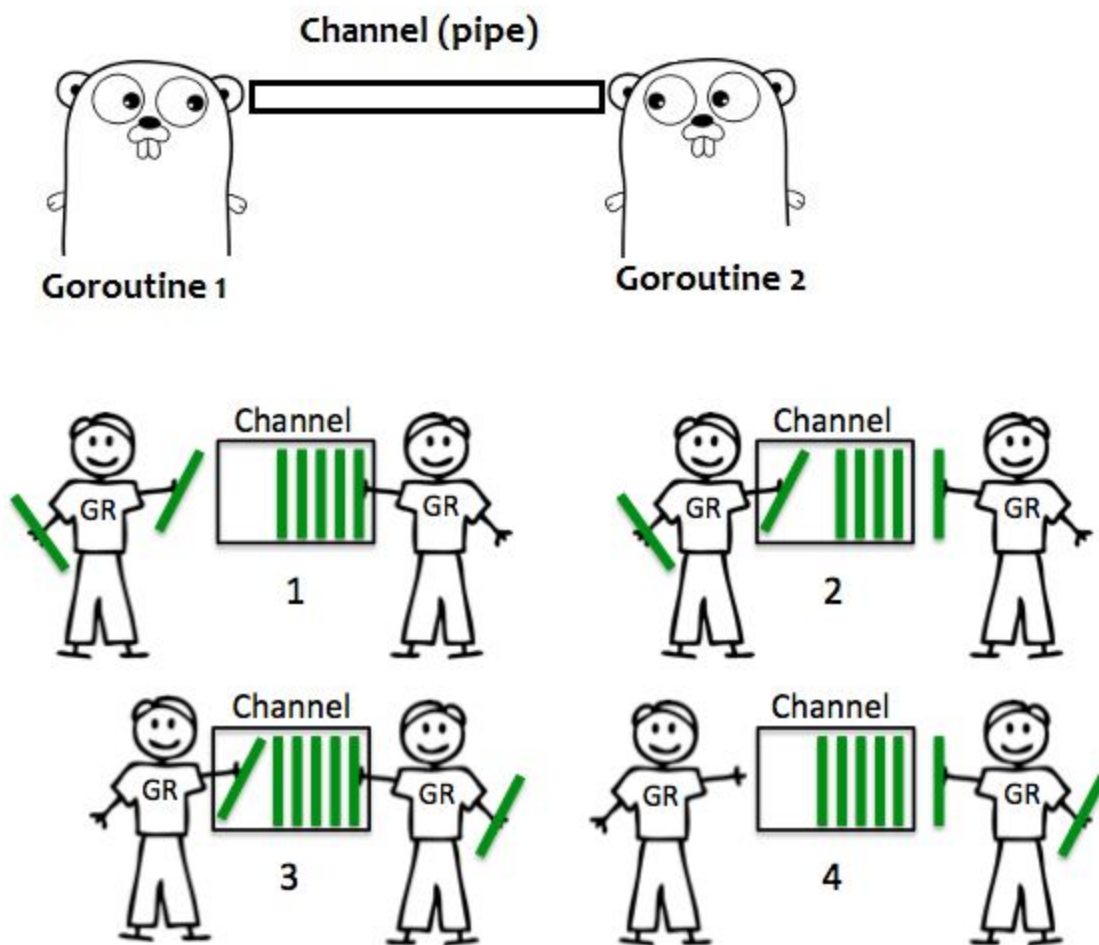
Goroutines се мултиплексират с по-малък брой нишки на OS. Може да има само една нишка в програма с хиляди Goroutines. Ако някой Goroutine в тази блокова нишка каже, че чака Input operation, тогава се създава друга нишка на OS и останалите Goroutines се преместват на нея.

Традиционните модели за нишки (обикновено използвани при писане на програми на Java, C и Python) изискват програмистът да комуникира между нишките с обща памет. Обикновено споделените структури от данни са защитени от ключалки, а нишките се борят за тези достъп до критичната секция, защитена от ключалката. В Go са въведени **каналите**.

Какво представляват каналите?

Каналът е вграден тип в GoLang, който се използва за комуникация между различни горутини и най-интересното е, че може да се използва и за синхронизация. Каналите могат да се разглеждат като pipes, които използват Goroutines. Подобно на това как водата тече от единия до другия край на тръба, данните могат да се изпращат от единия край и да се получават от другия край. Изпращането и получаването блокират, когато каналът се изпразни, докато някой "отсреща" не извърши "противоположната" операция. Каналът може да бъде използван във for цикли, където итерацията блокира докато някой "отсреща" не изпрати нещо по канала, а когато това се случи, се работи с получената стойност. Итерирането продължава, докато каналът не бъде затворен. GoLang насърчава следното:

Do not communicate by sharing memory; instead, share memory by communicating.

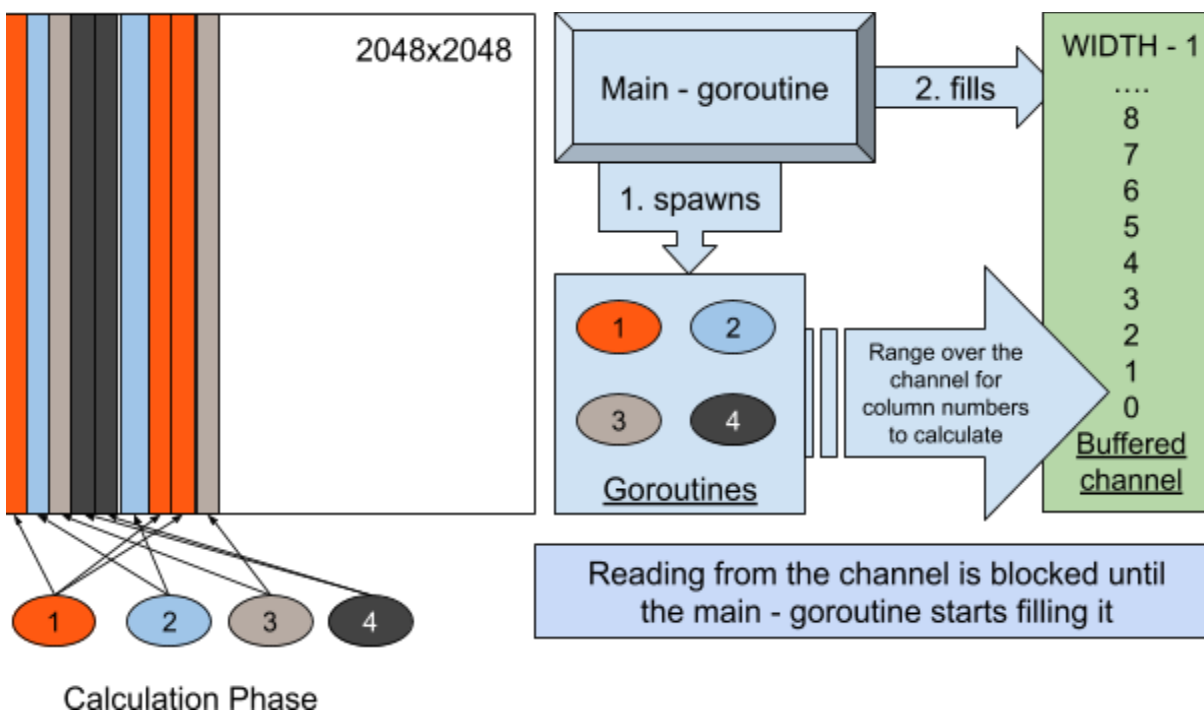


3. Паралелен алгоритъм

Програмата е имплементирана на GoLang 1.11.1. Реализиран е паралелизъм по данни - SPMD, като работата на горутините (нишките) е асинхронна. Архитектурата на програмата е по модела Master-Slaves. Main горутината влиза в ролята на Master, а другите горутини, които биват пускани от нея в по-късен етап от изпълнението, в ролята на Slaves. Идеята на алгоритъма е следната:

- В зависимост от параметъра за широчина, се създава **буфериран канал** за комуникация между горутини с размер **WIDTH**.
- Main горутината пуска **N-1** на брой нови горутини, които изпълняват функцията **calculateColumn**, която от своя страна започва да цикли канала.
- Main горутината започва да пълни канала с числата от **0** до **WIDTH-1**.
- Новите горутини започват да получават стойности по канала на всяка итерация.
- Всяка горутина изчислява цяла колона за една итерация и след това чете номера на новата колона за изчисляване от канала.
- По този начин картината се разделя на **WIDTH** на брой колони / стълбове, които се изчисляват едновременно от **N-1** на брой горутини, а същевременно Main горутината чака тяхното приключване използвайки WaitGroup.

По този начин задачата се разделя на **width** на брой подзадачи между различните горутини, които биват менажирани от буферирания канал, играещ ролята на load balancer. Следователно, най-фината възможна грануларност се постига при използване на 32 процесорни ядра, а най-едрата при използване на 1 ядро.



Main goroutine

```
c := make(chan int, width) // създава се буфериран канал
var w sync.WaitGroup

start := time.Now()

for n := 0; n < workers; n++ {
    w.Add(1)
    // пускат се горутините
    go calculateColumn(&w, &c, height, converter, pixels)
}

for i := 0; i < width; i++ {
    c <- i // пълни се каналаът със стойности от 0 до width
}

close(c)
w.Wait()
```

Worker goroutines

```
func calculateColumn(w *sync.WaitGroup, c *chan int, height int, converter
*Converter, pixels [][]color.NRGBA) {
    for x := range *c { // итерира се каналаът за стойности на колони
        for y := 0; y < height; y++ {
            complexNumber := converter.pixelToComplex(x, y)
            if iterations := converter.mandelbrot(complexNumber);
iterations < converter.MaxIterations {
                pixels[x][y] = color.NRGBA{iterations * 255, iterations
* 50, iterations * 20, 255}
            } else {
                pixels[x][y] = color.NRGBA{0, 0, 0, 255}
            }
        }
    }
    w.Done()
}
```

Резултати

Ще покажем и сравним две различни изпълнения на програмата със следните размери за **Width** x **Height** (*и параметрите по подразбиране*) на 1 ... 32 ядра:

1. **2048 x 2048**
2. **4096 x 4096**

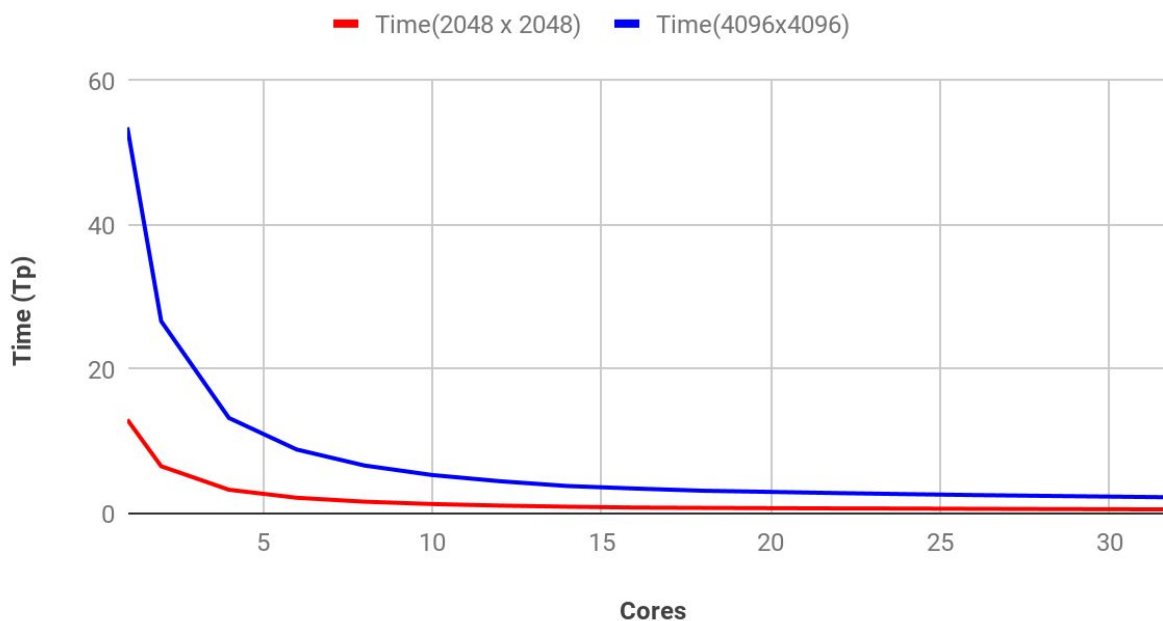
Cores	Time(2048x2048)	Speedup(2048)	Efficiency(2048)	Time(4096x4096)	Speedup(4096)	Efficiency(4096)
1	13.00012677	1	1	53.49606524	1	1
2	6.547454908	1.985524	0.9927618	26.64032828	2.008085812	1.004042906
4	3.290179727	3.95119	0.9877976	13.23491641	4.04204028	1.01051007
6	2.178843323	5.966527	0.9944211	8.865159239	6.034416731	1.005736122
8	1.634307742	7.954516	0.9943145	6.647784519	8.047202054	1.005900257
10	1.308399849	9.935897	0.9935897	5.328971791	10.03872179	1.003872179
12	1.092891055	11.89517	0.9912643	4.469509939	11.96911204	0.9974260035
14	0.936950409	13.87494	0.9910668	3.805298386	14.05831023	1.004165017
16	0.822074207	15.81381	0.9883632	3.456193561	15.47831864	0.9673949153
18	0.776222973	16.74793	0.9304405	3.151539024	16.97458443	0.9430324681
20	0.737524073	17.62672	0.8813358	2.98951668	17.89455319	0.8947276594
22	0.69968963	18.57985	0.8445385	2.825001679	18.93664901	0.860756773
24	0.682243625	19.05496	0.7939568	2.681465541	19.95030867	0.8312628613
26	0.645279365	20.14651	0.7748657	2.553662392	20.94876183	0.8057216087
28	0.612808719	21.214	0.757643	2.443689951	21.89151091	0.7818396755
30	0.586022807	22.18365	0.7394551	2.340974702	22.85204756	0.7617349188
32	0.561048159	23.17114	0.7240982	2.23689103	23.9153649	0.7473551533

Колоните **Time** показват времето за изпълнение в секунди на програмата при използване на **n** на брой процесорни ядра. $T_1 \dots T_{32}$

Колоните **Speedup** показват ускорението на нашата програма при използване на **n** на брой процесорни ядра. $S_p = T_1 / T_p$

Колоните **Efficiency** показват ефективността на нашата програма при използване на **n** на брой процесорни ядра. $E_p = S_p / p$

Time Chart

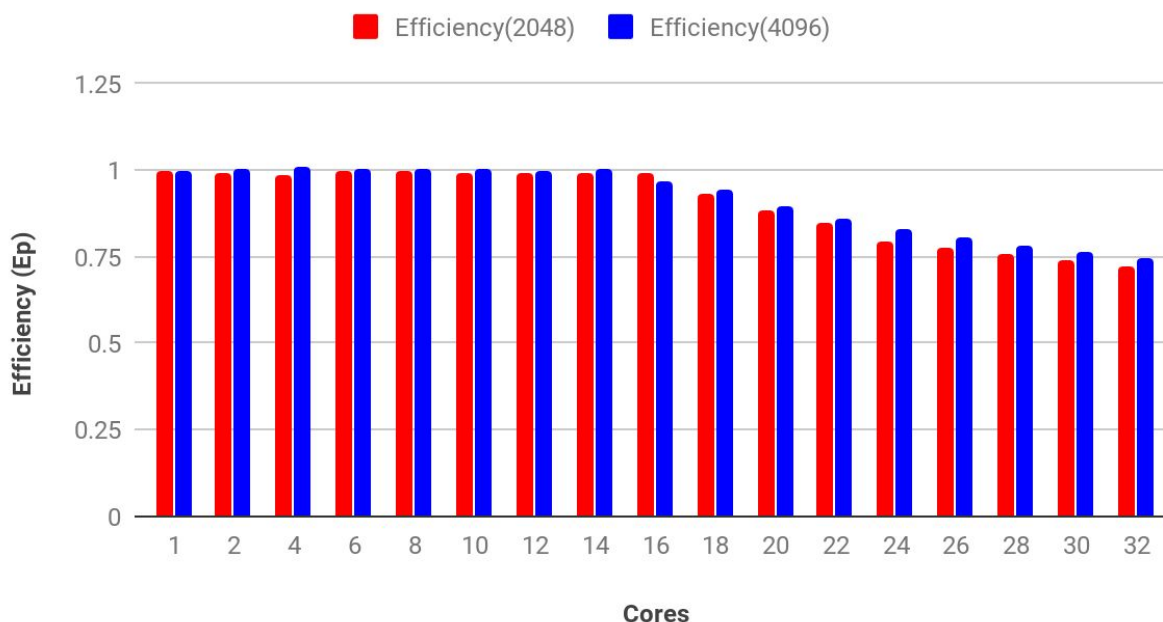


Speedup Chart



На тази графика със сив цвят и име Cores е изобразено **идеалното ускорение**.

Efficiency Chart



Извод

След анализиране на горните таблици, достигаме до извода, че при най-едрата грануларност (1 процесорно ядро) ускорението и ефективност са значително по-ниски в сравнение с останалите. С нарастване на броя процесорните ядра, ефектът на грануларността се увеличава, като при 32 ядра най-фината грануларност отчита най-високото ускорение - 23-24 пъти спрямо най-едрата.

Тези тестове са изпълнени на машина със следните параметри:

Architecture: x86_64
CPU(s): 32
Model name: Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
RAM: 62GB

Графичен резултат за размерност 2048x2048

2048.png

897,7 kB

