

Comparações de Performance: ARM vs. x86 Em Algoritmos Simples de Ordenação

Tiago Maluta
maluta@users.sf.net

April 17, 2008

Abstract

Referindo-se a parte do conteúdo abordado na disciplina de Análise de Algoritmos, ministrada pelo professor André Benardi, na Universidade Federal de Itajubá. Este artigo pretende traçar comparações no desempenho de algoritmos de ordenação simples (*bubble sort* e *quick sort*) nas arquiteturas Intel x86 e ARM utilizando a linguagem Python.

1 Introdução

Existem diversos fatores determinantes na eficiência de um algoritmo. Estes vão desde as configurações e projeto do hardware até a escolha da linguagem e da lógica do programa. Esse artigo irá analisar os tempos de execução de um programa feito em linguagem interpretada (Python) de dois algoritmos para ordenar uma sequência numérica, executado em duas arquiteturas diferentes: computador baseado na arquitetura x86 e em um *smartphone* baseado na arquitetura ARM. Contudo, vamos começar apresentando os dois algoritmos utilizados.

1.1 *Bubble Sort* [1]

A idéia básica por trás da classificação por bolha é percorrer o arquivo sequencialmente

vária vezes. Cada passagem consiste em comparar cada elemento no arquivo com seu sucessor ($x[i]$ com $x[i+1]$) e trocar os dois elementos se eles não estiverem na ordem correta.

1.2 *Quick Sort* [2]

Seja x um vetor e n o número de elementos no vetor a ser classificados. Escolha um elemento a numa posição específica dentro do vetor (por exemplo, a pode ser escolhido como o primeiro elemento de modo que $a = x[0]$). Suponha que os elementos de x sejam particionados de modo que a seja colocado na posição j e as seguintes condições sejam observadas:

1. Cada elemento nas posições 0 até $j - 1$ seja menor ou igual a a
2. Cada elemento nas posições $j + 1$ até $n - 1$ seja maior ou igual a a

Observe que, se essas duas condições forem mantidas para determinado a e j , a será iésimo menor elemento de x , de forma que a permanecerá na posição j quando o vetor estiver totalmente classificado. Se o processo anterior for repetido com os subvetores $x[0]$ até $x[j - 1]$ e $x[j + 1]$ e $x[n - 1]$ e com quaisquer vetores criados pelo processo em sucessivas iterações, o resultado final será um arquivo classificado.

2 Algoritmos

2.1 *Bubble Sort*

```
def bubble(a):
    n = len(a)
    changed = 1
    while changed:
        changed = 0
        for i in xrange(n-1):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                changed = 1
        n -= 1
```

2.2 QuickSort

```
def quicksort(l):
    if l:
        left = [x for x in l if x < l[0]]
        right = [x for x in l if x > l[0]]
        if len(left) > 1:
            left = quicksort(left)
        if len(right) > 1:
            right = quicksort(right)
        return left + [l[0]]
    * l.count(l[0]) + right

    return []
```

3 Metodologia

Primeiramente, iremos escolher uma pequena faixa (25 números) de valores aleatórios, para mostrar a estratégia utilizada:

```
[8, 14, 21, 17, 4, 13, 10, 9, 2, 52, 18, 7, 6, 44,
 63, 3, 95, 5, 79, 16, 1, 12, 11, 95, 1]
```

No código-fonte, existem duas variáveis para registrar o tempo do sistema antes da chamada da função e após seu retorno, sendo a diferença armazenada na variável **delta**.

```
t1 = time.time()
funcao(lista)
t2 = time.time()
delta = t2 - t1
```

Para cada vetor, será obtido um delta e montado uma tabela com as estatísticas. Os vetores foram criados, utilizando uma função da linguagem para gerar número aleatórios e inserí-los no vetor.

```
lista = range(tamanho)
random.shuffle(lista)
```

4 Configurações

Os testes foram efetuados utilizando a seguinte configuração de *hardware* sendo o foco principal o processador e sua frequência de operação, pois há diferenças significativas entre essas duas plataformas, das quais podemos destacar: a forma de armazenamento de dados (*hard disk* e *flash*), *scheduler*, *file system*, etc; e que postos em análise poderiam contribuir numa mudança.

4.1 x86

- Processador: Intel Pentium 4 (2.00 GHz)
- Sistema Operacional: Gentoo
- Kernel: Linux 2.6.24.1

4.2 ARM

- Modelo: Nokia E62
- Processador: ARM 926 (220 Mhz)
- Plataforma: TI OMAP 1710 [3]
- Sistema Operacional: Symbian OS

5 Resultados

5.1 Valores

Table 1: Intel x86 para 1.000 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	1.01714992523	0.0343110561371
2	0.953474998474	0.0415890216827
3	1.06153392792	0.0378048419952
4	1.20091104507	0.0291819572449
5	1.0036149025	0.0373599529266
6	1.05325102806	0.0351920127869

Table 2: Intel x86 para 1500 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	1.82341790199	0.0569121837616
2	1.51437902451	0.0601189136505
3	2.01726007462	0.0639839172363
4	1.45304584503	0.0574250221252
5	1.7247979641	0.0583050251007
6	1.67257595062	0.0598630905151

Table 3: Intel x86 para 2000 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	5.36400198936	0.0388009548187
2	4.98106908798	0.0391700267792
3	4.87079906464	0.0514640808105
4	6.46941709518	0.0369310379028
5	4.91478204727	0.0418050289154
6	5.24874901772	0.0378589630127

Table 4: Intel x86 para 10000 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	64.7194669247	0.219444036484
2	79.7865900993	0.224624156952
3	66.2718219757	0.265799045563
4	87.2412400246	0.267018079758
5	65.7647690773	0.226465940475
6	63.7938668728	0.245011091232

Table 5: ARM (Nokia E62) para 1000 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	21,0	1,0
2	20,0	1,0
3	26,0	0,9
4	22,0	1,0
5	19,0	1,0
6	21,0	1,0

Table 6: ARM (Nokia E62) para 1500 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	49,0	1,0
2	46,0	1,0
3	47,0	1,0
4	48,0	1,0
5	48,0	1,0
6	49,0	1,0

Table 7: ARM (Nokia E62) para 2000 inserções

	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	84,0	1,0
2	78,0	1,0
3	81,0	1,0
4	80,0	1,0
5	86,0	1,0
6	87,0	1,0

Table 8: ARM (Nokia E62) para 10000 inserções

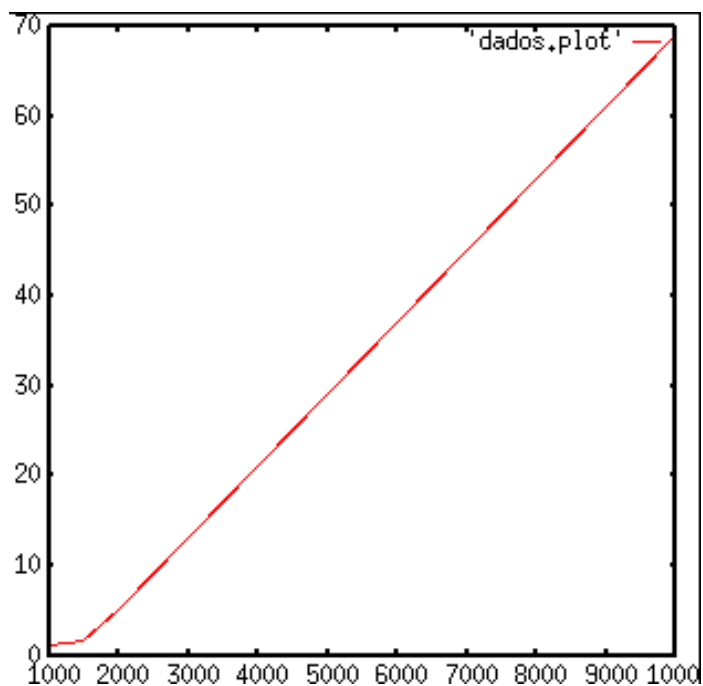
	<i>Bubble Sort</i> (s)	Quick Sort (s)
1	2148,0	9,0
2	2138,0	9,0
3	2148,0	9,0
4	2150,0	9,0
5	2147,0	9,0
6	2148,0	9,0

Podemos rapidamente perceber as diferenças de tempos na execução do algoritmo de *Bubble Sort* e Quick Sort analisando o aumento de tempo para os diferentes tamanhos de entrada.

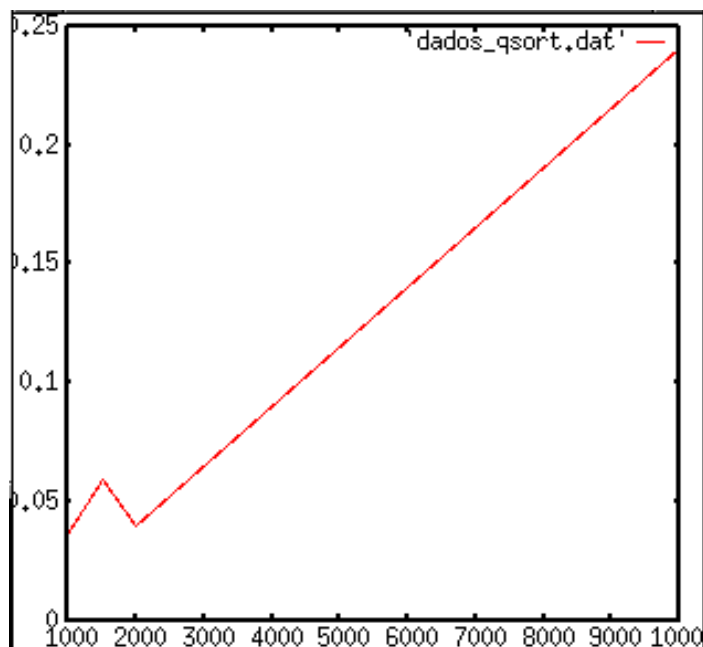
5.2 Gráficos

Utilizando o software **gnuplot** para criar os gráficos, obtemos:

5.3 *Bubble Sort*



5.4 Quick Sort



6 Outras abordagens

A linguagem Python possui um utilitário (`trace.py`) que retorna o o número de vezes que cada linha do código-fonte é executada, a sintaxe é bem simples e intuitiva¹:

```
python trace.py -c programa.py
```

Algumas abordagens podem ser feitas utilizando o número de vezes que cada linha é executada durante a execução do programa. O resultado é armazenado em um arquivo com a extensão `.cover`.

7 Conclusões

Através de uma análise empírica podemos avaliar a eficiência de um algoritmo. Nota-se que mesmo executando em uma plataforma menos eficiente, os resultados condizem proporcionalmente. Os testes feitos com o algoritmo *bubble sort* na plataforma com processador ARM mostraram-se muito ineficientes para o caso de vetores grandes (acima de 10000 entradas) além de pequenas distorções no tempo do x86, como visto no gráfico do *Quick Sort* podem ser consideradas desprezíveis. Finalmente, com os gráficos notamos comportamento, embora não exato devido a baixo número de pontos utilizados, próximo dos valores obtidos matematicamente.

1. $\Theta(n^2)$ para o *Bubble Sort*
2. $\Theta(n \ln n)$ para o *Quick Sort*

Outro ponto relevante é a escolha da linguagem Python, pois esta implica numa perda de performance se comparada com a linguagem C ou

¹É preciso especificar o caminho completo para o `trace`, normalmente algo como: `python /usr/lib/python2.5/trace.py -c programa.py`

C++, mas cumpre muito bem o papel de permitir testes em duas plataformas diferentes, sem alteração alguma no código-fonte.

8 Referências

- [1] http://www.codecodex.com/wiki/index.php?title=Bubble_sort#Python
- [2] [http://en.literateprograms.org/Quicksort_\(Python\)](http://en.literateprograms.org/Quicksort_(Python))
- [3] <http://tinyurl.com/ebux3>