# CP - Trabalho Prático 1

Duarte Serrão
UC de Computação Paralela
*Mestrado Integrado em Engenharia Informática*
*Universidade do Minho*
a83630@alunos.uminho.pt

Vasco Oliveira
UC de Computação Paralela
*Mestrado em Engenharia Informática*
*Universidade do Minho*
pg50794@alunos.uminho.pt

***Abstract*—Este relatório demonstra o processo de otimização de um algoritmo por meio de paralelismo a nível de instruções. Entre a primeira versão sem otimizações nenhumas e a última, documentou-se uma diferença de 20 segundos.**

***Index Terms*—algoritmo, otimização, paralelismo**

## I. Introdução

O presente relatório, do primeiro trabalho prático da unidade curricular de Computação Paralela, do curso de Mestrado (Integrado) em Engenharia Informática da Universidade do Minho, visa demonstrar a metodologia aplicada aquando a otimização de um dado algoritmo, dando máximo uso às capacidades de paralelismo a nível de instruções de um computador.

## II. Métodos de Otimização

### A. Algoritmo Original

Para chegar a uma primeira solução para o algoritmo, foi necessário a consulta e análise do documento fornecido pelos docentes bem como a visualização de vários exemplos da resolução do algoritmo k-means.

Optou-se por utilizar dois arrays de pontos (x e y) e dois de clusters (x e y) de forma a que fique o mais parecido à solução sugerida da função `init` no enunciado.

Quanto à a função principal `k-means`, aloca-se dois arrays para guardar a distância dos pontos aos clusters, dois para guardar a soma das coordenadas dos pontos em cada cluster e um para contar o número de pontos em cada cluster, sendo que nestes três últimos, se inicializa tudo a zero.

Seguidamente, começa-se o loop principal, que servirá para verificar se um ponto pertence a um dado cluster através da distância euclidiana. Ao somar todas as coordenadas pertencentes a um dado cluster, divide-se pelo número de elementos que lhe pertence, obtendo-se o seu novo centro.

De seguida, a função `has_converged` verificará se houve alterações nos centros dos clusters, comparando-os com os seus valores antigos.

Finalmente, a função principal, ou seja, a `main`, apenas irá iterar, verificando constantemente se os centros dos clusters não mudam.

Após o algoritmo correr, observou-se que os valores obtidos eram exatamente iguais aos valores esperados, indicando que o algoritmo foi desenvolvido corretamente.

### B. Alterações e Justificações

Sendo esta a fase principal do relatório, o grupo teve de adotar um conjunto de regras para a realização da mesma, nomeadamente que métricas se iriam documentar. Para uma maior compreensão dos resultados, optou-se por verificar o (1) tempo de execução, o (2) número de ciclos de relógio, o (3) número de instruções, o (4) CPI e (5) misses de loading na cache L1. O número de pontos será sempre 10 000 000 e o número de clusters será 4.

De acordo com as regras acima mencionadas, antes sequer de se alterar o código, documentou-se todas as as métricas para o algoritmo original.

A partir dos resultados na tabela 1, é possível ver que O2 é a otimização que reduz o tempo ao máximo. Em termos de misses, verifica-se que todas têm aproximadamente o mesmo, o que indica que se terá de fazer otimizações quanto a acessos de memória.

*1) Localidade Espacial:* Na versão original, os pontos estavam separados em dois arrays: um para as coordenadas associadas ao x, e outro para o y. Tendo esta informação em conta, sempre que o algoritmo tenha de aceder a um único ponto, terá de procurar em dois arrays, podendo resultar em dois misses de uma só vez.

Para tentar combater a separação entre as coordenadas, criou-se uma estrutura `Ponto` com dois floats – um para o x e outro para o y.

Ao executar o algoritmo, verificou-se que, em média, os tempos de execução entre esta nova versão e a versão anterior melhoraram ligeiramente.

*2) Menos alocações de memória:* Infelizmente, o grupo apercebeu-se de algumas falhas no algoritmo original que poderiam ter sido evitadas, embora esta fase de otimização tenha demonstrado o quão pesados são os acessos a memória.

Na versão original, na função `k_means`, o grupo alocou N floats a duas variáveis. O propósito destes arrays era guardar todas as distâncias dos pontos aos centros dos clusters.

Claramente isto foi um lapso do grupo, já que as distâncias são apenas usadas temporariamente dentro do loop, logo a solução para este problema foi algo tão básico como retirar o array e apenas ter uma variável temporária dentro do loop for mais aninhado.

Nos resultados, esta fase está em junção com a versão ”Localidade Espacial”. Infelizmente não se documentou a diferença entre as duas versões, mas é possível afirmar que

esta mudança simples retirou cerca de 1 segundo ao tempo de execução em comparação com o algoritmo original.

*3) Vetorização:* Para se vetorizar o algoritmo, teve de se seguir por um conjunto de regras. De todas as que existem, apenas duas não estavam a ser cumpridas.

Dentro do loop mais aninhado, observou-se uma condição *if-else*. Sendo isto um inibidor de vetorização, transformou-se para duas máscaras, sendo estas:

```
min_index = dist < min ? j : min_index;
min = dist < min ? dist : min;
```

Ao observar com mais atenção, apercebeu-se que apenas existe uma dependência da variável min. Logo, optou-se por guardar as k distâncias que um ponto poderá ter para cada cluster, e separou-se o loop em 3 diferentes.

O primeiro foca-se em obter todas as distâncias, o segundo em obter o mínimo, sendo que este loop não poderá ser vetorizado; por último, o terceiro apenas descobrirá o índice do mínimo.

Quanto ao último loop, utilizou-se a seguinte máscara:

```
min_index = dist[j] == min? j : min_index;
```

Optou-se por separar o código acima apresentado do loop que descobre o mínimo, para que o loop com dependências seja o mais curto e simples possível. Como é possível observar, este código não apresenta dependências, podendo ser vetorizado também, tal como o primeiro loop.

Ao efetuar esta mudança, os resultados irão diferir um pouco dos resultados mencionados no enunciado, pois caso um ponto esteja à mesma distância de dois clusters, irá escolher o de índice maior. O grupo considera que foi uma alteração necessária, pois ao efetuar os testes com e sem vetorização, a diferença de desempenho foi demasiado grande para ser ignorada. Em anexo irão estar imagens dos resultados, acompanhados pela porção de código que os gerou.

Quanto aos resultados desta fase, o tempo de execução e o número de ciclos era muito volátil. O tempo poderia chegar a 6 segundos, como também poderia demorar apenas 3.7 segundos. Nos resultados foi apresentado o tempo melhor que se observou.

### C. Algoritmo Final

Com todas as alterações feitas na fase anterior, o grupo considerou o algoritmo com vetorização o algoritmo final. Apesar de já não se ter mais alterado o código, ainda se tentou mais algumas otimizações com flags na compilação.

Primeiro testou-se com *unrolling*, algo que os docentes ensinaram e foi possível observar uma melhoria grande no desempenho. O tempo de execução e número de ciclos também ficaram mais estáveis, não havendo grande variação nos mesmos.

Por último, após uma pesquisa, ainda se adicionou umas otimizações às funções matemáticas, o que resultou num melhor desempenho.

## III. Resultados

Resultados obtidos com otimização -O2, N=100000 e K=4.

Na tabela abaixo encontram-se os resultados obtidos nas diferentes fases do projeto. Utilizando apenas as ferramentas lecionadas pelos docentes, é possível observar que se obteve um tempo de execução com cerca de 3.4 segundos.

É possível observar que o tempo de execução e o número de ciclos estão bastante correlacionados, sendo que quando um desce, o outro também.

Os L1-dcache-load-misses apenas se alteraram com a modificação do código, nunca pelas flags utilizadas na compilação.

Até à otimização de vetorização, focou-se mais em paralelismo a nível de instrução, logo o CPI irá ser mais pequeno. Ao fazer unrolling, observa-se que o CPI aumenta. Isto deve-se a que com unrolling, o número de instruções irá diminuir significativamente, pois irá remover instruções de controlo dos ciclos, mas não irá fazer mais paralelismo.

Em anexo, estará uma imagem com os resultados obtidos da melhor execução feita.

## IV. Conclusões e Trabalho Futuro

Por conclusão, o grupo pode sedimentar conhecimentos acerca de paralelismo entre instruções, e como é extremamente importante ter todas estas técnicas em conta para projetos futuros.

TABLE I
Resultados obtidos em versões diferentes de compilação.

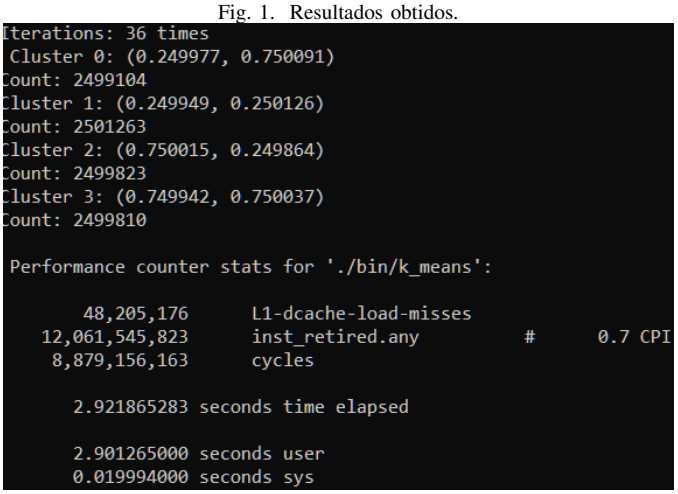
| Versão | Otimização | Tempo de Execução | #CC | #Instruções | CPI | L1-dcache-load-misses |
|---|---|---|---|---|---|---|
| Algoritmo Original | -O0 | 23,990 | 75 995 836 017 | 142 481 467 926 | 0.5 | 164 955 503 |
| | -O1 | 6,030 | 18 711 555 449 | 35 748 473 269 | 0.5 | 166 613 408 |
| | -O2 | 5,761 | 17 758 735 260 | 34 547 425 747 | 0.5 | 168 151 136 |
| | -O3 | 9,383 | 30 307 245 306 | 22 041 702 849 | 1.4 | 163 694 206 |
| | -Ofast | 8,418 | 26 947 752 604 | 22 110 487 072 | 1.2 | 165 795 375 |
| Localização Espacial | -O2 | 4,528 | 13 830 551 150 | 29 115 249 780 | 0.5 | 53 604 235 |
| Vetorização | -O2 -ftree-vectorize -mss4 | 3,797 | 12 326 785 989 | 27 391 340 282 | 0.5 | 48 405 545 |
| Unrolling | -O2 -ftree-vectorize -msse4 -funroll-loops | 3.433 | 10 613 597 626 | 16 563 725 162 | 0.6 | 48 304 610 |
| Otimizações Extras | -O2 -ftree-vectorize -msse4 -mavx2 -fprofile-use -funroll-loops –fast-math -funsafe-math-optimizations | 2.921 | 8 879 156 163 | 12 061 545 823 | 0.7 | 48 205 176 |

## V. Anexos

### A. Resultados Finais

A seguinte imagem representa os resultados obtidos na melhor execução, sendo esta com as flags todas.

Fig. 1. Resultados obtidos.




É de notar que os resultados nos números de pontos de cada cluster está ligeiramente diferente do que foi pedido no enunciado, sendo explicado com maior profundidade no anexo B.

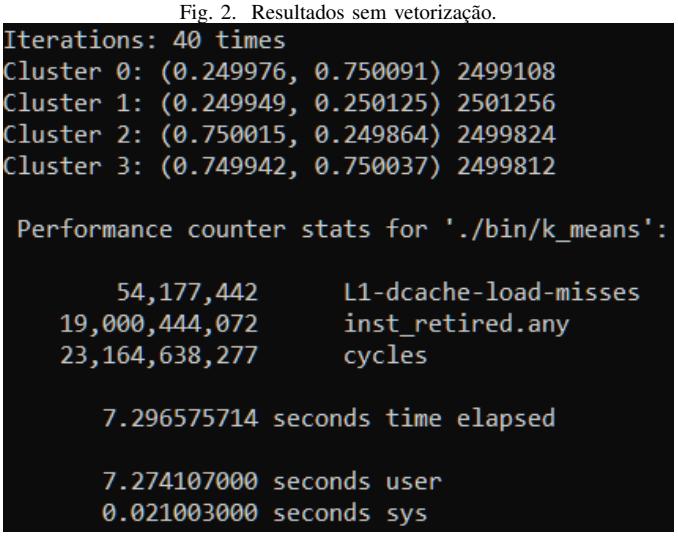
### B. Diferenças no número de pontos em cada cluster

Como já foi mencionado ao longo do relatório, os resultados obtidos foram ligeiramente diferentes dos que foram pedidos no enunciado.

O grupo conseguiu cumprir esse requisito, sendo apresentado a prova na seguinte imagem:

Estes resultados foram obtidos, pois quando se obteve o índice mínimo, este fica com o menor índice possível. Ou seja, se um ponto estiver à mesma distância de dois ou mais clusters, este estará associado ao cluster de índice menor.

Fig. 2. Resultados sem vetorização.




Os resultados acima foram obtidos com a seguinte porção de código:

```
if (dist[j] == min)
{
    min_index = j;
    break;
}
```

É de notar que havia outras formas de se realizar esta porção de código, mas em todas que o grupo se lembrou, não poderia haver vetorização devido à condição. Aliás, na tabela 1, os resultados antes da vetorização foram melhores do que a figura 2, mas mesmo assim, estavam piores que a figura 3.

Alterou-se apenas esta condição pela seguinte máscara:

```
min_index = dist[j] == min? j : min_index;
```

Esta máscara fará essencialmente o mesmo, com a exceção de que irá atribuir o índice maior, pois irá verificar todos as distâncias.

Como o algoritmo está a seguir as regras, nomeadamente a de associar o ponto ao cluster mais próximo, não especificando se terá de ser com o índice menor, maior, ou algo entre, o grupo decidiu implementar a solução com maior vetorização, obtendo os resultados abaixo.

Fig. 3. Resultados com vetorização.