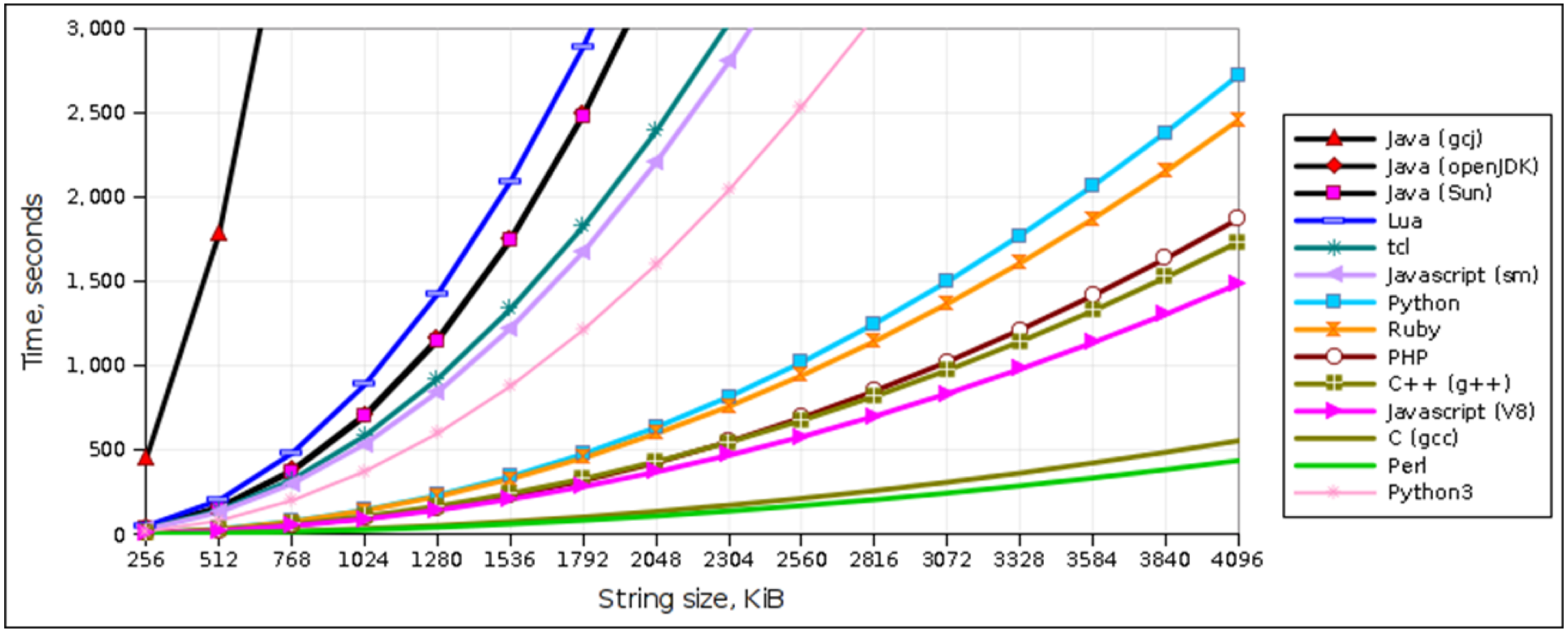
# NUMBA and Cython

Damian Podareanu

*SURFsara*

## At the speed of Python

One Python's major drawbacks is its speed. Even for optimized algorithms like string manipulation, Python falls way behind in the category of "faster" languages. www.raid6.com.au/~onlyjob/posts/arena/



## Cython and NUMBA

The most common two speedup options are:

- Cython, an optimizing static compiler as well as a compiled language which generates Python modules that can be used by regular Python code.
  - it is Python with C data types.
  - any piece of Python code is valid Cython code, which the Cython compiler will convert into C code.

- Numba, a Numpy-aware optimizing just-in-time/ahead-of-time compiler.
  - just-in-time compilation refers to the process of compiling during execution rather than before-hand. It uses the LLVM infrastructure to compile Python code into machine code.
  - ahead-of-time compilation produces a compiled extension module which does not depend on Numba: you can distribute the module on machines which do not have Numba installed (but Numpy is required).

## NUMBA basics

- Numba provides a Just-In-Time compiler for Python code.
- Just-in-time compilation refers to the process of compiling during execution rather than before-hand.
- It uses the LLVM infrastructure to compile Python code into machine code.
- Central to the use of Numba is the numba.jit decorator.

**Timing:**

```
3.98660914803 # N = 10.000.000
```

```
@numba.jit
def f(x):
    return x**2-x

def integrate_f(a, b,
N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit function
```

## NUMBA basics

- You can also specify the signature of the function.
- Otherwise Numba will generate separate compiled code for every possible type.

```
Timing:
0.0191540718079 # N =
10.000.000
```

```
import numba
from numba import float64, int32

@numba.jit
def f(x):
    return x**2-x

@numba.jit(float64(float64, float64,
int32))
def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit function
```

## Cython preparation

Cython code must, unlike Python, be compiled. This happens in two stages:

1. A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module.
2. The .c file is compiled by a C compiler to a .so file (or .pyd on Windows) which can be imported directly into a Python session.

The basic steps to compiling a Cython extension are as follows:

1. In helloworld.pyx: `print "Hello, World!`
2. Create setup.py, your python “makefile”.

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    ext_modules =
cythonize("helloworld.pyx") )
```

3. `$ python setup.py build_ext –inplace` → generates helloworld.so.
4. 
```
>>> import helloworld
Hello, World!
```

## Cython static typing

- Cython enforces static typing
- 
- Python is obviously dynamically-typed but for performance-critical code, this may be undesirable.
- Using static typing allows the Cython compiler to generate simpler, faster C code.
- The use of static typing, however, is not “pythonic” and results in less-readable code so you are encouraged to only use static typing when the performance improvements justify it.
- Cython supports all built-in C types as well as the special Cython types `bint`, used for C boolean values (int with 0/non-0 values for False/True), and `Py_ssize_t`, for (signed) sizes of Python containers.
- Also, the Python types list, dict, tuple, etc. may be used for static typing, as well as any user defined extension types.

## Cython static typing

The cdef statement is used to declare C variables, as well as C struct, union and enum types.

```
cdef int i, j, k
cdef float f, g[42], *h

cdef struct Node:
    int id
    float size

cdef union Data:
    char *str_data
    float *fl_data

cdef enum Color:
    red, blue, green
```

## Cython static typing

Consider the following purely Python code.

```
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

print(timeit.timeit("integrate_f(0.0,
5.0,10000000)", setup="from cydemo
import integrate_f", number=1))
```

Using Python’s timeit module, the call integrate_f(0, 5, 100000000) took about 4.198 seconds.

By just compiling with Cython, the call took about 2.137 seconds.

## Cython static typing

- Pure Python code took about 4.198 seconds.
- By just compiling with Cython, the call took about 2.137 seconds.
- By performing some static typing, the call
  took about 0.663 seconds.

A Cythonic version of this code might look like this:

```
def f(double x):
    return x**2-x

def integrate_f(double a, double b,
int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit code here
```

## Cython functions

- Within a Cython module, Python functions and C functions can call each other freely, but only Python functions can be called from outside the module by interpreted Python code.
  - any functions that you want to "export" from your Cython module must be declared as Python functions using def.
- Python functions take Python objects as parameters and return Python objects.
- There is also a hybrid function, called cpdef.
- A cpdef function can be called from anywhere, but uses the faster C calling conventions when being called from other Cython code.
- C functions are defined using the cdef statement. They take either Python objects or C values as parameters, and can return either Python objects or C values.

## Typing functions

When using Cython, Python function calls are extra expensive because one might need to convert to and from Python objects to do the call. We can obtain some more speedup just by typing our functions.

```
Time
0.0377948284149
```

```
cdef double f(double x):
    return x**2-x

def integrate_f(double a, double b,
int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit code here
```

## Some Results

- cydemo: pure Python implementation.
- cydemo2: pure Python compiled with Cython.
- cydemo3: static typing.
- cydemo4: static typing and function typing.
- numba: the numba implementation.

| module | N = 10.000.000 | N = 100.000.000 |
|---|---|---|
| cydemo | 4.198 | 41.69 |
| cydemo2 | 2.137 | 22.74 |
| cydemo3 | .663 | 5.90 |
| cydemo4 | .0377 | 0.382 |
| numba | 0.0191540718079 | |

## How is OpenMP typically used?

OpenMP is usually used to parallelize loops:
Find your most time consuming loops.
Split them up between threads.

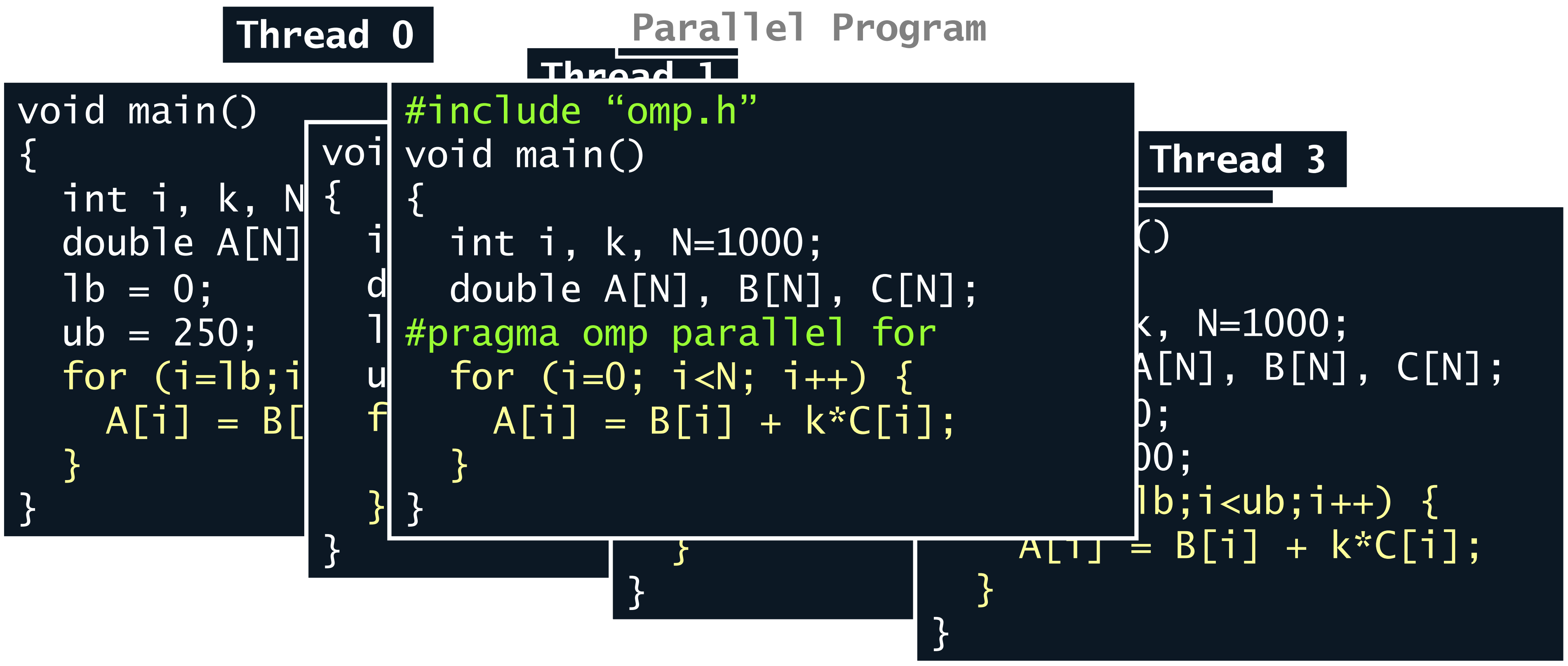
**Sequential Program**

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N],
C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
```

**Parallel Program**

```
#include “omp.h”
void main()
{
  int i, k, N=1000;
  double A[N], B[N],
C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] +
k*C[i];
  }
```

Single Program Multiple Data (SPMD)



```
void main()
{
  int i, k, N
  double A[N]
  lb = 0;
  ub = 250;
  for (i=lb;i
    A[i] = B[
  }
}
```

```
#include “omp.h”
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

```
()

k, N=1000;
A[N], B[N], C[N];
0;
00;
lb;i<ub;i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

## OpenMP Fork-and-Join model

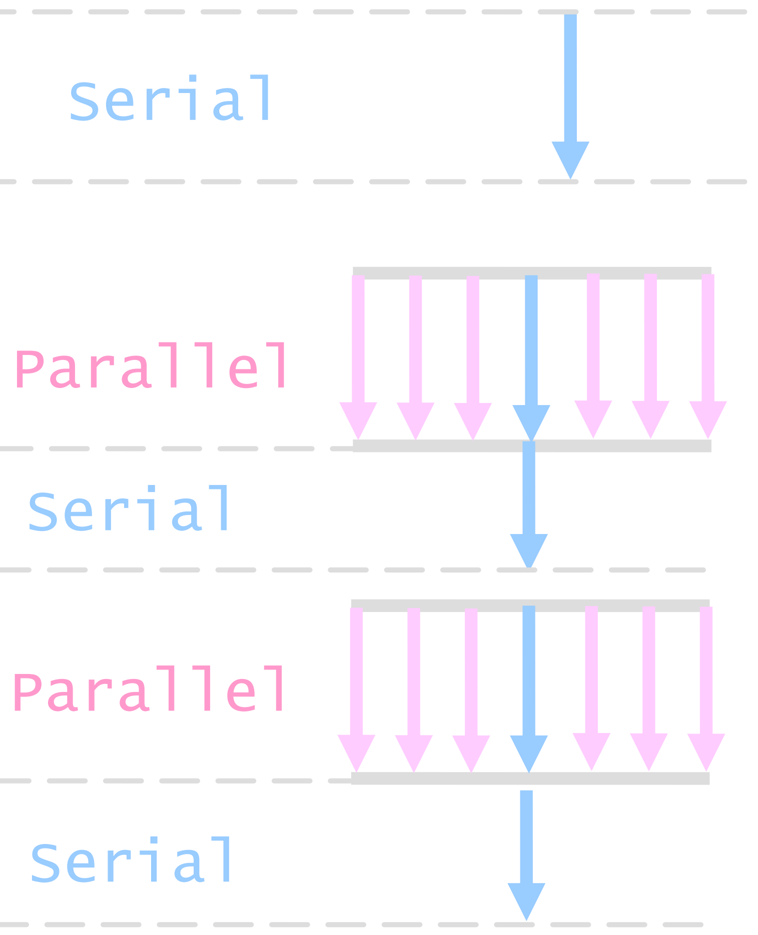
```
printf(“program begin\n”);
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf(“program done\n”);
```



OpenMP’s constructs:

1. Parallel Regions
2. Worksharing (for/DO, sections, …)
3. Data Environment  (shared, private, …)
4. Synchronization (barrier, flush, …)
5. Runtime functions/environment variables (omp_get_num_threads(), …)

**THANK YOU** FOR YOUR ATTENTION

**www.prace-ri.eu**