Date: 02/21/2020
Name: David Goerig (djg53)
School: School of Computing
Programme of Study: Advanced Computer Science (Computational Intelligence)
Module name: C0890 Concurrency and Parallelism
E-mail: djg53@kent.ac.uk
Academic Adviser: DR P Kenny (P.G.Kenny@kent.ac.uk)

Mandelbrot in Go

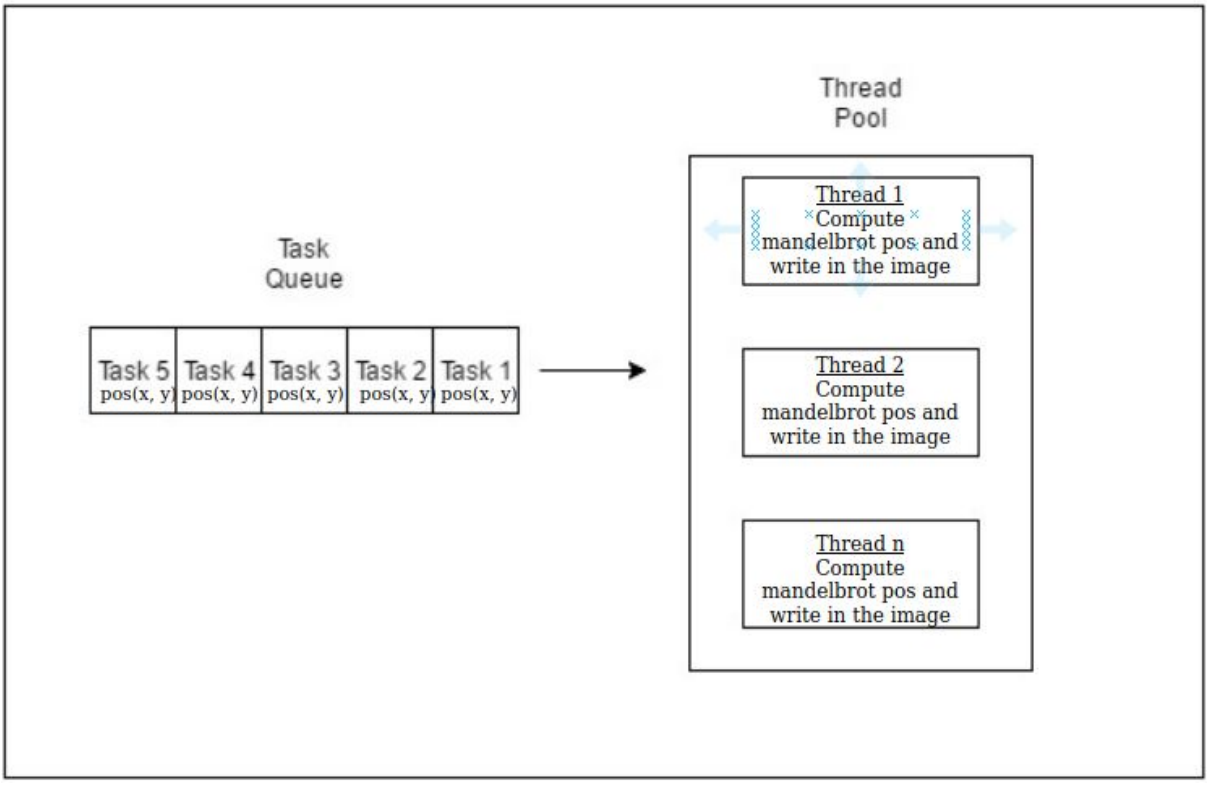
1. My Implementation
   a. Thread pool / workers implementation

For my implementation of thread pool I am using one channel.

The first step is to “initialize the worker”. In order to do that I set, for every worker (variable workerPool int is defining this number, it can change with input parameter), what they need to do with the data they will get through a chanel. This channel send to the worker the “position” (x, y) corresponding to the position in the image, where we need to calculate the mandelbrot value. It is also writing the calculated answer in the Image (so we don’t need to create another result channel, or compute it when job is done).

After this initialisation, we launch the work, so we iterate through all the mandelbrot positions and we send them to the workers via the channel. When a worker will be available he will take the positions value and compute the mandelbrot calculation.

Scheme of the task queue and thread pool:



b. Geometric distribution implementation

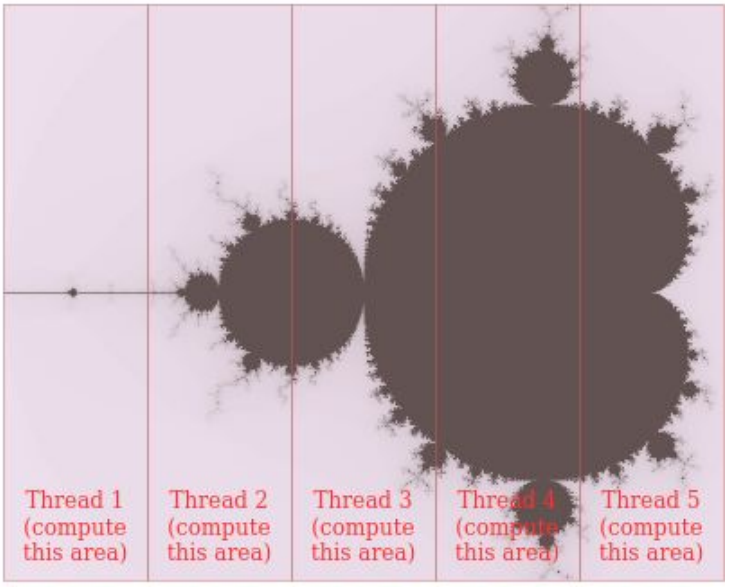
The geometric distribution implementation is way more basic. We already know our range (width = 750px in Graphical_geo_distrib.go). So what I did is to assign to each thread (number of thread can be set with the input parameters -> var nbrOfDistribution) a predefined range to compute. Each threads will so have the same number of computation.

In order to know where which thread need to start, and to end, I calculate for each one the starting position and ending position. In order to calculate these position you need to know the total number of computation (n), the number of threads you want (t), and the "thread number" (q: created thread number, from 1 to n):

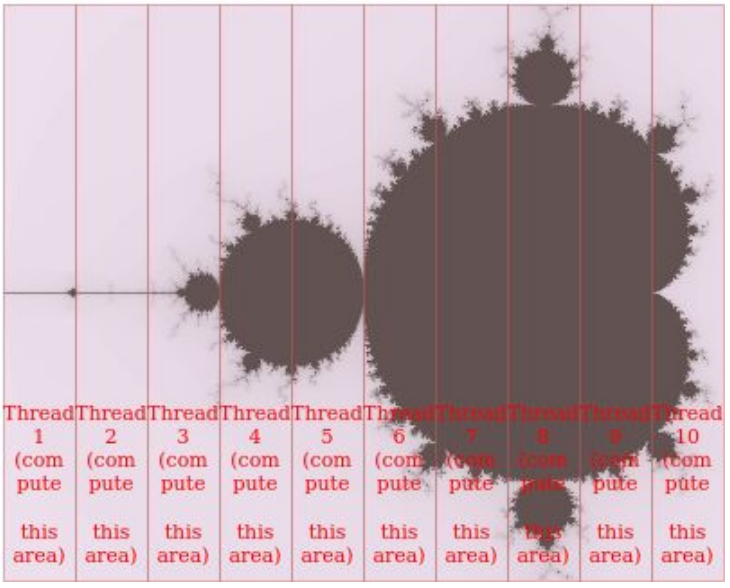
- start_position = n / t * q
- end_position = start position + n / t

Représentation of threads for the geometric distribution:

Example for 5 threads:



Example for 10 threads:



2. My results

In order to make good comparaisons I repeated each experiences 1000 times. (Why 1000 times? I tried with 20, 50, 100, 500, 1000, and it's almost constant after 100 iteration).

I compared the two implementations, I also compared them with different parameters:

- for the worker implementation I tried with 5, 10, 20, 50, 100 workers

```
---------  worker_with_5_workers ---------
Iteration nbr:                  1000
Average:                        0.302050673962
Standard deviation:             0.00879054567133
Standard error of the mean:     0.000879054567133

---------  worker_with_10_workers ---------
Iteration nbr:                  1000
Average:                        0.345632424355
Standard deviation:             0.0169195602689
Standard error of the mean:     0.00169195602689

---------  worker_with_20_workers ---------
Iteration nbr:                  1000
Average:                        0.352062489986
Standard deviation:             0.0143906055487
Standard error of the mean:     0.00143906055487

---------  worker_with_50_workers ---------
Iteration nbr:                  1000
Average:                        0.358691396713
Standard deviation:             0.00835621499202
Standard error of the mean:     0.000835621499202

---------  worker_with_100_workers ---------
Iteration nbr:                  1000
Average:                        0.399245557785
Standard deviation:             0.00769529516367
Standard error of the mean:     0.000769529516367
```

- for the geometric distribution I tried with 5, 10, 20, 50, 100 “intervals” (see 1.b. My geometric distribution implementation)

```
---------  geo_distrib_divide_in_5 ---------
Iteration nbr:                  1000
Average:                        0.118698720932
Standard deviation:             0.00555105448831
Standard error of the mean:     0.000555105448831

---------  geo_distrib_divide_in_10 ---------
Iteration nbr:                  1000
Average:                        0.119523575306
Standard deviation:             0.013191189316
Standard error of the mean:     0.0013191189316

---------  geo_distrib_divide_in_20 ---------
Iteration nbr:                  1000
Average:                        0.118268749714
Standard deviation:             0.0106666869469
Standard error of the mean:     0.00106666869469

---------  geo_distrib_divide_in_50 ---------
Iteration nbr:                  1000
Average:                        0.116515176296
Standard deviation:             0.0100529527967
Standard error of the mean:     0.00100529527967

---------  geo_distrib_divide_in_100 ---------
Iteration nbr:                  1000
Average:                        0.110143210888
Standard deviation:             0.00913484994708
Standard error of the mean:     0.000913484994708
```

- finally I compared all the results:

```
Fastest method in order according to the average:
geo_distrib_divide_in_100   ->  0.110143210888
geo_distrib_divide_in_50    ->  0.116515176296 (5.78516402142 % more than the previous one).
geo_distrib_divide_in_20    ->  0.118268749714 (1.50501717751 % more than the previous one).
geo_distrib_divide_in_5 ->  0.118698720932 (0.363554378607 % more than the previous one).
geo_distrib_divide_in_10    ->  0.119523575306 (0.694914290108 % more than the previous one).
worker_with_5_workers   ->  0.302050673962 (152.712214464 % more than the previous one).
worker_with_10_workers  ->  0.345632424355 (14.4286221319 % more than the previous one).
worker_with_20_workers  ->  0.352062489986 (1.86037685668 % more than the previous one).
worker_with_50_workers  ->  0.358691396713 (1.88287787407 % more than the previous one).
worker_with_100_workers ->  0.399245557785 (11.306142674 % more than the previous one).

Less deviation from the average (standard deviation):
geo_distrib_divide_in_5 ->  0.00555105448831
worker_with_100_workers ->  0.00769529516367 (38.6276279557 % more than the previous one).
worker_with_50_workers  ->  0.00835621499202 (8.58862219443 % more than the previous one).
worker_with_5_workers   ->  0.00879054567133 (5.19769632202 % more than the previous one).
geo_distrib_divide_in_100   ->  0.00913484994708 (3.91675657723 % more than the previous one).
geo_distrib_divide_in_50    ->  0.0100529527967 (10.0505520608 % more than the previous one).
geo_distrib_divide_in_20    ->  0.0106666869469 (6.10501374743 % more than the previous one).
geo_distrib_divide_in_10    ->  0.013191189316 (23.6671647123 % more than the previous one).
worker_with_20_workers  ->  0.0143906055487 (9.09255567423 % more than the previous one).
worker_with_10_workers  ->  0.0169195602689 (17.5736504742 % more than the previous one).

Most homogeneous method in order (standard error to the mean):
geo_distrib_divide_in_5 ->  0.000555105448831
worker_with_100_workers ->  0.000769529516367 (38.6276279557 % more than the previous one).
worker_with_50_workers  ->  0.000835621499202 (8.58862219443 % more than the previous one).
worker_with_5_workers   ->  0.000879054567133 (5.19769632202 % more than the previous one).
geo_distrib_divide_in_100   ->  0.000913484994708 (3.91675657723 % more than the previous one).
geo_distrib_divide_in_50    ->  0.00100529527967 (10.0505520608 % more than the previous one).
geo_distrib_divide_in_20    ->  0.00106666869469 (6.10501374743 % more than the previous one).
geo_distrib_divide_in_10    ->  0.0013191189316 (23.6671647123 % more than the previous one).
worker_with_20_workers  ->  0.0143906055487 (9.09255567423 % more than the previous one).
worker_with_10_workers  ->  0.0169195602689 (17.5736504742 % more than the previous one).
```

3. Result interpretation

a. Comparison of the results

We can easily see in the results that the fastest implementation is the geometric distribution. We can see a huge gap between the worker and the geometric: workers takes approximately 3 times more time.

We can also see that the number of workers don't reduce the computing time, but increase it. So we can conclude that the numbers of workers reduce the system time, and that's because of the "assignation" (made in the farmerWorkInit function in "Graphical_worker.go). Through the iteration on the width, is sent the calculations to be done through the channel and links these computations to workers thread. But if no worker is available, time is wasted. So the computation will be at every step limited to the number of workers, to the "pool size".

Of course the thread "pool size" can be optimized, and it will have impact on the execution. But as we saw in the results, it's needed to keep in mind that creating too many threads wastes resources and costs time creating the unused threads.

So for this type of problem (with known iteration time), geometric distribution is more efficient. Thread pool will find a lot of applications on web server for example.

b. The most stable and homogeneous implementation

We can see that all the results are quite homogeneous. The standard deviation is very low, we can conclude that every results (depending on the experiences of course), are close to each other.