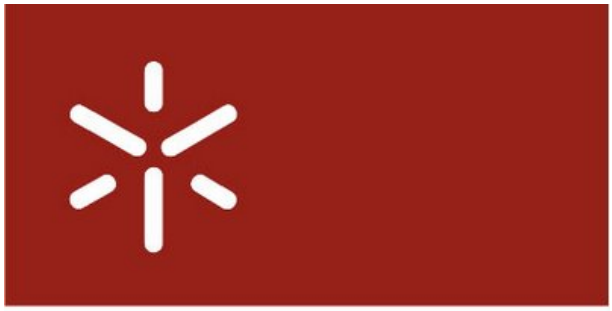
Mestrado Integrado em Engenharia Informática

Paradigmas da Computação Paralela

OpenOMP



Universidade do Minho

António Sérgio Alves Costa A78296
José Pedro Moreira Resende A77486

21 de Novembro de 2018

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 de um algoritmo skeleton. Este algoritmo de processamento de imagens binárias transforma a imagem que recebe no seu respetivo esqueleto. Apesar de haverem vários tipos de algoritmos skeleton, desenvolvemos um com base no Zhang-Suen Thinning Algorithm.

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, no entanto, pode ser paralelizado.

2 Descrição do Algoritmo

2.1 Representação dos Pontos da Imagem

As imagens processadas têm o seguinte formato:

```
P1
# feep.ascii.pbm
24 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

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 1: 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:

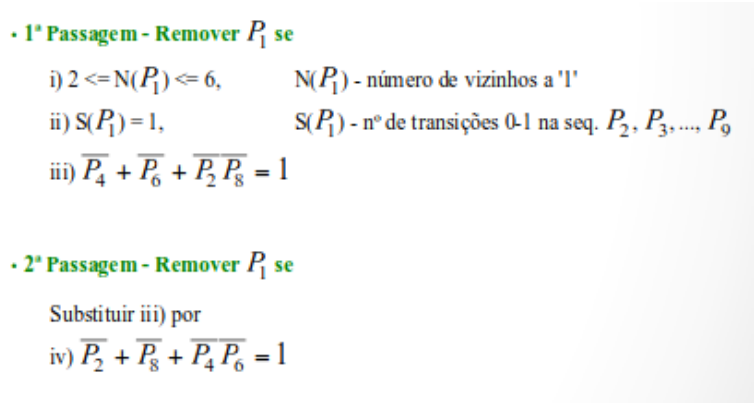


Figura 2: Condições para remover o ponto (P1)

Este processo é repetido até que nenhum dos pontos seja removido em ambas as passagens.

2.3 Desafios na Paralelização

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**.

3 Implementação

3.1 Implementação 1

A primeira implementação percorre todos os pontos linha a linha, como é normal em algoritmos que envolvam matrizes. Basicamente, percorremos cada ponto da matriz, verificamos as condições e removemos ou não. Com isto, percebemos que a paralelização pode ser efetuada ao percorrer todos os pontos da matriz, desde que não alteremos a matriz original até ao final da análise dos pontos.

3.2 Implementação 2

A segunda 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 de 32x32 (desta forma vai ter um tamanho inferior a cache L1) e percorre estes blocos um a um. 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 estão em cache. Assim sendo, percebemos que a paralelização pode ser implementada, tanto ao percorrer os blocos como ao percorrer os pontos dentro dos blocos.

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 assim.

4.1 Apresentação dos Resultados

4.1.1 Apresentação dos Resultados

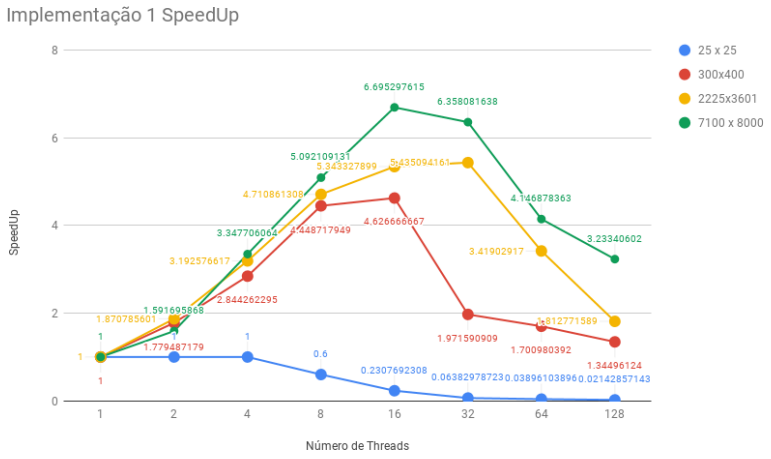


Figura 3: SpeedUp do algoritmo da implementação 1 dependendo do número de Threads

4.1.2 Apresentação dos Resultados da Implementação 2

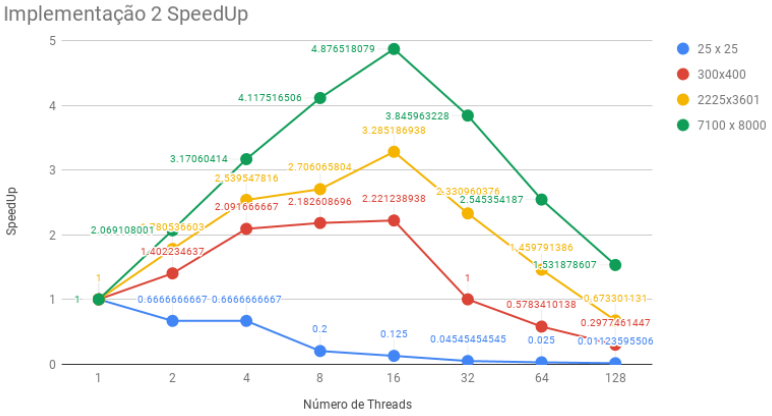


Figura 4: SpeedUp do algoritmo da implementação 2 dependendo do número de Threads

4.1.3 Apresentação da Implementação 1 vs Implementação 2

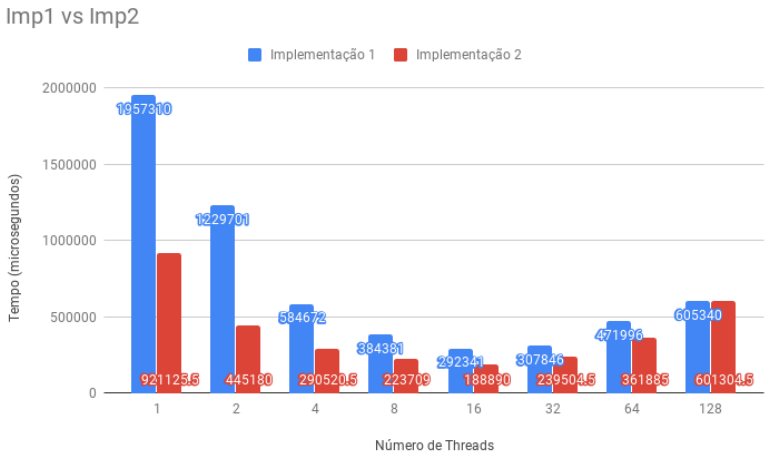


Figura 5: Tempos Implementação 1 vs Implementação 2 para 7100x8000

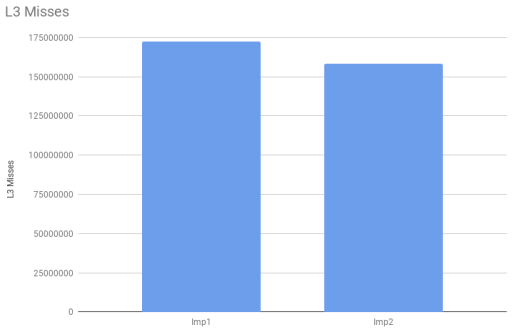


Figura 6: L3 Misses Implementação 1 vs Implementação 2 para 7100x8000

4.2 Análise dos Resultados

4.2.1 Threads e Tamanhos

Para o tamanho de 25x25 verifica-se que não há ganho com o aumento de threads, acontece exatamente o contrário. Isto acontece porque o overhead que cada thread acarreta não compensa, pois o tamanho é tão pequeno que **multithreading** é desnecessário.

Os restantes tamanhos verifica-se um aumento de perfomance com pico nas 16 threads, e depois disso uma descida constante no SpeedUp. Isto deve-se ao facto da a máquina utilizada conter 16 cores físicos e 32 lógicos, logo a partir das 16 threads está-se num ambiente de **HyperThreading**, o que pode provocar **cache poisining**, já que a L1 é partilhada por ambos os cores logicos, que leva a maior acessos a memória e substituição dos registos em cache L1.

4.2.2 Implementação 1 vs Implementação 2

Apesar de ambas as implementações paralelizarem bem, ao consular os tempos, percebe-se que a implementação 2 é bastante melhor. Este resultado deve-se ao **blocking**, já que este permite um menor número de acessos à memória, pois os blocos a serem processados em cache, logo há um menor número de **cache misses**.

5 Conclusão

Uma vez que este algoritmo se trata de um algoritmo **memory bound** no qual não existe reutilização dos dados da matriz, sendo que cada elemento é modificado apenas uma vez, no entanto os elementos são acedidos várias vezes. Sendo as operações sobre cada elemento e seus vizinhos cálculos bastante simples e rápidos encontramos facilmente o bottleneck do algoritmo que se trata dos acessos a memória que, como podemos ver pelos resultados, vão influenciar em muito a velocidade do algoritmo. Apesar de não conseguirmos um speed up linear conseguimos uma diminuição considerável no tempo de execução na paralelização do algoritmo.

6 Anexos

PCP - OpenOMP

Arquitetura: Ivy Bridge (Nó 641)
16 cores físicos
32 cores lógicos
gcc 5.3.0
papi 5.3.2
Unidades de tempo: microsegundos

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

L1

Implementação	Tamanho	#Repetição / #Threads	1	2	4	8	16	32	64	128
1	25 x 25	1	3	3	3	11	10	46	154	190
		2	3	3	3	5	13	64	77	133
		3	3	3	3	3	12	47	133	176
		4	3	3	3	6	13	41	62	129
		5	3	3	3	5	18	66	36	140
		Mediana	3	3	3	5	13	47	77	140
		SpeedUp	1	1	1	0.6	0.2307692308	0.0638297872	0.0389610389	0.0214285714
2		1	2	3	3	10	13	92	80	178
		2	2	2	3	12	16	32	81	326
		3	2	2	3	8	17	44	114	159
		4	2	3	4	12	16	23	79	152
		5	2	3	3	7	27	87	45	202
		Mediana	2	3	3	10	16	44	80	178
		SpeedUp	1	0.6666666667	0.6666666667	0.2	0.125	0.0454545455	0.025	0.0112359551

L2

Implementação	Tamanho	#Repetição / #Threads	1	2	4	8	16	32	64	128
1	305 x 400	1	422	211	121	81	76	119	205	212
		2	392	195	123	78	73	231	164	295
		3	339	194	123	104	75	176	204	335
		4	336	195	122	77	83	110	376	258
		5	347	195	122	78	72	231	175	254
		Mediana	347	195	122	78	75	176	204	258
		SpeedUp	1	1.779487179	2.844262295	4.448717949	4.626666667	1.971590909	1.700980392	1.344961240
2		1	125	90	59	61	57	126	221	409
		2	126	76	60	65	56	125	232	533
		3	131	89	67	56	56	117	250	404
		4	128	90	60	58	62	145	171	434
		5	125	90	60	57	86	136	213	557
		Mediana	125.5	89.5	60	57.5	56.5	125.5	217	421.5
		SpeedUp	1	1.402234637	2.091666667	2.182608696	2.221238938	1	0.5783410138	0.2977461447

L3

Implementação	Tamanho	#Repetição / #Threads	1	2	4	8	16	32	64	128
1	2225 x 3601	1	91777	49058	28747	22687	17864	17380	27742	50657
		2	88445	48765	28767	19425	17176	14579	27327	49983
		3	91795	49384	28474	19482	15160	16886	23961	50628
		4	88136	50078	31126	21810	18777	19029	24607	51771
		5	94052	47527	28653	19300	14447	16723	26843	50250
		Mediana	91777	49058	28747	19482	17176	16886	26843	50628
		SpeedUp	1	1.870785601	3.192576617	4.710861308	5.343327899	5.435094161	3.41902917	1.812771589
2		1	34315	21241	13451	12826	12236	14805	21646	53806
		2	35303	19493	13478	13180	10336	14958	21209	51549
		3	34708	19476	13667	10726	10236	14890	25091	51621
		4	31738	21473	13808	12684	10565	12381	23776	50411
		5	35610	19469	13726	13109	12492	16011	24539	47691
		Mediana	34708	19493	13667	12826	10565	14890	23776	51549
		SpeedUp	1	1.780536603	2.539547816	2.706065804	3.285186938	2.330960376	1.459791386	0.673301131

RAM

Implementação	Tamanho	#Repetição / #Threads	1	2	4	8	16	32	64	128
1	7100 x 8000	1	1957310	1283645	582463	381395	280434	306183	489420	605340
		2	1961218	1210623	596432	379672	298158	311360	468189	620928
		3	1967067	1293428	584672	389143	301233	315405	471996	599233
		4	1901459	1229701	588190	385653	292341	303090	467832	611293
		5	1953902	1215722	572453	384381	290363	307846	480451	603981
		Mediana	1957310	1229701	584672	384381	292341	307846	471996	605340
		SpeedUp	1	1.591695868	3.347706064	5.092109131	6.695297615	6.358081638	4.146878363	3.23340602
2		1	922702	441459	292904	222859	177357	242008	366111	602718
		2	943561	455763	288137	231410	193379	237227	359327	599891
		3	919549	439013	301290	224559	184401	224560	371080	610575
		4	930852	453168	281093	219343	199303	250989	361867	608004
		5	917001	448901	297674	226810	301379	241782	361903	598762
		Mediana	921125.5	445180	290520.5	223709	188890	239504.5	361885	601304.5
		SpeedUp	1	2.069106001	3.17060414	4.117516506	4.876518079	3.845963228	2.545354187	1.531878607