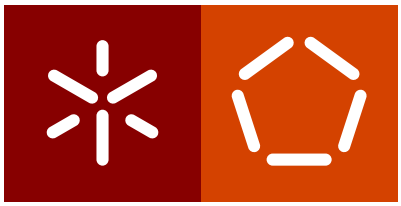
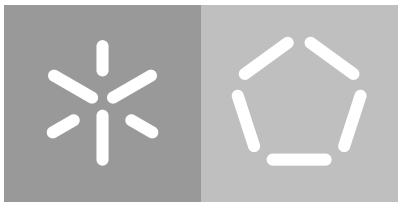


Universidade do Minho
Escola de Engenharia
Departamento de Informática

A72227 - Diogo Vilaça
A72424 - José Macedo

Distance Transform
OpenMP

November 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

A72227 - Diogo Vilaça
A72424 - José Macedo

Distance Transform OpenMP

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
João Luís Sobral
André Pereira

November 2016

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

1

ALGORITMO SEQUENCIAL

O primeiro passo do algoritmo é percorrer o array *Pixmap* que contem todos os pixeis da imagem, cada um representado com uma *struct Node*. À medida que o array é percorrido verifica-se se o pixel é preto, caso seja é invocada a função *EightNeighbour* que adicionará os pixeis brancos a uma lista. Para a transformada da distância queremos mudar a cor do pixel caso um vizinho não seja branco e por isso é que procuramos por pixeis pretos e adicionamos os seus vizinhos brancos. No fim deste ciclo temos uma lista dos pixeis brancos com vizinhos pretos.

No segundo passo percorremos a lista de pixeis brancos até esta ser vazia, calculando a cor de cada pixel branco. Em cada iteração removemos o pixel calculado e adicionamos os vizinhos brancos deste pixel que ainda não tenham tido a sua cor calculada, porque ao alterarmos a cor do pixel atual, os seus vizinhos ganham um vizinho não-branco. Cada nodo que representa um pixel tem uma flag que indica se já esteve na lista e é assim que filtramos os repetidos. Para impedir a adição de pixeis pretos, estes são criados com esta flag a indicar que já estiveram na lista.

No último ciclo o array dos pixeis é percorrido, alterando a cor de pixel de modo à imagem não ficar tão escura.

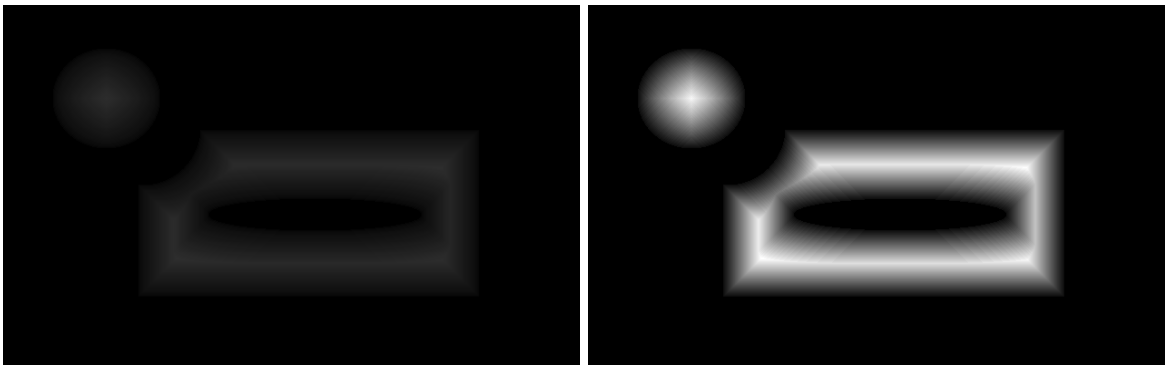


Figura 1: Output antes e depois de ser executado o último ciclo.

2

ALGORITMO PARALELO

Quanto à implementação da versão paralela começamos por paralelizar o último ciclo porque não tinha dependências, bastando assim adicionar *#pragma omp parallel for private(i)*. Apenas tivemos de assegurar que o iterador do ciclo fosse privado a cada uma das threads.

```
#pragma omp parallel for private(i)
for(i=0; i < width * height; i++){
  (Pixmap+i)->distance = ((Pixmap+i)->distance * Maxval)/currdist;
}
```

Ao paralelizar o primeiro ciclo deparamo-nos com três tipos de dependências:

1. Read after write (RAW)
2. Write after read (WAR)
3. Write after write (WAW)

Em cada iteração é chamada a função a função *Enqueue* várias vezes, que lê e escreve na lista de pixeis brancos. Daí poderem acontecer os três casos mencionados.

A solução para este ciclo foi criar uma lista local a cada thread. Cada thread faz o seu trabalho numa lista local e no fim junta os seus pixeis à lista principal.

```
#pragma omp parallel firstprivate(local_frontier) private(i)
{
  #pragma omp for
  for(i=0; i < height * width; i++)
    if ((Pixmap+i)->distance == 0) EightNeighbour(i, &local_frontier);

  #pragma omp critical
  mergeQ(&local_frontier, &Frontier);
}
```

O maior problema estava no segundo ciclo, que não conseguimos paralelizar.

```
while(Frontier.Head != NULL){
  EightNeighbour((Frontier.Head)->offset, &Frontier);
  Dequeue(&Frontier);
}
```

Tentamos primeiro transformar o *while* num *for* para usar *#pragma omp parallel for* mas o número de iterações é desconhecido, não podendo ser calculado no ciclo anterior. Isto deve-se ao facto de a função *EightNeighbour* poder adicionar novos pixeis à lista caso ainda não tenham sido calculados antes. Também não podemos separar a execução das duas funções porque a função *Dequeue* depende do pixel a ser tratado pela função *EightNeighbour*.

Não podendo separar as duas funções tentamos uma abordagem parecida com a do primeiro ciclo e cada thread em vez de juntar a sua lista de pixeis à lista principal (no primeiro ciclo) manteria essa lista e trabalharia sobre ela neste ciclo. O output gerado era errado e as threads apenas usavam os pontos pretos no domínio que lhes foi atribuído para calcular a transformada da distância.

Foi então que tentamos paralelizar a própria função *EightNeighbour* que pode executar até oito vezes a função *Enqueue* por cada vez que é chamada. Para isso, era feita uma task para cada um dos oito ifs que poderiam vir a executar a função *Enqueue*. Este método aumentou muito o tempo de execução do programa tratando-se de um problema em que eram criadas muitas threads para fazer pouco trabalho.

3

RESULTADOS

Para calcular os resultados, submetemos o trabalho a um nodo do cluster (compute-652-2), do qual reservamos todos os 20 cores físicos com hyperthreading resultando num total de 40 cores.

Os ficheiros de input usados para medir os tempos de execução eram todos semelhantes à figura seguinte, mas com dimensões diferentes.

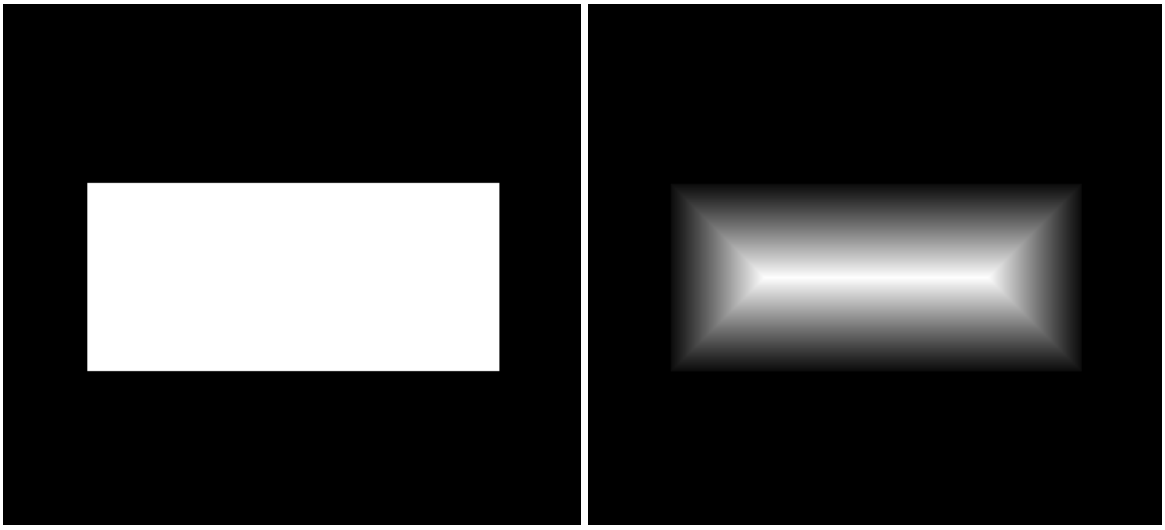


Figura 2: Ficheiro de input (esquerda) e ficheiro de output (direita).

Todos os ficheiros de input usados foram feitos recorrendo à ferramenta *gimp* e guardados em *ASCII* com extensão *pgm*. Para além destes ficheiros com rectângulos, foram testados outros que podem ser vistos a seguir.

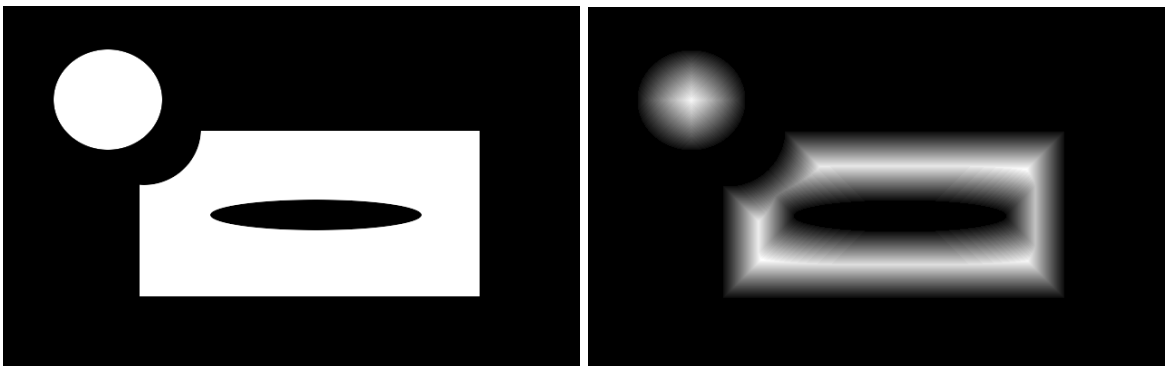


Figura 3: Ficheiro de input (esquerda) e ficheiro de output (direita).

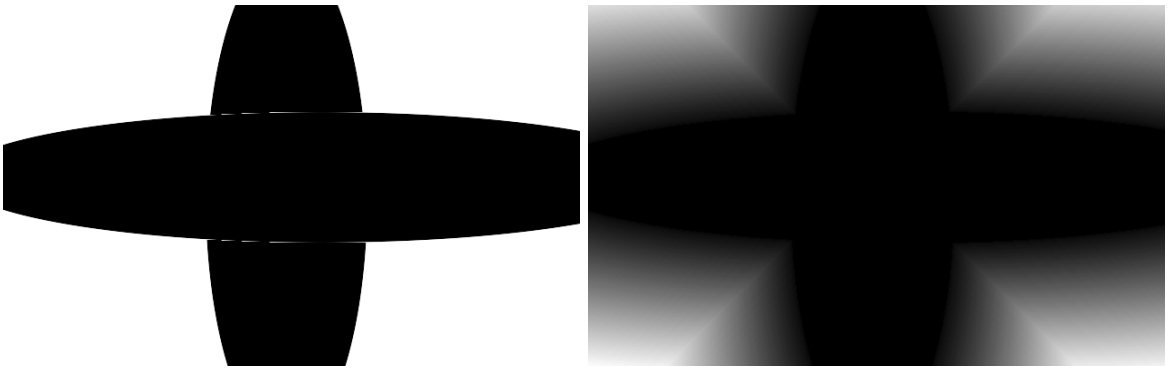


Figura 4: Ficheiro de input (esquerda) e ficheiro de output (direita).

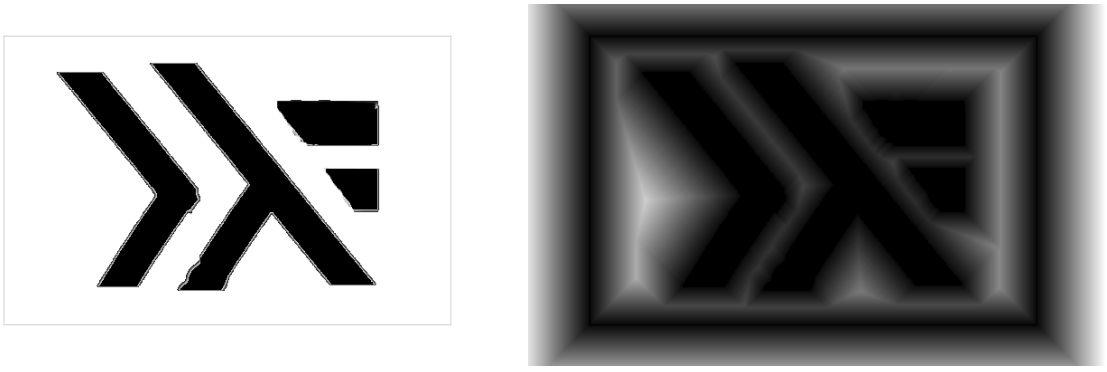


Figura 5: Ficheiro de input (esquerda) e ficheiro de output (direita).

4

ANÁLISE DE RESULTADOS

Para cada tamanho do ficheiro de input foram feitos 63 testes (9 testes para diversos números de threads), cujos resultados foram guardados em tabelas para posterior análise. Para cada conjunto de 9 testes calculamos a mediana dos tempos de execução e o speed-up, que podem ser vistos no seguinte gráfico.

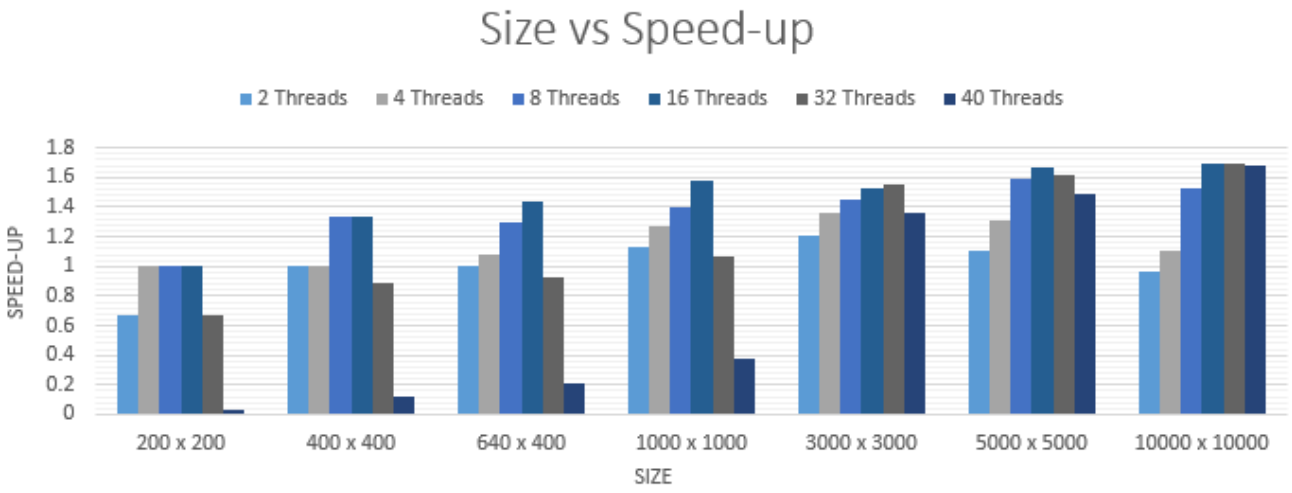


Figura 6: Gráfico do speed-up em função do tamanho para diversos números de threads.

Como esperado, os speed-ups são maiores para ficheiros maiores. Isto deve-se a haver mais trabalho para ser distribuído entre as várias threads.

Para o ficheiro de input 200x200, há tão pouco trabalho a ser feito que qualquer ganho obtido com a implementação de paralelismo é perdido pelo overhead de criação e distribuição de trabalho pelas threads.

Para ficheiros de input abaixo de 1000x1000, o uso de 8 ou 16 threads permitem algum ganho de performance e será óptimo, visto já haver mais trabalho a ser distribuído entre estas. Um número maior de threads irá traduzir-se num menor speed-up, devido uma granularidade demasiado fina, ou seja, pouco trabalho para cada thread. Para 40 threads

reparamos que o tempo de execução aumentou muito pois era gasto mais tempo a criar e a gerir estas threads do que a fazer os cálculos necessários para um ficheiro pequeno.

Para ficheiros acima de 1000x1000, o trabalho a ser executado já justifica o uso de mais threads, sendo que 16 threads continua a ser óptimo mas um numero de threads superior traduz-se em resultados praticamente iguais.

No entanto, não se atinge sequer um speed-up de 2. A introdução de paralelismo neste algoritmo traz ganhos de performance mas não muito significativos. Este algoritmo não é embaraçosamente paralelo e possui dependências que não conseguimos contornar, o que limita os ganhos de performance.

Para o cálculo da transformada da distância, existem vários algoritmos disponíveis. Este é um dos mais eficientes, mas não é perfeito para implementação de paralelismo. Foi feita a experiência em um outro algoritmo, este embaraçosamente paralelo, em que o speed-up era sempre próximo ou igual ao número de threads usadas. Este, porém, era incrivelmente ineficiente, e mesmo com esse paralelismo excelente e algumas optimizações do código, chegava a demorar mais de 10 minutos para resolver inputs que são praticamente instantâneos com este algoritmo.