Universidade de São Paulo
Escola de Engenharia de São Carlos

Trabalho de Conclusão de Curso

Desenvolvimento de Biblioteca Paralela em Python para Aplicações em Engenharia Biomédica

Lucas Mikilita Ribeiro
Orientador: Prof. Dr. Carlos Dias Maciel

São Carlos, 30 de novembro de 2011

LUCAS MIKILITA RIBEIRO

DESENVOLVIMENTO DE BIBLIOTECA PARALELA EM PYTHON PARA APLICAÇÕES EM ENGENHARIA BIOMÉDICA

Trabalho de conclusão de Curso apresentado no Curso de Engenharia de Computação como disciplina obrigatória para obtenção do diploma de graduação em 30/11/2011.

Orientador: Prof. Dr. Carlos Dias Maciel

São Carlos

2011

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca - EESC/USP

Ribeiro, Lucas Mikilita
R484d Desenvolvimento de biblioteca paralela em *python* para aplicações em engenharia biomédica. / Lucas Mikilita Ribeiro ; orientador Carlos Dias Maciel -- São Carlos, 2011.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos da Universidade
de São Paulo, 2011.

1. *Python*. 2. Programação paralela. 3. Engenharia Biomédica. 4. Portabilidade. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Lucas Mikilita Ribeiro

Título: "Desenvolvimento de Biblioteca Paralela em Python para Aplicações em Engenharia Biomédica"

Trabalho de Conclusão de Curso defendido e aprovado em 30 / 11 / 2011 ,

com NOTA 9,0 (NOVE , ZERO), **pela comissão julgadora:**

Prof. MSc. Wagner Endo - UTFPR

Prof. MSc. Daniel Espanhol Razera - IFSP/SBV

Prof. Associado Evandro Luís Linhari Rodrigues
Coordenador pela EESC/USP do
Curso de Engenharia de Computação

RESUMO

A pesquisa no campo da Engenharia Biomédica envolve a aplicação de métodos da engenharia para a melhoria da saúde humana, sendo o estudo de sinais biomédicos de interesse crescente na comunidade científica. O uso de computação paralela é importante nesse cenário, pois os cálculos necessitam de um alto poder computacional. Dentre as linguagens de programação, a linguagem Python e o ambiente IPython proporcionam facilidade de programação, versatilidade e eficiência comprovada por diversos trabalhos científicos. Sendo assim, o objetivo desse trabalho foi descrever a linguagem Python e o ambiente IPython para o desenvolvimento de aplicações computacionais paralelas. Foi feita uma adaptação de um código usado em análise de sinais biomédicos para funcionar na versão mais nova do ambiente.

ABSTRACT

The research on the field of Biomedic Engineering applies engineering methods to better the human health, and biomedic signals has a crescent interest on the scientific community. The use of parallel computing is important in this scenario, because the calculations need a high computer power. Among the computer languages, Python and the ambient IPython provides easy programming, versatility and comproved efficiency by many scientific studies. The objective of this work was to describe the Python language and the IPython IDE for development of parallel computing applications. A code used in analysis of biomedic signals was adapted to work on the newest version of the IPython interpreter.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A Engenharia Biomédica é a aplicação de conceitos e princípios da engenharia nas áreas de medicina e biologia. Esse campo procura diminuir a distância entre a medicina e a engenharia, combinando as habilidades de solução de problemas da engenharia com o conhecimento médico para melhorar o diagnóstico, monitoramento e terapia de diversas condições de saúde.

A pesquisa nesse campo envolve a aplicação de métodos da engenharia para a melhoria da saúde humana, sendo o estudo de sinais biomédicos de interesse crescente na comunidade científica.

Os sistemas biológicos são sistemas complexos interconectados exibindo tanto características não lineares determinísticas quanto estocásticas. Esses sistemas possuem uma estrutura hierárquica eventualmente conhecida fazendo com que seus sinais possuam características únicas. Embora as abordagens de sistemas lineares tenham sido aplicadas nos últimos anos, os sistemas biológicos não são um fenômeno puramente linear.

Diversos trabalhos têm utilizado conceitos da teoria de informação para diferentes tipos de sistemas dinâmicos. Resultados recentes da literatura mostraram que a teoria de informação sobre esses processos apresenta resultados superiores quanto ao poder discriminatório de patologias. O uso de computação paralela é importante nesse cenário, pois os cálculos necessitam de um alto poder computacional.

Há uma forte tradição entre os cientistas de computação para usar linguagens compiladas, como Fortran77 e C, para simulações numéricas. Entretanto, em anos recentes houve uma procura maior por ambientes de interpretação e interação, como Matlab, Maple e Octave.

Em 1991, Guido Van Rossum criou a linguagem Python, uma linguagem de programação de alto nível e de propósito geral, cuja filosofia enfatiza a facilidade de leitura de código.

A linguagem Python e o ambiente IPython proporcionam facilidade de programação, versatilidade e eficiência comprovada por diversos trabalhos científicos

Somando-se essas características ao suporte à computação paralela, Python se

mostra promissor para a computação científica. A linguagem atrai um interesse crescente entre os cientistas, e sua adoção é cada vez mais comum em trabalhos científicos.

Nesse contexto, o objetivo desse trabalho foi descrever a linguagem Python e o ambiente IPython para o desenvolvimento de aplicações computacionais paralelas, e em seguida portar o código-fonte, usado pelo Prof. Carlos Dias Maciel [7] em estudo de sinais biomédicos, para a versão mais atual do ambiente.

2. REVISÃO BIBLIOGRÁFICA

2.1. ENGENHARIA BIOMÉDICA

Muitos dos problemas que os profissionais da área de saúde enfrentam são de grande interesse para engenheiros, pois suas soluções dependem de aplicações práticas e design de sistemas, etapas fundamentais à engenharia.

A Engenharia Biomédica é a aplicação de conceitos e princípios da engenharia nas áreas de medicina e biologia. Esse campo procura diminuir a distância entre a medicina e a engenharia, combinando as habilidades de solução de problemas da engenharia com o conhecimento médico para melhorar o diagnóstico, monitoramento e terapia de diversas condições de saúde.

É uma área que apenas recentemente emergiu como um campo de estudo próprio, devido à interdisciplinariedade de seus assuntos. A maior parte dos trabalhos na área são de pesquisa e desenvolvimento, podendo ser citados: equipamentos de imagens como ressonância magnética e eletroencefalograma, diversos equipamentos de diagnósticos que variam desde equipamentos clínicos a micro implantes, remédios e terapias. São algumas atividades da engenharia biomédica:

- Aplicação de métodos de engenharia para problemas biológicos;
- Detecção, medição e monitoramento de sinais fisiológicos;
- Interpretação de diagnósticos via técnicas de processamento de sinais;
- Dispositivos de reposição de funções biológicas, como próteses e órgãos artificiais;
- Análise computadorizada de dados para tomada de decisão;
- Imagens médicas;
- Criação de novos produtos biológicos, como tecidos artificiais.

A engenharia biomédica contém desde abordagens teóricas e não-experimentais até aplicações usando o estado da arte da tecnologia. Envolve pesquisa, desenvolvimento, implementação e operação. Assim como na medicina, é improvável que uma pessoa só consiga obter fluência em todos os assuntos desse tema. Porém, devido à natureza interdisciplinar, especialistas em biosensores, por exemplo, podem interagir com especialistas em próteses para se informarem do mesmo sinal bioelétrico comum à ambos. Especialistas em automação laboratorial podem colaborar com especialistas em sistemas inteligentes, para a confecção de um sistema de tomada de decisão em uma clínica.

Usando a tecnologia existente e a metodologia da engenharia, podem-se criar ferramentas e técnicas voltadas à medicina, de modo a melhorar o sistema de saúde e torná-lo mais eficiente e com alto custo-benefício.

2.2. LINGUAGENS DE PROGRAMAÇÃO

Há uma forte tradição entre os cientistas de computação para usar linguagens compiladas, como Fortran77 e C, para simulações numéricas. A demanda por flexibilidade de software tambem popularizou as linguagens de alto nível, como C++ e Fortran95. Entretanto, em anos recentes houve uma procura maior por ambientes de interpretação e interação, como Matlab, Maple e Octave (Cai, 2005).

O sucesso do Matlab e outros ambientes similares é devido à sintaxe simples e limpa, integração entre simulação e visualização, interatividade na execução de comandos, e uma biblioteca rica de funcionalidade numérica (Cai, 2005).

Em 1991, Guido Van Rossum criou a linguagem Python, uma linguagem de programação de alto nível e de propósito geral, cuja filosofia enfatiza a facilidade de leitura de código. Essa linguagem clama combinar grande poder com uma sintaxe clara, direta e livre de ambiguidades. Seu uso de identação como delimitador de blocos é única entre as linguagens de programação.

A linguagem Python e o ambiente IPython proporcionam facilidade de programação, versatilidade e eficiência comprovada por diversos trabalhos científicos. Python foi

designado para ser extensível com código compilado para aumentar a eficiência, com várias ferramentas disponíveis para facilitar a integração do Python com códigos compilados.

Neste sentido, a linguagem Python se tornou uma alternativa competitiva ao Matlab. O Python, quando extendido por módulos numéricos e de visualização (IPython, bibliotecas Scipy, Numpy e Matplotlib), fornece todas as virtudes do Matlab. Uma vantagem particular do Python é de ser uma linguagem extremamente rica e poderosa, com eficiência similar a C e Fortran.

2.2.1. LINGUAGEM PYTHON

Em particular, Python é uma linguagem interpretada e orientada a objetos, que suporta sobrecarga de operadores e oferece interfaces multi-plataformas. Programadores C++ podem portar o seu código facilmente para o Python e obter mais flexibilidade e elegância. A criação de classes em Python é muito simplificada, sendo uma das principais razões para que cientistas escolham essa linguagem interpretada ao invés de linguagens compiladas.

Outra vantagem do Python é interfacear códigos legados em Fortran, C e C++, sendo muito mais simples que em outros ambientes. Isso se deve ao fato de que Python foi designado para ser extensível com código compilado para aumentar a eficiência, com várias ferramentas disponíveis para facilitar a integração do Python com códigos compilados.

Somando-se essas características ao suporte à computação paralela, Python se mostra promissor para a computação científica. A linguagem atrai um interesse crescente entre os cientistas, e sua adoção é cada vez mais comum em trabalhos científicos.

É importante notar que o Python também é usado em diversas áreas fora do meio acadêmico. É usado como uma linguagem de propósito geral em aplicações como: administração de sistemas, *sites* da internet, sistemas distribuídos, engenharia de software, interfaces gráficas, motores de busca (Google), rede de computadores, educação, e até aplicações empresarias de larga escala. As referências [1, 2, 5, 6] demonstram algumas dessas aplicações.

Muitos paradigmas de programação são suportados, como orientação a objetos, funcional e imperativo. Possui gerenciamento de memória automático similar à Ruby ou Perl.

Como se trata de uma linguagem de tipo dinâmico, Python é frequentemente usado como linguagem de *script*, mas isso não o limita a ser usado somente para esse fim. Programas feitos em Python podem ser transformados em executáveis, que são rodados através de interpretadores disponíveis na grande maioria dos sistemas operacionais. A Figura 1 apresenta um exemplo do console Python.

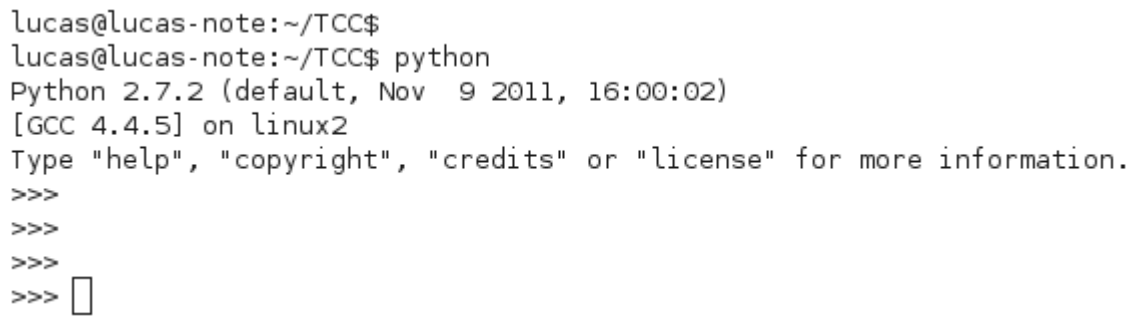

Figura 1: – O console Python

O Python pode ser usado em inúmeras aplicações:

- Desenvolvimento Web
- Escrever scripts CGI.
- Suporte a scripts HTML e XML.
- Suporte a vários protocolos de internet, como HTTP, TCP, UDP e outros.
- Banco de Dados
- Interfaces com MySQL, Oracle, MS SQL Server, PostgreSQL, SybODBC.
- API padrão de banco de dados.
- Banco de dados com objetos como ZODB e Durus.
- Interfaces gráficas
- Tk GUI, wxWidgets, GTK+, Qt via pyqt ou pyside.
- *Microsoft Foundation Classes* através de extensões win32.
- Suporte à Delphi.
- Computação Numérica e Científica
- Bioinformática.

- Física e Matemática através das bibliotecas Scipy e Numpy.
- Educação
- Frameworks para desenvolvimento de jogos: PyGame e PyKyra.

O Python é muito usado para o ensino de programação, tanto a nível iniciante como avançado. Diversas instituições de ensino a empregam: Universidade da Flórida, Universidade de Oxford, Universidade de Toronto, dentre outras.

Vários jogos comerciais utilizam Python em alguma área de funcionamento. Podemos citar: *Battlefield 2* e *1942*, *EVE Online*, *Sid Meyer Civilization IV* e outros. A figura 2 mostra uma imagem do jogo Civilization IV.

Figura 2*:* Civilization IV, famoso jogo de estratégia que utiliza Python

2.2.2. O AMBIENTE IPYTHON

O IPython é uma ferramenta que complementa o Python, de forma a torná-lo mais interativo e produtivo. Suas características são:

- Terminais Python interativos
- Suporte para visualização interativa de dados e interfaces gráficas.
- Ferramentas de alto nível para computação paralela interativa.
- Código aberto com licença BSD.

A figura 3 mostra um exemplo de interatividade.

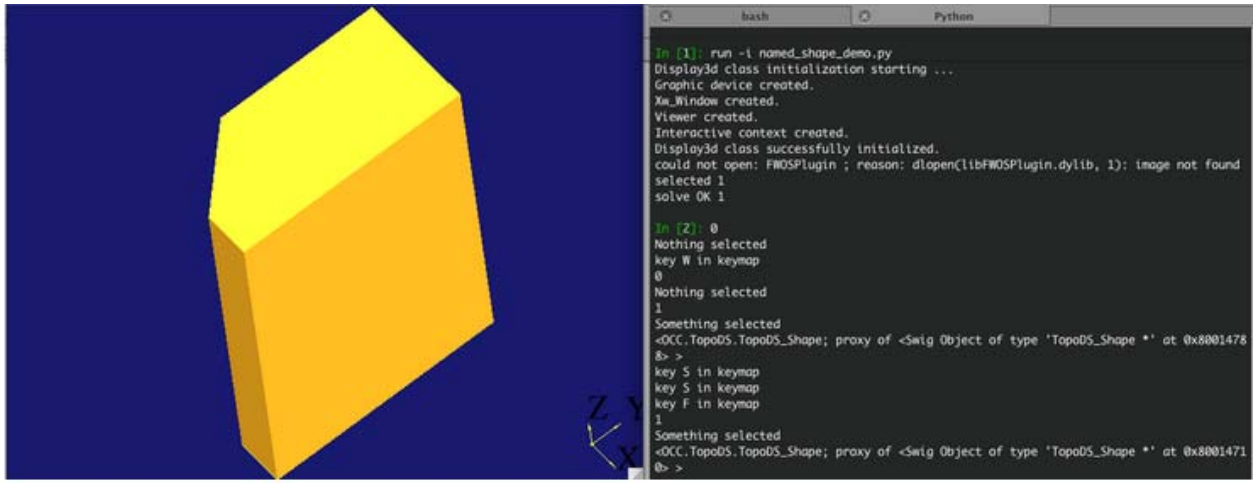

Figura 3: Terminal IPython e visualização de resultados em imagens

Uma aplicação poderosa é sem dúvida a capacidade de paralelização. Inúmeros projetos científicos utilizam o IPython como ferramenta por ser altamente eficiente e prático. Pode-se citar:

– O Departamento de Engenharia Mecânica da Universidade do Colorado, como parte de *Denver Aerosol Sources and Health* – DASH, coleta amostras de partículas do ar e as

compara com amostras antigas, para encontrar relações entre os poluentes encontrados e a saúde. Uma rede neural foi criada usando IPython para o reconhecimento de padrões.

– O trabalho de pesquisa do Prof. Dr. Carlos Dias Maciel[7], onde a paralelização é utilizada para analisar a resposta de impulsos nervosos gerados por movimentos gaussianos aleatórios na junção do tornozelo de um gafanhoto. Essa paralelização agiliza os cálculos de forma quase linear, onde quanto maior o número de núcleos de processamento, mais rápido será o cálculo.

2.2.2.1 - Desempenho

Em computação paralela, uma relação importante a ser levada em consideração é a razão entre o tempo de processamento e o tempo de comunicação. Para que uma aplicação funcione adequadamente, essa razão deve ser a maior possível. Com isso, a granularidade do sistema deve ser ajustada para que não se desperdice tempo demais em *overheads* ao invés de processamento útil.

2.2.2.2. - Latência

O envio e recebimento de mensagens pequenas nos dá uma estimativa do tempo em que o IPython gasta para para a comunicação. Pode-se obter uma estimativa do *overhead* mínimo do sistema ao analisar esses dados.

Os testes foram realizados pela interface *loopback,* que é uma interface onde onde os pacotes são enviados e imediatamente respondidos com os mesmos pacotes, em uma máquina de 8 núcleos de processamento e 4 engines de execução. A figura 4 mostra as latências obtidas.

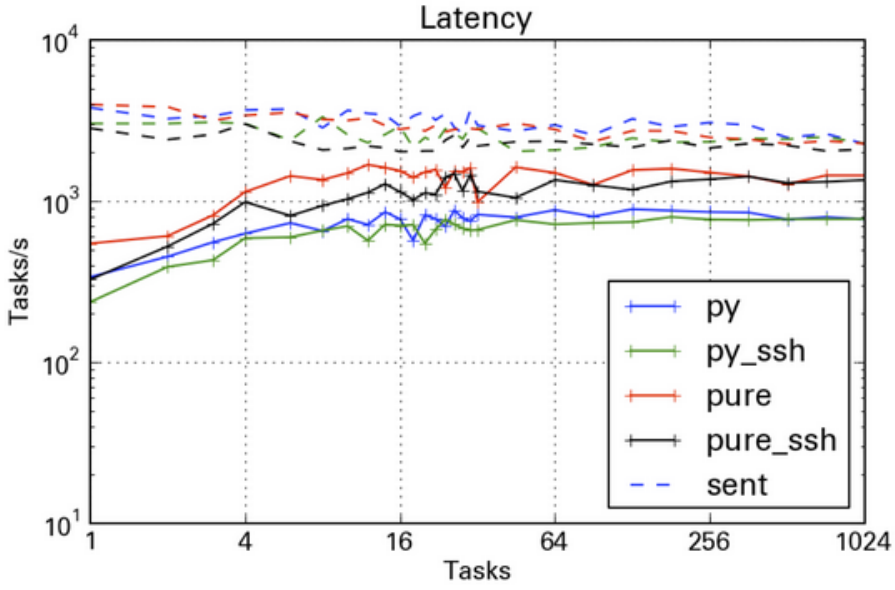

Figura 4: Gráfico de latência

Os testes foram realizados com o agendador do Python e pure-zmq, com e sem um túnel SSH.

Pode-se notar que o agendador Python pode realizar 800 tarefas por segundo, enquanto que o agendador zmq chega a 1500 tarefas por segundo. Um túnel SSH não influencia a performance consideravelmente.

O mesmo teste em um *cluster* dedicado com 128 CPUs mostra a boa escalabilidade da plataforma, visível na figura 5:

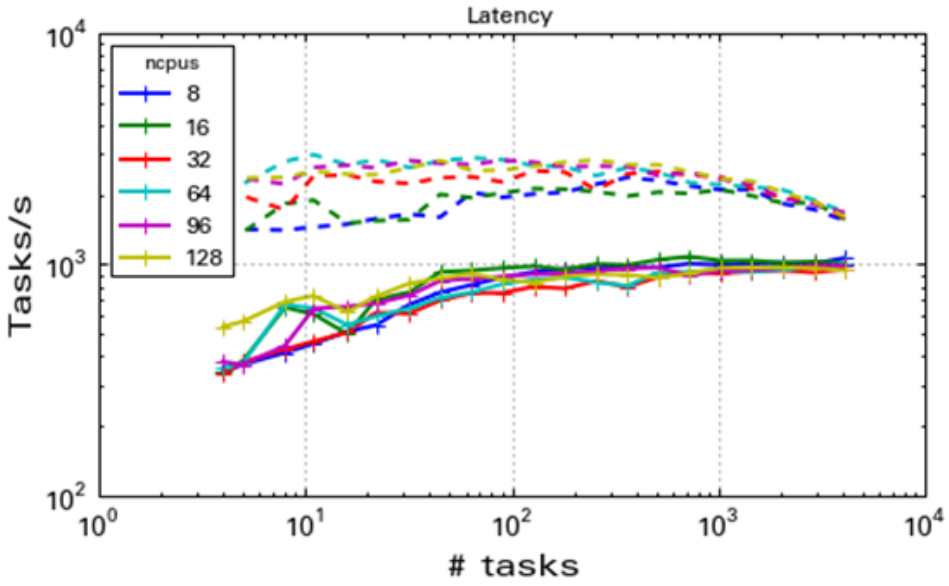

Figura 5: Latência em um cluster de 128 CPUs.

2.2.2.3 - Throughput

Para a medição de *throughput*, o mesmo método de eco foi utilizado, mas usando vetores de tamanho variável ao invés de variar o número de mensagens. O resultado é mostrado na figura 6.

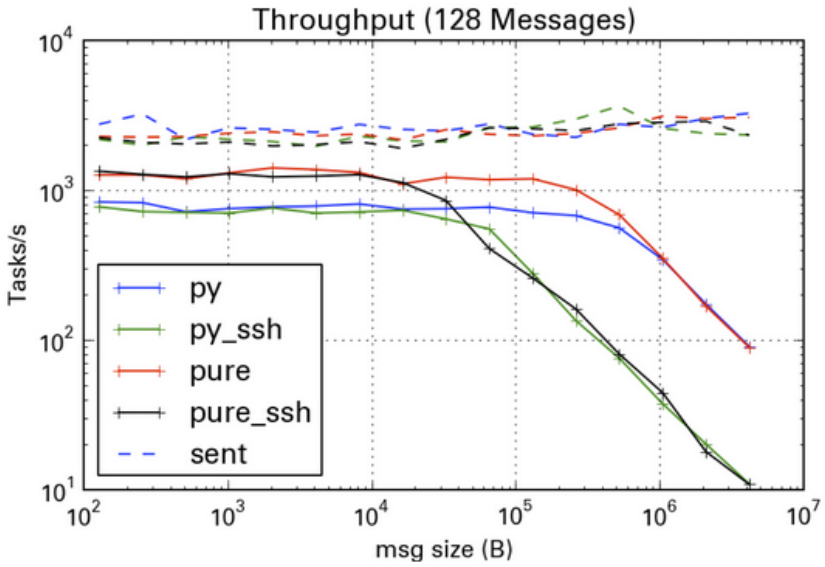

Figura 6: Throughput de acordo com o tamanho da mensagem.

As linhas pontilhadas, que medem o tempo para enviar os vetores, não são funções do tamanho da mensagem. Isso é possível devido ao envio de mensagens sem copiá-las antes. Pode-se enviar um elevado número de dados de maneira rápida, com pouco *overhead*.

3. MATERIAIS E MÉTODOS

O computador utilizado para a instalação e execução dos programas é um notebook Asus M6B00N, com processador Penium M 1.6 Ghz e 1.2GiB de RAM, rodando Debian Linux 6.0.3 (squeeze), kernel 2.6.32 e gnome 2.30.2.

Os softwares são: Python 2.7.2, IPython 0.12dev, NumPy 1.6.1, SciPy 0.9.0, Matplotlib 1.1.0.

3.1 – Instalação do Python

A instalação do Python em sistemas operacionais recentes é muito fácil. A maioria das distribuições Linux e UNIX já vem com uma versão do Python instalada. Até mesmo algumas versões do Windows incluem o interpretador Python na instalação.

Porém, algumas distribuições Linux podem não incluir a versão mais recente do Python em seu repositório. É o caso do Debian, conhecido por possuir somente versões antigas em prol da estabilidade do sistema.

Essa preocupação do Debian é legítima, porém, existem aplicações em que são necessárias versões atuais do software, pela correção de *bugs* e novas funcionalidades. Nesse caso, é necessário baixar o código-fonte, compilar para o sistema e instalar. A seguir, seguem instruções para a instalação do Python versão 2.7.2, que é a versão mais recente da árvore 2.x, no Debian 6.0.3, versão mais recente da distribuição Linux considerada mais universal pela comunidade.

– Primeiramente baixar e instalar a versão 6.0.3 do Debian para arquiteturas x86 , obtida em “http://cdimage.debian.org/debian-cd/6.0.3/i386/iso-cd/debian-6.0.3-i386-CD-1.iso”

– Baixar e descompactar o código-fonte do Python no site oficial: “http://www.python.org/ftp/python/2.7.2/Python-2.7.2.tar.bz2”

– Deve-se instalar as dependências necessárias para compilar o código. Com um terminal bash aberto em superusuário (su), inserir “visudo” para editar o arquivo de super-usuários. É preciso adicionar o usuário atual abaixo de ROOT, para obter permissões de instalação. A figura 7 mostra o arquivo de permissões.

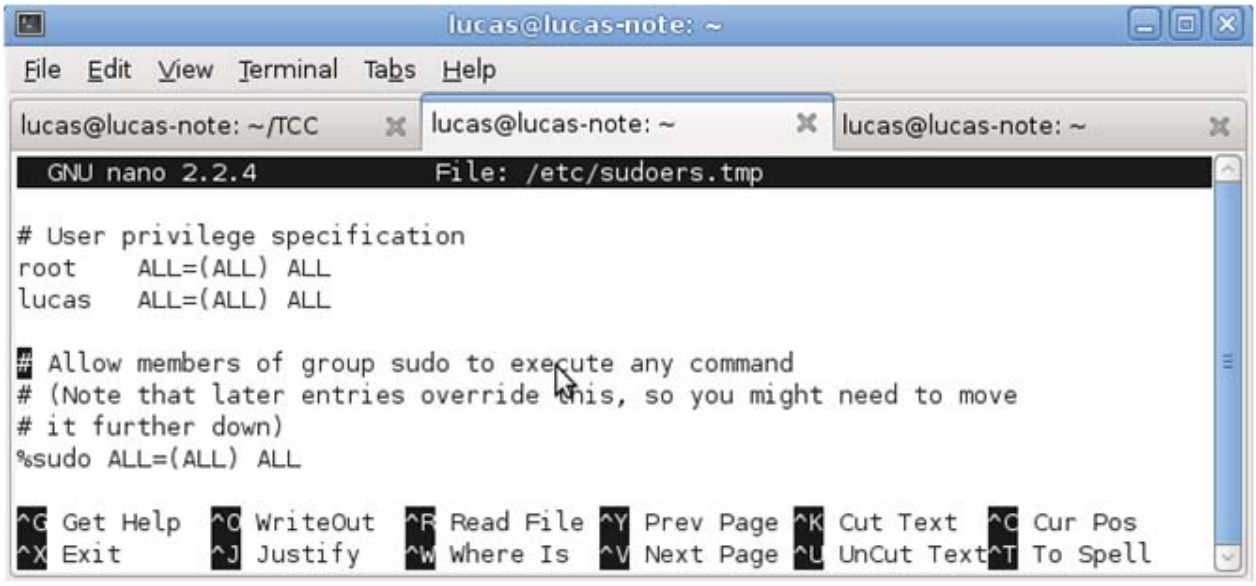

Figura 7: Obtendo permissões

– Todas os comandos seguintes podem ser executados em modo usuário, com a adição de “sudo” para obter permissões. É importante que o usuário da máquina seja o atual e não ROOT, para não haver diferenças de versões entre o usuário e root.

– Digitar “sudo apt-get install build-essential” para obter todas as dependências básicas de compilação, como gcc e g++.

– Em seguida, mudar para o diretório onde foi descompactado o Python e digitar “./configure”. As dependências serão checadas e se não houver pendências, digitar “make”.

– Após a compilação, digitar “sudo make install” para instalar nos diretórios do sistema.

Se tudo ocorrer bem, a versão 2.7.2 estará instalada, e para executá-la basta digitar “python” em um terminal. O comando já chamará a versão mais recente instalada.

3.2 - Instalação do IPython e bibliotecas

Assim como o Python, a ferramenta IPython está disponível pré-compilada na maioria das distribuições Linux. Porém, as versões mais novas só estão disponíveis em código-fonte, sendo necessário compilar suas dependências.

Para obter a funcionalidade desejada, é preciso instalar antes 3 bibliotecas de uso geral do IPython: NumPy, SciPy e Matplotlib.

3.2.1 - NumPy

NumPy é o pacote fundamental para computação científica no Python. Essa biblioteca contém, entre outras implementações:

– Vetores N-dimensionais

– Funções sofisticadas de *broadcasting*

– Ferramentas para integração de código C/C++ e Fortran

– Álgebra linear, transformada de Fourier, implementações de números aleatórios.

Além da sua utilidade científica, NumPy também pode ser usado para manusear dados multi-dimensionais, com tipos de dados arbitrários. Isso faz com que a biblioteca suporte os mais variados tipos de banco de dados.

É distribuída sob a licença BSD, permitindo o seu uso com poucas restrições.

Passos de instalação:

– Baixar o código-fonte no site: http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/numpy-1.6.1.tar.gz/download

– Instalar as dependências gfortran e libatlas-dev: “sudo apt-get install gfortran libatlas-dev”

– Mudar para o diretório onde foi descompactado o NumPy, e executar “./configure” e em seguida, “make” e “sudo make install”.

3.2.2 - SciPy

A biblioteca SciPy foi construída para trabalhar com os vetores fornecidos pela biblioteca NumPy, provendo rotinas mais amigáveis e eficientes ao usuário. É de fácil uso e altamente eficiente, sendo comumente usada para manipulação de dados científicos.

Ela fornece módulos para: estatística, otimização, integração numérica, álgebra linear, transformada de Fourier, processamento de sinais, processamento de imagens e outras funções especiais.

– Baixar o código-fonte no site: http://sourceforge.net/projects/scipy/files/scipy/0.9.0/scipy-0.9.0.tar.gz/download

– Mudar para o diretório onde foi descompactado o SciPy, e executar "./configure" e em seguida, "make" e "sudo make install".

3.2.3 - Matplotlib

O Matplotlib é uma biblioteca para plotar gráficos 2D, produzindo imagens de qualidade em uma variedade de formatos e ambientes interativos. Pode-se gerar gráficos, histrogramas, espectros, gráficos de barra, *scatterplots* e outros com poucas linhas de código. A figura 8 mostra alguns exemplos.

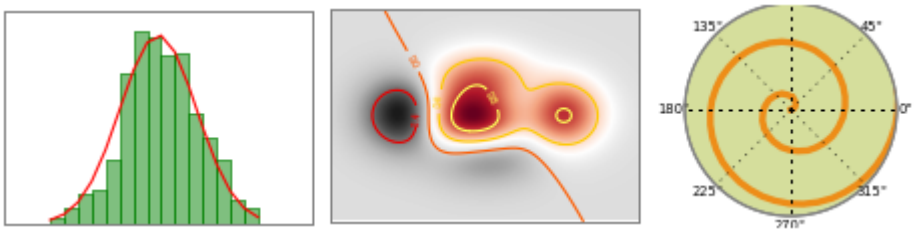

Figura 8: Exemplos de gráficos gerados pela matplotlib

Quanto à instalação:

– Baixar o código fonte no site: http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.1.0/matplotlib-1.1.0.tar.gz/download

– Mudar para o diretório onde foi descompactado o matplotlib, e executar “python setup.py build” e em seguida “sudo python setup.py install”.

3.2.4 – ZeroMQ

A biblioteca ZeroMQ é a responsável pela estrutura de paralelização do IPython. Sua utilização torna a troca de dados mais rápida do que usando o protocolo TCP convencional em aplicações paralelas. Possui uma comunidade de desenvolvimento grande e ativa, suportando mais de 30 linguagens, como C, C++, Java, .NET e Python.

É liberada como software livre sob a licensa LGPL, com suporte comercial oferecido pela empresa Imatix. Para instalar:

– Baixar o código fonte no site: http://download.zeromq.org/zeromq-2.1.10.tar.gz

– Instalar as dependências: “sudo apt-get install uuid-dev”

– Mudar para o diretório onde foi descompactado o ZeroMQ e executar “./configure”, e em seguida, “make” e “sudo make install”.

3.2.5 – PyZeroMQ

o Pyzmq integra a biblioteca zeromq à versão do Python instalada na máquina. Para instalar:

– Baixar o código fonte no site: https://github.com/zeromq/pyzmq/downloads/pyzmq-2.1.10.tar.gz

– Mudar para o diretório onde foi descompactado o Pyzmq e executar: “sudo python setup.py configure” e em seguida “sudo python setup.py install”.

3.2.6 – IPython

Depois de instalar todas as bibliotecas anteriores, o IPython estará com todas as dependências necessárias. Para instalar:

– Criar uma pasta com o nome python-dev e mudar para esse diretório, em seguida, executar “git clone https://github.com/ipython/ipython.git”, em seguida “cd ipython” e depois “python setup.py configure”

– A figura 9 mostra a saída desejada, mostrando a versão do python e do pyzmq instalados.

```
=============================================================================
BUILDING IPYTHON
               python: 2.7.2 (default, Nov  9 2011, 16:00:02)  [GCC 4.4.5]
             platform: linux2

OPTIONAL DEPENDENCIES
               sphinx: Not found (required for building documentation)
             pygments: Not found (required for syntax highlighting
                       documentation)
                 nose: 1.1.2
              pexpect: no (required for running standalone doctests)
                pyzmq: 2.1.10
             readline: yes
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

Figura 9: Dependências do IPython.

– Executar “sudo python setup.py install”.

Após a compilação e instalação, pode-se digitar “ipython” em um terminal e o ambiente deve abrir como na figura 10:

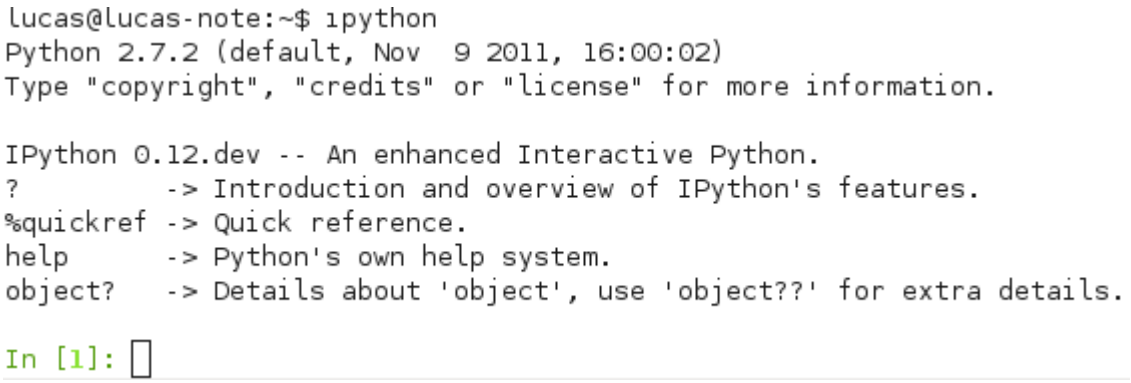

Figura 10: Ambiente IPython.

3.3. Código-fonte

O código usado na análise de sinais pelo Prof. Maciel funciona na versão antiga do IPython (0.10), mas precisa de modificações nas versões mais novas. Abaixo, o código original:

```
#!/usr/bin/env python

# bibliotecas

from   IPython.kernel  import client

from   sys import argv

from   os import chdir, getcwd

import numpy

import scipy

import scipy.stats

import scipy.signal

import pylab

import matplotlib

import scipy.signal

import scipy.integrate

from matplotlib import pyplot, mlab, pylab
```

```
# Inicializa as maquinas paralelas
mec = client.MultiEngineClient()
print 'Paralel machines ID -> ', mec.get_ids()
mec.activate()
mec.execute('import numpy')
mec.execute('from scipy import *')
mec.execute('from numpy import *')
###########  fecha todas as janelas
pylab.close('all')



NBins =   128
epson =   0

DirInicio   = argv[1]
DirTrabalho = getcwd()
chdir(DirInicio)

arquivo = '00_te.txt'
fid = open(arquivo, 'r')



Mlag= 3000
lag = range(-Mlag,Mlag)

mec.scatter('lag',lag)
mec.push({'NBins':NBins})
mec.push({'Mlag':Mlag})

import library15
reload(library15)

#mec.push_function({'Ixyz':Ixyz})
```

```
mec.push_function({'Ixy':library15.Ixy})


Surrogate = 'nok'
NExec = 14
Tol = 0.00001
mec.push({'Tol':Tol})




mec.push_function({'erro':library15.erro})            # erro(X,Y):
mec.push_function({'substitui':library15.substitui})  # substitui(x,y):
mec.push_function({'AAFT':library15.AAFT})            # IAAFT(y, Tol, grafico):
mec.push_function({'SurrogateData':library15.SurrogateData})

I = range(NExec)    # gera as repeticoes em funcao do numero de cores ativos e NExec
mec.scatter('I',I)



import matplotlib.mlab
import scipy.signal
import time
NN = 0
for line in fid:
               T0 = time.time()
               print NN,' ',line,
               F = line.split()
               nome = F[0]
               Nome = nome[0:nome.find('.')]
               Fs, Canais, Ruim = library15.LeituraCanais(Nome)
               if Ruim == True:
                              break
```

```
                t1 = time.time()

                x, y, valor = library15.LeituraSinais(nome, F)

                C = library15.InfCapacity(x, y, Fs)

                print 'Capacidade Canal = {0:.2f}'.format(C)


                yd = scipy.signal.decimate(y, 60, ftype='fir')

                m = 40 # tamanho maximo para entropy rate

                NS = 2  # numero de simbolos


                HR = library15.EntropyRate(yd,m,NS)

                RespPred = library15.Predictability(HR,NS)


                print 'G = {0:.2f}'.format(RespPred[0]),' R = {0:.2f}'.format(RespPred[1]),' T =
{0:.2f}'.format(RespPred[2]),     '     E     =     {0:.2f}'.format(RespPred[3]),     '     hmu     =
{0:.2f}'.format(RespPred[4])


                st = Nome+'_HR'+'_'+F[1]+'_'+F[2]


                numpy.savez(st, a = RespPred, b = HR, c = C)


#               t2 = time.time()

                print 'Comprimento yd = ', len(yd)

#               print 'tempo = ', t2 - t1

#               raise SystemExit

#               break


                mec.push({'x':x})

                mec.push({'y':y})

                mec.push({'valor':valor})


                NL  = len(x) - 2*Mlag

                NL  = int(numpy.exp2(int(numpy.log2(NL))))

                print 'NL = ',NL

                mec.push({'NL':NL})
```

```
                xd  = x[Mlag:Mlag+NL].T
                yd  = y[Mlag:Mlag+NL].T

                mec.push({'xd':xd})
                mec.push({'yd':yd})

                print 'Calculando: x -> yt'
                mec.execute('MI = [Ixy(xd, y[Mlag+i:Mlag+NL+i], valor) for i in lag]')
                MI = mec.gather('MI')
                if Surrogate == 'ok':
                                print 'Geracao dos dados surrogate'
                                Erro = []
                                xs  =  xd.copy()  #  faz  uma  copia  do  xd  apenas  por  precaucao  de
estragar esse dado
                                mec.push({'xs':xs})
                                x_s            =            library15.SurrogateData(xs,'IAAFT',NExec)
#'PhaseRandom',NExex)
                                print 'Calculando: xs -> yt',
                                VSMI = []
                                for i in I:
                                                x_surr = x_s[i]
                                                mec.push({'x_surr':x_surr})
                                                mec.execute('SMI  =  [Ixy(x_surr,  y[Mlag+i:Mlag+NL+i],
valor) for i in lag]')
                                                SMI = mec.gather('SMI')
                                                VSMI.append(SMI)
                                                pylab.grid()
                                print ' '
                                st = Nome+'_MI'+'_'+F[1]+'_'+F[2]
                                numpy.savez(st,a = lag, b = MI, c = VSMI)
                else:
                                st = Nome+'_MI'+'_'+F[1]+'_'+F[2]
                                numpy.savez(st,a = lag, b = MI, c = [])
```

```
                T1 = time.time()
                print '  duracao = ',str(numpy.floor(T1-T0)),' sec'
                print ' '
                NN = NN + 1
chdir(DirTrabalho)
```

Tal código não funciona na versão 0.12dev devido à mudança da biblioteca paralela, que na versão atual utiliza ZeroMQ. Esse novo design resultou em um ganho de performance em aplicações paralelas, além de clarear a execução de código em máquinas remotas.

4. RESULTADOS E DISCUSSÃO

4.1 – Processos

O modelo de processamento para o novo código paralelo é similar ao Ipython.kernel, utilizado nas versões 0.10 e anteriores. Ainda há o controlador, *engine* e clientes. Entretanto, o controlador agora é dividido em múltiplos processos e pode ser dividido em várias máquinas. O script ipcontroler utilizado para iniciar o funcionamento do cluster ainda é executado da mesma forma.

4.2 – Criando um cliente

Criar um cliente com configurações padrão não mudou muito. Uma mudança significativa é de que não há mais múltiplas classes Client para representar os modelos de execução. Agora há somente um Client de baixo nível para conectar com o cluster, e objetos View são criados através desse Client que fornece as interfaces de execução.

Para criar um novo cliente:

```
# old
In [1]: from IPython.kernel import client as kclient

In [2]: mec = kclient.MultiEngineClient()

In [3]: tc = kclient.TaskClient()

# new
In [1]: from IPython.parallel import Client

In [2]: rc = Client()

In [3]: dview = rc[:]

In [4]: lbview = rc.load_balanced_view()
```

4.3 – Outra diferença

Foi observado que o “push” de funções existentes em arquivos separados não estavam se comportando da maneira esperada. A solução encontrada foi manter todas as funções em um único arquivo, de modo a enviar as funções para os motores corretamente.

4.4 – Alterações

As alterações necessárias para executar o código foram poucas. A seguir, as alterações feitas estão destacadas em vermelho.

```
#!/usr/bin/env python

# bibliotecas
from IPython.parallel import Client

from sys import argv
from os import chdir, getcwd
from matplotlib import pyplot, mlab, pylab

import numpy
import scipy
import scipy.stats
```

```
import scipy.signal
import pylab
import matplotlib
import scipy.signal
import scipy.integrate
import library15
import non_parallel
import time


def Ixy(x,y, valor):
               from numpy import sum, histogramdd, log
               global NBins
               R = [x.T]
               px, edges = numpy.histogramdd(R,bins= NBins, range = [valor[0]])
               px = px/sum(px)

               R = [y.T]
               py, edges = numpy.histogramdd(R,bins= NBins, normed=True, range = [valor[1]])
               py = py/sum(sum(py))

               R = [x.T,y.T]
               pxy, edges = numpy.histogramdd(R,bins= NBins, normed=True, range =
[valor[0],valor[1]])
               pxy = pxy/sum(sum(pxy))
               Ixy = 0
               escala = range(NBins)
               for j in escala:
                              for i in escala:
                                             if px[i]!=0 and py[j]!=0 and pxy[i,j]!=0:
                                                            Ixy = Ixy +
pxy[i,j]*(log(pxy[i,j]/(px[i]*py[j])))
               return Ixy


# Inicializa as maquinas paralelas
rc = Client()
dview = rc[:]
dview.block = True
print 'Paralel machines ID -> ', rc.ids

dview.execute('import numpy')
dview.execute('from scipy import *')
dview.execute('from numpy import *')
############  fecha todas as janelas
pylab.close('all')



##############################################################################################
# programa principal
##############################################################################################

############ parametros globais ########################################################
NBins =   128
epson =   0

#dir passado ao iniciar
DirInicio   = argv[1]
DirTrabalho = getcwd()
chdir(DirInicio)

arquivo = '00_te.txt'
fid = open(arquivo, 'r')
```

```
#max lag
Mlag= 30
lag = range(-Mlag,Mlag)

#espalha variavel lag entre as engines
dview.scatter('lag',lag)
dview.push({'NBins':NBins})
dview.push({'Mlag':Mlag})

#carrega funcao Ixy nas engines
dview.push({'Ixy':Ixy})

Surrogate = 'nok'
NExec = 14
Tol = 0.00001
dview.push({'Tol':Tol})



dview.push({'erro':library15.erro})            # erro(X,Y):
dview.push({'substitui':library15.substitui})  # substitui(x,y):
dview.push({'AAFT':library15.AAFT})            # IAAFT(y, Tol, grafico):
dview.push({'SurrogateData':library15.SurrogateData})

I = range(NExec)    # gera as repeticoes em funcao do numero de cores ativos e NExec
dview.scatter('I',I)

#iteracao
NN = 0
#executa para cada arquivo de entrada
for line in fid:
               #registra tempo inicial
               T0 = time.time()
               #imprime numero da iteracao
               print NN,' ',line,
               F = line.split()
               nome = F[0]
               Nome = nome[0:nome.find('.')]
               #le a descricao da entrada
               Fs, Canais, Ruim = non_parallel.LeituraCanais(Nome)
               if Ruim == True:
                              break

               t1 = time.time()
               x, y, valor = non_parallel.LeituraSinais(nome, F)
               C = non_parallel.InfCapacity(x, y, Fs)
               print 'Capacidade Canal = {0:.2f}'.format(C)

               yd = scipy.signal.decimate(y, 60, ftype='fir')
               m = 40 # tamanho maximo para entropy rate
               NS = 2  # numero de simbolos

               HR = non_parallel.EntropyRate(yd,m,NS)
               RespPred = non_parallel.Predictability(HR,NS)

               print 'G = {0:.2f}'.format(RespPred[0]),' R = {0:.2f}'.format(RespPred[1]),' T =
{0:.2f}'.format(RespPred[2]), ' E = {0:.2f}'.format(RespPred[3]), ' hmu =
{0:.2f}'.format(RespPred[4])

               st = Nome+'_HR'+'_'+F[1]+'_'+F[2]

               numpy.savez(st, a = RespPred, b = HR, c = C)

               print 'Comprimento yd = ', len(yd)
```

```
                dview.push({'x':x})
                dview.push({'y':y})
                dview.push({'valor':valor})

                NL  = len(x) - 2*Mlag
                NL  = int(numpy.exp2(int(numpy.log2(NL))))
                print 'NL = ',NL
                dview.push({'NL':NL})

                xd  = x[Mlag:Mlag+NL].T
                yd  = y[Mlag:Mlag+NL].T

                dview.push({'xd':xd})
                dview.push({'yd':yd})

                print 'Calculando: x -> yt'
                dview.execute('MI = [Ixy(xd, y[Mlag+i:Mlag+NL+i], valor) for i in lag]')
                MI = dview.gather('MI')

                if Surrogate == 'ok':
                                print 'Geracao dos dados surrogate'
                                Erro = []
                                xs = xd.copy() # faz uma copia do xd apenas por precaucao de
estragar esse dado
                                rc[:].push({'xs':xs})
                                x_s = library15.SurrogateData(xs,'IAAFT',NExec)
#'PhaseRandom',NExex)
                                print 'Calculando: xs -> yt',
                                VSMI = []
                                for i in I:
                                                x_surr = x_s[i]
                                                rc[:].push({'x_surr':x_surr})
                                                rc[:].execute('SMI = [Ixy(x_surr,
y[Mlag+i:Mlag+NL+i], valor) for i in lag]')
                                                SMI = rc[:].gather('SMI')
                                                SMI = list(SMI)
                                                VSMI.append(SMI)
                                                pylab.grid()
                                print ' '
                                st = Nome+'_MI'+'_'+F[1]+'_'+F[2]
                                numpy.savez(st,a = lag, b = MI, c = VSMI)
                else:
                                st = Nome+'_MI'+'_'+F[1]+'_'+F[2]
                                numpy.savez(st,a = lag, b = MI, c = [])

                T1 = time.time()
                print '  duracao = ',str(numpy.floor(T1-T0)),' sec'
                print ' '
                NN = NN + 1

chdir(DirTrabalho)
```

4.5 – Execução

Para rodar o código, primeiramente devemos criar o cluster de *engines*. Pode-se criar um cluster na mesma máquina ou em máquinas diferentes, mas por motivos de praticidade, foi criado um cluster de 2 motores na mesma máquina. Cada motor estará alojado em um processo separado, podendo usufruir do desempenho de um processador com dois ou mais núcleos. Deve-se executar o comando em um terminal separado: “ipcluster start –n=2”. Depois de criado, o terminal deve exibir a seguinte mensagem indicando sucesso: “[IPClusterStart] Engines appear to have started successfully”.

Em outro terminal, deve-se ir para a pasta onde se encontram os códigos e executar o ipython. Ao abrir o *prompt*, executar “run transfe_15.py /home/lucas/TCC/dados”, substituindo o último diretório para onde os dados de entrada se encontram.

A figura 11 mostra a saída do programa:

```
In [1]: run transfe_15.py /home/lucas/TCC/dados/
Paralel machines ID ->  [0, 1]
0   ID-620.txt.gz 3 1 185806 200000
Channels:  ('X', 'FETi', '27', '200')
F Sampl :  24000.0
Capacidade Canal = 0.50
G = -0.96  R = 0.96  T = 40.88  E = 4.83  hmu = 0.04
Comprimento yd =  237
NL =  8192
Calculando: x -> yt
  duracao =  39.0  sec

1   ID-620.txt.gz 2 3 185806 513588
Channels:  ('X', 'FETi', '27', '200')
F Sampl :  24000.0
```

Figura 11: Saída do programa de acordo com os arquivos de entrada.

5. CONCLUSÃO

O uso da versão mais nova do IPython traz benefícios de performance, como demonstrado no desenvolvimento do trabalho. Novas aplicações podem usufruir de um código antigo, contanto que ele ainda funcione nas versões atuais. Devido à portabilidade realizada, o código criado pelo Prof. Maciel continuará sendo usado na análise de sinais.

O trabalho de conclusão de curso foi condizente com o Curso de Engenharia de Computação, que possui disciplinas de programação e Engenharia Elétrica. A aplicação em si é interdisciplinar, por se tratar de análise de sinais elétricos e tratamento com algoritmos e computação paralela.

Esse estudo é importante para a formação de um graduando, pois a confecção de um material científico é uma experiência importante para todos que vivem no meio acadêmico.

6. REFERÊNCIAS BIBLIOGRÁFICAS.

D. Arnold, M.A. Bond, Chilvers and R. Taylor, Hector: Distributed objects in Python, In Proceedings of the 4th International Python Conference, 1996.

D. Blank, L. Meeden and D. Kumar, Python Robotics: An environment for exploring robotics beyond LEGOs, In Proceedings of the 34th SIGCSE technical symposium on Computer Science Education, 2003, pp. 317–321.

Cai, X., Langtangen, H.P. & Moe, H., 2005. On the performance of the Python programming language for serial and parallel scientific computations.*Scientific Programming*, 13(1), p.31-56. Available at: http://portal.acm.org/citation.cfm?id=1239665.

Fernando Pérez, Brian E. Granger, *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

K. Jackson, pyGlobus: A Python interface to the Globus toolkit, Concurrency and Computation: Practice and Experience 14 (2002), 1075–1084.

R. Plosch, Design by contract for Python, In Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference, IEEE Computer Society, 1997, pp. 213–219.

C. D. Maciel, D. M. Simpson, P. L. Newland. Inference about Multiple Pathways in Motor Control Limb in Locust. A publicar.

Links acessados no dia 07/12/2011:
http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=4000007661
http://minrk.github.com/scipy-tutorial-2011/performance.html