Explorando recursos multicore com Python Numba

Lucas Ferreira da Silva

Motivação

- Explorar recursos de programação paralela utilizando Python;
- Soluções que contornem o problema do paralelismo real em Python;
- Alternativas ao multiprocessing;
- Numba.

Recursos anunciados no Numba

- Paralelização automática, apenas informando um decorator;
- Evita o bloqueio do GIL;
- **@jit(nopython=True, parallel=True)**
- Fácil paralelização de laços explicitamente paralelos
- Permite reduções e operações aritméticas básicas
- **@njit(parallel=True)**
- **prange**

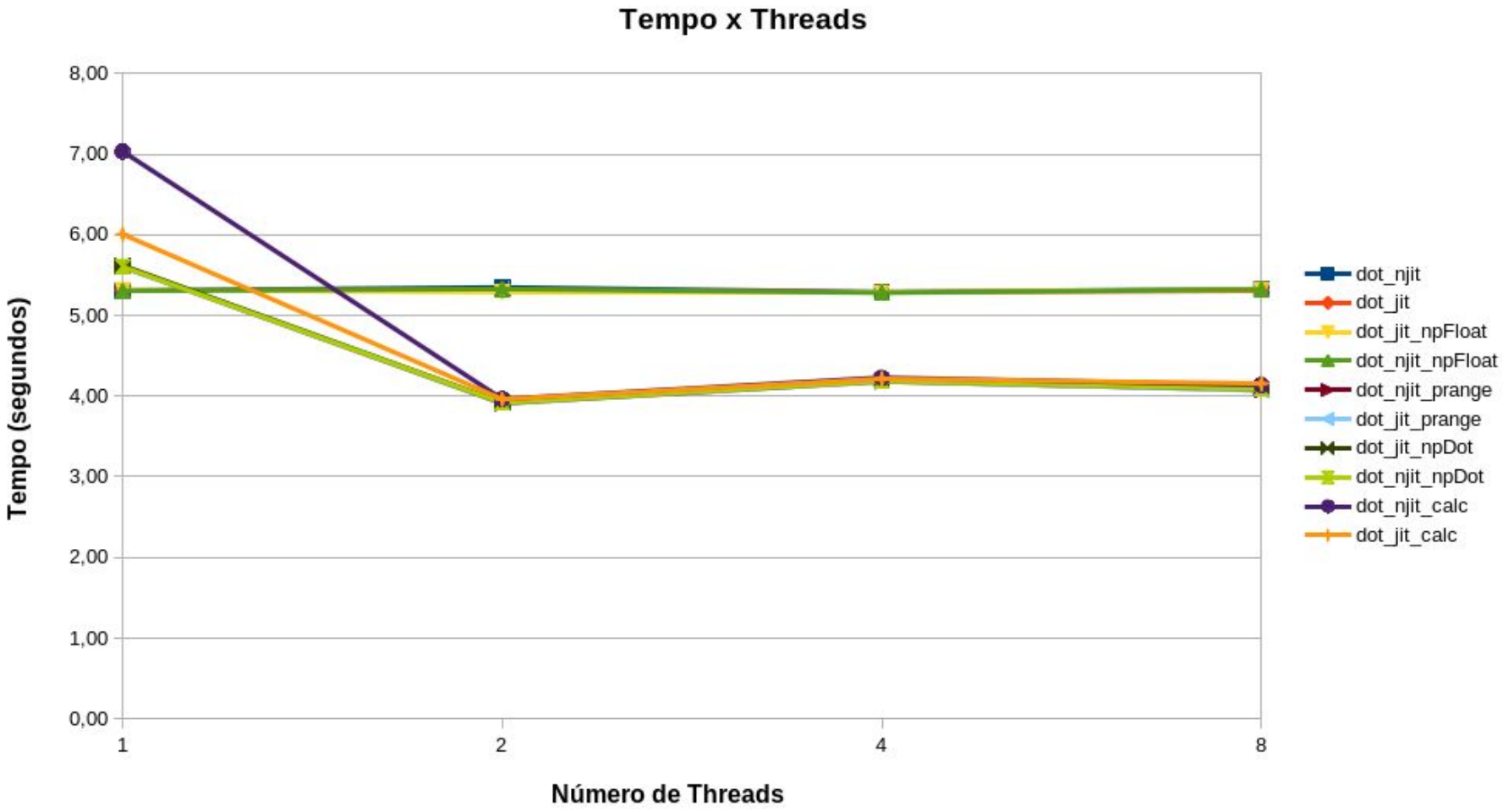
Proposta

- Explorar os recursos anunciados pelo Numba;
- Comparar os resultados com outras alternativas;
- Realizar testes de desempenho com algoritmos potencialmente paralelizáveis e com considerável carga de trabalho;
- Obter conclusões sobre o uso da ferramenta em HPC.

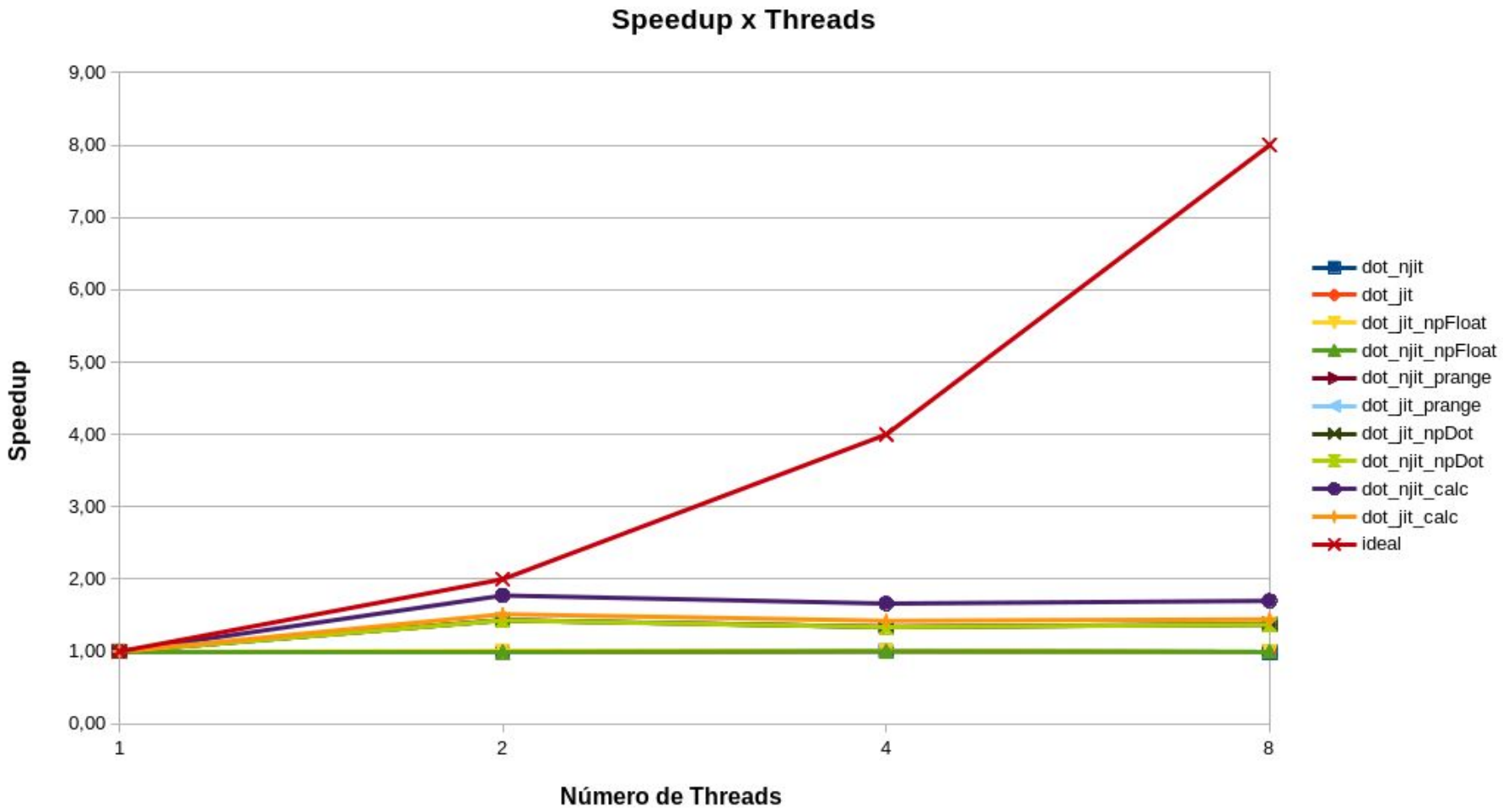
Benchmark

- 10 execuções de cada configuração;
- Variação entre **2**, **4** e **6** threads;
- Hardware:
  - Intel® Core™ i5-2410M
  - 2.30GHz
  - 2 Cores
  - 4 Threads
  - 6 GB de RAM
- Sistema Operacional:
  - Debian GNU/Linux Buster
  - Versão do Linux: 4.15.0-2-amd64
  - Versão do Python: 3.6

Tempos de execução



Speedup


Speedup x Threads
Speedup
9,00
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
1
2
4
8
Número de Threads
dot_njit
dot_jit
dot_jit_npFloat
dot_njit_npFloat
dot_njit_prange
dot_jit_prange
dot_jit_npDot
dot_njit_npDot
dot_njit_calc
dot_jit_calc
ideal

Implementações

- A partir dos resultados obtidos com a execução de uma primeira versão, notou-se anormalidades dos tempos e no paralelismo explorado pelo Numba;
- Pensando nas possíveis causas, o código foi adaptado de diversas maneiras objetivando-se um paralelismo efetivo, como o proposto na documentação;
- Escolha de um algoritmo simples e explicitamente paralelizável;
- Variações, na tentativa de se deixar mais próximo do anunciado na documentação do Numba.

Implementações

```
@njit(parallel=True)
def prange_test(A):
    s = 0
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

```
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iterations):
    for i in range(iterations):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
    return w
```

Implementações

```python
# Implementação utilizando njit (como a documentação sugere)
@numba.njit(parallel=True)
def dot_njit(a, b, n, repeat):
    for k in range(repeat):
        dot = 0.0
        for i in range(n):
            dot += a[i] * b[i]
    return dot

# Implementação utilizando jit (como a documentação sugere)
@numba.jit(nopython=True, parallel=True)
def dot_jit(a, b, n, repeat):
    for k in range(repeat):
        dot = 0.0
        for i in range(n):
            dot += a[i] * b[i]
    return dot
```

Implementações

```python
# Implementação utilizando variável float do numpy
@numba.njit(parallel=True)
def dot_njit_npFloat(a, b, n, repeat):
    for k in range(repeat):
        dot = np.float32(0.0)
        for i in range(n):
            dot += a[i] * b[i]
    return dot
```

```python
@numba.jit(nopython=True, parallel=True)
def dot_jit_npFloat(a, b, n, repeat):
    for k in range(repeat):
        dot = np.float32(0.0)
        for i in range(n):
            dot += a[i] * b[i]
    return dot
```

Implementações

```python
# Implementação utilizando numba.prange
@numba.njit(parallel=True)
def dot_njit_prange(a, b, n, repeat):
    for k in range(repeat):
        dot = 0.0
        for i in numba.prange(n):
            dot += a[i] * b[i]
    return dot


@numba.jit(nopython=True, parallel=True)
def dot_jit_prange(a, b, n, repeat):
    for k in range(repeat):
        dot = 0.0
        for i in numba.prange(n):
            dot += a[i] * b[i]
    return dot
```

Implementações

```python
# Implementação utilizando a operação dot do numpy
@numba.njit(parallel=True)
def dot_njit_npDot(a, b, n, repeat):
    dot = 0.0
    for k in range(repeat):
        dot = np.dot(a, b)
    return dot


@numba.jit(nopython=True, parallel=True)
def dot_jit_npDot(a, b, n, repeat):
    dot = 0.0
    for k in range(repeat):
        dot = np.dot(a, b)
    return dot
```

Implementações

```python
def dot_njit_calc(a, b, n, repeat):
    dot = 0.0
    for k in range(repeat):
        dot = calc_njit(a, b, n, dot)
    return dot
```

```python
@numba.jit(nopython=True, parallel=True)
def calc_jit(a, b, n, dot):
    for i in numba.prange(n):
        dot += a[i] * b[i]
    return dot
```

```python
def dot_jit_calc(a, b, n, repeat):
    dot = 0.0
    for k in range(repeat):
        dot = calc_jit(a, b, n, dot)
    return dot
```

```python
@numba.njit(parallel=True)
def calc_njit(a, b, n, dot):
    for i in numba.prange(n):
        dot += a[i] * b[i]
    return dot
```

Conclusões

- Resultados totalmente inesperados!
- O pouco de speedup que se obteve em algumas implementações se deve a otimizações de código e a nível de compilação;
- As implementações que utilizaram mais de um core do processador, utilizavam todos sempre, independente do número de threads atribuído para a variável de ambiente;

Conclusões

- Antes de ser adotado o produto vetorial como benchmark, outros algoritmos foram testados, mas descartados pelo não sucesso:
  - Determinação do PI pelo método de Monte Carlo;
  - Fatoração LU
  - Cálculo fatorial simples.

Principais referências

- https://numba.pydata.org/numba-doc/dev/user/parallel.html
- https://numba.pydata.org/numba-doc/dev/user/jit.html#parallel
- https://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html