



A 1st introduction to Large Scale Computing Concepts & Techniques

Chapter 5-B: OpenMP/Numba



孔令波

mlinking@126.com

+86 15010255486

□ 前言

- 为什么需要“大规模计算” [HPC, DL, Business platform system, Cloud已经合流]
 - 导入 – 科学计算(天气预报), DL, 互联网平台(Google, Amazon, Alibaba, MeiTuan, ...)

□ 基础篇 – 概念和样子/例子

- 并发程序的样子 – Divide & Conquer, Model & Challenges, PCAM, Data/Task, ...
 - 天气预报的计算
- 运行环境
 - 硬件 – 自己梳理的3个方案 – Shared/Non-shared Memory, Hybrid
 - 系统软件 – 协议栈, Modern OS, Distributed Job Scheduler, GTM等
 - 大数据环境 – 开源运动的新贡献 – 分享是有威力的!

□ 算法级篇 – 并行编程(Parallel Programming)的原理和实现

- OpenMP, MPI, CUDA (DL的实现), Big Data 中的MR/Spark等 (只涉及在Big Data SDK之上的编程; 大数据本身的介绍放到后一部分)
 - Numba (OpenMP), MPI4PY, PyCUDA, PySpark, ...

□ 系统级篇 – 架构+集成 (Architecture + Integration Programming) 的威力

- “秒杀”的技术架构; 计算广告; 系统架构 (HTAP等)
 - Flink, ClickHouse, MaxCompute, ELK ...

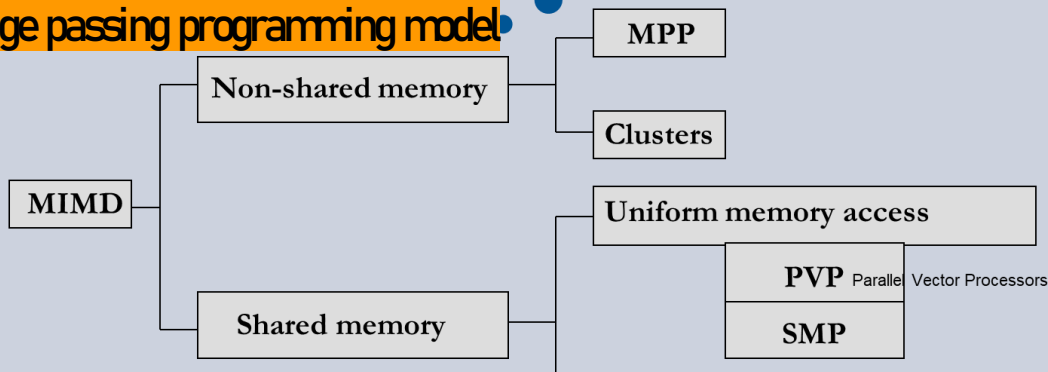
□ 并行编程模型 – 对计算系统的简单抽象，以便于指导并行程序的编写

- Shared Memory model
- Non-shared Memory model

Current focus is on MIMD
purpose processors or multicomputers

They lead to different
programming
(discussed later)

Message passing programming model



Shared Memory programming model

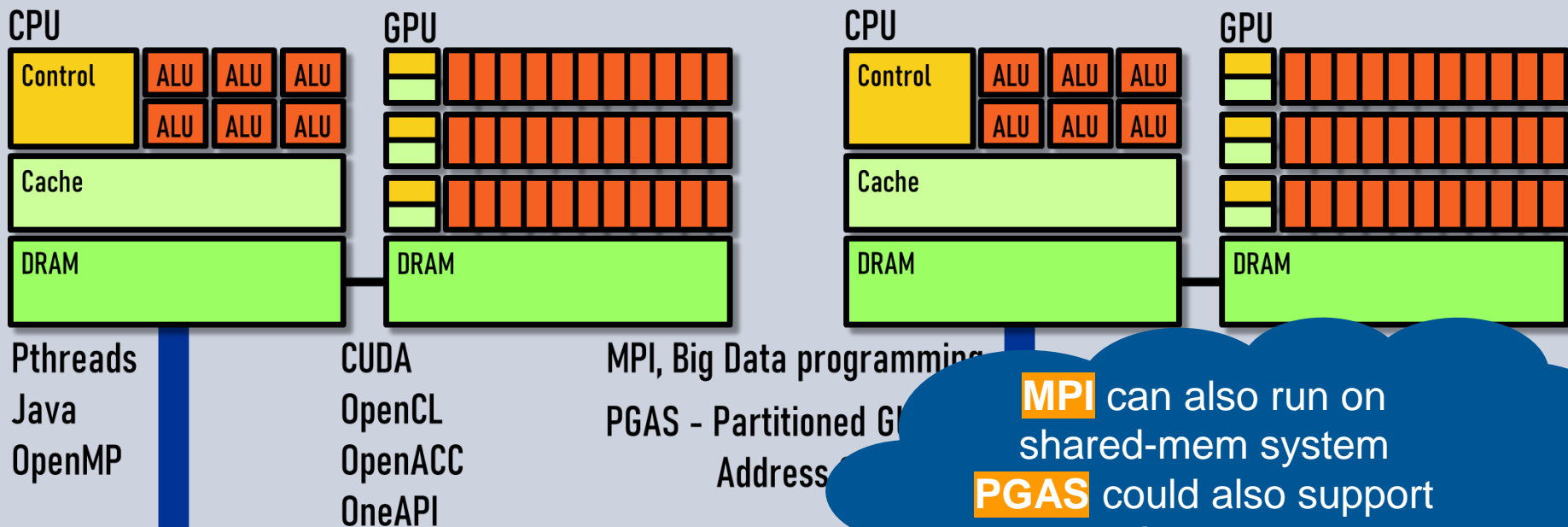
Non-Uniform memory access

CC-NUMA

NUMA

COMA





Chapter 5-B: OpenMP/Numba

□ Algorithmic Level (算法级) programming

- Shared Memory based programming – Multi-processed or -threaded
 - Multi-threaded: Pthreads, OpenMP, Java, CUDA, OpenCL, OpenACC, etc.
 - ✓ Python Numba
 - Multi-processed: Multi-processed, MPI
- Non-shared Memory based programming

□ System Level (系统级) programming

- Dedicated programs
- Big Data based Integration programming



Chapter 5-B: OpenMP/Numba

□ OpenMP with Python (Numba)

- OpenMP's history and overview
- Numba!

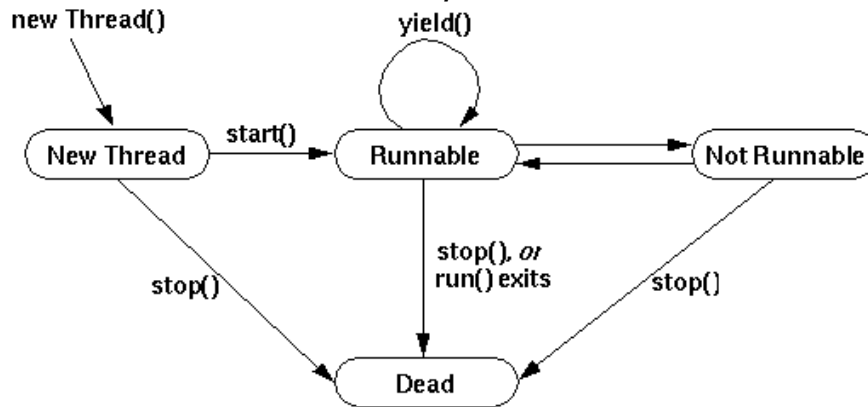
□ Resources

OpenMP? – ease the multi-threaded programming!

❑ OpenMP – Open Multi-Processing

❑ Do you remember the multi-threaded programming with Java, Pthreads, ...?

- You have to considerate carefully the synchronization, master-workers, Deadlocks and race conditions, ...



OpenMP

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

- 起源于ANSI X3H5（1994）标准
- 由设备商和编译器开发者共同制定，"工业标准"（1997）
- 编程简单，增量化并行，移植性好，可扩展性好
- 支持 Fortran, C/C++（编译器自带 OpenMP）
(<http://openmp.org/wp/openmp-compilers/>)
- 支持 Unix, Linux, Windows 等操作系统
- AMD, Intel, IBM, Cray, NEC, HP, NVIDIA,



基础知识

OpenMP



Association for
Computing Machinery



ACM中国
国际并行计算挑战赛

- OpenMP: Open Multi-processing
- 支持共享内存并行的应用开发接口 (API) 和规范
- 支持多种编程语言、指令级架构和操作系统
- 1997年, Fortran 1.0 版本发布
- 1998年, C/C++ 1.0 版本发布
- 2005年, C/C++、Fortran两个版本开始合并发布



北京大学 叶子凌锋



← → ↻ openmp.org

🐦 f in 📡 ✉ ⋮

OpenMP®

The OpenMP API specification for parallel programming

Home Specifications Community ▾ Resources ▾ News & Events ▾ About ▾ 🔍

IWOMP 2022

27-30 SEPTEMBER 2022
University of Tennessee at Chattanooga, USA
Co-located with EuroMPI

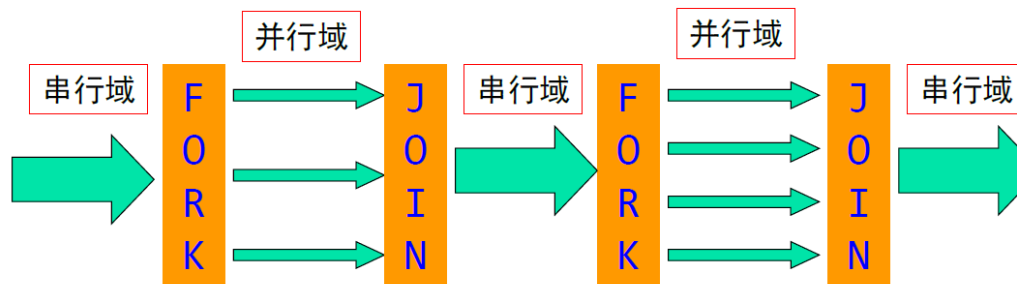
CALL FOR PAPERS

Image courtesy of Angela Foster, University of Tennessee at Chattanooga

OpenMP 采用 Fork-Join 并行执行方式

- OpenMP 程序开始于一个单独的主线程（Master Thread），然后主线程一直串行执行，直到遇见第一个**并行域**（Parallel Region），然后开始并行执行并行域。并行域代码执行完后再回到主线程，直到遇到下一个并行域，以此类推，直至程序运行结束。

Fork-Join



- Fork:** 主线程创建一个并行线程队列，然后，并行域中的代码在不同的线程上并行执行
- Join:** 当并行域执行完之后，它们或被同步，或被中断，最后只有主线程继续执行

† 并行域可以嵌套



OpenMP: Work sharing example

```
answer1 = long_computation_1();  
answer2 = long_computation_2();  
if (answer1 != answer2) { ... }
```

How to parallelize?

OpenMP: Work sharing example

```
answer1 = long_computation_1();  
answer2 = long_computation_2();  
if (answer1 != answer2) { ... }
```

How to parallelize?

```
#pragma omp sections  
{  
    #pragma omp section  
    answer1 = long_computation_1();  
    #pragma omp section  
    answer2 = long_computation_2();  
}  
if (answer1 != answer2) { ... }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual
parallelization

```
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    int nt = omp_get_num_threads();  
    int i_start = id*N/nt, i_end = (id+1)*N/nt;  
    for (int i=i_start; i<i_end; i++) { a[i]=b[i]+c[i]; }  
}
```


OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual
parallelization

```
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    int nt = omp_get_num_threads();  
    int i_start = id*N/nt, i_end = (id+1)*N/nt;  
    for (int i=i_start; i<i_end; i++) { a[i]=b[i]+c[i]; }  
}
```

- Launch nt threads
- Each thread uses id and nt variables to operate on a different segment of the arrays

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
```

```
{
```

```
    int id = omp_get_thread_num();
```

```
    int nt = omp_get_num_threads();
```

```
    int i = id * N / nt;
```

```
    for (; i < N; i++) {
```

```
        a[i] = b[i] + c[i];
```

```
    }
```

Increment:

var++, *var--*,

var += incr, *var -= incr*

Comparison:

var op last, where

op: *<*, *>*, *<=*, *>=*

Automatic parallelization of the for loop using

#parallel for

```
#pragma omp parallel
```

```
    #pragma omp for schedule(static)
```

```
    for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

```
}
```

Initialization:
var = init

One signed variable in the loop ("*i*")

Challenges of #parallel for

Load balancing

- If all iterations execute at the same speed, the processors are used optimally
- If some iterations are faster, some processors may get idle, reducing the speedup
- We don't always know distribution of work, may need to re-distribute dynamically

Granularity

- Thread creation and synchronization takes time
- *Assigning work to threads on per-iteration resolution may take more time than the execution itself*
- Need to coalesce the work to coarse chunks to overcome the threading overhead

Trade-off between **load balancing** and **granularity of parallelism**

Schedule: controlling work distribution

`schedule(static [, chunksize])`

- Default: chunks of approximately equivalent size, one to each thread
- If more chunks than threads: assigned in round-robin to the threads
- Why might want to use chunks of different size?

`schedule(dynamic [, chunksize])`

- Threads receive chunk assignments dynamically
- Default chunk size = 1

`schedule(guided [, chunksize])`

- Start with large chunks
- Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize

OpenMP: Data Environment

Shared Memory programming model

- Most variables (including locals) are shared by threads

```
{  
    int sum = 0;  
    #pragma omp parallel for  
    for (int i=0; i<N; i++) sum += i;  
}
```

- Global variables are shared

Some variables can be private

- Variables inside the statement block
- Variables in the called functions
- Variables can be explicitly declared as private

Overriding storage attributes

private:

- A copy of the variable is created for each thread
- There is no connection between original variable and private copies
- Can achieve same using variables inside { }

```
int i;  
#pragma omp parallel for private(i)  
for (i=0; i<n; i++) { ... }
```

firstprivate:

- Same, but the initial value of the variable is copied from the main copy

lastprivate:

- Same, but the last value of the variable is copied to the main copy

```
int idx=1;  
int x = 10;  
#pragma omp parallel for \  
firstprivate(x) lastprivate(idx)  
for (i=0; i<n; i++) {  
    if (data[i] == x)  
        idx = i;  
}
```

Reduction

```
for (j=0; j<N; j++) {  
    sum = sum + a[j]*b[j];  
}
```

How to parallelize this code?

- sum is not private, but accessing it atomically is too expensive
- Have a private copy of sum in each thread, then add them up

Use the reduction clause

#pragma omp parallel for reduction(+: sum)

- Any associative operator could be used: +, -, ||, |, *, etc
- The private value is initialized automatically (to 0, 1, ~0 ...)

#pragma omp reduction

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i = 0; i < N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```


Summary

OpenMP: A framework for code parallelization

- Available for C++ and FORTRAN
- Provides control on parallelism
- Implementations from a wide

Relatively easy to use

- Write (and debug!) code for
- Parallelization can be incremental
- Parallelization can be turned off at runtime or compile time
- Code is still correct for a serial machine

OpenMP with Python? –
23q0rwejgvw9485g3q
2rq2f3....

OpenMP* Overview

`C$OMP FLUSH` `#pragma omp critical`
`C$OMP THREADPRIVATE(/ABC/)` `CALL OMP_SET_NUM_THREADS(10)`
`C$OMP parallel do shared(a, b, c)` `call omp_test_lock(jlok)`
`call OMP_INIT_LOCK (ilok)` `C$OMP ATOMIC` `C$OMP MASTER`
`C$OMP SINGLE PRIVATE(X)` `setenv OMP_SCHEDULE "dynamic"`
`C$OMP PARALLEL DO ORDERED PRIVATE (A, B, C)` `C$OMP ORDERED`
`C$OMP PARALLEL REDUCTION (+: A, B)` `C$OMP`
`#pragma omp parallel for private(A, B)` `SECTIONS`
`C$OMP PARALLEL COPYIN(/blk/)` `!$OMP BARRIER`
`C$OMP DO lastprivate(XX)`
`Nthrds = OMP_GET_NUM_PROCS()` `omp_set_lock(lck)`

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP* Overview

OpenMP: An API for Writing Multithreaded Applications

A set of compiler directives and library routines for parallel application programmers

Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++

Standardizes established SMP practice + vectorization and heterogeneous device programming

C\$OMP FLUSH

#pragma omp critical

C\$OMP THREADPRIVATE(/ABC/)

CALL OMP SET NUM THREADS(10)

C\$ON

C\$ON

C\$O

C

#p

C\$OMP PARALLEL COPYIN(/blk/)

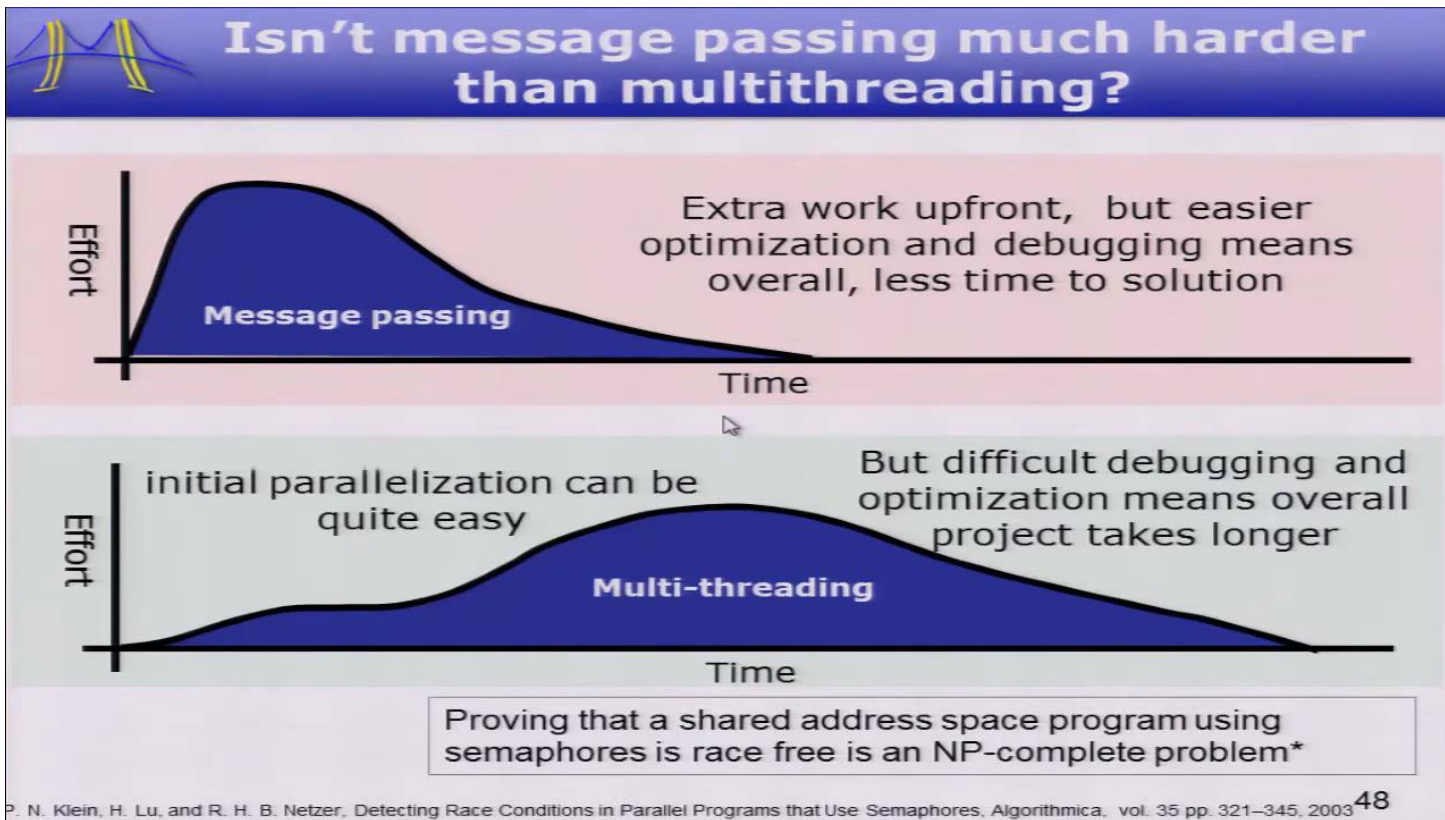
C\$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

MPI vs. Multi-threading



Chapter 5-B: OpenMP/Numba

□ OpenMP with Python (Numba)

- OpenMP's history and overview
- Numba!

□ Resources

但是 ... Python's GIL!

<https://stackoverflow.com/questions/11368486/openmp-and-python>

□ GIL – Global Interpreter Lock

- Due to GIL there is no point to use threads for CPU intensive tasks in CPython.
- You need either multiprocessing or use C extensions that release GIL during computations e.g., some of numpy functions.
- You could easily write C extensions that use multiple threads in Cython.

OpenMP and Python

Asked 10 years, 7 months ago Modified 1 year, 3 months ago Viewed 112k times



83



I have experience in coding OpenMP for Shared Memory machines (in both C and FORTRAN) to carry out simple tasks like matrix addition, multiplication etc. (Just to see how it competes with LAPACK). I know OpenMP enough to carry out simple tasks without the need to look at documentation.

Recently, I shifted to Python for my projects and I don't have any experience with Python beyond the absolute basics.

My question is :

What is the *easiest* way to use OpenMP in Python? By easiest, I mean the one that takes least effort on the programmer side (even if it comes at the expense of added system time)?

The reason I use OpenMP is because a serial code can be converted to a working parallel code with a few `!$OMP`s scattered around. The time required to achieve a *rough* parallelization is fascinatingly small. Is there any way to replicate this feature in Python?

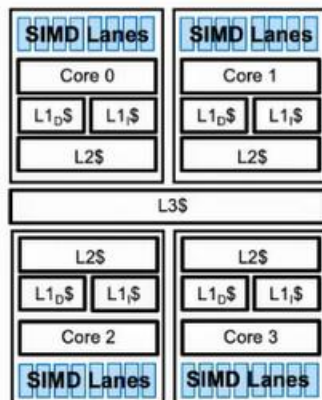
From browsing around on SO, I can find:

- C extensions
- StackLess Python

Are there more? Which aligns best with my question?



How do you get high performance for a modern CPU?



Three simple principles:

- Lots of threads ... at least one per hardware thread (often two hardware threads per core)
- Exploit SIMD lanes from each thread
- Maximize cache utilization

Why not embed parallelism inside Numpy? This works, but it suffers from two problems:

1. Overhead of creating/destroying threads at each operation ... increases parallel overhead and limits scalability (due to Amdahl's law)
2. Lost opportunity for parallelism from running multiple Numpy operations in parallel

... We want threads, but the **GIL** (Global Interpreter Lock) prevents multiple threads from making forward progress in parallel. The GIL is great for supporting thread safety and making it hard to write code that contains data races, but it prevents parallel multithreading in Python

What is the most common way in HPC to create multithreaded code? Something called OpenMP

anaconda.org/conda-forge/openmp

Conda Files Labels Badges

License: NCSA
Home: <http://openmp.llvm.org/>
684339 total downloads
Last upload: 2 years and 11 months ago

Installers

Info: This package contains files for the following operating systems:

conda install ?

- linux-64 v8.0.1
- win-32 v6.0.0
- osx-64 v8.0.1
- win-64 v8.0.1

To install this package with conda run one of the following:

```
conda install -c conda-forge openmp
```

```
conda install -c conda-forge/label/cf201901 openmp
```

```
conda install -c conda-forge/label/cf202003 openmp
```

Description

Q2vaknesr3qhrv
naweruv3qivre...!

conda install -c conda-forge openmp

Are you kidding to use Python for HPC?

The view of Python from an HPC perspective
(from the "Room at the top" paper).

```
for l in range(4096):  
  for j in range(4096):  
    for k in range (4096):  
      C[i][j] += A[i][k]*B[k][j]
```

A proxy for computing
over nested loops ...
yes, they know you
should use optimized
library code for
DGEMM

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6.727	18.4	4.33
6	plus vectorization	1.10	124.914	23.224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62.806	2.7	40.45

Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60*GB RAM





But ... now we have Numba!

- Modern technology should be able to map Python onto low-level code (such as C or LLVM) and avoid the “Python performance tax”.
- We’ve worked on ...
 - Numba (2012): JIT Python code into LLVM
 - Parallel accelerator (2017): Find and exploit parallel patterns in Python code.
 - Intel High-Performance Analytics Toolkit and Scalable Dataframe Compiler (2019): Parallel performance from data frames.
 - Intel numba-dppy (2020): Numba ParallelAccelerator regions that run on GPUs via SYCL.



Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

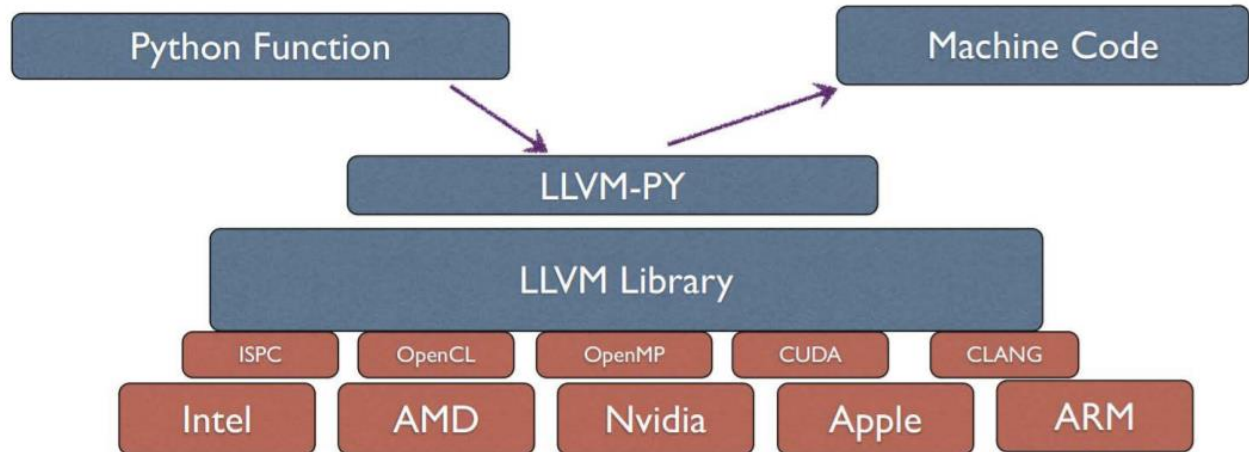
[Learn More](#)

[Try Numba »](#)

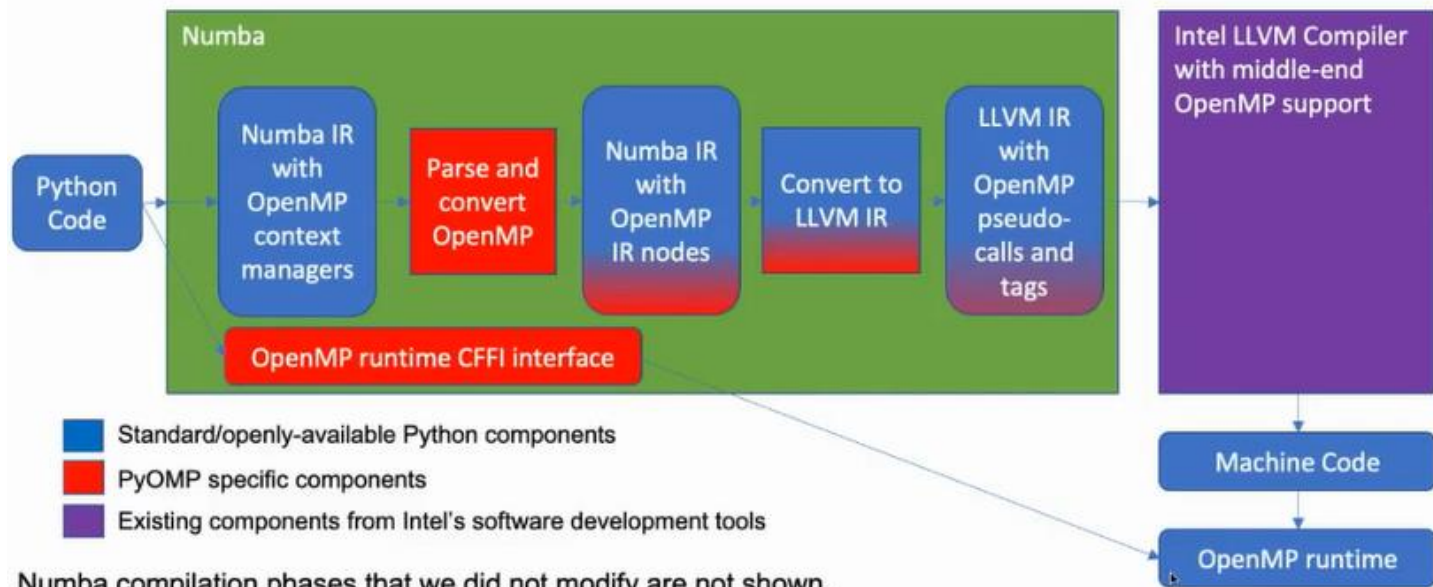
Numba

JIT Compiler for Python with LLVM

- Numba
 - open source compiling environment for Python and NumPy
 - generates optimized machine code by use the LLVM compiler
 - supports OpenMP, OpenCL and CUDA



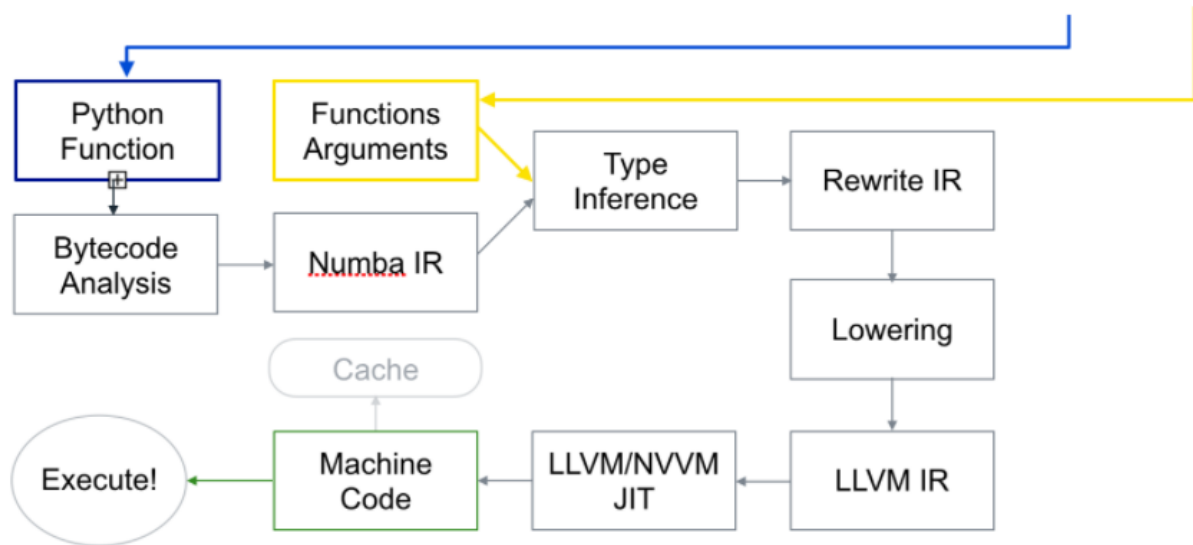
PyOMP Implementation in Numba: Overview



Numba compilation phases that we did not modify are not shown.

Numba的原理

```
@jit
def do_math(a, b):
    ...
>>> do_math(x, y)
```



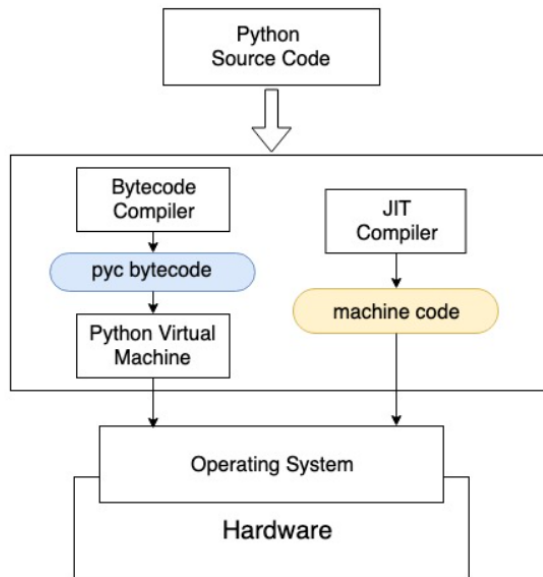
Numba编译过程

Numba使用了LLVM和NVVM技术。LLVM技术可以将Python、Julia这样的解释语言直接翻译成CPU可执行的机器码。NVVM主要是基于LLVM在GPU上的编译器技术。



Python解释器工作原理

Python是一门解释语言，Python为我们提供了基于硬件和操作系统的一个虚拟机，并使用解释器将源代码转化为虚拟机可执行的字节码。字节码在虚拟机上执行，得到结果。



Python解释器工作原理

十分钟入门Python Numba

Numba是一个针对Python的开源JIT编译器，由Anaconda GPU加速。Numba对NumPy数组和函数非常友好。

使用Numba非常方便，只需要在Python原生函数上堆用即时编译JIT方式编译成机器码，这些代码将以近乎

目前，Numba对以下环境进行了支持：

- 操作系统：Windows (32位和64位) , macOS, Linux
- CPU微架构：x86, x86_64, ppc64, armv7l和armv8
- GPU：NVIDIA CUDA和AMD ROCm
- CPython
- NumPy 1.15以后的版本

安装方法

使用 conda 安装Numba：

```
1 $ conda install numba
```

或者使用 pip 安装：

```
1 $ pip install numba
```



Numba的使用场景

Numba简单到只需要在函数上加一个装饰就能加速程序，但也有缺点。目前Numba只支持了Python原生函数和部分NumPy函数，其他一些场景可能不适用。比如Numba官方给出这样的例子：

此外，Numba不支持：

- `try...except` 异常处理
- `with` 语句
- 类定义 `class`
- `yield from`

注 Numba当前支持的功能：<http://numba.pydata.org/numba-doc/latest/reference/pysupported.html> 

pandas是更高层次的封装，Numba其实不能理解它里面做了什么，所以无法对其加速。一些大家经常用的机器学习框架，如 `scikit-learn`，`tensorflow`，`pytorch` 等，已经做了大量的优化，不适合再使用Numba做加速。



编译开销

□ 编译源代码需要一定的时间

- C/C++等编译型语言要提前把整个程序先编译好，再执行可执行文件
- Numba库提供的是一种懒编译（Lazy Compilation）技术，即在运行过程中第一次发现代码中有@jit，才将该代码块编译
- 用到的时候才编译，看起来比较懒，所以叫懒编译
- 使用Numba时，**总时间 = 编译时间 + 运行时间**

相比所能节省的计算时间，编译的时间开销很小

```
1 from numba import jit
2 import numpy as np
3 import time
4
5 SIZE = 2000
6 x = np.random.random((SIZE, SIZE))
7
8 """
9 给定n*n矩阵，对矩阵每个元素计算tanh值，然后求和。
10 因为要循环矩阵中的每个元素，计算复杂度为 n*n。
11 """
12 @jit
13 def jit_tan_sum(a): # 函数在被调用时编译成机器语言
14     tan_sum = 0
15     for i in range(SIZE): # Numba 支持循环
16         for j in range(SIZE):
17             tan_sum += np.tanh(a[i, j]) # Numba 支持绝大多数NumPy函数
18     return tan_sum
19
20 # 总时间 = 编译时间 + 运行时间
21 start = time.time()
22 jit_tan_sum(x)
23 end = time.time()
```

```
C:\ProgramData\Anaconda3\python.exe D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-03.py
Elapsed (with compilation) = 0.9473755359649658
Elapsed (after compilation) = 0.020324230194091797
Process finished with exit code 0
```



原生Python速度慢的另一个重要原因是变量类型不确定。声明一个变量的语法很简单，如 `a = 1`，但没有指定 `a` 到底是一个整数和一个浮点小数。Python解释器要进行大量的类型推断，会非常耗时。同样，引入Numba后，Numba也要推断输入输出的类型，才能转化为机器码。针对这个问题，Numba给出了名为Eager Compilation的优化方式。

```
1  from numba import jit, int32
2
3  @jit("int32(int32, int32)", nopython=True)
4  def f2(x, y):
5      return x + y
```

py

`@jit(int32(int32, int32))` 告知Numba你的函数在使用什么样的输入和输出，括号内是输入，括号左侧是输出。这样不会加快执行速度，但是会加快编译速度，可以更快将函数编译到机器码上。

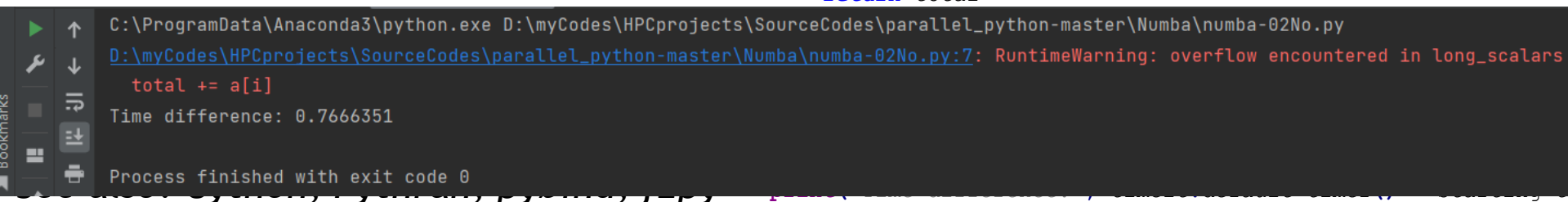


Numba: JIT Compiler for Python with LLVM

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- Wrap it in @numba.jit
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds

```
import numba
import numpy
import timeit

@numba.jit
def sum(x):
    total = 0
    for i in range(x.shape[0]):
        total += x[i]
    return total
```



```
C:\ProgramData\Anaconda3\python.exe D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-02No.py
D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-02No.py:7: RuntimeWarning: overflow encountered in long_scalars
    total += a[i]
Time difference: 0.7666351
Process finished with exit code 0
```

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- **Wrap it in @numba.jit**
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds

```
import numba
import numpy
import timeit

@numba.jit
def sum(x):
    total = 0
    for i in range(x.shape[0]):
        total += x[i]
    return total
```

```
(base) D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba>python numba-02.py
Time difference: 0.2639573

Structure
(base) D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba>
```

See also: Cython, Pythran, pybind, f2py

```
sum(x)
print("Time difference:", timeit.default_timer() - starting_time)
```



- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- Wrap it in `@numba.jit`
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds
- **Supports**
 - Normal numeric code
 - Dynamic data structures
 - Recursion
 - CPU Parallelism (thanks Intel!)
 - CUDA, AMD ROCm, ARM
 - ...

```
import numba
```

```
@numba.jit
def sum(x):
    total = 0
    for i in range(x.shape[0]):
        total += x[i]
    return total
```

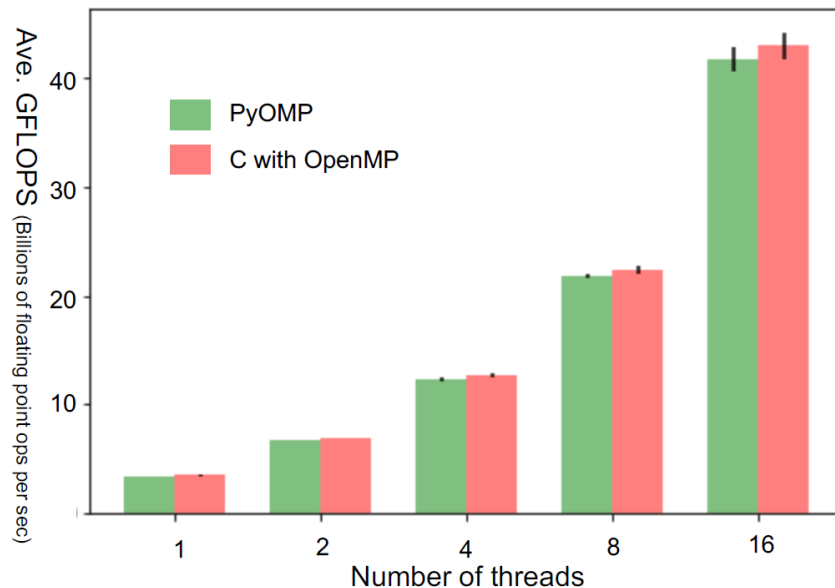
```
>>> x = numpy.arange(10_000_000)
>>> %time sum(x)
5.09 ms ± 110 µs
```

Performance Improvements

- Automatic SIMD vectorization of loops:
 - `@jit(nopython=True, fastmath=True)`
 - Much improved in LLVM 6, works with SSE, AVX, AVX, AVX-512 (Skylake, Xeon Phi)
 - Even better with SVML (see below)
- ParallelAccelerator (contributed by Intel):
 - `@jit(nopython=True, parallel=True)`
 - Automatic multithreading / loop fusion / optimization of NumPy array expressions.
 - `numba.prange()`: Parallel loops!
 - `@stencil`: like ufuncs, but for windowed functions
- SVML support (contributed by Intel):
 - SIMD vectorization of loops with common math function calls (sin, cos, exp, etc..)
- Combining all the above can be 10x over regular nopython mode in some cases:
 - <https://numba.pydata.org/numba-doc/dev/user/performance-tips.html>

DGEMM PyOMP vs C-OpenMP

Matrix Multiplication, double precision, order = 1000, with error bars (std dev)



250 runs for
order 1000
matrices

PyOMP times
DO NOT include
the one-time JIT
cost of ~2
seconds.

Intel® Xeon® E5-2699 v3 CPU, 18 cores, 2.30 GHz, threads mapped to a single CPU, one thread/per core, first 16 physical cores.
Intel® icc compiler ver 19.1.3.304 (icc -std=c11 -pthread -O3 xHOST -qopenmp)



```
from numba import jit
import numpy as np
import time
```

<https://numba.readthedocs.io/en/stable/user/5minguide.html#>

```
x = np.arange(100).reshape(10, 10)
```

```
@jit(nopython=True)
```

```
def go_fast(a): # Function is compiled and runs in machine code
```

```
    trace = 0.0
```

```
    for i in range(a.shape[0]):
```

```
C:\ProgramData\Anaconda3\python.exe D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-04.py
```

```
Elapsed (with compilation) = 0.4604744s
```

```
Elapsed (after compilation) = 4.400000000015503e-06s
```

```
Process finished with exit code 0
```

```
start = time.perf_counter()
```

```
go_fast(x)
```

```
end = time.perf_counter()
```

```
print("Elapsed (with compilation) = {}".format((end - start)))
```

```
# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
```

```
start = time.perf_counter()
```


```
go_fast(x)
```


```
end = time.perf_counter()
```

```
print("Elapsed (after compilation) = {}".format((end - start)))
```







 Numba




stable


FOR ALL USERS


 User Manual


 A ~5 minute guide to Numba


Overview


 Installation


 Compiling Python code with `@jit`


 Flexible specializations with `@generated_jit`


 Creating NumPy universal functions


 Compiling Python classes with `@jitclass`


 Creating C callbacks with `@cfunc`


 Compiling code ahead of time

 Automatic parallelization with `@jit`

 Using the `@stencil` decorator

 Callback into the Python Interpreter from within JIT'ed code

 Automatic module jitting with

 Read the Docs

v: stable ▾

User Manual

- A ~5 minute guide to Numba
 - How do I get it?
 - Will Numba work for my code?
 - What is `nopython` mode?
 - How to measure the performance of Numba?
 - How fast is it?
 - How does Numba work?
 - Other things of interest:
 - GPU targets:
- Overview
- Installation
 - Compatibility
 - Installing using conda on x86/x86_64/POWER Platforms
 - Installing using pip on x86/x86_64 Platforms
 - Installing on Linux ARMv7 Platforms
 - Installing on Linux ARMv8 (AArch64) Platforms
 - Installing from source
 - Build time environment variables and configuration of optional components
- Dependency List
- Version support information
- Checking your installation

Other things of interest:

Numba has quite a few decorators, we've seen `@jit`, but there's also:

- `@njit` - this is an alias for `@jit(nopython=True)` as it is so commonly used!
- `@vectorize` - produces NumPy `ufunc`s (with all the `ufunc` methods supported). [Docs are here.](#)
- `@guvectorize` - produces NumPy generalized `ufunc`s. [Docs are here.](#)
- `@stencil` - declare a function as a kernel for a stencil like operation. [Docs are here.](#)
- `@jitclass` - for jit aware classes. [Docs are here.](#)
- `@cfunc` - declare a function for use as a native call back (to be called from C/C++ etc). [Docs are here.](#)
- `@overload` - register your own implementation of a function for use in nopython mode, e.g. `@overload(scipy.special.j0)`. [Docs are here.](#)

Extra options available in some decorators:

- `parallel = True` - enable the automatic parallelization of the function.
- `fastmath = True` - enable fast-math behaviour for the function.

ctypes/cffi/cython interoperability:

- `cffi` - The calling of CFFI functions is supported in `nopython` mode.
- `ctypes` - The calling of ctypes wrapped functions is supported in `nopython` mode.
- Cython exported functions are callable.

GPU targets:

<https://numba.readthedocs.io/en/stable/user/5minguide.html#>

Numba can target [Nvidia CUDA](#) GPUs. You can write a kernel in pure Python and have Numba handle the computation and data movement (or do this explicitly). Click for Numba documentation on [CUDA](#).



The Threading Layers

This section is about the Numba threading layer, this is the library that is used internally to perform the parallel execution that occurs through the use of the `parallel` targets for CPUs, namely:

- The use of the `parallel=True` kwarg in `@jit` and `@njit`.
- The use of the `target='parallel'` kwarg in `@vectorize` and `@guvectorize`.

Note

If a code base does not use the `threading` or `multiprocessing` modules (or any other sort of parallelism) the defaults for the threading layer that ship with Numba will work well, no further action is required!

Which threading layers are available?

There are three threading layers available and they are named as follows:

- `tbb` - A threading layer backed by Intel TBB.
- `omp` - A threading layer backed by OpenMP.
- `workqueue` - A simple built-in work-sharing task scheduler.

Intel® Threading Building Blocks (Intel® TBB)

<https://numba.readthedocs.io/en/stable/user/threading-layer.html>

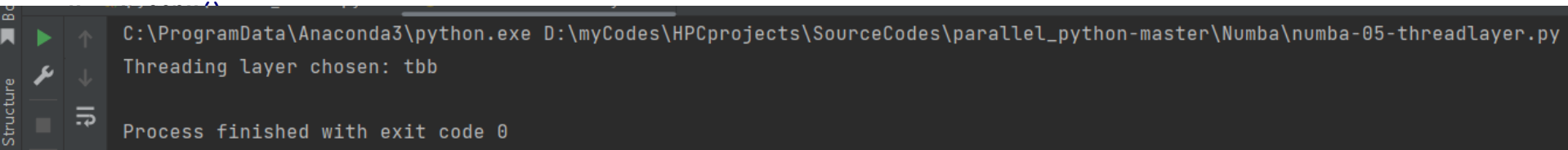


```
from numba import config, njit, threading_layer
import numpy as np

# set the threading layer before any parallel target compilation
config.THREADING_LAYER = 'threadsafe'

@njit(parallel=True)
def foo(a, b):
    return a + b

x = np.arange(10.)
```



```
C:\ProgramData\Anaconda3\python.exe D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-05-threadlayer.py
Threading layer chosen: tbb
Process finished with exit code 0
```

```
foo(x, y)

# demonstrate the threading layer chosen
print("Threading layer chosen: %s" % threading_layer())
```



Windows 11 – OMP

Setting the threading layer

The threading layer is a component of the Numba runtime that provides a way to select a specific threading layer for their use case, this is done by directly supplying the threading layer name to the `numba.config.THREADING_LAYER` setting through the `numba.config` module.

Selecting a named threading layer

Advanced users may wish to select a specific threading layer for their use case, this is done by directly supplying the threading layer name to the `numba.config.THREADING_LAYER` setting through the `numba.config` module. The options and requirements are as follows:

Threading Layer Name	Platform	Requirements
<code>tbb</code>	All	The <code>tbb</code> package (<code>\$ conda install tbb</code>)
<code>omp</code>	Linux Windows OSX	GNU OpenMP libraries (very likely this will already exist) MS OpenMP libraries (very likely this will already exist) Either the <code>intel-openmp</code> package or the <code>llvm-openmp</code> package (<code>conda install</code> the package as named).
<code>workqueue</code>	All	None

Note that it

compilation for a parallel target has occurred.

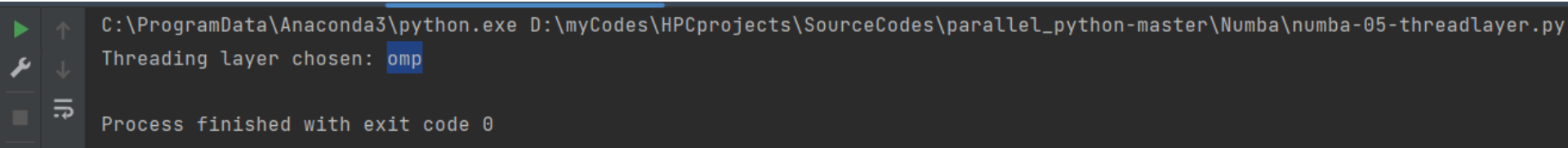
For example, to instruct Numba to choose `omp` first if available, then `tbb` and so on, set the environment variable as `NUMBA_THREADING_LAYER_PRIORITY="omp tbb workqueue"`. Or programmatically, `numba.config.THREADING_LAYER_PRIORITY = ["omp", "tbb", "workqueue"]`.



```
from numba import config, njit, threading_layer
import numpy as np

# set the threading layer before any parallel target compilation
# config.THREADING_LAYER = 'threadsafe'
config.THREADING_LAYER_PRIORITY = ["omp", "tbb", "workqueue"]

@njit(parallel=True)
def foo(a, b):
    return a + b
```



C:\ProgramData\Anaconda3\python.exe D:\myCodes\HPCprojects\SourceCodes\parallel_python-master\Numba\numba-05-threadlayer.py
Threading layer chosen: omp

Process finished with exit code 0

```
# and then execute in parallel
foo(x, y)

# demonstrate the threading layer chosen
print("Threading layer chosen: %s" % threading_layer())
```



A more complicated example

```
'''
https://python-programs.com
'''

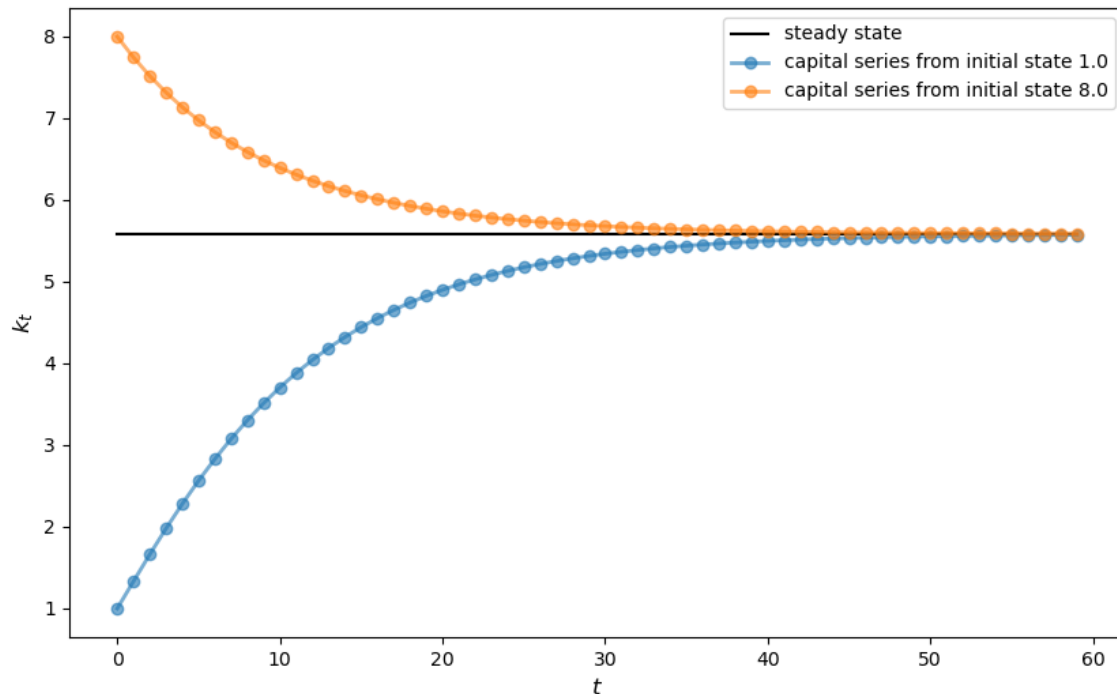
from numba.experimental import jitclass
from numba import float64

import matplotlib.pyplot as plt

solow_data = [
    ('n', float64),
    ('s', float64),
    ('delta', float64),
    ('alpha', float64),
    ('z', float64),
    ('k', float64)
]


@jitclass(solow_data)
class Solow:
    r"""
```

Implements the Solow growth model with the update rule



更复杂的例子

<https://gist.github.com/SkBlaz/04f19b909f3f570a7c73b91e2ff1a496>

 **SkBlaz / README.md**
Forked from [sklam/README.md](#)
Created 6 years ago • Report abuse

SubscribeStar 0Fork 2

<> CodeRevisions 1Forks 2Embed<script src="https://gi!>Download ZIP

Randomwalk PageRank in numba. Multithreaded CPU and single CUDA-GPU implementation.

<> README.mdRaw

README

Setup

Create conda environment with:

```
conda create -n pagerank35 python=3.5 numba pyculib cudatoolkit networkx
```

Activate environemnt with:

```
source activate pagerank35
```

Usage



<https://gist.github.com/SkBlaz/04f19b909f3f570a7c73b91e2ff1a496>

```
1 """
2 Implementation of CPU random walk pagerank.
3
4 The implementation is optimized with the numba JIT compiler
5 and uses multi-threads for parallel execution.
6 """
7
8 import multiprocessing
9 from concurrent.futures import ThreadPoolExecutor
10 import os
11
12 from numba import jit, float32, uint64, uint32
13 import numpy as np
14
15
16 CPU_COUNT = int(os.environ.get('CPU_COUNT', multiprocessing.cpu_count()))
17
18 MAX32 = uint32(0xffffffff)
19
20
21 @jit("(uint64[:,1], uint64)", nogil=True)
22 def xorshift(states, id):
23     x = states[id]
24     x ^= x >> 12
25     x ^= x << 25
26     x ^= x >> 27
27     states[id] = x
28     return uint64(x) * uint64(2685821657736338717)
29
30
31 @jit("float32(uint64[:,1], uint64)", nogil=True)
32 def xorshift_float(states, id):
33     return float32(float32(MAX32 & xorshift(states, id)) / float32(MAX32))
34
35
36 @jit("boolean(intp, uint32[:,1], uint32[:,1], uint32[:,1], uint32[:,1], "
37       "float32, uint64[:,1])", nogil=True)
```



Chapter 5-B: OpenMP/Numba

□ OpenMP with Python (Numba)

- OpenMP's history and overview
- Numba!

□ Resources



- > 第1章动机和历史
- > 第2章并行体系结构
- > 第3章并行算法设计
- > 第4章消息传递编程
- > 第5章Eratosthenes筛法
- > 第6章Floyd算法
- > 第7章性能分析
- > 第8章矩阵向量乘法
- > 第9章文档分类
- > 第10章蒙特卡洛法
- > 第11章矩阵乘法
- > 第12章线性方程组求解
- > 第13章有限差分方法
- > 第14章排序
- > 第15章快速傅立叶变换
- > 第16章组合搜索
- > 第17章共享存储编程
- > 第18章融合OpenMP和MPI
- > 附录A MPI函数

□ Parallel programming in C with MPI and Open MP

□ Michael J Quinn

□ 2003

