Universidade do Minho

Mestrado Integrado em Engenharia Informática

Paradigmas de Computação Paralela

Otimização do Algoritmo *Bucket Sort*, recorrendo a OpenMP

Gonçalo Esteves
.(A85731).

António Gonçalves
.(A85516).

6 de dezembro de 2020

1 Introdução

Neste primeiro trabalho prático decidimos desenvolver e analisar o algoritmo de ordenação *Bucket Sort*, utilizando *OpenMP* para paralelizar o mesmo. Este algoritmo é utilizado para ordenar vetores de inteiros, começando por dividir os valores pelos *buckets*, tendo estes intervalos de valores associados (0-10, 11-20, ..., por exemplo). Em seguida, cada *bucket* é ordenado, e finalmente os valores são inseridos no vetor de forma ordenada.

Iremos começar por explicar as principais diferenças entre a nossa implementação sequencial e paralela, analisando de seguida os resultados obtidos para vários tamanhos diferentes, bem como os *speedups* calculados.

2 Implementação do Código

A ordenação de um *array* recorrendo a "baldes", também conhecida como *Bucket Sort*, baseia-se no princípio de **divisão e conquista**: a divisão do problema inicial em muitos outros mais pequenos facilita a resolução final do problema. Neste caso, o *array* recebido é dividido por diversos *buckets* ou "baldes", sendo mais fácil ordenar cada um destes e posteriormente reinserir os elementos ordenadamente no *array* final.

Devido à natureza embaraçosamente paralela do algoritmo, é fácil deduzir que a paralelização do código do mesmo poderá torná-lo mais eficiente.

2.1 Versão Sequencial

Por forma a implementar um algoritmo de ordenação recorrendo a "baldes", são necessárias três fases:

- Alocação de memória para cada *bucket*;
- Distribuição dos elementos do *array* pelo *bucket* respetivo;
- Ordenação de cada um dos *buckets* e reinserção dos elementos no *array*.

Tal como pode ser observado no ficheiro *bucket_sort_seq.c*, por forma a implementar as três fases referidas recorre-se a três ciclos *for*, um para cada fase: no primeiro, percorre-se todos os *buckets* e aloca-se a memória necessária para cada um; no segundo, percorre-se todos os elementos do *array* inicial e distribui-se os mesmos pelos *buckets* respetivos; no terceiro, percorre-se novamente todos os *buckets*, ordenando cada um deles e reinserindo os elementos de forma ordenada no *array* final. De realçar que, por forma a ordenadar cada "balde", recorreu-se ao algoritmo *merge sort* em deterimento do *quick sort*, uma vez que o primeiro escala melhor para *arrays* maiores, muito devido ao facto de não ser necessário recorrer ao uso de *pivots* para a ordenação do *array*.

2.2 Versão Paralela

Numa primeira abordagem à paralelização do código previamente produzido, optou-se por manter a estrutura base da versão sequencial, paralelizando apenas os ciclos utilizados, tal como pode ser consultado no ficheiro *proto_bucket_sort_par.c*. No entanto, observou-se que não era possível paralelizar de forma eficiente o ciclo responsável pela distribuição dos elementos pelos *buckets*. Tal se deve, principalmente, ao facto de ser permitida a criação de um número de *buckets* superior ao número de *threads* e, como tal, deve-se ter em conta os acessos em simultâneo tanto ao *array* inicial, bem como a cada um dos *buckets*, pois duas *threads* não poderão inserir em simultâneo no mesmo *bucket*. Manteve-se, também, o uso do algoritmo *merge sort* na ordenação de cada um dos *buckets*.

No entanto, os *speedups* obtidos nesta primeira abordagem não eram muito satisfatórios, o que levou a uma nova tentativa. Desta vez, tal como se pode ver em *bucket_sort_par.c*, optamos por limitar o número de *buckets* ao número de *threads*. Assim, cada *thread* é responsável por um único *bucket*, resolvendo todos os problemas de concorrência de dados, permitindo às *threads* que trabalhem efetivamente em paralelo.

Agora, cada *thread* aloca a memória necessária para o seu "balde", percorre o *array* inicial por forma a recolher os elementos que lhe pertencem, ordena o seu "balde"e insere os elementos no *array* final. É importante realçar que, nesta implementação, passou-se a recorrer à função **qsort** da biblioteca **stdlib.h**. Tal se sucedeu uma vez que a implementação criada do algoritmo *merge sort* recorre ao uso de dois arrays definidos estaticamente, L e M. Ao tentar ordenar um *array* inicial de grande dimensão, e devido à execução de múltiplas intâncias da função *mergesort*, não so devido à sua natureza recursiva, mas também devido à existencia de múltiplas *threads*, obtem-se *segmentation fault*, uma vez que a stack ficava sem espaço para novas estruturas. Ao alocar L e M com *malloc*, apesar de resolver os *segmentation fault*, detriorava os tempos de execução imenso, comparando com os obtidos recorrendo a *qsort*.

Posto tudo isto, assumiu-se esta nova abordagem como a implementação final do *Bucket Sort*, tendo obtido otimizações satisfatórias em diversos testes, apresentados mais tarde.

2.2.1 Partição do problema e dos dados a processar

Tendo em vista a **decomposição dos dados** a processar, podemos afirmar que o paralelismo obtido aquando da consulta do *array* inicial, por forma a preencher os *buckets*, bem como o paralelismo no momento de preencher o *array* final recaem neste caso. No primeiro, o facto de o *array* apenas estar a ser lido permite que múltiplas *threads* o consultem ao mesmo tempo, podendo mesmo estar a consultar a mesma posição em simultâneo; já no segundo caso, o facto de que cada *thread* apenas escreve a partir de uma dada posição do *array* final permite garantir que não existirão duas *threads* a tentar escrever na mesma posição de memória.

Olhando para a **decomposição funcional**, podemos afirmar que um exemplo da mesma será a execução em paralelo da função *qsort*, aquando da ordenação dos diferentes *buckets*.

2.2.2 Problemas de escalabilidade

Tendo em conta os problemas de escalabilidade do algoritmo, há dois que são detetáveis:

- **Overhead de Sincronização**, que ocorre aquando do uso da chamada *malloc* para alocar a memória necessária a cada *bucket* (apesar de ser executado em paralelo, os *malloc's* ocorrem sempre sequencialmente);
- **Granularidade de paralelismo**, que ocorre aquando do percurso de todo o *array* inicial por todas as *threads* (na versão sequencial, era fácil distribuir todos os elementos do *array* pelos *buckets* numa só travessia do *array*, no entanto, na versão paralela, esta é a forma mais eficiente de o fazer).

3 Análise de resultados

Com as implementações sequencial e paralela do nosso algoritmo completas, efetuamos alguns testes para conseguirmos visualizar os ganhos entre as duas. Utilizamos o nodo 662 do *cluster* uma vez que este tem uma frequência fixa.

Após observar as características deste nodo, decidimos realizar testes para 4 tamanhos de *array* diferentes: **65536**, **524288**, **4194304** e **33554432** inteiros, para que seja possivel guardar o vetor nas *caches L1*, *L2*, *L3* e memória central, respetivamente. Para além disto, definiu-se o número de *threads* a utilizar (e, consequentemente, o número de *buckets*) como **2**, **4**, **8**, **16**, **32**, **48** e **64**. Os gráficos com os tempos obtidos, para o algoritmo sequencial e paralelo, usando cada um destes tamanhos encontram-se nos anexos.

Observando o gráfico apresentado em seguida, que compara os *speedups* obtidos com a versão paralela para os diferentes tamanhos (os *speedups* foram calculados tendo em conta o melhor tempo da versão sequencial e paralela para um mesmo número de *buckets*), é facilmente visivel que, há medida que o tamanho do vetor aumenta, o valor de *speedup* obtido também aumenta.

No entanto, deteta-se que após um certo número de *buckets*, o *speedup* obtido começa a diminuir. Isto acontece porque a criação de *buckets* adicionais, bem como a criação da *thread* a ele associada, aumenta o peso do programa, não compensando os ganhos (quão maior for o número de *buckets* para um mesmo tamanho de vetor, menos elementos possui cada *bucket*, podendo a maior distribuição da carga não compensar os custos associados à criação de cada *bucket*).

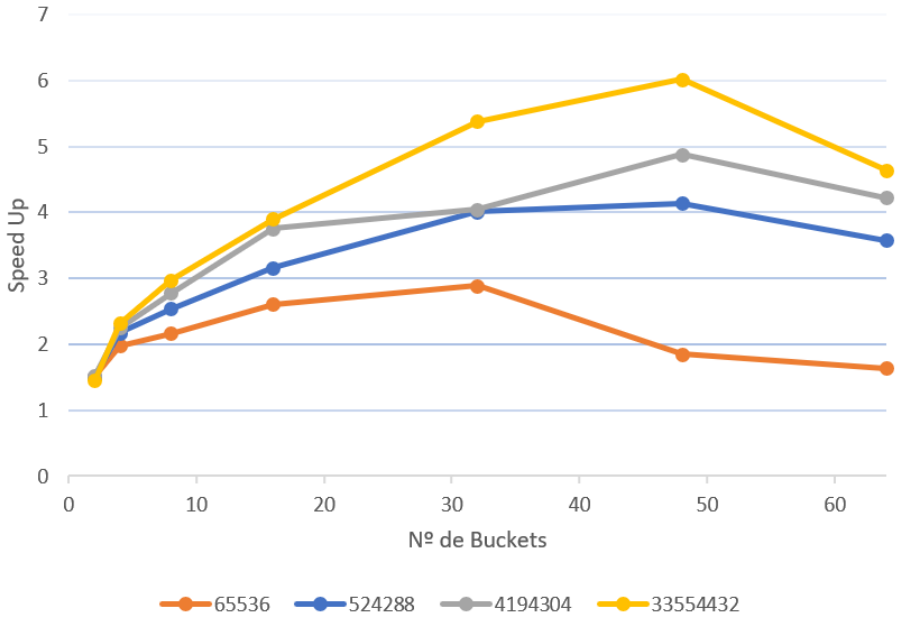

Figura 1: *Speedups* obtidos para cada tamanho

3.1 Vetor com Tamanho 65536

Obtem-se um *speedup* razoável, no entanto, pouco aumenta com o aumentar do número de *buckets*, sendo que a partir de 32 *buckets* o valor começa a diminuir. No entanto, se compararmos o melhor tempo sequencial (independentemente do número de *buckets*) com os tempos paralelos, verificamos que o valor ótimo de *buckets* é 16 (estes valores estão calculados nos anexos).

3.2 Vetor com Tamanho 524288

Os valores obtidos são superiores aos do tamanho anterior, notando-se a maior eficiência. O *speedup* máximo é obtido quando utilizamos 48 *buckets*, no entanto, a variação é pouca, comparando com o uso de 32. Apesar disto, e mais uma vez, usando o valor ótimo sequencial (ignorando o número de *buckets*), a quantidade ótima de *buckets* é 16.

3.3 Vetor com Tamanho 4194304

Mais uma vez, os valores de *speedup* são superiores, sendo que se nota uma maior vantagem no uso de 48 *buckets* começando, no entanto, a partir daqui a diminuir. Novamente, utilizando o melhor tempo sequencial obtido, averiguamos que a quantidade ótima de *buckets* é, outra vez, 16.

3.4 Vetor com Tamanho 33554432

Tem os maiores *speedups* de todos os tamanhos, mas, mesmo assim, o valor começa a diminuir para mais de 48 *buckets*. Por fim, e tendo em conta o melhor tempo sequencial obtido, concluímos que, neste caso, o número ótimo de *buckets* a utilizar é 32.

4 Conclusão

Acreditamos que tenham sido atingidos os principais objetivos deste trabalho, sendo eles o desenvolvimento e a análise de resultados do algortimo por nós escolhido. Para além disso, conseguimos desenvolver as nossas capacidades e raciocínio referentes à computação paralela e à utilização da ferramenta *OpenMP*.

Anexos

Tamanho 65536

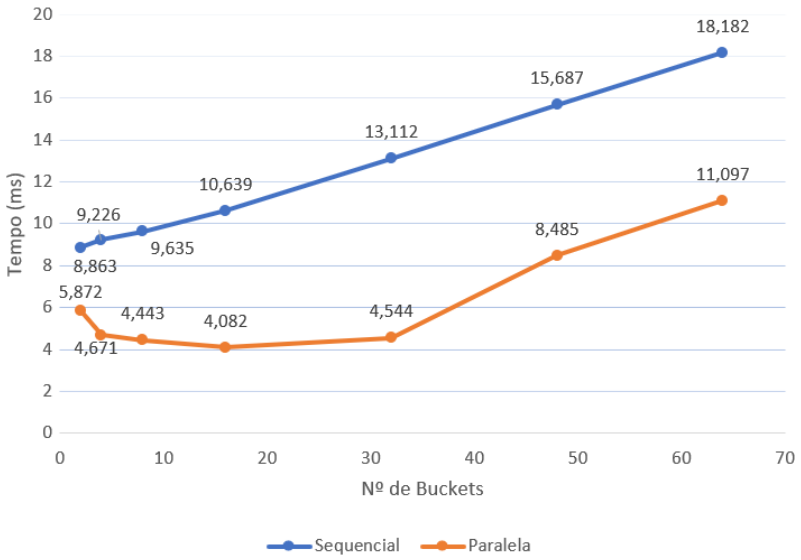

Figura 2: Tempos obtidos para um *array* de tamanho 65536

	2	4	8	16	32	48	64
Speedup(*)	1,509	1,975	2,168	2,606	**2,886**	1,849	1,638
Speedup(**)	1,509	1,897	1,995	**2,171**	1,951	1,045	0,799

Tabela 1: *Speedups* obtidos comparando o melhor tempo paralelo para um dado número de *buckets* com (*) o melhor tempo sequencial para o mesmo número de *buckets*, (**) bem como o melhor tempo sequencial geral

Tamanho 524288

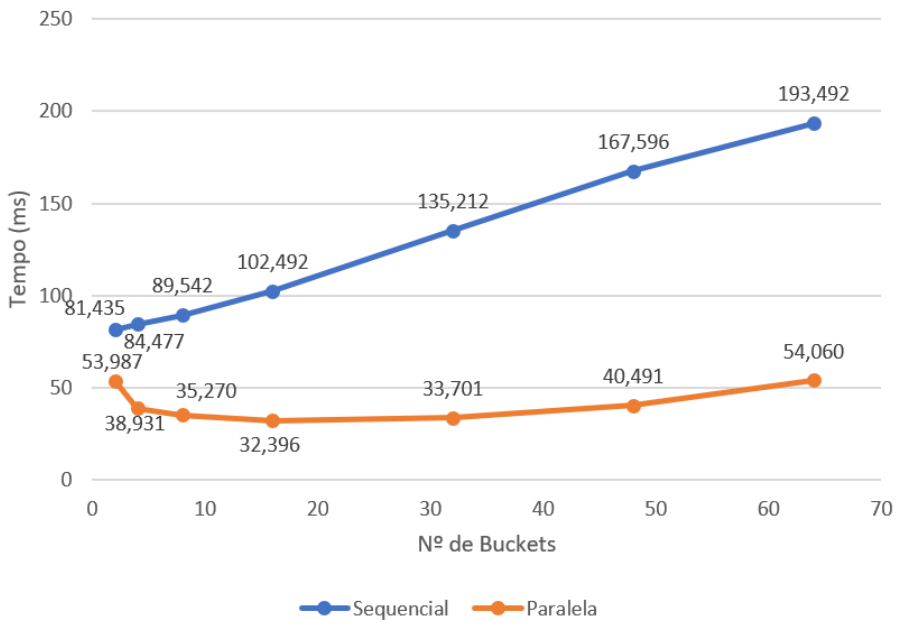

Figura 3: Tempos obtidos para um *array* de tamanho 524288

	2	4	8	16	32	48	64
Speedup(*)	1,508	2,170	2,539	3,164	4,012	**4,139**	3,579
Speedup(**)	1,508	2,091	2,309	**2,514**	2,416	2,011	1,506

Tabela 2: *Speedups* obtidos comparando o melhor tempo paralelo para um dado número de *buckets* com (*) o melhor tempo sequencial para o mesmo número de *buckets*, (**) bem como o melhor tempo sequencial geral

Tamanho 4194304

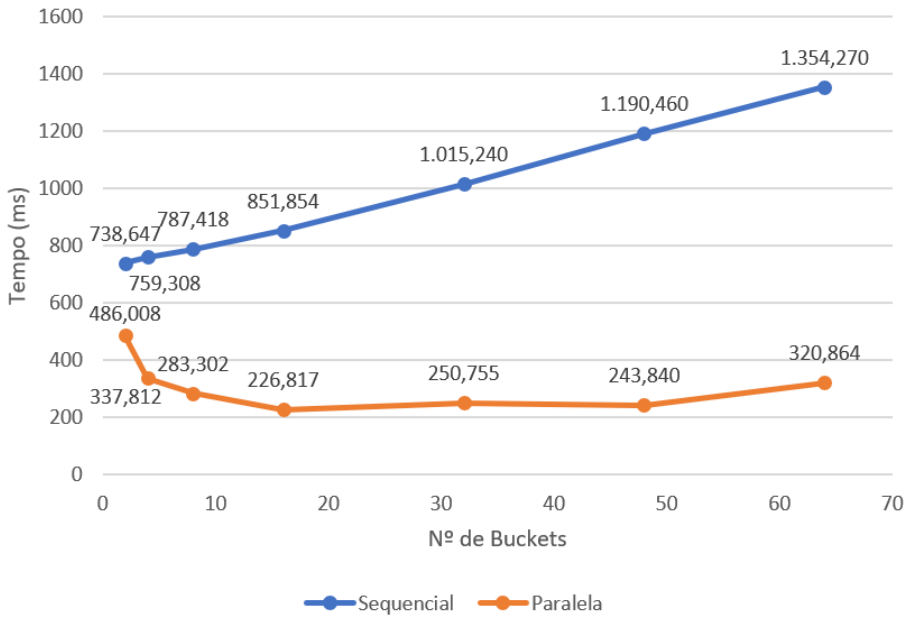

Figura 4: Tempos obtidos para um *array* de tamanho 4194304

	2	4	8	16	32	48	64
Speedup(*)	1,520	2,248	2,779	3,756	4,049	**4,882**	4,221
Speedup(**)	1,520	2,187	2,607	**3,257**	2,946	3,0292	2,302

Tabela 3: *Speedups* obtidos comparando o melhor tempo paralelo para um dado número de *buckets* com (*) o melhor tempo sequencial para o mesmo número de *buckets*, (**) bem como o melhor tempo sequencial geral

Tamanho 33554432

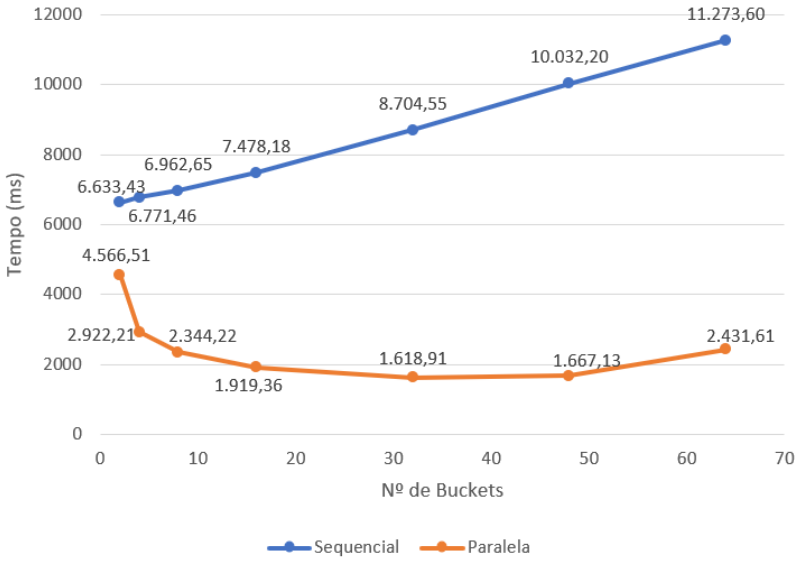

Figura 5: Tempos obtidos para um *array* de tamanho 33554432

	2	4	8	16	32	48	64
Speedup(*)	1,453	2,317	2,970	3,896	5,377	**6,018**	4,636
Speedup(**)	1,453	2,270	2,830	3,456	**4,097**	3,979	2,728

Tabela 4: *Speedups* obtidos comparando o melhor tempo paralelo para um dado número de *buckets* com (*) o melhor tempo sequencial para o mesmo número de *buckets*, (**) bem como o melhor tempo sequencial geral