Universidade do Minho

Computação Paralela
Trabalho Prático-OpenMP

Márcio Rocha PG41086
Tiago Pereira A61032

5 Dezembro 2019

Introdução

A fim da cadeira de Computação Paralela foi nos proposto a criação, sequencial e paralela, de um dos algoritmos apresentados na aula, com o objectivo de avaliar a aprendizagem do paradigma de computação paralela em OpenMP.

Computação paralela é uma forma de computação que permite distribuir várias instruções para sejam executadas em paralelo por um computador. Isto apenas pode ser feito, correctamente, se não existir dependências entre essas instruções, caso contrário poderá levar a potenciais defeitos, como a condição de corrida.

Algoritmo

O algoritmo seleccionado é o 6 que realiza a transformada de distância de imagens. Este algoritmo recebe uma imagem *ASCII* em formato *pgm*.

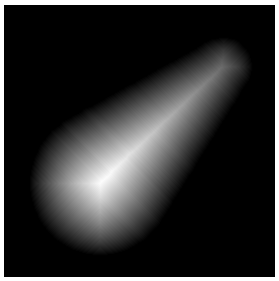


Figura 1: Transformada da distância

Durante a leitura da matriz esta é representada por um *array*, caso um ponto não tenha o valor de 0, preto, é lhe atribuído o valor de 65365, valor máximo do formato *pgm* ou seja branco, para assim iterar por todos os pontos que não sejam pretos. A razão pelo uso do valor 65365 e não de 255 acontece pois ao trabalhar com imagens grandes o valor dos pontos pode ultrapassar 255 quando estamos a calcular a sua transformada da distância.

Após a leitura da imagem, a matriz resultante é processada, onde todos os pontos são verificados se o valor deve ou não ser modificado. Este processo é executado da seguinte forma, a matriz é percorrida se o ponto encontrado é preto este é apenas adicionado à nova matriz, caso seja branco este irá assumir o valor mínimo de um dos seus vizinhos. De seguida, é verificado se esse valor excede ou não 65535 para se somar 1, caso não seja, é ainda verificado se é maior que o valor mais alto que foi encontrado na leitura da matriz("*max_gray*") para depois ser guardado na nova imagem. Caso o valor do ponto seja igual a 65365 este é apenas adicionado à nova matriz.

Este processo repete-se até que não seja encontrado mais pontos brancos na matriz para modificar.

Implementação

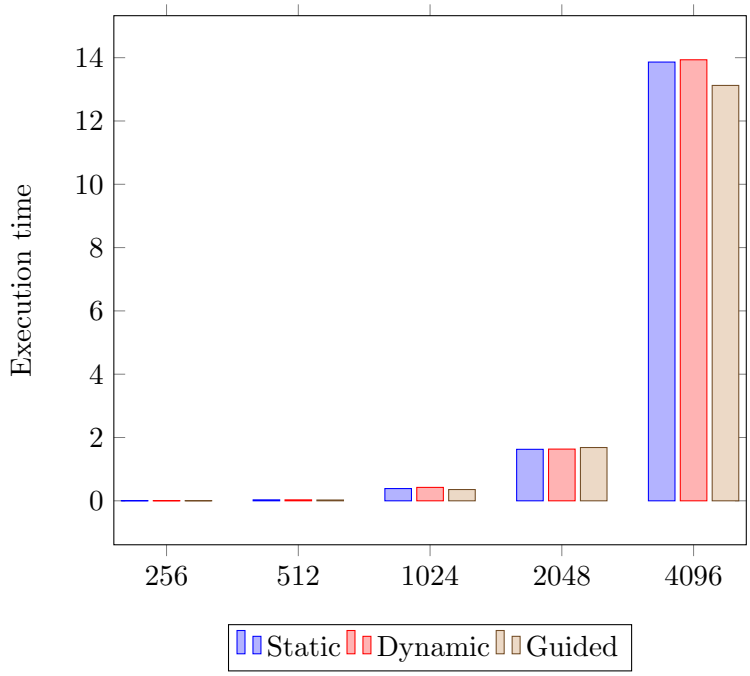
Foi optado não vetorizar o código, visto que a vetorização é usada para aplicar uma operação sobre múltiplos pares de operandos de uma só vez e neste algoritmo percorremos vários operandos, um de cada vez, que podem assumir diferentes operações, sendo assim, não existe necessidade para tal.

O paralelismo do algoritmo foi apenas efectuado no processo de transformação, sendo assim, o programa começa com uma única *thread* que carrega a matriz da imagem e a passa para memória. De seguida esta encontra a directiva OpenMP *for*: *#pragma_omp_parallel for schedule(guided) num_threads(MAX_THREADS)* que cria a equipa de *threads* e que divide o trabalho entre estas.

Durante a fase de experimentação da implementação paralela deste trabalho foram testados o *static*, *dynamic* e *guided* de forma a verificar a melhor maneira de efectuar o balanceamento de carga.

Static vs Dynamic vs Guided

Para os seguintes gráficos foram usadas 20 *threads* e uma imagem com tamanhos diferentes.



A maneira como o *schedule* funciona é dividir o trabalho em múltiplos *chunks* e atribuí-lo a *threads* disponíveis. No *static* a divisão dos *chunks* é feita antes do algoritmo iniciar e atribuída a cada uma das *threads*, isto pode ser ineficiente em alguns algoritmos pois o tempo de execução de um *chunk* nem sempre demora o mesmo tempo que outro levando a mau balanceamento de carga, sendo uma melhor opção optar pelo *dynamic* é o caso, pois à medida que as *threads* trabalham caso acabe o seu trabalho mais cedo esta "rouba"o trabalho de outras *threads* para se manter ocupada e completar a execução do algoritmo mais rápido, acabando assim com o mau balanceamento.

No entanto, neste algoritmo o tempo de execução de cada *chunk* é relativamente o mesmo, fazendo com que os resultados da utilização da diretiva *static* e *dynamic* sejam semelhantes, apesar disto o *guided* apresenta melhores resultados.

À partida tudo aponta para o *static*, contudo definindo o k no *schedule(dynamic, k)* os valores torna-se melhores ou iguais ao do *static*.

	k = 1	k = 2	k = 4	k = 8	k = 16	k = 32
256	0.00452	0.00454	0.00455	0.00458	0.00456	0.00454
512	0.02782	0.02753	0.02725	0.02715	0.02736	0.02797
1024	0.39457	0.40699	0.38428	0.40631	0.38689	0.39840
2048	1.57927	1.61117	1.58781	1.59674	1.58099	1.59339
4096	13.18661	13.379499	13.42573	13.49609	13.45234	13.27744

Foram testados vários valores de k para o algoritmo, a correr com 20 *threads*, é de notar que existe melhoramento dos valores sendo o melhor obtido pelo k = 1, no entanto, isto mantêm-se apenas até um certo numero de *threads* pois existe um *bottleneck* na condição de *high concurrency*. Isto acontece porque o *scheduler* atribui uma iteração do ciclo por *thread*, isto faz com que elas regressem rapidamente a pedir por mais trabalho. Nos anexos encontra-se um gráfico de escalabilidade desta diretiva.

Com isto em mente optou-se por usar o *guided* para a paralelização do algoritmo.

Resultados

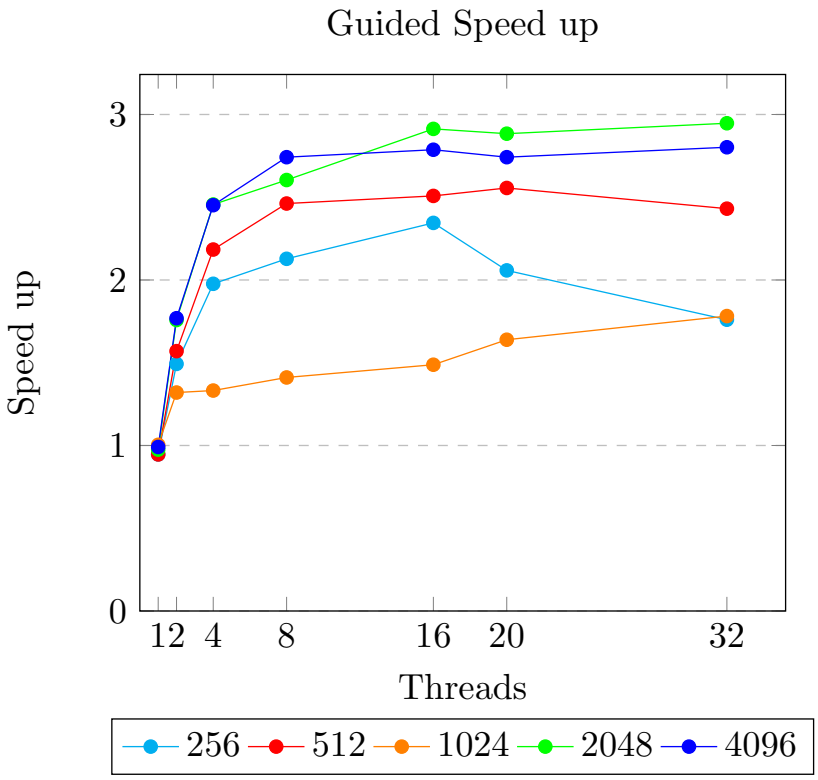
Esta secção apresenta os resultados obtidos com a implementação paralela final do algoritmo. Como referido antes apenas o processo da transformada da distância da imagem foi paralelizado pois é o que se pretende analisar e estudar neste trabalho, sendo assim, os valores representados nesta secção são equivalentes ao calculo da transformada e não da execução total do programa.

Para os testes, foi usado a máquina *Ivy Bridge dual-processor* com "gcc -03 std=c99 -fopenmp".

A Tabela 1 contêm os tempos médios de execução do algoritmo sequencial e da implementação paralela que foram testados com imagens de vários tamanhos e executados com diferentes números de *threads*. Para o calculo das médias o algoritmo foi executado 20 vezes.

	sequencial	2 threads	4 threads	8 threads	16 threads	20 threads	32 threads
256	0.0090	0.0057	0.0043	0.0040	0.0042	0.0041	0.0048
512	0.0692	0.0417	0.0300	0.0266	0.0261	0.0256	0.0269
1024	0.5814	0.4425	0.4386	0.4141	0.3926	0.3563	0.3278
2048	4.6496	2.5786	1.8458	1.7414	1.5564	1.5721	1.5387
4096	36.3160	20.3411	14.6761	13.1271	12.9125	13.1232	12.8456

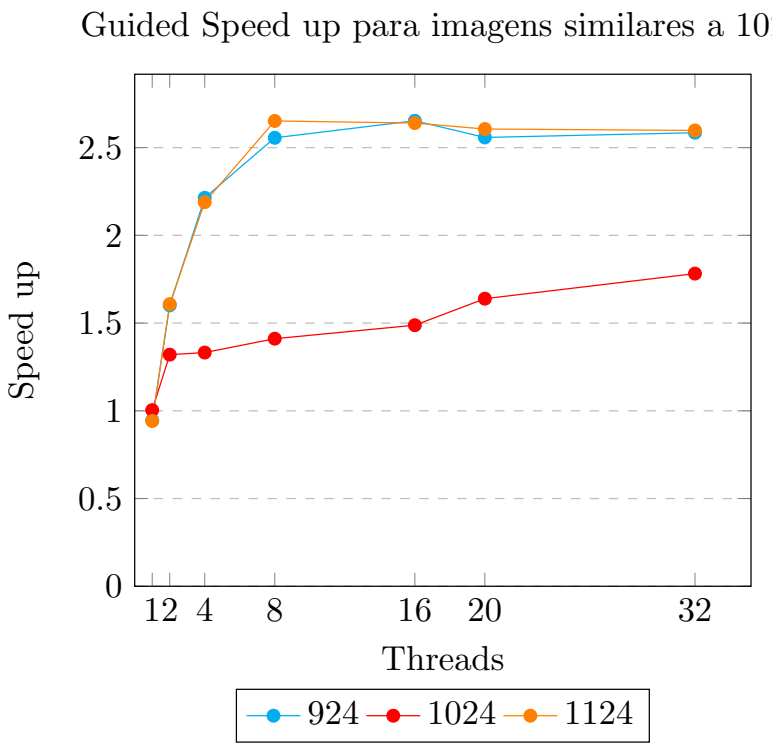
É de notar que o tempo de execução aumentou drasticamente em imagens com tamanho igual ou maior a 2048 pixeis, isto acontece pois a memória de um nó não consegue guardar tamanhos tão grandes, é então que o método de *swapping* é iniciado. Além deste aumento, o tempo de execução tende a normalizar a partir das 8 *threads*, isto acontece pois chegou ao limite de *memory bandwidth*, sendo assim por mais *threads* que tenha este o tempo de execução irá manter-se igual ou mais pequeno.



Á medida que o numero de *threads* aumenta nota-se também um aumento do *speed up*, até

que atinge 8 *threads* onde normaliza, isto acontece devido à normalização do tempo de execução referido anteriormente, por causa da *memory bandwidth*.

No entanto, na imagem 1024x1024 o *speed up* é pequeno pois esta não cabe inteiramente na cache, por isso tem de ir mais vezes à memória diminuindo a performance do algoritmo. Apenas esta imagem é afectada, visto que as imagens 2048x2048 e 4096x4096 mantêm um grau de *speed up* elevado, uma possível razão disto acontecer é que a imagem 1024x1024 não está a conseguir tirar partido do alojamento de dados na cache como as outra imagens estão. Com o gráfico a baixo é possível concluir que a razão do baixo *speed up* é o tamanho da imagem e não do algoritmo.



Quando entra em *hyperthreading*, a partir das 20 *threads*, como as imagens pequenas cabem num só core fazendo com que o segundo core tenha de ir buscar informação da imagem ao outro, aumentando o tempo de execução. No caso das imagens, 2048x2048 e 4096x4096, grandes acontece o contrário, em que o uso de um segundo core ajuda no *speed up*, isto justifica-se porque,este segundo core recolhe da memória mais informação da imagem, sendo assim como esta variável é partilhada as *threads* podem aceder á cache um do outro precisando de aceder à memória menos vezes, melhorando o tempo de execução e consequentemente o *speed up*.

Este algoritmo é *memory bound*, no qual sobre cada iteração existem múltiplos acessos à memória, isto acontece pois não existe reutilização dos dados da matriz, sendo cada elemento modificado ou passado para uma nova matriz. É ainda verificado para cada ponto da matriz os seus vizinhos, são cálculos simples e rápidos mas que são executados varias vezes tendo de aceder á memória por cada um dos pontos.

Conclusão

Em suma, uma vez que neste algoritmo existem tantas *threads* a irem à memória encontra-se facilmente o *bottleneck* que se trata dos acessos à memória e a incapacidade da *memory bandwidth* acompanhar o tamanho de tantos pedidos, *memory bound*. Apesar de não ter sido atingido o *speed up* ideal conseguiu-se uma diminuição considerável do tempo de execução do algoritmo.

Anexos

Static:

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
256	0.0090	0.0060	0.0046	0.0038	0.0037	0.0045
512	0.0692	0.0444	0.0330	0.0273	0.0257	0.0291
1024	0.5806	0.4112	0.4136	0.4160	0.4120	0.3936
2048	4.4541	2.7390	2.0786	1.6902	1.6015	1.6961
4096	35.3565	22.1.90	17.1608	13.8523	12.9859	13.6398

Dynamic:

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
256	0.0091	0.0082	0.0045	0.0040	0.0036	0.0046
512	0.0699	0.0526	0.0331	0.0270	0.0259	0.0291
1024	0.5871	0.5289	0.3951	0.4164	0.4277	0.4143
2048	4.6218	2.8017	2.0801	1.7081	1.5769	1.6946
4096	36.2886	21.4665	16.8477	14.1828	13.1190	13.6556

Guided:

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
256	0.0090	0.0060	0.0044	0.0036	0.0037	0.0045
512	0.0693	0.0441	0.0332	0.0274	0.0256	0.0293
1024	0.5789	0.4514	0.4204	0.4081	0.4339	0.4063
2048	4.4592	2.7780	2.0945	1.7396	1.5721	1.7156
4096	35.3917	21.9067	17.4501	14.0027	13.1232	13.5325