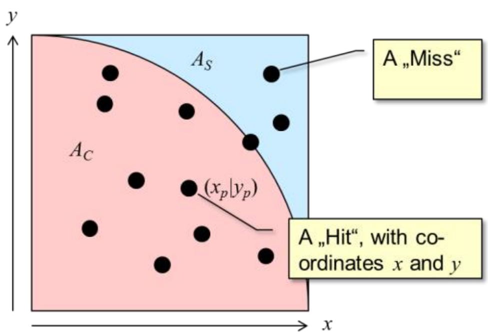
"System Design" – Parallel Programming

Threads | Collaboration Models [05]

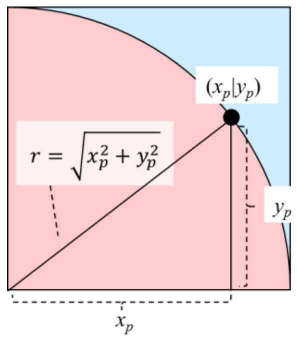
1 Master/Worker

π can be calculated with the help of the Monte-Carlo-Method. The basic idea is like this:

- We have a square with a quarter circle and we do random shots on the square.
- The area of the square is A_S, the area of the circle is A_C
- Each shot is either in the circle or outside
- We count the total number of shots (n) and the number of hits ($hits$).
- According to the "Law of Large Numbers" $\frac{hits}{n} = \frac{A_C}{A_S} = \frac{r^2\pi}{4r^2}$, with r being the radius of the circle and the width/height of the square
- This evaluates to $\frac{4 \cdot hits}{n} = \pi$



- We assume r is 1 and the random shots are within the domain $[0..1; 0..1]$
- How can we decide, if a shot is a "Hit" or a "Miss"?
- Answer: With the help of the Pythagorean theorem:
- If $x_p^2 + y_p^2 \le r^2$ it is inside, otherwise it is outside.



To calculate this, you would do the following (pseudo-code)-calculation:

```
int hits = 0;
for(int i = 0; i < n; i++){
      double x = random();
      double y = random();
      if(x*x + y*y <= 1)
            hits++;
}
double pi = 4.0 * hits / n;
```

1.1 Sequential Implementation

First do a sequential implementation. For generating the random numbers use the class `java.util.concurrent.ThreadLocalRandom` (not `java.util.Random`) with the following call `ThreadLocalRandom.current().nextDouble()`.

Print your calculated pi and compare it to `Math.PI`. What is a reasonable number of shots to get a good approximation of pi?

Measure the computation time with the help of `System.currentTimeMillis()`.

1.2 Low-Level-Parallelization

Now do a parallel implementation.

- Divide the calculation into segments
- Assign the segments as `Runables` to threads
- Let the threads contribute to the overall result – be aware of guarding common data with `synchronized` access-methods.
- Wait for the result with `join()`

Again, measure the computation time. Which number of threads gives the fastest result?

1.3 High-Level-Paralelization

Now do an implementation with `ExecutorService`

- Use the same segmentation as above
- Assign the segments as `Callables` to the service
- Retrieve the `Future`-Results
- And don't forget to `shutdown`.

Again, measure the computation time. Compare your results with the previous measurements.

2 Consumer/Producer

Implement the Consumer-Producer-Scenario given in the lecture slides. Refactor it in the following way:

- Extract an abstract class `Actor` which serves as a superclass to the producer and consumer.
- Introduce a class `Product`, products are created by the producers holds the message as an internal data
- Use an Executor for starting the threads

3 Pipes/Filters

In Moodle you find a simple framework for Pipes/Filters with a concrete sample for String. It has a Source-Pipe, which reads from a file and a Drain-Pipe which writes to the console, as well as one simple (neutral) filter which just copies the value.

A pipeline is built the following way:

```
Pipeline pipeline = Pipeline.source(aSourcePipe).map(aFilter)
    .map(anotherFilter).drain(aDrainPipe);
```

Add two additional filters which

- convert the string to an uppercase
- remove all the vowels from the string