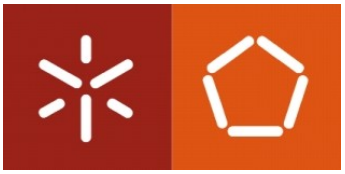
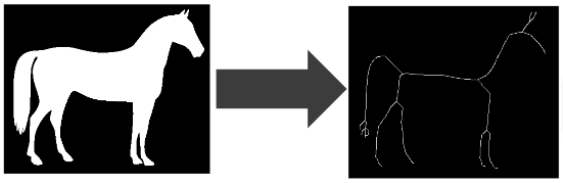


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Algoritmo Esqueletização em memória partilhada

Implementação em openMP



Paradigmas de Computação Paralela - 18/19

Luís Pedro Pereira Fonseca a60993
Filipe de Sousa Marques a57812

Conteúdo

1 Caso de Estudo

1.1 Esqueletização de uma imagem binária

O algoritmo em estudo será a esqueletização de uma imagem binária, que tal como o nome indica, permite o fabrico de um esqueleto representativo de uma imagem original a preto e branco. Para alcançar esse efeito, a imagem será iterada para se proceder à remoção dos pixeis em excesso. O resultado final será uma linha com 1 pixel de espessura. Para proceder à paralelização do algoritmo, utilizaremos a linguagem C++, e sua extensão OpenMP, que permitirá distribuir as computações a fazer pelas diversas threads.

2 Ambiente de testes

2.1 Plataformas de teste

Para os testes utilizamos uma das nossas máquinas pessoais (portátil) e um dos nós do Cluster do DI, neste caso o node 641. Equipado com dois Xeon E5-2650v2, arquitectura Ivy Bridge, 16 cores físicos no total, 32 virtuais. Em termos de cache, tem 20MB de L3 partilhada, 256K de L2 exclusiva a cada core (totalizando 4MB por chip) e 64K de cache L1, dividida em 32K para dados e 32K para código. Cada linha de cache tem 64 Bytes. Abaixo podemos ver uma descrição mais pormenorizada de cada máquina.

System	compute-641	personal-laptop
#Sockets	2	1
CPU	Xeon E5-2650v2	Core i7-7700HQ
#Cores per CPU	8	4
#Threads per CPU	16	8
Architecture	Ivy Bridge	Skylake
Lithography	22nm	14nm
Base Frequency	2.60 GHz	2.80 GHz
Turbo-Boost	3.40 GHz	3.80 GHz
Extensions	SSE4.2, AVX	SSE4.2, AVX2
Peak FP Performance	400 GFlops/s	36.8 GFlops/s
L1 Cache	32KB per core	32KB per core
L2 Cache	256KB per core	256KB per core
L3 Cache	20MB	6MB
Total memory	64GB	16GB
#memory channels	4	2
Vendor Peak Mem. BW	59.7 GB/s	21.3 GB/s

2.2 Métricas

A metodologia utilizada para o cálculo de resultados foi o k-best, com k=3 de 10 amostras, e feita a média desses 3 resultados, que permite eliminar os valores excessivamente desviados do resultado real/expectável num cenário real de uso.

As medicões não incluem I/O de leitura e escrita de ficheiros. Para medir os tempos foram usadas as funções disponibilizadas pelo openMP para cronometragem. Todos os tempos foram registados em micro-segundos.

2.3 Inputs utilizados

Um script de submissão para o Cluster correu extensivamente cada uma das versões do kernel de forma iterativa. Para cada uma das suas versões correu imagens de diferentes tamanhos, várias vezes, para extracção do k-best

score. As imagens de input utilizadas foram círculos brancos de fundo preto, de diâmetro igual à largura da imagem.

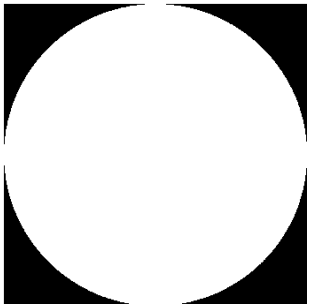


Figura 1: Exemplo de um input. Um círculo de 1024 de diâmetro será representado por uma imagem 1024x1024 de 1048576 pixels. Assumindo 1 pixel = 4 bytes em memória, necessitaremos de 4MB para armazenar a sua matriz.

2.3.1 Tabela inputs

tamanho em disco	altura*largura	malloc (Img_inicial + changing_matrix)
128K	horse 402*330	1MB
256K	circle 512*512	2MB
1MB	circle 1024*1024	8MB
2MB	circle 1536*1536	18.5MB
4MB	circle 2048*2048	32MB
16MB	circle 4096*4096	131MB

Figura 2: Lista dos inputs utilizados para teste. Cada tamanho de imagem foi escolhida de forma a encaixar em diferentes níveis de cache

2.4 Compilador utilizado e optimizações

Foram compiladas várias versões dos executáveis em g++ versão 5.3 e versão 6.5, ambas as versões com nível de optimização de nível=O3. As flags de compilação foram:

-O3 -std=c++11 -march=native -fopenmp

Foi também utilizada a versão 5.4.1 do papi para extração de algumas métricas extra na execução de cada kernel.

3 Versão sequencial

3.1 Descrição do algoritmo

Para a realização desta função procedemos a seguir o mais fielmente possível os cálculos e verificações fornecidos com o enunciado do trabalho para o algoritmo de esqueletização.

Ao iniciar o cálculo, este prepara algumas variáveis para auxiliar no processamento da imagem, sendo que, as importantes de referir são, a variável a que chamaremos ch image (changing image), representando o espaço alocado com o mesmo tamanho da imagem original, ao qual se trata as bordas de padding internamente e, dois arrays de tamanho fixo a representar as localizações relativas dos vizinhos ao pixel pela ordem dada pela imagem.

P_9	P_2	P_3
P_8	P_1	P_4
P_7	P_6	P_5

Figura 3: Ordem pré-definida dos vizinhos do pixel a calcular

Chegando ao ciclo principal, é contabilizado o numero de iterações, representantes do número de vezes que a imagem é corrida, sendo que será sempre corrida um número par de iterações para poder ser atingida pelas condições de ambos os sentidos. Neste ciclo, os pixeis da imagem são seguidos sequencialmente, e por cada um, as seguintes verificações são feitas:

- caso o valor do pixel seja 0 (representando um pixel preto), na mesma posição da ch image, é colocado o valor 0 na mesma sem mais algum cálculo realizado para este pixel;
- caso o valor do pixel seja 1 (representando um pixel a branco), é verificado se se trata de um pixel da borda da imagem e, no caso afirmativo, é retirado e marcado como pixel preto no ch image, caso contrário, continua como pixel branco.

A verificação do pixel como borda passa por:

- verificar se tem no máximo 6 vizinhos e no mínimo 2 que tenham o valor 1;
- verificar se, no sentido dos ponteiros do relógio, o número de transições dos vizinhos de 0 para 1 apenas acontece uma vez, isto examina que o pixel não represente um pixel de ligação entre partes da imagem;
- e, finalmente, examinando os valores de vizinhos específicos para testar, quer num sentido nas iterações ímpares, quer noutro, nas iterações pares, se o pixel é uma borda lateral, de uma secção da imagem não linear, e que retirar-lo não cortará a imagem em pedaços diferentes. As funções lógicas representativas deste teste podem ser vistos na imagem a seguir, retirada do enunciado do trabalho.

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

No fim de cada iteração, a imagem de onde se lê e a imagem para onde se escreve trocam de posição, para continuar o ciclo, sendo que este só termina quando nenhum valor for removido durante uma iteração completa pela imagem, e o número de iterações for par. Este passo, é importante pois cada iteração é a base para a iteração seguinte.

3.2 Optimizações

3.2.1 Matriz uni-dimensional

Os arrays em c/c++ são guardados em memória como um bloco contíguo. Assim aceder a[x][y] é equivalente a a aceder a[x*N_COLUNAS + y]. Ao iterarmos em 2 dimensões podemos estar a não utilizar largos espaços de memória podendo causar cache misses e as respetivas penalizações. Adicionalmente, ao reduzirmos de duas para uma dimensão estamos também a reduzir a complexidade do código e a facilitar o compilador no processo de optimização. Para o paralelismo e vetorização isto é uma mais valia uma vez que o compilador poderá tentar vetorizar código que anteriormente considerava não vetorizável. Por último podemos ainda referir o facto de nestes casos necessitarmos de menos sincronização entre processos/threads.

3.2.2 Alinhamento dos recursos de memória

De forma a não desperdiçar linhas de cache alinhamos as alocações de memória, para a imagem original e para a matriz resultate com a função memalign fazendo com que o apontador para esses chuncks de memória sejam múltiplos de 32. Com isto reduzimos as cache misses, e potencializamos as oportunidades de vectorização do código.

3.2.3 Redução nas acessos à memória e aproveitamento da L1

Tal como explicitado anteriormente, para cada pixel teremos que conhecer os seus vizinhos. Estando os mesmos em diferentes linhas da matriz, podemos ser tentados em extrair os mesmos para uma sub-matriz 3x3, num outro espaço de memória, para depois ser testada pelas funcões de teste e remoção do pixel. No entanto tal operação revela-se extremamente cara do ponto de vista computacional. Não só não tiramos partido da L1, como estamos a acrescentar um overhead desnecessário de leitura/escrita na memória para um array temporário. A solução adoptada foi a criação de dois arrays estáticos (XX-index e YY-index) com as coordenadas relativas a um dado pixel. Para conhecer os vizinhos de um pixel apenas precisaremos de saber as suas coordenadas, ao qual depois somamos os valores de XX-index e YY-index.

3.2.4 Eliminação de chamadas de funcões dentro do ciclo

Dentro do ciclo while principal, optou-se por desdobrar todas as funções de teste de remoção de pixel, eliminando qualquer overhead provocado pela inicialização de uma nova stack frame e mudança de contexto.

4 Versão paralela

4.1 Profiling

% time	cumulative seconds	seconds	calls	self ms/call	total ms/call	name
100.06	0.17	0.17	1	170.09	170.09	skeletonize_serial
0	0.17	0	1	0	0	output_ppm
0	0.17	0	1	0	0	__static_initialization_and_destruction_0
0	0.17	0	1	0	0	new_img
0	0.17	0	1	0	0	free_img
0	0.17	0	1	0	0	read_ppm
0	0.17	0	1	0	0	read_inp_image
0	0.17	0	1	0	0	write_out_image
0	0.17	0	1	0	0	verify_command_line

Figura 4: Profiling em Gprof

Pela seguinte tabela obtida com gprof, executanto o programa para uma imagem de reduzidas dimensões, verificamos que a totalidade do tempo é consumido pela função skeletonize. Pela lei de Amdahl sabemos que existe um grande potencial para um speedup, se melhorar-mos os tempos de execução de skeletonize, uma vez que o programa não é limitado pelas restantes funções. Skeletonize será então o nosso foco.

4.2 Descrição do algoritmo

Sendo este um algoritmo iterativo, onde a iteração seguinte depende sempre da anterior, revela-se difícil apostar na paralelização do ciclo while mais externo sem que haja grande contenção entre threads. Uma forma mais limpa de tentar paralelizar o algoritmo será dividir o trabalho de 1 iteração por diversas threads. Com efeito, na solução adotada, o algoritmo parte a matriz input e a matriz output em fatias iguais (caso o escalonamento seja static e o número de linhas seja múltiplo do número de threads) e atribui-as a cada uma das threads. Cada thread trabalhará independentemente na sua porção da imagem.

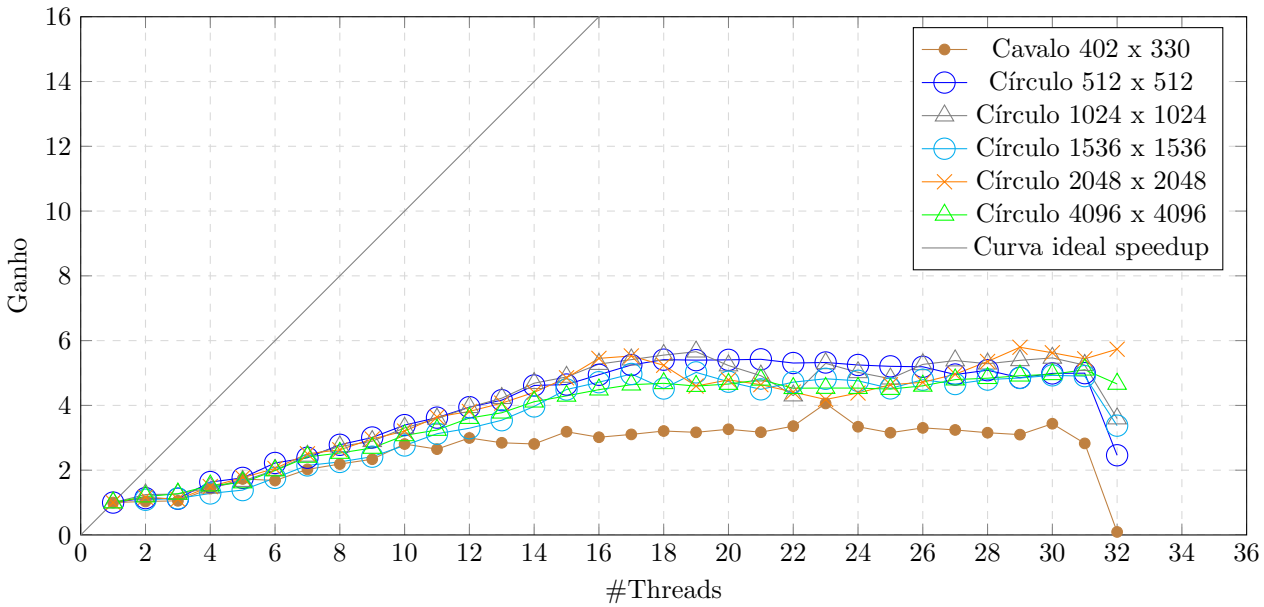
A cláusula OpenMP que nos permite fazer esta divisão equitativa do trabalho será então `#pragma omp parallel for schedule(static)reduction(+:cont0)reduction(+:cont1)` que actuará no ciclo for exterior, que itera sobre as linhas da matriz. Quando cada uma das threads termina o que tem a fazer, é feito uma espécie de "broadcast"das variáveis cont0 e cont1 o que vai permitir à master thread saber se continua ou não a iterar sobre a imagem.

Tentamos também outros formatos de escalonamento, sendo que o dynamic foi o que nos ofereceu resultados mais interessantes.

5 Análise dos resultados

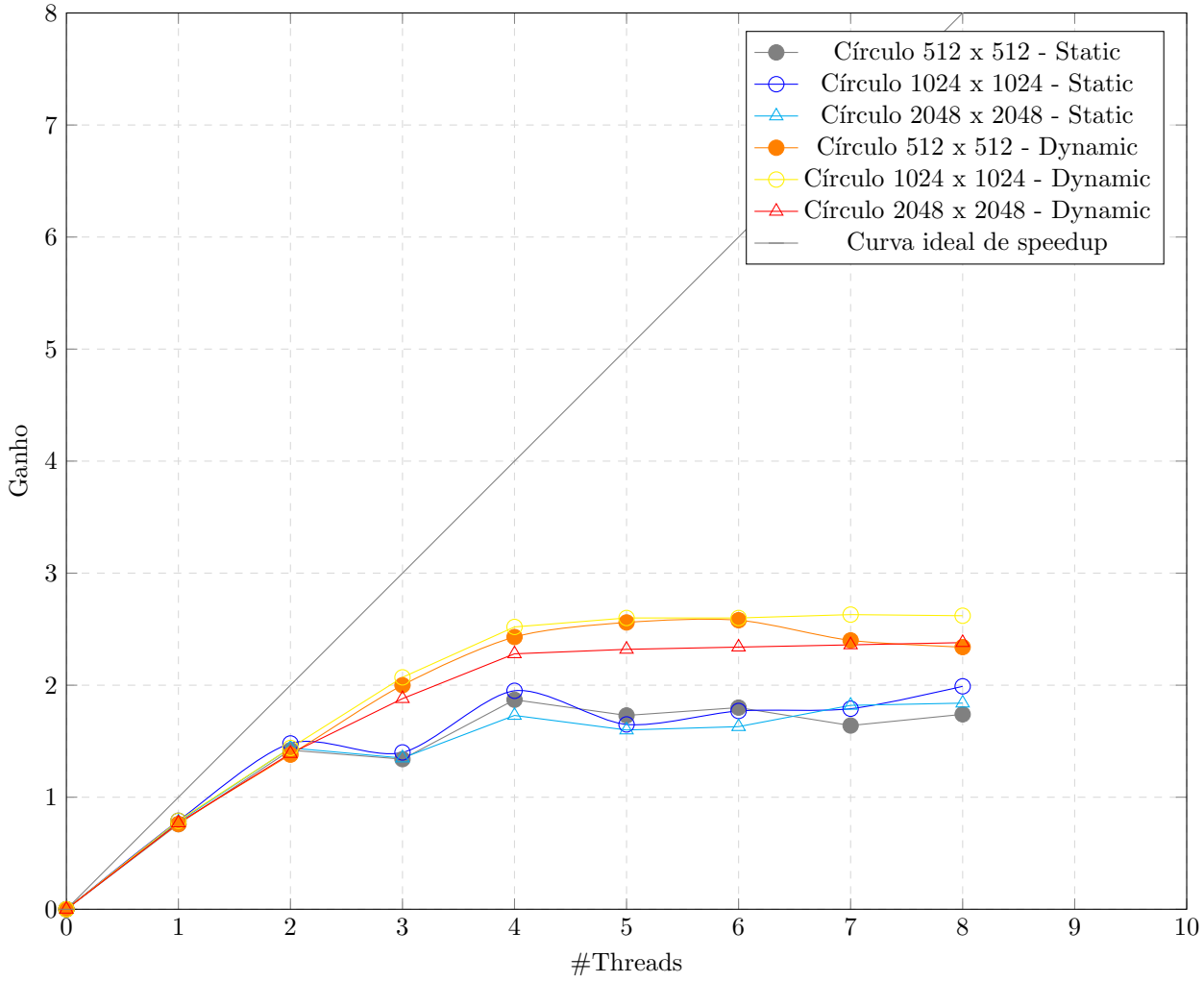
5.1 Resultados no Node-641 do cluster

Podemos observar no gráfico que o algoritmo não escala linearmente com o aumento do número de threads. Os ganhos obtidos com duas threads em simultâneo são negligenciáveis. Os ganhos da divisão da matriz não compensam o overhead adicional de paralelismo. O mesmo se pode dizer para 3 threads. A partir das quatro threads começamos a observar alguns ganhos substanciais, no entanto ainda longe do linear esperado. A partir das 16 threads deixamos de ter cores físicos por onde atribuir threads, e passamos a partilhar as unidades funcionais por 1 ou mais threads de cada core. Para um número superior a 32 threads, deixamos de ter cores virtuais para atribuir às threads e entramos em contenção de recursos com o sistema operativo, algo que se mostra bem patente na queda abrupta da curva em quase todas as imagens testadas.



Resultados foram obtidos com a versão 5.3.0 do gcc, com a cláusula de escalonamento static.

5.2 Resultados no portátil pessoal



Uma vez mais, podemos ver que o algorimo não escala linearmente para qualquer tamanho de imagem. Sendo esta uma máquina de 4 cores, podemos também observar que a partir de 4 threads não compensa aumentar o número de threads do programa. Estando várias threads a correr no mesmo core, cache poisoning poderá ocorrer, além de luta pelas mesmas unidades funcionais. Algo que nem sempre o compilador e/ou o CPU conseguem resolver com a exploração do ILP.

O algoritmo melhora consideravelmente com a utilização de um escalonamento dinâmico, pois o número de computações a serem feitas vai variar de thread para thread, o que leva a que umas acabem mais cedo do que outras. Algo que pode variar bastante de acordo com a imagem que se está a processar. Ao incutir no escalonador a função de distribuição do trabalho, haverá um maior rendimento por parte de cada uma das threads.

6 Conclusões e dificuldades encontradas

O algoritmo paralelo criado mostrou-se capaz de obter um *speedup* na ordem das 6 vezes em determinadas condições. Ainda que não linear, este ganho parece-nos razoável tendo em conta as dependências que existem entre iterações.

A criação do algoritmo sequencial foi um processo interessante tendo-se primeiro criado uma versão em python para assegurar a sua correção (e compreensão). As maiores dificuldades surgiram na sua tentativa de paralelização. Inicialmente estava-mos a utilizar cláusulas openMP 4.5 o que nos dificultou o processo de testes no cluster devido à inexistência de uma versão mais recente do gcc. Após compilado o gcc-6.5, e testado o programa, reparamos que as cláusulas utilizadas não eram as mais indicadas. Com o uso das diretivas omp target e omp map, pretendia-mos sinalizar o compilador das zonas de memória onde deveria haver contenção, nomeadamente no aray de vizinhos. Tal solução de alocação suplementar de um array de vizinhos acabou por ser abandonada pelos motivos já explicados e a diretiva openmp foi simplificada.

Devido à complexidade assimptótica do algoritmo, testes com imagens de maiores dimensões não foram possíveis, sob pena de ocuparmos o nó durante vários dias.

Nas otimizações da versão sequencial, faltou ainda produzir variantes com loop unrolling e explorar oportunidades de vectorização.