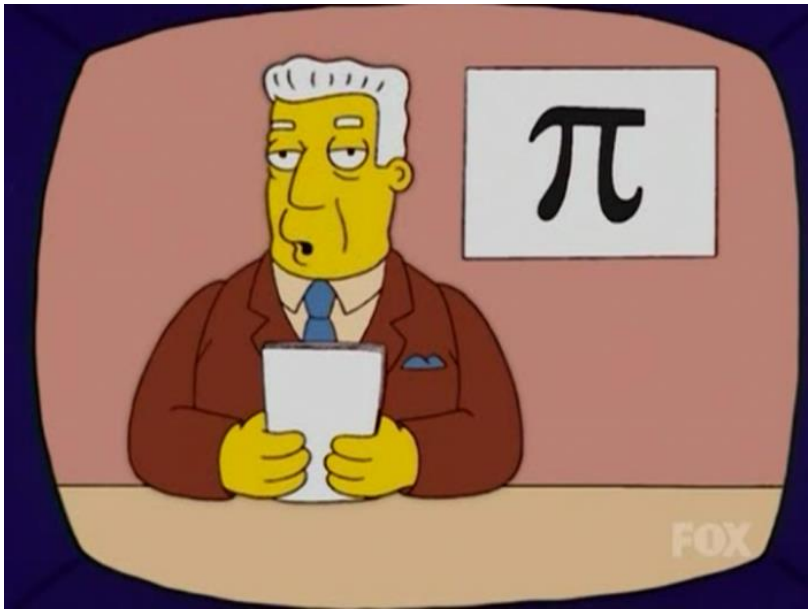
π computation with Montecarlo

Paolo Burgio
paolo.burgio@unimore.it

PI

- No need to explain…
  - 3.14159265358979323846264338327950288419716939937510582097...
- Never end
  - No algorithm can compute its value in finite time



Save the date!!

Monte-Carlo methods

- Random-based experiments

Used in

- Solving deterministic problems (e.g., π computation)
- Studying random systems

Monte-Carlo PI computation

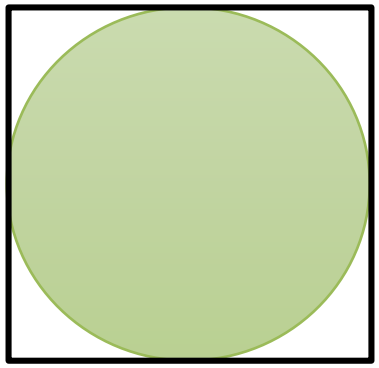
Rationale behind

› Correlation between π and the area of circle

$$A_{circle} = \pi * {R_{circle}}^2$$

› Mmmm…

Of squares and circles



- Consider a square enclosing the circle. Its area is…

$$Side_{square} = R_{circle} * 2$$

- So, its area is…

$$A_{square} = Side_{square}{}^{2} = (Rcircle * 2)^{2}$$

- Given that..

$$A_{circle} = \pi * R_{circle}{}^{2}$$

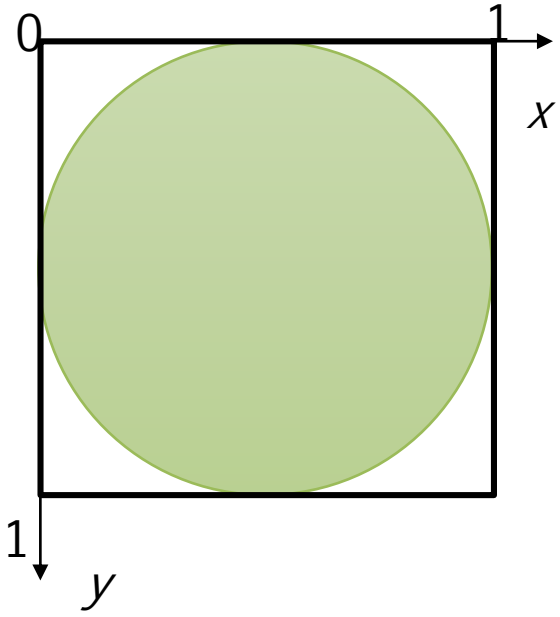
Hmmmmm…

› Hmmmmm….

$$\frac{A_{circle}}{A_{square}} = \frac{\pi * {R_{circle}}^2}{{R_{circle}}^2 * 4} = \frac{\pi}{4}$$

› …so….

$$\pi = 4 * \frac{A_{circle}}{A_{square}}$$

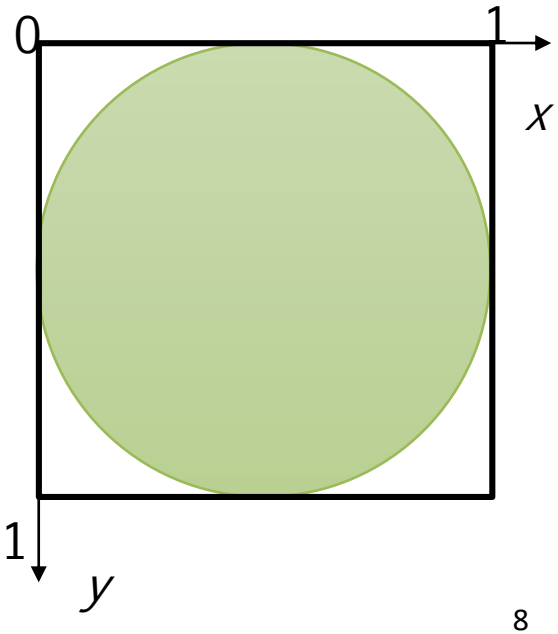


Now, the problem is..

- How to find the area of a circle
  - ..without knowing how much is π??

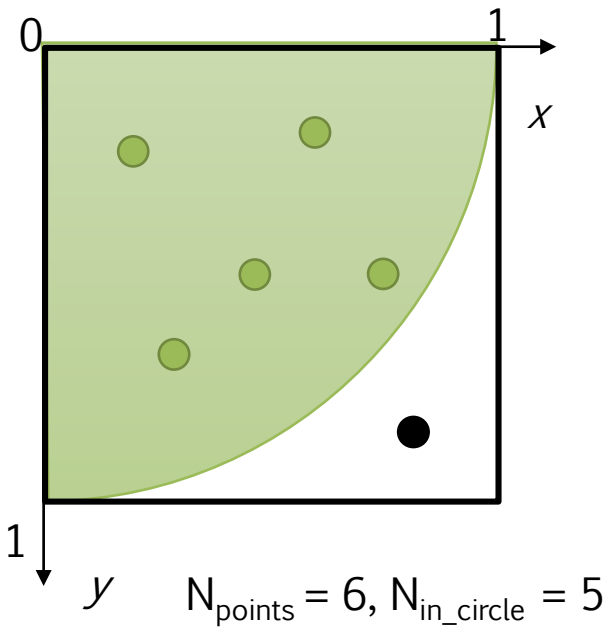
Go random!

1. Throw randomly (many) point inside the square
   - Point = (x, y) with x, y { 0, Rcircle }
2. Count how many are within circle
3. Count how many are within the square
4. Compute the ratio $\frac{A_{circle}}{A_{square}}$
5. Multiply by 4



Optimizing the algorithm

- Consider only ¼ of the circle
  - R_{circle} from 0 to 1
  - $Side_{square}$ from 0 to 1
- Here, 5/6 are inside the circle
  - $N_{points} = 6 \Rightarrow \pi = 4 * 5 / 6 = 3.33$

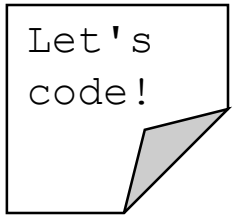


How to compute it?

1. All points are always inside the square
2. A given point is inside the circle iff ….?
3. The more N_{points}, the more precision!

- It's sooo parallel!
  - Each thread generates a point and performs 2.

Exercise



- Compute PI using Montecarlo Method
  - 10k iterations
  - Parametrizable

Random number generation

- Generate random (`float`) number between 0 and 1
  - `Code/utils.c`
  - Credits: Francesco Bellei

```c
#include <stdlib.h>
float randNumGen()
{
  //Generate a random number
  int random_value = rand_r( /* Add thread-unique seed here */ );

  //make it between 0 and 1
  float unit_random = random_value / (float) RAND_MAX;

  return unit_random;
}
```

`rand` vs. `rand_r`: thread safeness

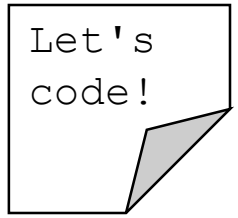
The `rand()` function returns a pseudo-random integer in the range 0 to RAND_MAX inclusive

...

The function `rand()` is not reentrant or thread-safe, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. In order to get reproducible behavior in a threaded application, this state must be made explicit; this can be done using the reentrant function `rand_r()`.

- (Reentrant function: can be interrupted/resumed in the middle of its execution)
- Thread safe function: can be accessed by multiple threads at the same time

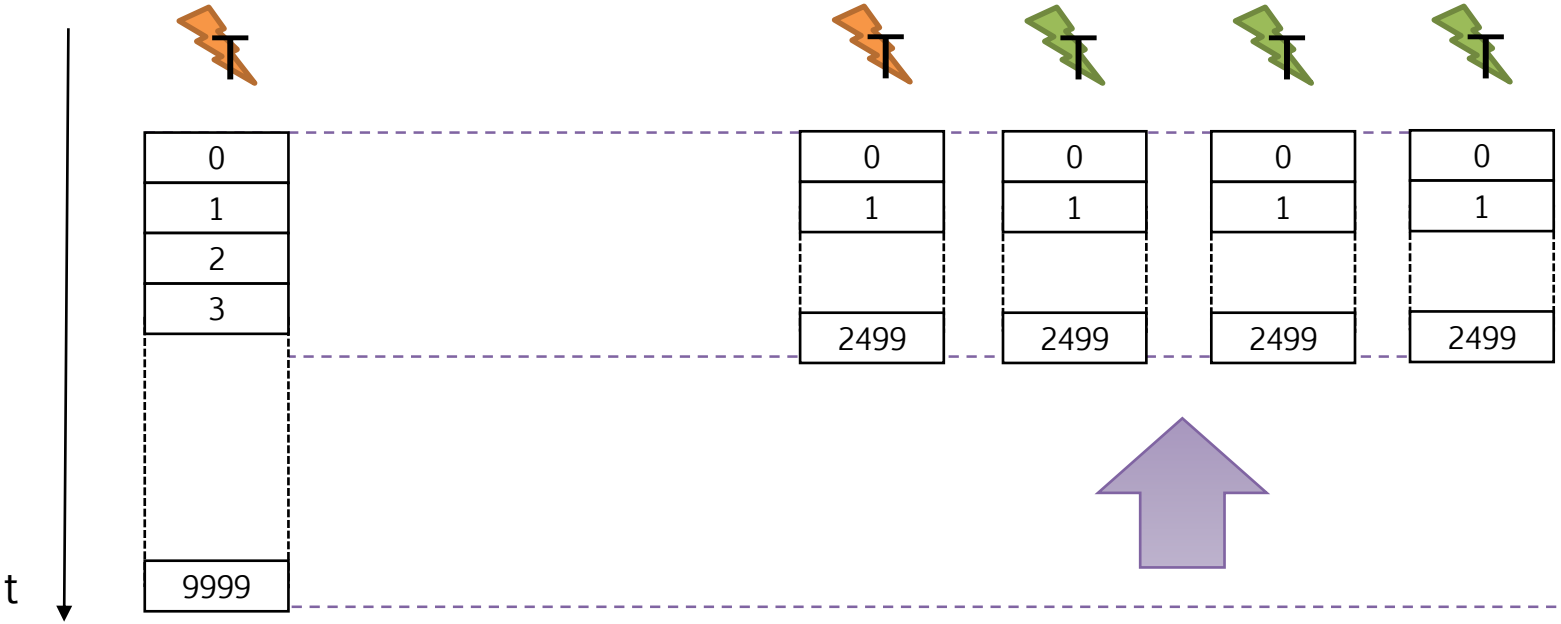
Exercise



- Now, parallelize Montecarlo over N threads
- Need only 10k/N iterations!
  - In my laptop, w/Cygwin, N = 4
- Potentially, N times faster!
  - …it won't be…
- How to know the number of (virtual) cores in Unix systems
  - `$ cat /proc/cpuinfo`

Sequential vs parallel

- 10k iterations
  - 1 rectangle = 1 iteration
  - You save N time!



Timing measurements

- Enable timing analysis of sequential/parallel code
  - Code/utils.c

```
#include <time.h>
#include <sys/time.h>

#define SECONDS 1000000000

unsigned long long gettime(void)
{
  struct timespec t;
  int r;

  r = clock_gettime(CLOCK_MONOTONIC, &t);
  if (r < 0)
  {
    printf("Error to get time! (%i)\n", r);
    return -1L;
  }

  return (unsigned long long) t.tv_sec * SECONDS + t.tv_nsec;
 }
```

Some hints…

- Create `_printf` macro
  - Enables you adding/removing debug prints
  - "The problem of parallel debugging"
  - Printf "`THREAD_ID/NTHREADS`"
- First, test with Sequential!
  - Enable/disable OMP code dis/abling `-fopenmp` switch
- Rely on pre-processor macro to know whether `-fopenmp` was set

```
#define _printf(...) printf(__VA_ARGS__)
// #define _printf(...)

void foo(void)
{

  #pragma omp parallel
  { // Parallel code
    _printf("[Thread %d/%d]\n",
      omp_get_thread_num(),
      omp_get_num_threads());

  }
}
```

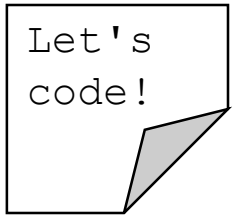
Standard-defined macro

OpenMP specifications

In implementations that support a preprocessor, the `_OPENMP` macro name is defined to have the decimal value yyyymm where yyyy and mm are the year and month designations of the version of the OpenMP API that the implementation supports.

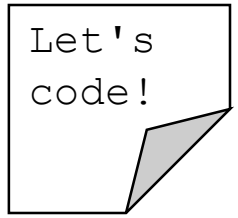
- But…
  - "In implementations that support a preprocessor"

Exercise



- Do this at home, varying N
  - N = 1, 2, 4, 8, 16
  - Run each experiment 3-5 times
  - Put avg values in an excel, and let's discuss

How to run the examples



- Download the `Code/` folder from the course website

- Compile
- `$ gcc -fopenmp code.c -o code`

- Run (Unix/Linux)

```
$ ./code
```

- Run (Win/Cygwin)

```
$ ./code.exe
```

References

- "Calcolo parallelo" website
  - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- Useful links
  - http://www.google.com
  - http://www.openmp.org
  - https://gcc.gnu.org/
- A "small blog"
  - http://www.google.com