dx.doi.org/10.7437/NT2236-7640.2011.01.004
Notas Técnicas, v. 1, n. 1, 2011

Uma Introdução a Programação Paralela com *Parallel Python*.

Pedro Otávio Teixeira Mello,* Marcelo Giovani Mota Souza,† and Nilton Alves Júnior‡
Coordenação de Atividades Técnicas (CAT), Centro Brasileiro de Pesquisas Físicas (CBPF), Rua Dr. Xavier Sigaud, 150, Urca, Rio de Janeiro, Brasil

A presente nota técnica tem como objetivo apresentar conceitos necessários para o entendimento e uso do processamento paralelo e demonstrar a implementação e utilização do conjunto de bibliotecas e *scripts* conhecido como PP (*Parallel Python*). Este documento apresenta também algumas arquiteturas paralelas e os principais propósitos de se utilizar o processamento paralelo. Aborda os conceitos de passagem de mensagens que é um modelo de programação paralela utilizando a biblioteca MPI (*Message Passing Interface*) como exemplo. O Parallel Python é apresentado como uma alternativa ao padrão *MPI*. Utilizado em conjunto com a linguagem de programação Python, possibilita a realização de processamento paralelo e distribuído. A preparação dos diferentes computadores de modo que se possa realizar um processamento paralelo também serão demonstrados. O envio de determinadas tarefas que necessitam de cálculos para diferentes processadores será exemplificado como também a requisição das respostas calculadas. Uma comparação dos resultados obtidos utilizando um número variável de processadores e uma demonstração que as tarefas foram enviadas e processadas em diferentes computadores serão ilustradas neste documento.

Keywords: Processamento Paralelo, Paralelismo, Programação Paralela, Python, Cluster, Parallelpython, Python Paralelo.

1. INTRODUÇÃO

O poder computacional dos equipamentos acessíveis aliado ao elevado custo dos equipamentos com alto poder computacional, gerou a necessidade de tecnologias de distribuição de tarefas mais acessíveis e simplificadas. Isto determinou o surgimento do conceito de processamento paralelo e/ou distribuído, que basicamente é a utilização racional de uma máquina com várias *CPUs* ou várias máquinas dedicadas com uma ou mais *CPUs* [1].

A linguagem de programação *Python* [2] está entre as linguagens que mais crescem em popularidade e número de programas escritos [3]. Desde dispositivos embarcados a servidores, a linguagem *Python* é suficientemente versátil para ser utilizado em todo tipo de aplicação. Praticamente não há nada que se possa fazer em outra linguagem que não seja possível com *Python*, de forma rápida e eficaz [4].

2. CONCEITOS E TERMINOLOGIAS

Este capítulo descreve a computação procedimental e a computação paralela, apresenta também dois tipos de arquiteturas computacionais paralelas e retrata a importância da utilização da passagem de mensagens por meio de programas exemplos. Um dos exemplos segue o modelo adotado como padrão por troca de mensagens, o outro exemplo utiliza um conjunto de bibliotecas e *scripts Parallel Python*.

*Electronic address: pmello@cbpf.br
†Electronic address: mgm@cbpf.br
‡Electronic address: naj@cbpf.br

2.1. Modelos de Programação

Um programa é uma solução computacional para um problema. Essa solução é descrita por meio de alguma linguagem de programação e cada linha de código é uma instrução a ser realizada. Ao executar um programa, o mesmo é dividido em N tarefas que vão de acordo com o número de instruções e essas tarefas são executadas por uma *CPU*, instrução por instrução (figura 1).

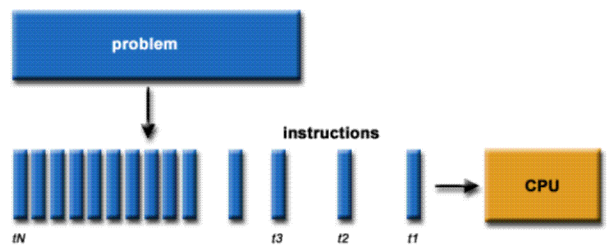

Figura 1: Execução Procedimental

Todo esse processo é conhecido como computação procedimental, pois todo o processo de execução ocorre em procedimentos, etapa por etapa. Tarefas que exigem um alto custo computacional, ou seja, tarefas que necessitem de maiores recursos computacionais para seus cálculos, podem ser impraticáveis se executadas em sua forma procedimental. Partindo desse princípio surge o paradigma da computação paralela.

A computação paralela consiste na divisão de um problema em partes independentes para que possam ser executadas em diferentes *CPUs* simultaneamente [1]. Com o aumento do número de *CPUs* trabalhando em conjunto, o trabalho realizado por cada *CPU* é menor e em conseguência desse fato, o tempo total de execução do programa diminui (figura 2).

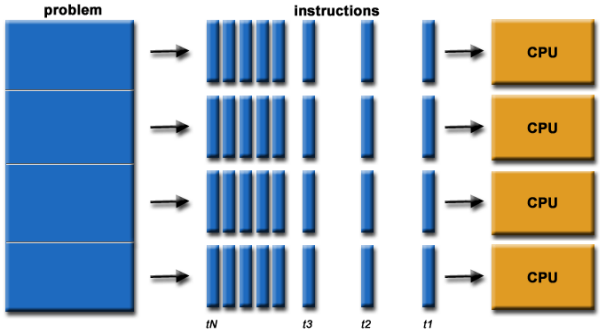

Figura 2: Execução Paralela

Programar em paralelo pode não ser uma tarefa simples, pois o modelo de programação (algoritmo) adotado diverge do modelo de programação procedimental. Primeiro o programador elabora uma lógica especial para a descrição de seu problema e divisão das tarefas, em seguida o programador deve se preocupar com a distribuição das tarefas de modo que os diferentes processadores computem o que lhes foram designados.

A programação paralela não é a solução para todos os problemas computacionais. Existem fatores que podem contribuir para que o processamento paralelo se torne menos eficiente se comparado ao processamento procedimental [5]. Logo abaixo seguem dois fatores que podem contribuir diretamente no desempenho do processamento paralelo.

- Muitas tarefas - Uma simples divisão de tarefas, pode exigir muitas comunicações entre os diferentes processadores e, como conseguência, gerar um custo de processamento elevado para o gerenciamento de todas as informações.
- Largura de banda - A velocidade da entrega das tarefas aos diferentes processadores está relacionada à largura de banda. A largura de banda pode causar atrasos no tempo da velocidade dessas entregas.

2.2. Arquiteturas Computacionais Paralelas

As arquiteturas computacionais paralelas consistem na utilização de diversas unidades de processamento.

A primeira arquitetura paralela abordada será a arquitetura *SMP* (*Simetrical Multi Processors*) onde sua principal característica é o compartilhamento de um mesmo banco de memória entre os processadores locais [6] e toda a comunicação se dá por meio do barramento[1] de memória (figura 3).

Os sistemas operacionais tratam as arquiteturas *multicore* como arquiteturas *SMP*, portanto as mesmas serão tratadas assim a partir daqui.

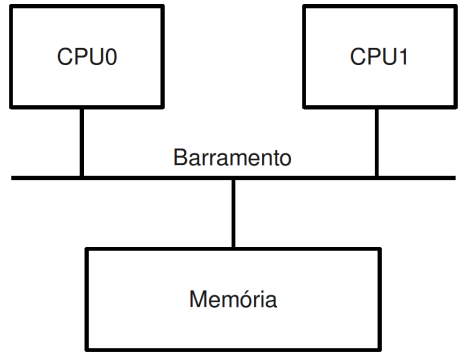

Figura 3: Arquitetura *SMP*

Um segundo tipo de arquitetura paralela é o *cluster* de computadores e é definido como um conjunto de computadores (heterogêneos ou não) conectados em rede. Em arquiteturas desse tipo são necessários mecanismos de troca de mensagens entre os diferentes computadores.

O projeto pioneiro em *clusters* de computadores ficou conhecido como *Beowulf* [7] e se desenvolveu no *CESDIS (Center of Excellence in Space Data and Information Sciences)* em 1994 e contava com 16 *PCs 486 DX-100* ligados em rede rodando *GNU/LINUX*. Na figura 4 pode ser visto um modelo de um *cluster* do tipo *Beowulf*.

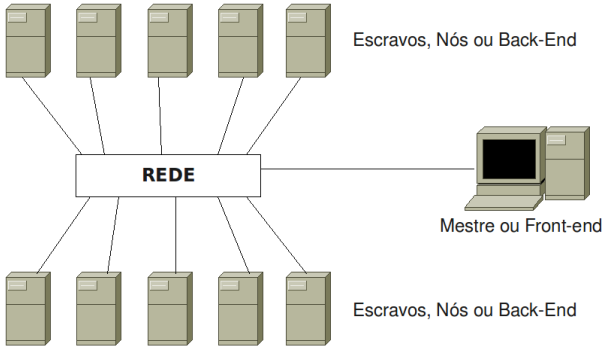

Figura 4: Arquitetura *Cluster* Beowulf

Uma das características de um *cluster Beowulf* é a sua arquitetura em redes que adota um modelo cliente-servidor. Basicamente todas as tarefas passam primeiramente pelo computador mestre (cliente) que faz requisições e envia tarefas para os computadores escravos (servidor).

Os diferentes tipos de arquiteturas computacionais podem receber diferentes classificações, de acordo com *Flynn* [8], essas classificações podem ser obtidas por

[1] O conjunto de linhas de comunicação que permitem a troca de informações entre os diferentes dispositivos que compõem um computador é chamado de barramento.

meio de uma análise do fluxo de dados onde quatro casos são possíveis (tabela 1).

- SISD - Single Instruction Single Data.
- SIMD - Single Instruction Multiple Data.
- MISD - Multiple Instruction Single Data.
- MIMD - Multiple Instruction Multiple Data.

Tabela I: Taxonomia de Flynn

Legenda	Instrução	Dados	Descrição
SISD	Única	Único	Processadores Tradicionais, onde para cada instrução há um único dado sendo operado.
SIMD	Única	Múltiplos	Mesma instrução sendo executada sobre diversos conjuntos de dados de forma paralela, muito utilizado em processamento de GPU ou máquinas multiprocessadas.
MISD	Múltiplas	Único	Várias instruções sendo operadas sobre o mesmo dado; não se tem um exemplo dessa classificação.
MIMD	Múltiplas	Múltiplos	Múltiplas instruções sendo executadas sobre diversos conjuntos de dados, o exemplo que mais se adéqua a esta classificação é o *cluster* de computadores.

2.3. Passagem de Mensagens

A partir do ano de 1980, diversas bibliotecas de funções foram criadas com o objetivo de realizarem trocas de informações entre diferentes computadores. Em consequência desse fato, não existia um padrão em relação à utilização e implementação dessas bibliotecas.

No ano de 1992, diversos desenvolvedores se reuniram e estabeleceram um padrão para o modelo de programação por troca de mensagens, esse modelo recebeu o nome de *MPI (Message Passing Interface)* [9].

O modelo de programação *MPI*, não é uma biblioteca propriamente dita, é um padrão que deve ser seguido por bibliotecas que adotarem esse modelo de programação.

A troca de mensagens entre diferentes computadores pode garantir a execução de uma, ou várias partes de uma tarefa nos diferentes computadores. A figura 5 exemplifica um modelo por trocas trocas de mensagens, onde a máquina A envia uma tarefa para a máquina B, esta máquina por sua vez tem que estar preparada para receber a mensagem e o mesmo ocorre inversamente.

Um exemplo de um código em C utilizando a biblioteca *mpi.h* se encontra na listagem 1. A seguir uma breve descrição deste código.

A função *MPI_Init(&argc,&argv)* (lin. 16) inicia o ambiente de execução *MPI*. Essa função deve ser chamada antes de qualquer outra função da biblioteca *mpi.h* e a última rotina que deve ser chamada é a função *MPI_Finallize()* (lin. 46), pois sua função é finalizar um ambiente de execução *MPI*. Os argumentos que foram

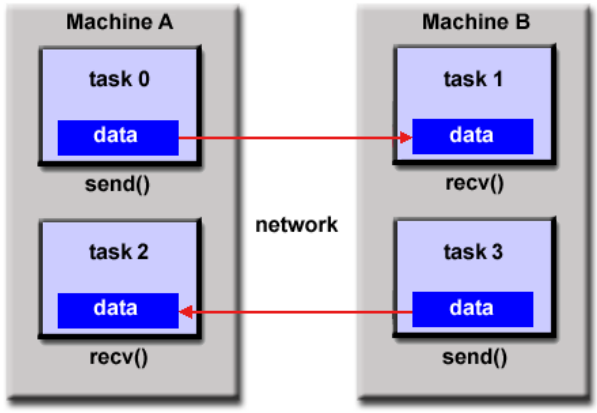

Figura 5: Passagem de mensagens em diferentes computadores

fornecidos em *argc* e *argv*, serão transmitidos aos diversos processos criados.

A função *MPI_Comm_size(MPI_COMM_WORLD, &numtasks)* (lin. 22), obtém o número de processos que será utilizado pelo programa exemplo e o valor será armazenado na variável *numtasks*.

A função *MPI_Get_processor_name(name,&resultlen)* (lin. 24), obtém o *hostname* do computador e armazena-o à variável *name*. O número total de caracteres do *hostname* é armazenado em *resultlen*.

As informações relativas a todos os processos *MPI*, estão inclusas no tipo definido *MPI_COMM_WORLD*.

Para dimensionar o vetor de caracteres da variável *name* (lin. 10), foi utilizado o valor já definido *MPI_MAX_PROCESSOR_NAME*, este valor é 128.

O número identificador dos processos é chamado de *rank*. Partindo disso, foi criada uma variável homônima ao identificador. A função *MPI_Comm_rank(MPI_COMM_WORLD,&rank)* (lin. 23) obtém o identificador dos diferentes processos e armazena-o na variável.

As variáveis de controle *dest* e *source* demarcam o destino e a origem das informações e seus valores nada mais são do que o próprio *rank* dos processos.

A variável *tag* pode assumir valores inteiros não negativos, atribuídos pelo programador. Essa variável é utilizada em operações de envio e recebimento de informações e o seu valor deve coincidir durante essas operações, sendo um identificador da informação enviada.

O tipo *MPI_Status* é uma estrutura utilizada em operações de recebimento de informações, toda essa estrutura contém o identificador da origem (*rank*) e o identificador da mensagem enviada (*tag*).

Por meio da variável *rank* é possível controlar as operações de envio e recebimento de informações. No programa exemplo, o processo 0 recebe os resultados obtidos pelos outros processos por meio da função *MPI_Recv* (lin. 30 até lin. 36). O valor da variável *source*

foi modificado dinamicamente por meio de um laço, o que possibilitou que o processo 0 pudesse receber diferentes resultados (lin. 31 até lin. 35).

Os processos que possuem o identificador diferente de 0 enviam seus resultados por meio da função *MPI_Send* (lin. 37 até 40).

As funções (lin. 23 e lin. 24) utilizadas pelo programa exemplo fornecem apenas um resultado, mas o programa é replicado por todos os processos que serão utilizados, com isso é possível obter diferentes resultados, como por exemplo: diferentes *ranks*, diferentes *hostnames*.

Listagem 1: Código exemplo com MPI

```
1  /*
2   * Programa exemplo MPI
3   */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include "mpi.h"
7  int main(int argc, char** argv){
8      int numtasks, rank, resultlen, dest, source,
9      tag = 1;
10     char name[MPI_MAX_PROCESSOR_NAME];
11     MPI_Status Stat;
12
13     /*
14      * INICIO APLICAÇÃO MPI
15      */
16     MPI_Init(&argc,&argv);
17     /*
18      * Obter o numero de processadores
19      * Obter o rank ou ID dos processadores
20      * Obter o hostname
21      */
22     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
23     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
24     MPI_Get_processor_name(name,&resultlen);
25
26     /*
27      * Testes para permitir que o rank 0 receba
            informações
28      * rank diferente de 0 envia informações
29      */
30     if (!rank){
31         for(source = 1; source < numtasks; source
               ++){
32             MPI_Recv(name, MPI_MAX_PROCESSOR_NAME,
33             MPI_CHAR, source, tag, MPI_COMM_WORLD,
                   &Stat);
34             fprintf(stderr,"Nome dos nodes: %s\n",
                   name);
35         }
36     }
37     else{
38         dest = 0;
39         MPI_Send(name, resultlen+1,
40         MPI_CHAR, dest, tag, MPI_COMM_WORLD);
41     }
42
43     /*
44      * FIM APLICAÇÃO MPI
45      */
46     MPI_Finalize();
47     return 0;
48 }
```

Todo esse código fornece a seguinte saída:

```
Nome dos nodes: node16.estat
Nome dos nodes: node2.estat
Nome dos nodes: node1.estat
Nome dos nodes: node9.estat
```

Figura 6: Paralelismo com MPI.

Para a execução do programa exemplo (listagem 1), foram utilizados 5 *nodes* de um *cluster* e o nome de um deles não aparece, pois é o *rank* 0.

Uma das alternativas à biblioteca *MPI*[2] é a utilização de um conjunto de bibliotecas e *scripts* denominado *Parallel Python*(PP) e todo esse conjunto é capaz de realizar processamento paralelo ou processamento distribuído em arquiteturas *SMP* e *Clusters* [10].

Todo esse conjunto de bibliotecas e *scripts PP* incorpora conceitos de *MPI*, porém, o estabelecimento da comunicação entre os diferentes processadores ocorre de forma não explícita, sendo totalmente simplificado ao programador e suas demais atribuições serão abordadas na seção seguinte.

2.4. Parallel Python

Criado por Vitalii Vanovschi [11], o conjunto de bibliotecas e *scripts* conhecido como *Parallel Python*(PP), teve sua primeira versão disponibilizada em seu site oficial no ano de 2005 e se encontra em código aberto. O conjunto será chamado apenas de módulo PP a partir daqui.

De modo a organizar suas atividades paralelas, o módulo PP trabalha com múltiplos processos e *IPC (Inter Process Communications)*[3]. Diante disso, é possível paralelizar tarefas em arquiteturas computacionais do tipo *SMP* e *Clusters*.

O módulo tem semelhanças com o *MPI* no conceito da mudança de parâmetros, mas seu modelo de programação diverge do *MPI*. A mudança de parâmetros ocorre ainda no computador cliente e depois são enviadas tarefas a diferentes *CPUs*. Basicamente, tem-se uma função que representa uma tarefa paralelizável e a mesma é enviada a diferentes *CPUs* por meio de um método *submit()* (figura 7).

[2] A biblioteca para a linguagem *Python* pode ser encontrada com o nome de *pyMPI*

[3] É o grupo de mecanismos que permite aos processos transferirem informação entre si.

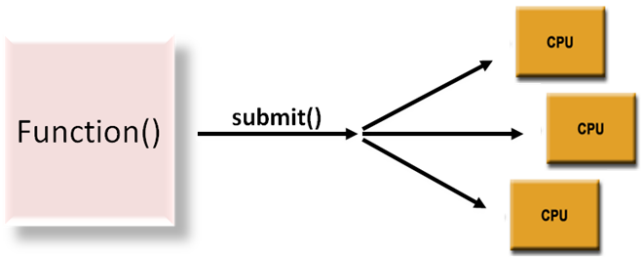

Figura 7: O método de envio de tarefas.

Diferente das bibliotecas que seguem o padrão *MPI*, o módulo PP se encontra disponível apenas para a linguagem de programação *Python*.

O módulo PP possui uma biblioteca que deve ser inserida ao código fonte de um programa (biblioteca pp) [4], nela estão classes e métodos que serão utilizados para escrever programas em paralelo.

O programa referente à listagem 2 foi escrito na linguagem de programação *Python* utilizando a biblioteca pp (lin. 7) e tem o mesmo comportamento que o programa exemplo *MPI* (listagem 1).

A função *f()* (lin. 11) será enviada para os diferentes processadores e terá como retorno o resultado da função *socket.gethostname()* (lin. 12) que pertence à biblioteca *socket* (lin. 8), cuja função é adquirir o *hostname* dos computadores por onde a função foi executada.

A classe pp.Server(ncpus, ppservers=ppservers) (lin. 20), determina o número de processadores que será utilizado pelo computador cliente durante a execução das tarefas, além de iniciar um servidor de tarefas que faz parte do módulo PP. O número de processadores que serão utilizados pelo computador cliente pode ser imposto pelo programador, bastando atribuir valores inteiros à variável *ncpus*.

O método *submit(f,(),(),("socket",))* (lin. 27) envia as tarefas para o servidor de tarefas, que por sua vez será responsável por enviar os processos aos demais computadores que serão utilizados. Em *ppservers* (lin. 16) é inserido uma lista dos nós que compõem o *cluster* que participarão durante o processamento da tarefa.

O método *print_stats()* (lin. 33) fornece a saída de algumas informações relativas à execução das tarefas, tais como: a quantidade de tarefas que cada *node* processou, tempo total para o término da execução do programa, tempo que cada *node* levou para processar as tarefas e o endereço dos servidores.

Por meio da inserção de valores à variável *recv* (lin. 30), o computador cliente requisita as tarefas de computadores servidores. As variáveis *send* e *recv* não fazem parte da sintaxe da biblioteca ou da linguagem em si, portanto seu nome não importa, podendo ser atribuído qualquer nome para esta finalidade.

Listagem 2: Código exemplo com PP

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #================================
4  #    Programa Exemplo em Python
5  #================================
6
7  import pp
8  import sys,socket
9
10
11 def f():
12     return socket.gethostname()
13
14 def main():
15     #tupla de servidores
16     ppservers = ("*",)
17     if len(sys.argv) > 1:
18         ncpus = int(sys.argv[1])
19         # criar lista de jobs utilizando ncpus
20         job_server = pp.Server(ncpus, ppservers=
               ppservers)
21     else:
22         # criar lista de jobs com detecção
               automática
23         job_server = pp.Server(ppservers=ppservers
               )
24     #Enviando a função
25     jobs = []
26     for x in range(4):
27         jobs.append(job_server.submit(f,() ,() ,("
               socket",)))
28     #Requisitando
29     for send in jobs:
30         recv=send()
31         print recv
32
33     job_server.print_stats()
34
35     return 0
36
37 if __name__ == '__main__': main()
```

A figura 8 é a saída do código referente à listagem 2.

```
node15.estat
node9.estat
node12.estat
node13.estat
Job execution statistics:
 job count | % of all jobs | job time sum | time per job | job server
         1 |         25.00 |       0.0007 |     0.000731 | 192.168.0.12:60000
         1 |         25.00 |       0.0008 |     0.000750 | 192.168.0.9:60000
         1 |         25.00 |       0.0005 |     0.000481 | 192.168.0.15:60000
         1 |         25.00 |       0.0007 |     0.000702 | 192.168.0.13:60000
Time elapsed since server creation 1.4560790062
```

Figura 8: Paralelismo com a biblioteca PP.

O *script ppserver.py* é um dos componentes do módulo PP e sua função é criar um canal de comunicação entre as diferentes *CPUs* de um *Cluster* para que possibilite o envio de tarefas.

Para a realização bem sucedida do programa exemplo (listagem 2), o *script ppserver.py* foi executado nos *nodes* que fizeram parte do processamento paralelo, no cliente foi executado o programa principal (figura 9).

Para a execução do programa exemplo (listagem 2), foram utilizados 5 *nodes* de um *cluster* e um deles foi utilizado como cliente. O *hostname* do cliente não aparece (figura 8), pois ele não fez parte do processa-

[4] A biblioteca pp é um dos componentes do módulo PP

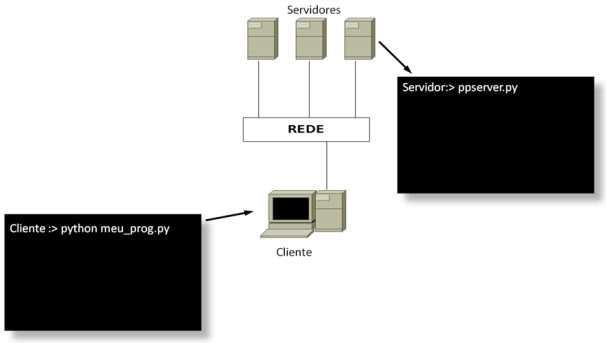

Figura 9: Preparação do ambiente de execução.

mento da função. O cliente apenas submeteu as tarefas e isso foi possível passando o valor 0 como argumento para *argv[1]*, logo, a variável *ncpus* recebeu o valor 0, não permitindo o processamento na máquina local ou cliente (lin. 18).

3. COMPARAÇÃO

Diversos testes foram executados sobre uma aplicação simples (listagem 3), do qual o objetivo é comparar as diferenças de tempo. A tabela II demonstra os resultados obtidos da aplicação executada em 3 diferentes formas de processamento: processamento paralelo em uma arquitetura computacional em *cluster* do tipo *Beowulf* com 3 *nodes*, processamento paralelo em uma arquitetura computacional do tipo *SMP* e por último sobre a forma procedimental.

A tabela III não considerou testes referentes ao processamento paralelo do tipo *SMP*. As especificações técnicas dos computadores utilizados podem ser encontradas do Apêndice 'A'.

As tabelas II e III mostram a média temporal de 10 testes realizados entre as diferentes formas de processamento.

Tabela II: Medidas de desempenho das diferentes arquiteturas computacionais.

Média aritmética aproximada em segundos		
Cluster	*SMP*	Procedimental
11	16	30

Listagem 3: Código para somar os índices do vetor em paralelo

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #
4
5  import random, time, sys
6  import pp
7
8  '''
9  A função constrói um vetor onde os valores dos 2
       primeiros índices são obtidos aleatoriamente e
        depois esses valores são somados e
       armazenados no terceiro índice
10 '''
11 def construirvetor(start,end):
12     vet = []
13     i = 0
14     for x in xrange(start,end):
15         vet.append([random.randint(0,100),random.
               randint(0,100),None])
16         vet[i][2]=vet[i][0]+vet[i][1]
17         i+=1
18     return vet
19
20 def main():
21     v1 = []
22     jobs = []
23     #TUPLA DE SERVIDORES
24     ppservers = ("node1","node2","node3")
25
26     if len(sys.argv) > 1:
27         ncpus = int(sys.argv[1])
28         job_server = pp.Server(ncpus, ppservers=
               ppservers)
29     else:
30         job_server = pp.Server(ppservers=ppservers
               )
31
32     start = 1
33     end = 2000000
34
35     #DIVISÃO DO PROBLEMA
36     parts = 100
37     step = (end - start) / parts + 1
38
39     #GERANDO TAREFAS COM DIFERENTES PARAMETROS
40     for x in xrange(parts):
41         starti = start+x*step
42         endi = min(start+(x+1)*step,end)
43         jobs.append(job_server.submit(
               construirvetor,(starti,endi),(),("
               random",)))
44
45     #TRAZENDO OS RESULTADOS DAS TAREFAS ENVIADAS
46     for x in jobs:
47         v1.append(x())
48
49     job_server.print_stats()
50     return 0
51
52 if __name__ == '__main__': main()
```

O objetivo dos testes referentes à tabela III é produzir um caso onde o desempenho do processamento paralelo mostra ser inferior se comparado ao processamento procedimental, com base nos fatores vistos na seção 2.1.

A linha que se refere à divisão das tarefas (lin. 36) foi modificada, ficando assim:

```
parts = 20000
```

Com essa modificação, o número de partes aumentou e, em consequência, houve um custo maior de pro-

cessamento para gerenciar as informações.

Tabela III: Medidas de desempenho com comunicações excessivas

Média aritmética aproximada em segundos	
Cluster	Procedimental
74	29

O *node master* do *cluster* não participou do processamento paralelo, isto foi possível passando o valor 0 para *argv[1]* no código referente à listagem 3.

Para realizar os testes sobre a arquitetura *SMP* e procedimental, bastou modificar a tupla[5] que contém os identificadores dos computadores envolvidos (lin. 24), ficando assim:

```
ppservers = ()
```

Para a reprodução da execução procedimental, foi passado como parâmetro para *argv[1]* o valor 1 (lin. 27), referindo-se ao número de processadores locais.

Os testes referentes à execução *SMP* e procedimental, foram realizados sobre um dos *nodes* do *cluster* utilizado.

4. CONCLUSÕES

O desenvolvimento de aplicações com o uso do módulo pp apresentou um método simples, como visto nas seções 2.4 e 3. O uso deste módulo pode reduzir o tempo de desenvolvimento do programa de forma significativa. O módulo consegue estabelecer as devidas comunicações para a realização de processamento paralelo de forma não explícita.

O programador não tem o trabalho lógico em estabelecer e sincronizar a comunicação entre os diferentes processadores, o programador passa a se preocupar apenas no problema a ser paralelizado, ou seja, no algoritmo. Como visto no capítulo 3, o aumento do número de processadores em computadores locais ou distribuídos por meio de uma rede, pode proporcionar um maior desempenho na execução de tarefas, tornando possível a execução das mesmas quando esta seria impraticável se executada de forma procedimental.

No processamento paralelo, existem fatores que podem influenciar diretamente no desempenho da aplicação, como vistos nas seções 2.1 e 3. De acordo com a tabela III, uma divisão não adequada das tarefas resultou em comunicações excessivas entre os diferentes processadores e, como resultado desse custo computacional, o processamento paralelo mostrou um tempo gasto superior ao processamento procedimental.

O processamento paralelo não é aplicável a todos os problemas computacionais, o problema deve ser analisado de modo que possa fazer uma escolha da melhor técnica de programar, seja ela procedimental ou paralela.

5. APÊNDICE 'A'

5.1. Especificações Técnicas

5.1.1. Cluster

Especificações técnicas do cluster utilizado nos testes referentes ao capítulo 3. O cluster é composto por um *node* cliente (ou *node* master) e três *nodes* servidores (ou *nodes* escravos).

Cliente (*Master*)

- AMD Athlon 64 3800+ Processor with 512KB cache L2.
- 2GB DDR 400MHz (2x1GB).
- 600Watt power supply.
- 8MB DDR SDRAM AGP Graphics Adapter.
- 52X CDROM IDE Drive.
- 1x80GB IDE Ultra ATA 7200rpm.
- 2x160GB SATA II 7200rpm.
- Motherboard ASUS A8N-Premium, 8 controladoras SATAII (3Gbps).
- 2 Gigabit Ethernet Adapter (Marvell, NVidia) on board.
- PS/2 Keyboard, PS/2 Mouse.

Servidor (*Nodes*)

- AMD Opteron Duo Core, 2200MHz, 1MB cache L2 cada.
- 2GB DDR 400v(2x1GB).
- 450Watt power supply.
- 80GB SATAII 7.200RPM.
- 2 Gigabit Ethernet Adapter (BroadCom, NVidia) on board.

[5] Tupla é um tipo de estruturas de dados da linguagem de programação *Python*.

[1] BARNEY, Blaise. **Introduction to Parallel Computing**. LAWRENCE LIVERMORE NATIONAL LABORATORY, disponível em <https://computing.llnl.gov/tutorials/parallel_comp/>, acesso em 13 abr. 2011.

[2] Disponível em <http://www.python.org/>, acesso em 13 abr. 2011.

[3] Disponível em <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html >, acesso em 13 abr. 2011.

[4] Disponível em <http://www.python.org.br>, acesso em 13 abr. 2011

[5] CALLAHAN, David. **Mudança de Paradigma**: considerações sobre design de programação paralela. MSDN Magazine, disponível em <http://msdn.microsoft.com/pt-br/magazine/cc872852.aspx>, acesso em 31 maio 2011.

[6] STALLINGS, William. **Arquitetura e organização de computadores**. 8.ed. Prentice Hall, 2010.

[7] **BEOWULF**. Disponível em <http://www.beowulf.org/>. Acesso em 13 abr. 2011.

[8] Flynn, M. **Some Computer Organizations and Their Effectiveness**. IEEE Trans. Comput. C-21: 948, 1972.

[9] BARNEY, Blaise. **Message Passing Interface (MPI)**. LAWRENCE LIVERMORE NATIONAL LABORATORY, disponível em <https://computing.llnl.gov/tutorials/mpi/>, acesso em 13 abr. 2011.

[10] Disponível em <http://www.parallelpython.com/>, acesso em 13 abr. 2011.

[11] Disponível em <http://www.vitalii.com/>, acesso em 13 abr. 2011.