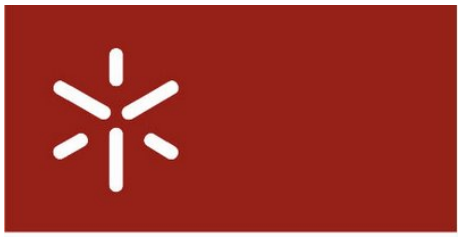
Universidade do Minho

Departamento de Informática

Paradigmas da Computação Paralela

Otimização OpenMP



Universidade do Minho

António Sérgio Alves Costa A78296
José Pedro Moreira Resende A77486

14 de Janeiro de 2019

Conteúdo

1 Introdução **2**

2 Descrição do Algoritmo **2**
2.1 Representação dos Pontos da Imagem 2
2.2 Análise dos Pontos 2
2.3 Desafios na Paralelização com OpenMP 3

3 Implementação **4**

4 Demonstração e Análise dos Resultados **5**
4.1 Apresentação dos Resultados 5
4.1.1 Apresentação dos Resultados de SpeedUp 5
4.1.2 Apresentação dos Resultados dos tempos de comunicação 5
4.2 Análise dos Resultados 6

5 Conclusão **6**

6 Anexos **7**

7 L3 Architecture **7**
7.1 Tabelas 7
7.2 Gráficos 9
7.3 Script 9
7.3.1 testing.py 9
7.3.2 notify.py 10
7.3.3 env.py 11

1 Introdução

A computação paralela permite distribuir as instruções executadas por um computador, mas isto apenas pode ser feito, corretamente, se não houver dependência entre essas instruções. Assim sendo, quanto mais independência, mais partido podemos tirar da computação paralela e dos múltiplos processadores presentes em qualquer máquina hoje em dia.

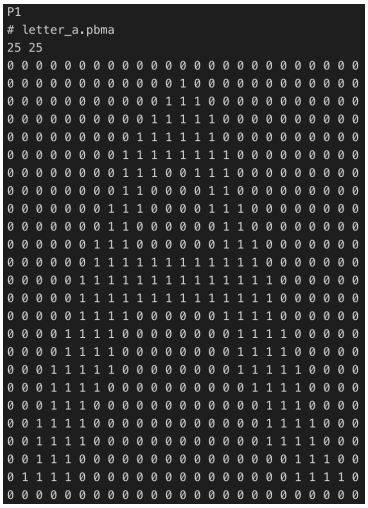
O tema deste relatório, sugerido pelo docente desta Unidade Curricular, será a otimização da nossa implementação com blocking da primeira fase através do uso de blocos de tamanho variável para constatar qual será o melhor tamanho.

Modificar o tamanho dos blocos irá interferir, não só na manipulação da memória, como no trabalho efetuado por cada bloco.

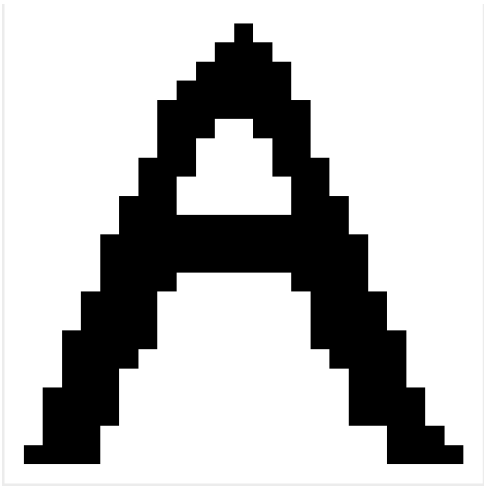
2 Descrição do Algoritmo

2.1 Representação dos Pontos da Imagem

As imagens processadas têm o seguinte formato:



(a) Formato da imagem



(b) Imagem equivalente

Como previsto, a imagem foi processada para uma matriz, no entanto esta matriz é representada por um array, por motivos apresentados mais à frente no relatório.

2.2 Análise dos Pontos

Após processar a imagem, terá de se analisar todos os pontos para ver se este é ou não removido. Para tal, como usamos por base o Zhang-Suen Thinning Algorithm, vamos ter de analisar todos os pontos à volta do ponto que está a ser analisado de momento.

P9	P2	P3
P8	P1	P4
P7	P6	P5

Figura 2: Enumeração dos pontos envolventes do ponto analisado(P1)

De seguida, temos de aplicar algumas condições para saber se devemos ou não remover o ponto analisado (P1). As condições são as seguintes:

- **1ª Passagem - Remover P_1 se**

  i) $2 <= N(P_1) <= 6$, $N(P_1)$ - número de vizinhos a '1'

  ii) $S(P_1) = 1$, $S(P_1)$ - nº de transições 0-1 na seq. $P_2, P_3, ..., P_9$

  iii) $\overline{P_4} + \overline{P_6} + \overline{P_2}\,\overline{P_8} = 1$

- **2ª Passagem - Remover P_1 se**

  Substituir iii) por

  iv) $\overline{P_2} + \overline{P_8} + \overline{P_4}\,\overline{P_6} = 1$

Figura 3: Condições para remover o ponto (P1)

Este processo é repetido até que nenhum dos pontos seja removido em ambas as passagens. No final, o resultado deverá ser algo deste género.

(a) Imagem Original

(b) Esqueleto da imagem

2.3 Desafios na Paralelização com OpenMP

Quando trabalhamos com memória partilhada, diferentes threads podem tentar aceder ao mesmos dados simultaneamente, originando conflitos, que geram resultados inconsistentes. Portanto, é importante alterar o algoritmo efetuando uma gestão nos acessos aos dados.

O processador ao requerer acesso a uma posição de memória irá copiar um bloco inteiro para a cache com o propósito de tirar partido da localidade espacial nos acessos, reduzindo assim, em princípio, o número de vezes que é necessário ir buscar dados à memória. Em problemas resolvidos com recurso a paralelismo podemos ter duas threads a tentar escrever em valores que se encontram na mesma linha de cache invalidando toda a linha de cache, obrigando uma das threads a pedir os dados à memória novamente, ou seja, **False Sharing**.

Para além destes problemas, a paralelização é bastante dificultada, devido ao algoritmo em si. Neste algoritmo não podemos paralelizar as passagens pela matriz, visto que cada passagem depende da passagem anterior, nem sequer podemos paralelizar as fases, já que a segunda fase depende da primeira fase. Sendo assim, percebemos que a paralelização apenas poderá ser feita dentro de cada passagem e dentro de cada fase.

3 Implementação

Esta implementação tem em conta os acessos a memória deste algoritmo. Este algoritmo acede a vários endereços de memória, já que precisamos de verificar condições com os pontos envolventes. Tendo isto em conta, esta implementação divide a matriz em blocos mais pequenos e percorre estes blocos em paralelo por cada fase da iteração, como se pode ver pela seguinte imagem:

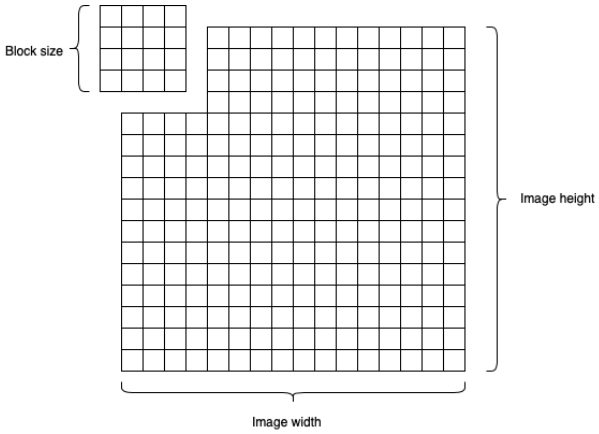


Figura 5: Algoritmo de blocking

Para além disto, garante-se que todos os pontos analisados (inclusive os envolventes) estão dentro do bloco que está a ser processado. Isto permite uma diminuição de CM (Cache Misses), visto que garantimos que todos os dados precisos, como demonstra a seguinte imagem:

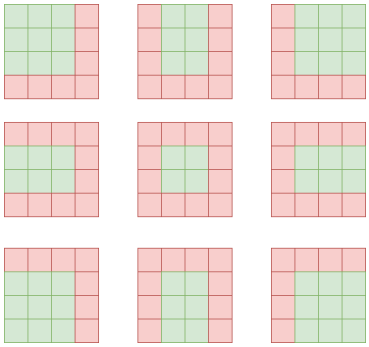


Figura 6: Dados processados e não processados

■ - pontos não processados, apenas para efetuar o algoritmo
■ - pontos processados
É importante referir que, caso a divisão do block size pela height ou pela width não seja inteira, o resto das colunas ou linhas ficará nos blocos que ficam no extremo horizontal ou no extremo vertical.

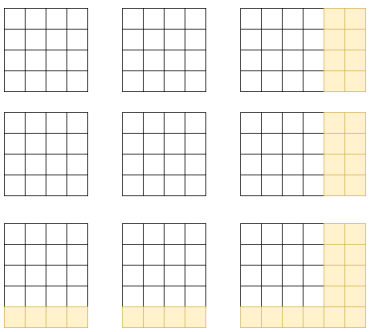


Figura 7: Divisão não inteira

4 Demonstração e Análise dos Resultados

Concluída a implementação, é necessário realizar vários testes para perceber a performance do algoritmo para os diferentes tamanhos de blocos, e tentar comprovar o porquê dele se comportar de certa maneira.

Primeiramente, iremos apresentar gráficos com resultados (as tabelas relativas a estes gráficos estarão nos anexos finais) e, de seguida, iremos discutir os resultados presentes nos gráficos.

4.1 Apresentação dos Resultados

4.1.1 Apresentação dos Resultados de SpeedUp

Devido ao elevado range dos valores, a google sheets não nos deixou retirar a escala logarítmica do eixo horizontal.

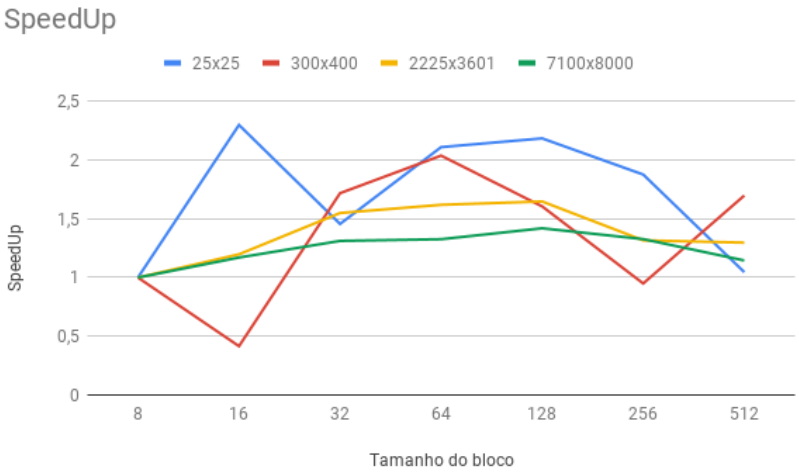


Figura 8: SpeedUp em função do tamanho dos blocos

4.1.2 Apresentação dos Resultados dos tempos de comunicação

A seguinte tabela representa as L3 Misses para a imagem 300x400, ou seja, os acessos à RAM para essa imagem. As restantes tabelas estão em anexo.

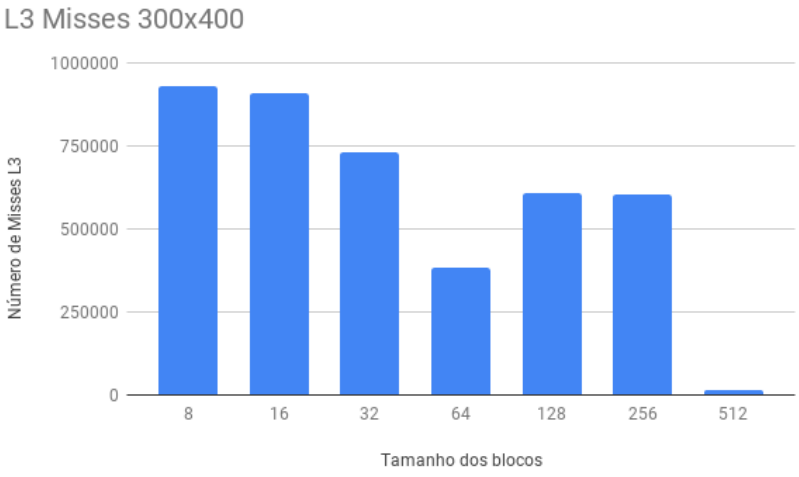


Figura 9: Tempo necessário para comunicação

4.2 Análise dos Resultados

Como podemos visualizar no gráfico do *SpeedUp*, todos os processos têm um speedup bastante diferente. Isto deve-se ao diferente tamanho dos blocos e o diferente tamanho das imagens, fazendo com que para imagens mais pequenas, blocos mais pequenos apresentam os melhores resultados. Também devemos ter em atenção ao facto da L3 ser partilhada por 8 cores em cada processador(Arquitetura nos anexos).

- 25x25 - o maior *SpeedUp* para esta imagem verifica-se para blocos 16x16, isto deve-se a uma melhor distribuição do trabalho e maior paralelismo, já que para blocos maiores apenas teriamos um bloco e enquanto 8 por 8, é demasiado paralelismo para o trabalho de cada bloco. Todos os blocos têm L3 Misses muito parecidas, visto que todos os blocos cabem em cache.

- 305x400 - o maior *SpeedUp* para esta imagem verifica-se para blocos 64x64, visto que este tamanho de blocos não só permite um baixo custo em termos de parelelismo já que não temos demasiados blocos, como também permite um nível de carga de trabalho considerável por bloco. Para além disto, o número de L3 Misses é bastante pequeno a relação há maioria dos outros blocos. Neste caso, há alguma variância nas L3 Misses, no entanto, não conseguimos repetir os testes porque o cluster está com problemas. Por exemplo, os blocos 512x512 têm um número extramamente pequeno de L3 Misses em relação aos restantes tamanhos. A razão que encontramos para justificar tal acontecimento, é apenas haver um bloco e este é colocado totalmente em cache e apenas um core vai processá-lo, nos outros casos, como a L3 é partilhada por 8 cores, haveria bastantes conflitos.

- 2225x3601 - o maior *SpeedUp* para esta imagem verifica-se para blocos 128x128, visto que este tamanho de blocos não só permite um equilibrio entre parelelismo e distribuição de carga de trabalho por cada bloco. Como esta imagem já só cabe em L3, o mais importante será tirar partido de cada bloco para fazer um trabalho eficiente, e conseguir um paralelismo que permita ter vários blocos em cache. Como podemos constatar, o número de L3 Misses para este tamanho é o mais baixo de todos.

- 7100 x 8000 - o maior *SpeedUp* para esta imagem verifica-se para blocos 128x128, visto que este tamanho de blocos não só permite um equilibrio entre parelelismo e distribuição de carga de trabalho por cada bloco. Como esta imagem já só cabe em RAM, o mais importante será encontrar um tamanho que ofereça paralelismo, e ao mesmo tempo, permita tem vários blocos em cache. Como podemos ver, o número de L3 Misses para este tamanho é dos mais baixos de todos.

5 Conclusão

Após terminar a otimização da primeira fase, através do teste do algoritmo blocking para vários tamanhos, o grupo conclui que o tamanho do bloco é variável para o tamanho da imagem a processar e o tipo de imagem.

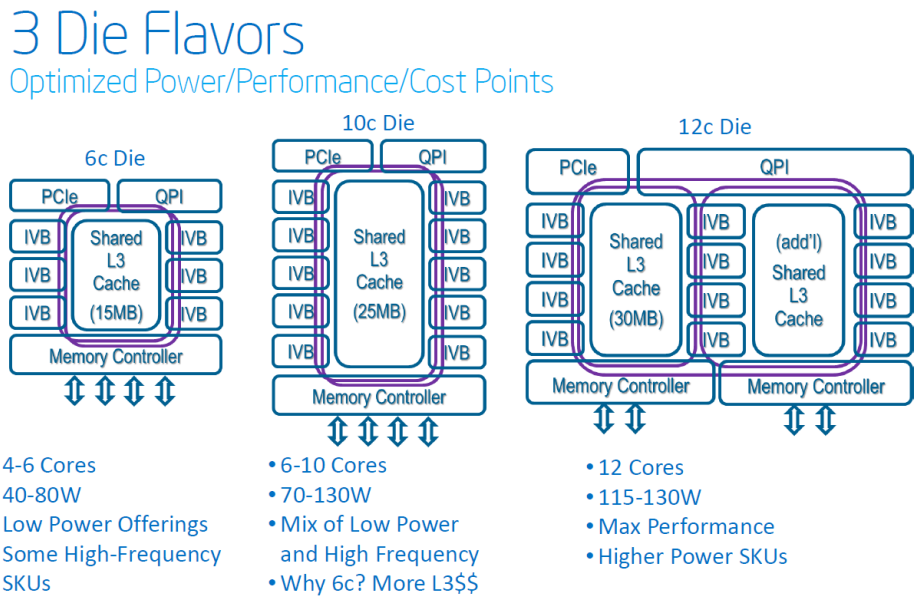
Para as imagens mais pequenas, conclui-se que blocos de tamanhos mais pequenos são favoráveis, no entanto não devemos sobrecarregar o sistema com paralelismo, visto que isto apenas irá piorar os resultados devido ao overhead,.

Para as imagens maiores, temos de ter uma maior consideração pelos acessoa à RAM e, normalmente, tamanhos de bloco maiores são melhores, já que permitem um equilíbrio entere o paralelismo e a carga de trabalho que cada bloco irá ter.

Em suma, não há um tamanho indicado para os blocos, este tamanho irá variar e só perceberemos qual será o melhor, após efetuarmos testes com vários tamanhos.

6 Anexos

7 L3 Architecture



7.1 Tabelas

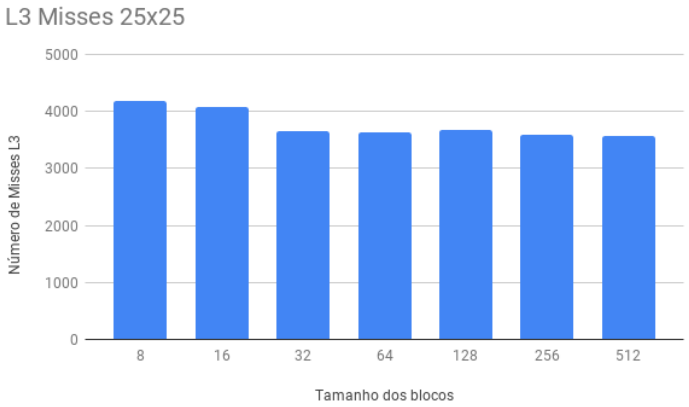
PCP - MPI
Arquitetura: Ivy Bridge (Nó 641)
24 cores físicos
48 cores lógicos
gcc 5.3.0
gnu openmpi_eth 1.8.4
Unidades de tempo: microsegundos

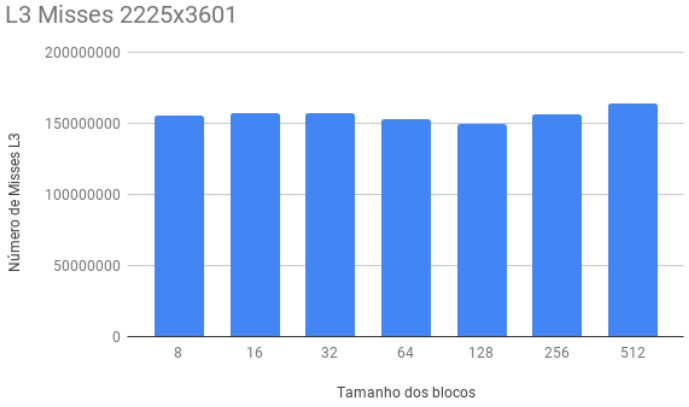
Nível de Memória	Tamanho
L1	32 KB
L2	256 KB
L3	30 MB

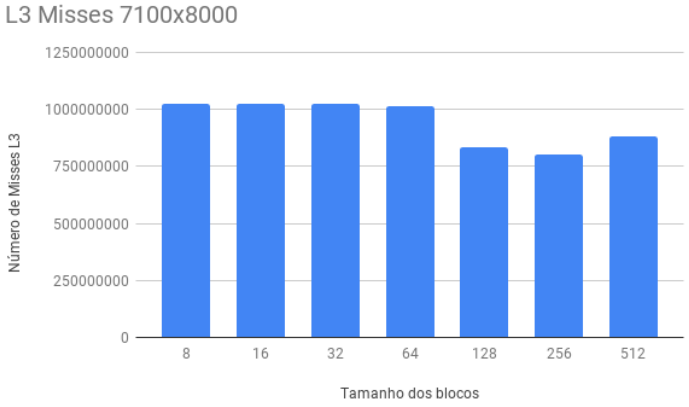
Salientar desde já, que utilizamos um script para obeter os resultados. O script repetia 8 vezes o programa para cada número de processos e para cada mensagem, e efetuava a mediana dos resultados automaticamente.

	Tamanho do bloc	Time	MR L1	MR L2	MR L3	Access Ram	Flops	SpeedUp
letter_a.ascii.pbn	8	64324,33333	4.04649395407e	2049611752.67	313420939.333	4189,666667	31706.6666667	1
	16	27973,33333	203695320254.0	2036411250.67	313747770.667	4076,666667	29338.6666667	2,299487607
	32	44160	3.31621132014e	1976344003.0	301283848.667	3651	42510.3333333	1,456619867
	64	30477	2.08482465336e	2003344474.33	306057519.333	3638	60571.0	2,110586125
	128	29437,33333	2.83950688704e	1981011042.33	308040474.667	3670,666667	34412.0	2,185127729
	256	34265,33333	2.37932678663e	2004344467.33	307550639.667	3579,333333	5236.66666667	1,877242305
	512	61478	1.96670171998e	1985277097.67	309517410.333	3572,333333	6519.0	1,046298405
						Access Ram		SpeedUp
washington.ascii.	8	280470,6667	1.80152424351e	3.33095732607e	4.2503192161e+	927786,6667	134002.333333	1
	16	674603,6667	3.78544680656e	3.16428698753e	4.08096552499e	909430	268486.666667	0,4157562144
	32	163218	1.36556096334e	3.23525464817e	3.87172342174e	730194,6667	359266.666667	1,718380734
	64	137593,6667	1.16318309029e	3.53416165249e	3.93369822531e	382733,6667	76866.6666667	2,038398085
	128	174639	1.14714953741e	4.0220860009e+	4.00325262154e	609820	162528.0	1,606002477
	256	295353	1.64700033162e	4.9132550022e+	438736080859.0	602773,6667	44615.0	0,9496117076
	512	165133	2.18874335976e	6.42390267048e	510609279951.0	12856	944.666666667	1,698453166

						Access Ram		SpeedUp
seahorse.ascii.pt	8	25403426	1.36687959693e	6.11428803078e	4.39403005492e	155201327,7	9449962.33333	1
	16	21203982,33	9.71142477257e	5.70399338075e	4.43507266827e	157214693	6107046.0	1,198049763
	32	16392651	9.12358640222e	5.45097386078e	4.41015086823e	156932322,7	4934107.33333	1,549683819
	64	15686044,33	8.49675333588e	5.4159421775e+	4.43446175159e	153156982	4329560.66667	1,619492172
	128	15414671,67	8.06897392955e	5.33290013407e	4.42009488485e	149116226,3	3280665.33333	1,648003055
	256	19290767,33	9.63857275056e	5.54563178071e	4.56273755491e	156023162,7	6125729.33333	1,316869649
	512	19580649,33	1.23362555153e	5.87026788417e	4.63848776496e	163949494,7	6081316.66667	1,297374033
						Access Ram		SpeedUp
dog.pbm	8	320911041,3	1.60574052604e	9.63511775968e	3.0419845331e+	1024706313	103641979.667	1
	16	274141513,7	1.26806963974e	8.70847818665e	3.04808245644e	1023456944	68101742.0	1,170603595
	32	244571117	1.12323345316e	8.30723937298e	3.0852839471e+	1026979015	56689998.0	1,312137939
	64	241902281,3	1.08419419543e	8.15427371132e	3.12303237977e	1013583400	51871411.0	1,326614365
	128	226.062.005	1.03593558061e	8.16930175497e	2.96824237917e	834551442	49850818.3333	1,419570887
	256	241770129,7	1.16109771931e	8.13206277528e	3.0083174638e+	799555079,3	80490982.6667	1,327339493
	512	280.131.097	1.43662268526e	8.62362588861e	3.15894380156e	879013504	106910644.667	1,145574501







7.2 Gráficos

7.3 Script

7.3.1 testing.py

```
import time
import subprocess
import datetime
from notify import send_mail_notify


def run_command(cmd):
    result = subprocess.Popen(cmd, stdout=subprocess.PIPE,
        stderr=subprocess.PIPE, stdin=subprocess.PIPE, shell=True)
    result.wait()
    try:
        return result.stdout.read().decode("ascii").strip()
    except Exception as e:
        print e
        return None


def run_program(program):
    while True:
        cmd = 'ps -eo user=|sort|uniq -c | grep -P "a[0-9]{5}" | wc -l'
        number = run_command(cmd)
        if number == '1':
            result = run_command(program)
            if result is not None:
                return result
            else:
                print "ERROR"
                return None
        else:
            time.sleep(1)


def k_best(k, values):
    error = (1, -1)
    values.sort()
    for i in range(len(values)-k):
        maximum = values[i+k-1]
        minimum = values[i]
        e = (maximum - minimum) / float(maximum)
        if e < 0.05:
            return sum(values[i:i+k]) / float(k)
        if e < error[0]:
            error = (e, i)
    if error[1] != -1:
        return sum(values[error[1]:error[1]+k]) / float(k)
    return -1


def run_func(table, matrix, measures, nreps, k, func):
    outf= "images/out_" + datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S") + func
    for m in matrix:
        print m
        table.write(","+m)
```

```
        for ms in measures:
            print ms
            tmp = []
            for r in range(nreps):
                out = run_command(' '.join(["mpirun -np",m,"-mca btl",
                "self,sm,tcp bin/skeleton_mpi","images/"+func,outf]))
                if out is not None:
                    print(out)
                    tmp.append(float(out))
                else:
                    print "Error 1"

            try:
                table.write("," + str(k_best(k, tmp)))
            except:
                table.write(",")
                print "Error 2"

        table.write("\n")


def run_tests(funcs, matrix, measures, nreps, k):
    fname = datetime.datetime.now().strftime("%Y-%m-%d_%H:%M") + ".csv"
    table = open(fname, "w")
    table.write(",,Time\n")

    for func in funcs:
        print func
        table.write(func)
        run_func(table, matrix, measures, nreps, k, func)
        table.write("\n")
    table.close()


if __name__ == '__main__':
    images = ["washington.ascii.pbm"]
    #images = ["letter_a.ascii.pbm"]
    numero_processos = ["2", "4", "8", "16", "32","48"]
    measures = ["time"]
    nreps = 8
    k = 3
    run_tests(images, numero_processos, measures, nreps, k)
    send_mail_notify()
```

7.3.2 notify.py

```
import urllib2 as rr
from env import url,phone,url_mail,mail

def send_phone_notify():
    r = rr.urlopen(url+phone).read()

def send_mail_notify():
    r = rr.urlopen(url_mail+mail).read()
```

7.3.3 env.py

```
url = "https://spneauth.herokuapp.com/msg/"
url_mail = "https://spneauth.herokuapp.com/mail/"
phone = "963003977"
mail = "asergioalvesc@gmail.com"
```