Relatório para o trabalho de Sistemas Distribuídos

Alessandro S. S. Júnior[1], **Jaylson P. Oliveira**[1], **Thalles C. Fontainha**[1]

[1]Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro – RJ – Brasil

alessandrossjunior@gmail.com, jaylson.oliveira@gmail.com, cfthalles@gmail.com

Abstract. *Regarding high performance and parallel programming, python has been gaining ground with several library projects for this area. One of them is python dataflow (pyDF) also called Sucuri. It creates the environment required for programming using more than one thread. The user uses it as shared memory making its use less dependent on hardware knowledge. The present work makes use of this library for an email verification program using parallelism with pipeline concept.*

Resumo. *No que diz respeito a programação de alto desempenho e paralela, o python vem ganhando espaço com vários projetos de bibliotecas para essa área. Uma delas é o python dataflow (pyDF) também chamado de Sucuri. Ela cria o ambiente necessário para programação utilizando mais de uma thread. O usuário a utiliza como memória compartilhada tornando o seu uso menos dependende de conhecimento do hardware. O presente trabalho faz uso dessa biblioteca para um programa de verificação de emails utilizando paralelismo com conceito de um pipeline.*

1. Introdução

A Sucuri é uma biblioteca para python para programação paralela na qual permite o uso de mais de um ”worker”trabalhando em conjunto. Ela é baseada em duas partes. A primeira relacionada ao grafo de execução, na qual são instanciados nós e arestas para orquestrar o fluxo do programa. Esses nós podem ser feeders(alimentadores) que levam entradas ao grafo, Nodes(Nós) estes atrelados a execução de uma função, Souces(Fontes) que levam um objeto iterável,FilterTagged(Filtro) que age sobre cada item do source, é atrelado a uma função e pode ser serializado, Serializer(Serializador) que ordena a saída de acordo com a chegada dos itens de entrada, entre outros. Cada item desses pode ser instanciado o atribuindo a uma variável respeitando seus parâmetros e depois precisam ser adicionados ao grafo já previamente instanciado. A segunda parte é o escalonador que irá organizar a execução do programa de acordo com o descrito no grafo. Esta parte é realizada automaticamente para o programador como uma abstração.

2. Metodologia

O funcionamento do programa assim como as partes específicas da sucuri serão explicadas nesta seção de acordo com o código abaixo:

Código em Python

Filtragem de Email

```
import sys, os, time, math
from pyDF import *
sys.path.append(os.environ['PYDFHOME'])

def readFile(args):
    filename = args[0]
    f = open(filename, "r")

    vector = []

    for line in f:
        vector.append(line)

    f.close()

    return nprocs, vector

def filterMails(args):
    sp1 = args
    sp = sp1[0].split("@")
    if len(sp) == 2:
        aux = sp[1][:-1]

        if(aux == "gmail.com"):
            ret = sp1[0][:-1]
        else:
            ret = ""
    else:
        ret = ""

    return ret

def printMails(args):
    if args[0] != "":
        print args[0]


nprocs = int(sys.argv[1])
filename = sys.argv[2]

graph = DFGraph()
sched = Scheduler(graph, nprocs, mpi_enabled = False)

fp = open(filename, "r")

src = Source(fp)
graph.add(src)
```

```
nd = FilterTagged(filterMails, 1)
graph.add(nd)

ser = Serializer(printMails, 1)
graph.add(ser)


src.add_edge(nd, 0)
nd.add_edge(ser, 0)


t0 = time.time()
sched.start()
t1 = time.time()

print "Execution time %.3f" %(t1-t0)
```

Primeiramente é preciso configurar o ambiente da sucuri. Isso é feito através da variável PYDFHOME que precisa receber o valor do diretório da Sucuri. Particularmente, este programa precisa estar no mesmo diretório que contém a pasta pyDF. O arquivo a ser lido pelo programa deve estar no mesmo diretório do programa caso seja chamado só pelo nome. É recebido em linha de comando os valores da quantidade de workers e o nome do arquivo.

A função readFile para ler o arquivo recebe o nome dele como parâmetro. Em seguida o arquivo é aberto e cada linha é passada para um vetor. O arquivo é fechado e é retornado o numero de workers e o vetor no qual cada indice é uma linha do arquivo.

Já a função filterMails para filtrar os emails recebe uma linha do vetor retornado pela função anterior. Tenta-se separar a string em duas pelo caracter @. Se não conseguir retorna vazio pois já não é email válido. Se conseguir checa se a segunda parte da string(depois do @) é "gmail.com". Se não for, retorna vazio pois não é um gmail válido. Caso seja, retorna o email completo.

A função print simplesmente imprime os emails úteis válidos. Embaixo há duas linhas para a leitura dos parâmetros da linha de execução.

Agora ocorre a parte da Sucuri para organizar a execução das funções acima. Primeiramente, o grafo e o escalonador são instanciados nas variáveis graph e sched. Após, é criado o nó Source na variável src. Lembrando, esse nó introduz um objeto iterável no grafo. Para isso também é preciso adicionar o src ao grafo. Para adicionar itens ao grafo é utilizado o comando graph.add(nomeVariavelDesejada), neste caso src. Da mesma forma é criado o FilterTagged na variável nd e adicionado ao grafo. Por ultimo é gerado o serializer e adicionado também ao grafo.

A partir deste momento, o grafo tem os nós porém ainda desconectados. É feita agora a ligação do Source ao FilterTagged e do FilterTagged ao Serializer com o comando src.add.edge(nd,0) e nd.add.edge(ser,0).

Até aqui, o programa não realiza nenhuma função para a solução do problema, somente sendo realizada a criação do grafo, além do escopo das funções. A partir do

comando sched.start() é dado inicio do programa para realizar a função em si da listagem dos emails obedecendo ao fluxo estipulado no grafo.

3. Resultados

3.1. Testes

3.1.1. Instancia

Foram gerados arquivos com emails de vários formatos, inclusive inválidos para o programa filtrar todos os de final ”gmail.com”.Há um email por linha de arquivo. Os Arquivos utilizados para testes foram o database1.txt , database2.txt e o database3.txt

3.1.2. Medidas

Foram realizadas 5 execuções com 1, 2, 3 e 4 workers devido à arquitetura do computador utilizado que conta com 4 núcleos e em seguida foi capturada a média de tempo das execuções. A medida do tempo foi gerada através do ”time”do python para quando o escalonador começa, após o grafo estar montado. Medindo apenas a parte na qual o programa está sendo executado para a função que deve desempenhar.

Além do tempo, também foram geradas tabelas em relação ao melhor speed up e quanto ao throughput.

3.2. Tabelas

Arquivo	1 worker	2 workers	3 workers	4 workers
database1	5,7026	1,8392	1,6260	1,5968
database2	70,4852	23,8704	23,1710	23,4528
database3	120,3044	42,7848	40,1814	41,2420

Tabela 1. Média de 5 execuções. Tempo em segundos

Arquivo	Speed Up	Melhor worker
database1	3.5712	4
database2	3.0420	3
database3	2.9940	3
Speed Up Médio	3.2024	

Tabela 2. MleSpeed Up

Arquivo	Throughput 1w	Throughput 2w	Throughput 3w	Throughput 4w
database1	1753.59	5437.15	6150.06	6262.52
database2	1418.74	4189.29	4315.74	4263.88
database3	1246.84	3505.92	3733.07	3637.07
Throughput Médio	1473.05	4377.45	4732.96	4721.16

Tabela 3. Throughput, w = worker

3.3. Gráficos

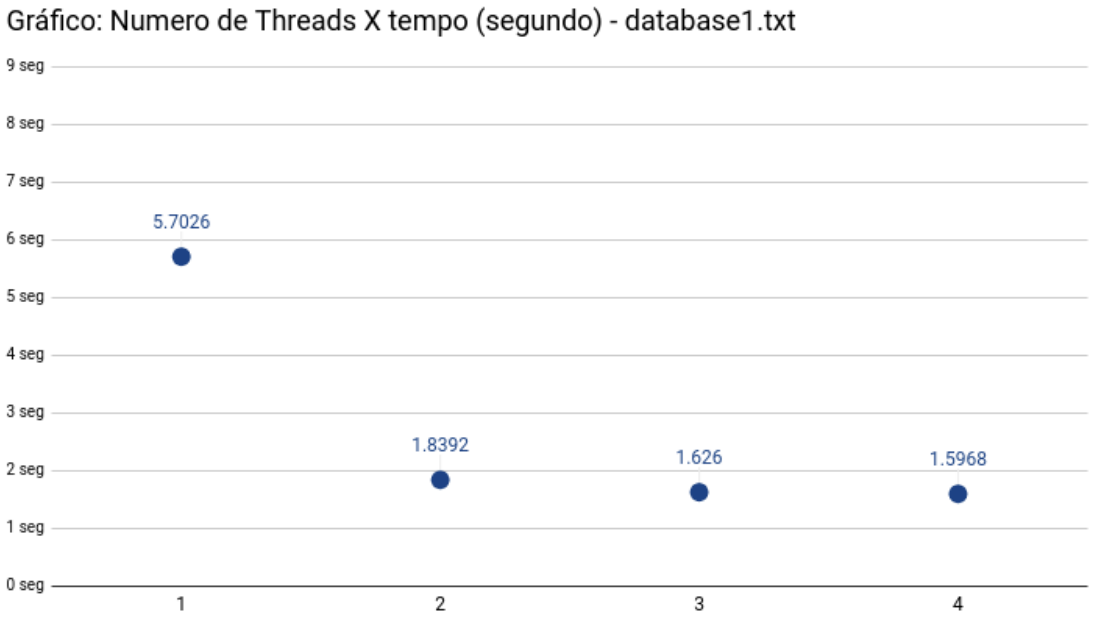

Figura 1. Gráfico de tempo database1

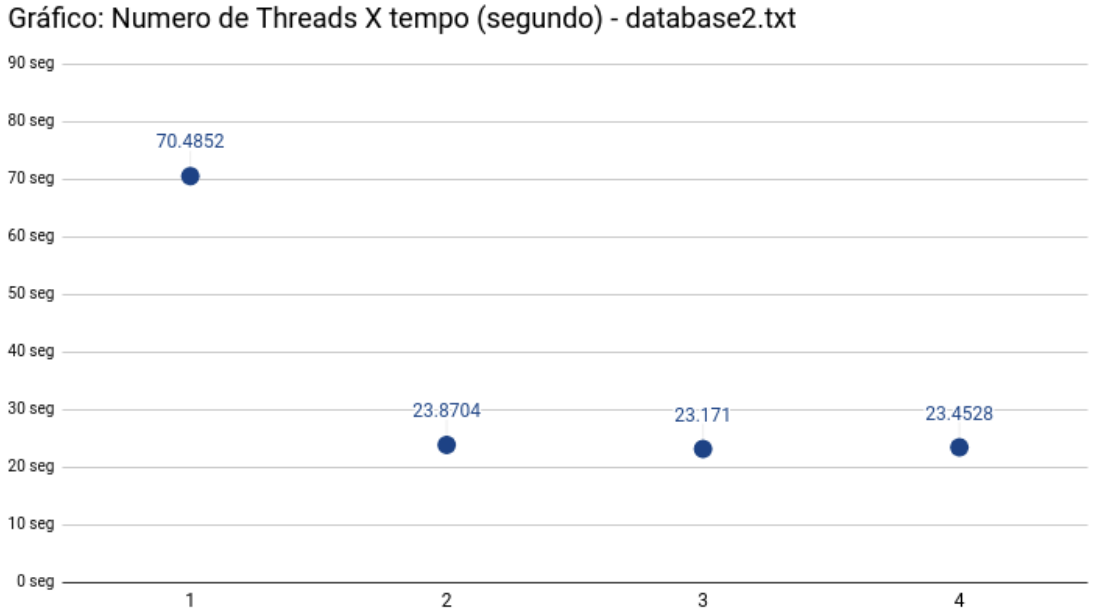

Figura 2. Gráfico de tempo database2

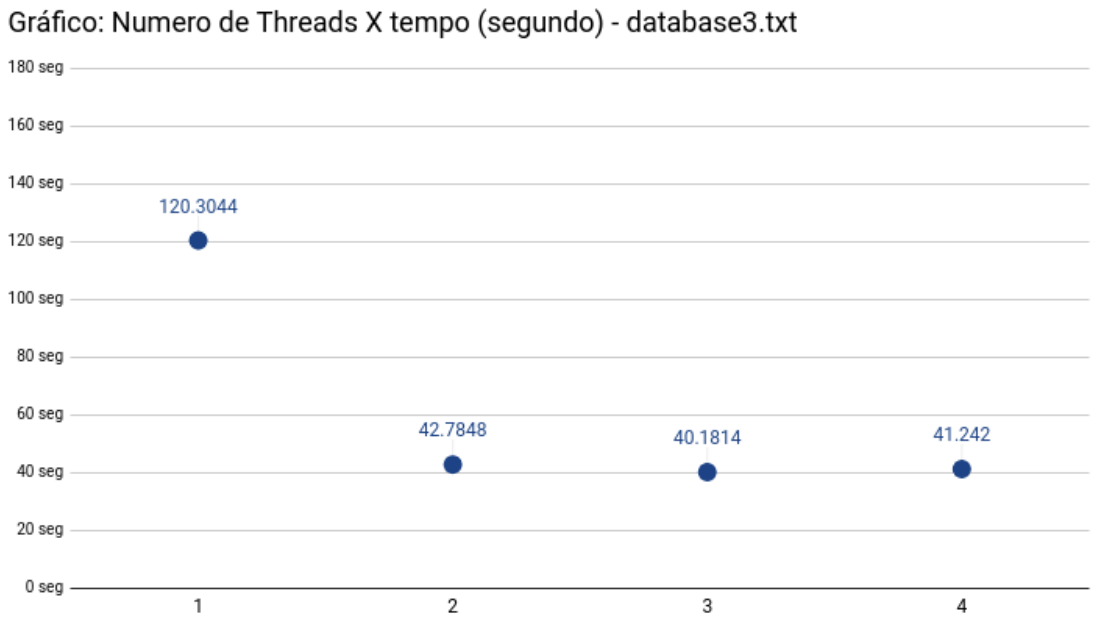

Figura 3. Gráfico de tempo database3

4. Conclusão

Pode-se observar dos gráficos e tabelas que neste programa em questão ocorre um grande ganho de dois workers em relação a um só. Quando adicionado mais um worker, totalizando 3, o ganho continua a crescer, porém cresce muito menos. Com quatro workers já foram obtidos resultados divididos. Em uma das instâncias ainda se obteve desempenho melhor, porém em duas já ficou pior que em relação a 3 workers. Conclui-se que há uma curva de desempenho na qual existe um valor de workers limite para continuar obtendo-se ganho. Neste caso já o limite seria 3 workers pois foi o melhor em garantir o equilíbrio entre o desempenho do throughput da pipeline do grafo e o custo do trabalho de escalonar os workers resultando no maior desempenho médio.

5. Agradecimentos

Ao monitor da disciplina, Rodrigo Cezar Leão, que sempre se mostrou dedicado e manteve empenho em colaborar para que não só nosso grupo, como outros grupos, também conseguissem instalar a biblioteca Sucuri, e ajudou-nos a pensar na ideia do trabalho para que conseguissemos êxito ao criá-lo.

6. Bibliografia e referências

T. A. Alves, L. A. J. Marzulo, and F. M. G. Franca, “Unleashing parallelism in longest common subsequence using dataflow,” in 4th Workshop on Applications for Multi-Core Architectures, 2013.

https://stackoverflow.com/questions/11835046/multithreading-and-multicore-differences

https://stackoverflow.com/questions/37308705/multi-threading-vs-clustering

https://github.com/tiagoaoa/Sucuri

https://stackabuse.com/how-to-permanently-set-path-in-linux/

https://youtu.be/Gyw32-1LgKc

http://www.linhadecodigo.com.br/artigo/3256/teste-de-desempenho-conceitos-objetivos-e-aplicacao-parte-1.aspxixzz68NDvrWp4