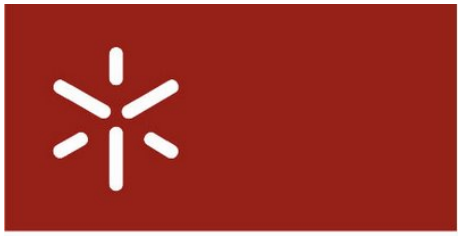
Universidade do Minho

Departamento de Informática

Paradigmas da Computação Paralela

MPI



Universidade do Minho

António Sérgio Alves Costa A78296
José Pedro Moreira Resende A77486

4 de Janeiro de 2019

Conteúdo

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.

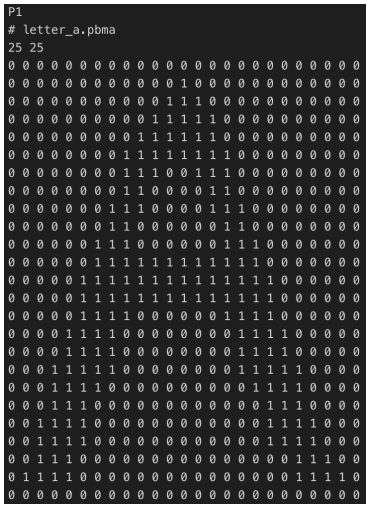
O tema deste relatório será a paralelização através do **Message Passing Interface (MPI)** de um algoritmo skeleton. Este algoritmo de processamento de imagens binárias transforma a imagem que recebe no seu respetivo esqueleto.

Tendo em conta os conceitos do algoritmo base, terá de se aceder duas vezes, por cada iteração, a todos os pontos à volta do ponto a analisar. Este ponto, após a análise, será apagado ou mantido. Pode-se perceber que este algoritmo tem algumas dependências, logo a implementação do *MPI* terá de ser cuidadosa.

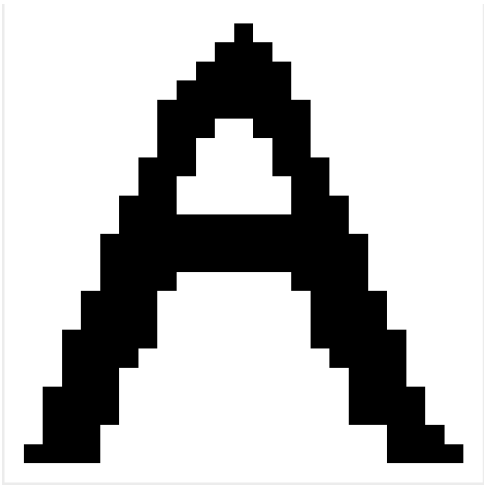
2 Descrição do Algoritmo

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 MPI

Quando trabalhamos com o **Message Passing Interface**, diferentes processos vão fazer partes diferentes do algoritmo. Isto dificulta muito a execução pois, como explicamos anteriormente, este algoritmo tem duas fases distintas que dependem uma da outra, logo todos estes passos terão de ser levados em conta na implementação.

Mais ainda, como podemos ver pela descrição do algoritmo, para decidir se um ponto deve ser ou não mantido teremos de analisar os seus pontos envolventes. Assim sendo, se repartimos o trabalho de análise da matriz por todos os processos, percebemos que vão haver muitas partes críticas, visto que teremos de trocar linhas ou colunas, ou ambas, dependendo do algorimto que decidimos implementar. Esta comunicação terá de ser minuciosamente controlada para que o algoritmo descrito não seja erradamente afetado.

3 Implementação

Na implementação desta fase, decidimos utilizar a *Implementação 1* realizada na fase anterior, no entanto dividimos a matriz em blocos verticais, e cada um desses blocos será processado por um processo.

Dividimos os blocos da seguinte forma:

$$h = \frac{height}{size}$$

Em que size representa o número de processos, e a height o número de linhas da matriz.

O h irá servir para saber quantas linhas o processo irá processar da matriz. Sendo que o último processo irá ficar com a sobra de linhas, caso a divisão tenha resto.

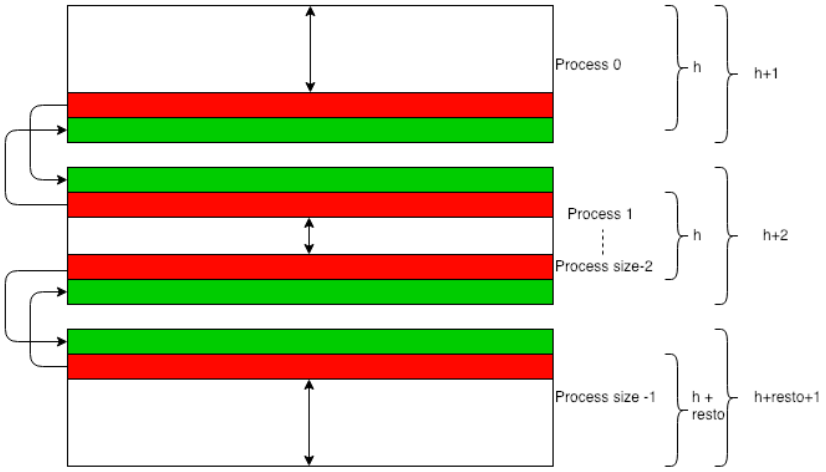


Figura 5: Algoritmo de comunicação

■ - linhas processadas e enviadas a um processo vizinho
■ - linhas não processadas e recebidas por um processo vizinho

Todavia, como dissemos anteriormente para processarem h linhas, o processo terá de ter mais linhas, dependendo da posição do processo.

- **Processo 0** irá processar h linhas, mas precisa de ter h+1 linhas, sendo que a última apenas serve para conseguirmos analisar os pontos envolventes e é recebida pelo processo 1.
- **Processo 1.. Processo size-2** irá processar h linhas, mas precisa de ter h+2 linhas, sendo que a última e a primeira apenas servem para conseguirmos analisar os pontos envolventes e é recebida pelo processo anterior e posterior.
- **Processo size-1** irá processar h linhas, mas precisa de ter h+1 linhas, sendo que a primeira apenas serve para conseguimos para podermos analisar os pontos envolventes e é recebida pelo processo size-2.

É importante referir que nenhum dos processos pode parar quando não modifica nada, visto que a execução dos processos vizinhos poderá ter influência e futuramente poderá ter de modificar a sua matriz, logo todos os processos só param quando nenhum deles alterar nada em todas as fases. Assim sendo, tivemos de implementar um MPI_REDUCE para descobrir quando o processamento da imagem terminava.

Para além disto, é importante referir que no final temos de juntar todos os resultados para imprimir e obter um resultado final. Nesta porção do programa, o Processo 0 irá receber todas as matrizes finais e juntá-las, para que no final fosse possível dar print à imagem total.

4 Demonstração e Análise dos Resultados

Concluída a implementação, é necessário realizar vários testes para perceber a performance do algoritmo e se este se comporta como é previsto, e tentar comprovar o porquê dele se comportar de certa maneira.

Primeiramente, iremos apresentar gráficos com resultar (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

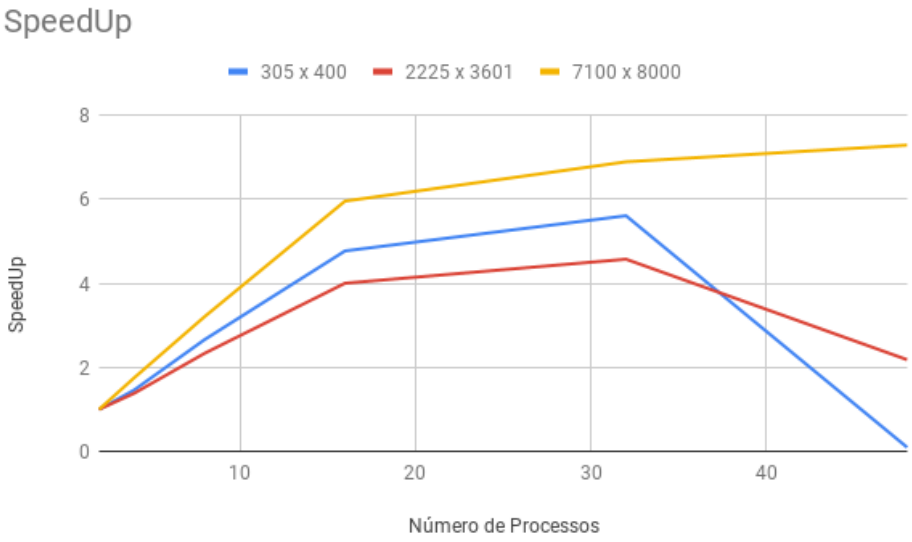


Figura 6: SpeedUp do algoritmo

4.1.2 Apresentação dos Resultados dos tempos de comunicação

A seguinte tabela representa o tempo dispendido em comunicação ao executar apenas uma passagem pela matriz (2 fases), mas sem realizar os cálculos. Apenas são realizadas as comunicações.

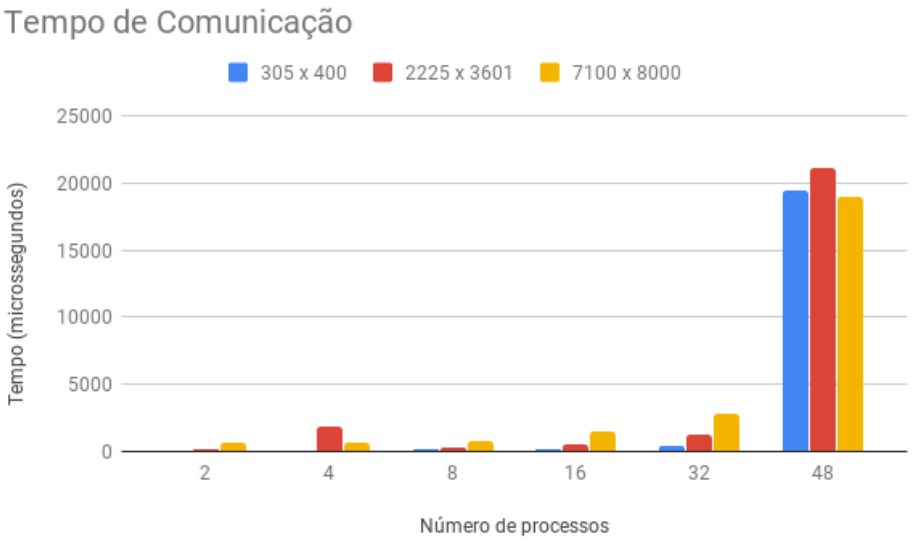


Figura 7: Tempo necessário para comunicação

4.2 Análise dos Resultados

Como podemos visualizar no gráfico do *SpeedUp*, todos os processos têm um speedup significativo até aos 32 processos. Isto deve-se à distribuição de trabalho pelos processos e à paralelização do proces-

samento da matriz. No entanto, quando passamos para 48 processos nota-se uma grande diferença de comportamentos.

- 305x400 - com 48 processos o speedup para esta matriz diminui drasticamente, demorando mais do que com 2 processos. Isto deve-se essencialmente ao facto do trabalho realizado já não ser significante, pois como podemos calcular o h é 6 para 48 processos, e para 32 processos é 9, logo o não há uma grande melhoria. No entanto, há um aumento drástico do custo de comunicação com 48 processos. O aumento do custo de comunicação é 43 vezes maior com 48 processos, sendo assim, percebemos que o overhead das comunicações para este número de processos não compensa.

- 225x3601 - com 48 processos o speedup para esta matriz diminui bastante, no entanto, contnua a ser melhor do que com 2 ou 4 processos. Isto deve-se às mesmas razões, as comunicações para 48 processos demoram bastante, logo toda a paralelização acaba por não compensar este tempo dispendido na comunicação.

- 7100 x 8000 - com 48 processos o speedup para esta matriz aumenta ao contrário do que aconteceu com as outras matrizes. Isto deve-se essencialmente ao facto do trabalho realizado ainda ser significante para a paralelização compensar, mesmo com os custos de comunicação. O h, neste caso, é 147, utilizar 48 processos paralelos acaba por compensar, já que diminui bastante o tempo de análise.

5 Conclusão

Após terminar a implementação, o grupo pensa que todo o algoritmo foi concluído com sucesso, uma vez que o resultado do processamento de imagens foi o esperado, e os resultados foram concisos. Este projeto serviu também para aprofundar todo o conhecimento sobre os temas lecionados na Unidade Curricular e perceber a utilidade dos mecanismo e técnicas aprendidas ao longo do semestre sobre o **Message Passing Interface (MPI)**. No entanto, também nos apercebemos de todos os problemas e dificuldades na implementação dessas técnicas e mecanismos.

Primeiramente, foi bastante difícil arranjar um algoritmo onde a quantidade de comunicações fosse e mínima e, ao mesmo tempo, distribuir a carga de trabalho aproximadamente igualmente pelos processos. Decidimos dividir então verticalmente, pois apenas precisavamos de trocar as linhas. Isto ajudou, não só bastante na diminuição do número de comunicações, como na execução do algoritmo, visto que os bits duma linha estão seguidos no array da matriz.

De seguida, tivemos dificuldade a escrever o código do mesmo, já que havia alguns *deadlocks* caso os **Send** e **Receive** fossem implementados na mesma hora. Logo definimos que uma ordem que impedisse esses *deadlocks* e assegurasse que o algoritmo só era executado após as comunicações.

O último grande problema da implementação, foi a escrita da matriz no ficheiro de output, porque o *Send* e *Receive* tinham limites de dados que podiam ser enviados, e tivemos de descobrir uma forma de ultrapassar este pequeno problema.

Por último, implementamos um script que nos permitia fazer várias execuções do programa para vários números de processos, e repetia isto 8 vezes para que no final pudessemos fazer a mediana. Este script demorou algum tempo para ser feito, mas no final poupou-nos bastante tempo.

6 Anexos

6.1 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.

7100 x 8000	Processes	Time	SpeedUp
dog.pbm	2	992154017.7	1
	4	566021608.7	1.752855372
	8	309381840.3	3.206891576
	16	166435042.7	5.96120866
	32	143827683.3	6.898213158
	48	136002695	7.295105569

305 x 400	Processes	Time	SpeedUp
washington.asc	2	240897	1
	4	164738.6667	1.462297862
	8	90561	2.66005234
	16	50455.33333	4.774460579
	32	42919	5.612828817
	48	2506055	0.09612598287

2225 x 3601	Processes	Time	SpeedUp
seahorse.ascii.p	2	36293773	1
	4	26230264	1.383660225
	8	15551229	2.33382024
	16	9061943	4.005076284
	32	7928309	4.57774451
	48	16618283.33	2.183966435

305 x 400	Processes	Time
washington.asc	2	34.33333333
	4	71.33333333
	8	133.6666667
	16	213.3333333
	32	453.6666667
	48	19500

2225 x 3601	Processes	Time
seahorse.ascii.p	2	125
	4	1838.666667
	8	351
	16	580.3333333
	32	1293.333333
	48	21177.33333

7100 x 8000	Processes	Time
dog.pbm	2	695.6666667
	4	630.3333333
	8	745.6666667
	16	1474.333333
	32	2842.333333
	48	18968.33333

6.2 Script

6.2.1 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()
```

6.2.2 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()
```

6.2.3 env.py

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