Escola de Engenharia
Universidade do Minho

Otimização de um algoritmo de ordenação

Computação Paralela

Gilberto Cunha
Tomás Carneiro

7 de junho de 2021

full code here

1. Introdução

No trabalho anterior foi feita uma melhoria ao algoritmo **Bucket Sort**, de forma a tornar a sua versão sequencial mais rápida. A melhor versão desenvolvida consiste na utilização do `mergesort` para ordenar os arrays de cada um dos baldes. De modo a melhorar ainda mais este algoritmo foi substituído o `mergesort` pelo `quicksort`.

No entanto, como a execução deste algoritmo é sequencial, este ainda não aproveita todas as threads físicas do CPU, de forma a correr várias tarefas em simultâneo e reduzir ainda mais o tempo de execução do nosso algoritmo.

De forma a integrar este algoritmo previamente desenvolvido com a execução em múltiplas threads, foi então feita uma paralelização do mesmo.

2. Paralelização do Bucket Sort

Foram feitos paralelismos distintos em diferentes segmentos do algoritmo anterior:

- **A criação dos baldes**: Distribuir pelas threads a criação dos diferentes baldes;
- **A ordenação por balde**: Permitir que várias threads possam ordenar baldes em simultâneo;
- **A união dos baldes após estarem ordenados**: Distribuir baldes por diferentes threads, de forma a paralelizar a união dos diferentes arrays contidos em cada balde.

2.1. Paralelização da criação dos baldes

Para a criação dos baldes, diferentes threads alocam espaço na memória correspondente àquele ocupado por cada balde. Para este algoritmo funcionar para qualquer distribuição de elementos no array, cada balde tem a mesma dimensão que o array original.

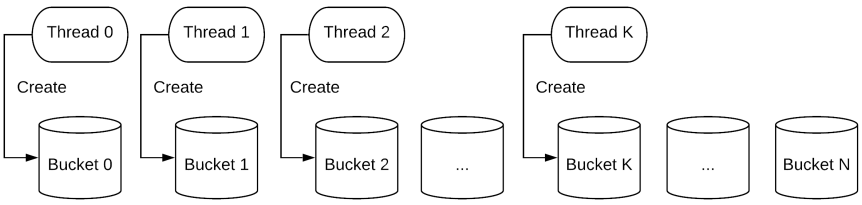

Fig. 1: Paralelização da alocação de memória dos baldes no Bucket Sort

2.2. Paralelização da Ordenação dos baldes

Para agilizar o processo de ordenação, o maior ganho deste algoritmo deverá ser obtido pela ordenação em paralelo dos diferentes baldes. Isto é, cada balde deverá ser ordenado por uma thread, podendo haver múltiplas threads a ordenar um balde cada, em simultâneo.

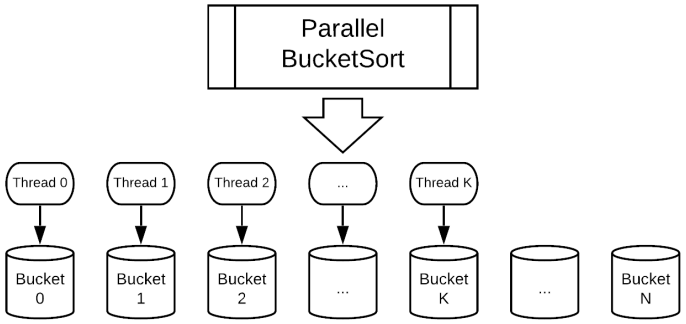

Fig. 2: Paralelização do algoritmo Bucket Sort

Cada um destes baldes é independente dos restantes e neste processo não é necessária a troca de informação entre threads, não havendo então nenhuma *data-race*.

2.3. Paralelização da união dos baldes

Uma vez que os baldes já se encontram ordenados, resta agora apenas copiar os elementos dos baldes para o array final. Para paralelizar este passo cada thread faz a inserção dos elementos de cada balde.

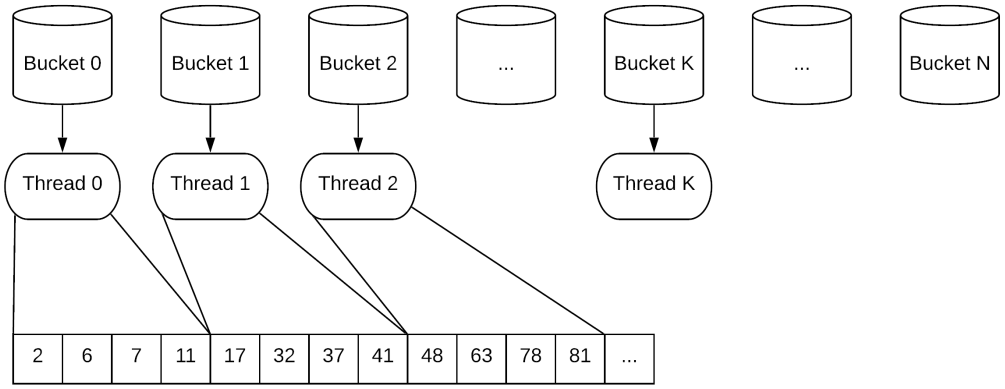

Fig. 3: Paralelização da união dos buckets para o array final

Como a posição de cada elemento no array final pode ser definida à partida, e nenhum dos elementos de baldes diferentes irá ocupar a mesma posição, este passo também não cria *data-races*.

2.4. Outras possíveis paralelizações

Dos quatro passos constituintes do *Bucket Sort* (criação dos baldes, distribuição dos elementos do array pelos baldes, ordenação dos baldes e união de elementos no array final) apenas a distribuição dos elementos nos baldes não foi paralelizada nas implementações apresentadas.

Para paralelizar este passo é necessário dividir o array original entre as threads, percorrer cada partição com threads distintas, determinar o balde correto para cada elemento e colocá-lo nesse balde.

No entanto, para cada uma destas partições, diferentes threads podem querer escrever ao ”mesmo tempo” no mesmo balde, criando então um data-race. De modo a evitar estas colisões entre threads a escrever no mesmo balde, é necessário criar um lock (ou usar um `pragma omp critical`) para cada balde antes de se fazer a escrita neste, apenas permitindo uma thread de cada vez fazer esta escrita.

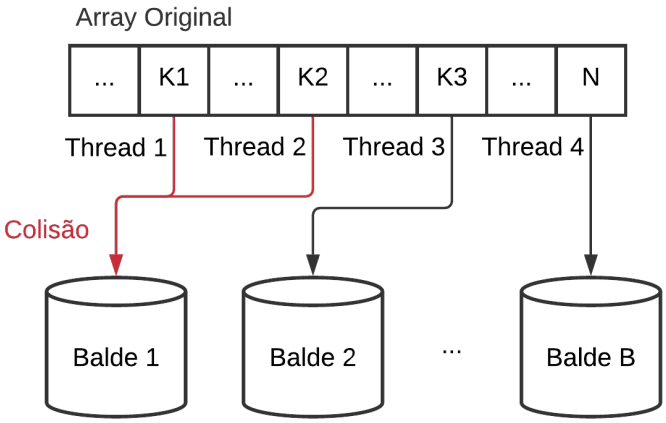

Fig. 4: Exemplo de uma colisão na paralelização da distribuição dos elementos do array original pelos baldes

Devido a este mecanismo de locks, bons ganhos na paralelização deste passo apenas serão obtidos para um número de baldes elevado (onde as colisões serão menos frequentes), mas pelos testes que foram feitos e apresentados mais à frente, o nosso algoritmo funciona melhor para um número reduzido de baldes, pelo que a paralelização deste passo não foi incluída no nosso algoritmo final.

3. Obtenção de Dados

De acordo com as possíveis paralelizações acima descritas, foram então testados os seguintes algoritmos:

1. **PB**: Paralelizar a ordenação pelos baldes
2. **PBPU**: Paralelizar a ordenação pelos baldes, a criação dos baldes e a união dos elementos dos baldes no array final
3. **SB**: Bucket Sort sequencial

E foram variados os diferentes parâmetros do programa:

- Tamanho total do array
- Número de baldes utilizados
- O algoritmo BucketSort a usar (dos enumerados acima)
- O número de threads utilizadas

Para obter métricas do nosso programa de acordo com todos os diferentes parâmetros enumerados acima, foi feita uma **grid search**. Isto é, definiu-se uma lista de valores que queríamos testar para cada parâmetro e efetuaram-se testes para todas as combinações de parâmetros possíveis, guardando a configuração e as métricas em formato `.csv`.

Deste modo, estes dados podem posteriormente ser consultados e processados para análise estatística e obtenção de resultados.

Utilizando o `Papi`, foram obtidas as seguintes métricas para cada execução:

- Tempo de execução (em μs)
- Número de Instruções
- Número de Ciclos

Com estas métricas e configurações podemos também inferir o CPI e ganho em tempo de execução.

Para cada configuração do programa, foram feitas 50 execuções distintas com o array inicial aleatoriamente ordenado, para permitir medir a média e o desvio padrão das métricas obtidas com alguma significância estatística.

4. Métricas e discussão de resultados

A grande vantagem de paralelizar o nosso algoritmo é permitir que o processador execute múltiplas operações em simultâneo. Para verificar o quanto a nossa versão paralelizada do Bucket Sort melhorou a performance da versão sequencial, foi obtido o seu ganho em tempo de execução ao variar o número de threads utilizadas:

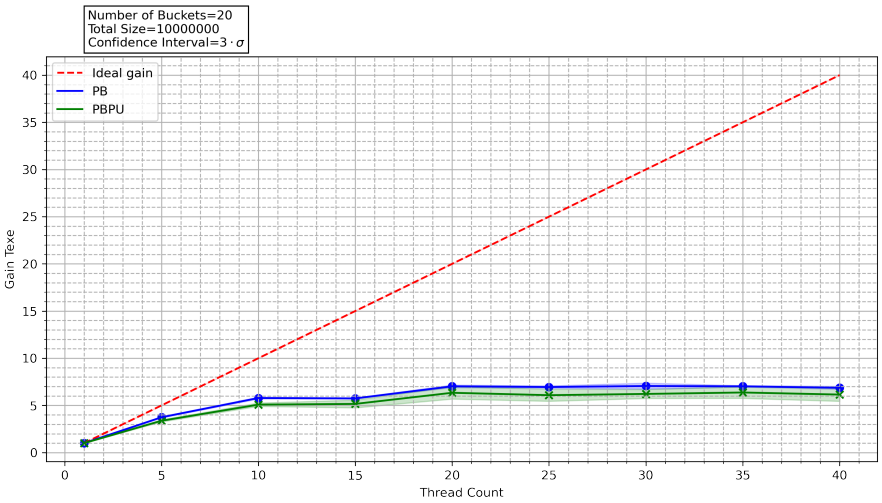

Fig. 5: Ganho em tempo de execução contra ganho ideal

Como se pode verificar, com o aumento do número de threads conseguimos também melhorar o tempo de execução do nosso algoritmo em cerca de 7 vezes. Nota-se também que este ganho se mantém praticamente inalterado a partir das 20 threads. A figura 5 está também representada com um intervalo de confiança de 3 desvios padrões, medidos para cada ponto utilizando as 50 runs por configuração.

Este curva de ganho, no entanto, não é ideal. Um dos motivos pelos quais este é o caso deve-se a nem todo o código se encontrar paralelizado. Para obtermos uma referência da fração do nosso código que foi paralelizada, utilizou-se a lei de Amdahl:

$$S_M = \frac{1}{(1-p) + \frac{S}{p}}$$

Onde S_M é o ganho medido do algoritmo, S é o ganho teórico da parte paralelizada e p é a fração de código paralelizada que queremos encontrar. Reescrevendo em ordem a p (e determinando o seu desvio padrão δp):

$$p(S_M) = \frac{S}{1-S}\left(\frac{1}{S_M} - 1\right), \quad \delta p(S_M) = \left|\frac{\partial p}{\partial S_M}\delta S_M\right| = \frac{S}{S_M^2(1-S)}\delta S_M$$

Sabendo o ganho ideal (é igual ao número de threads) e utilizando o ganho medido apresentado nas figuras 5 e 6, podemos então obter um gráfico relativo a esta fração do código paralelizado.

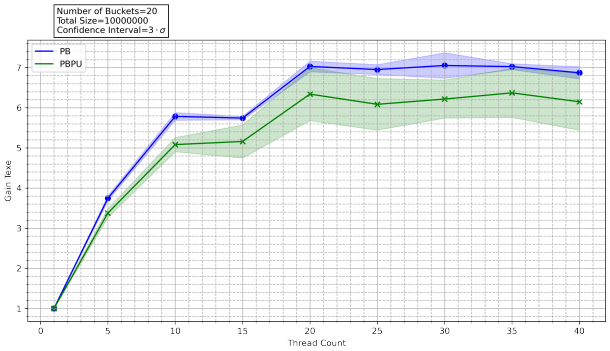

Fig. 6: Ganho em tempo de execução

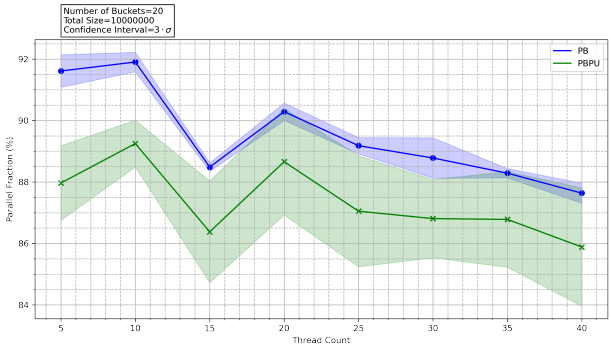

Fig. 7: Fração paralelizada do algoritmo com o número de threads

Pela figura 7 conseguimos verificar que por volta de 90% do código é paralelizado em ambas as versões implementadas. Embora a versão **PBPU** também paralelize a criação dos baldes e a união dos elementos ordenados, estas tarefas são consideravelmente menos dispendiosas do que a ordenação, sofrendo mais com o overhead da sua paralelização e consequentemente reduzindo ligeiramente o seu ganho em tempo de execução.

Para demonstrar o motivo pelo qual uma percentagem tão elevada de código paralelizado apenas origina um ganho de 7 (onde o teórico é de 40), analisemos a Lei de Amdahl:

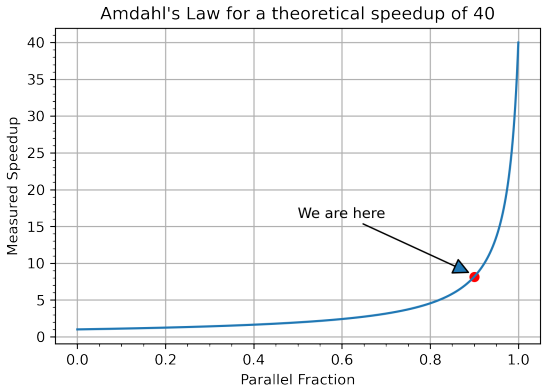

Fig. 8: Representação da Lei de Ahmdal

Pela figura ao lado, podemos concluir que com um aumento da fração paralelizada do código para além dos 0.9 (ou 90%), o aumento no ganho é muito superior a um aumento linear.

Deste modo, pequenos aumentos na fração paralelizada do código podem dar origem a elevados aumentos no ganho medido.

Pela análise do tempo de execução com o tamanho do array original, podemos observar que nos algoritmos de ordenação paralelos, o tempo de execução cresce mais lentamente.

Deste modo, os algoritmos paralelos desenvolvidos apresentam uma melhor escalabilidade face ao aumento do número de elementos do array original.

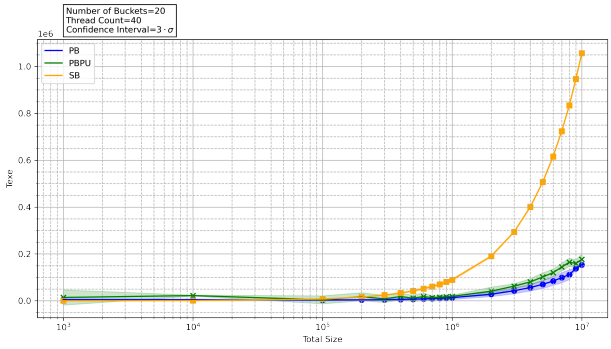

Fig. 9: Tempo de execução com o tamanho do array

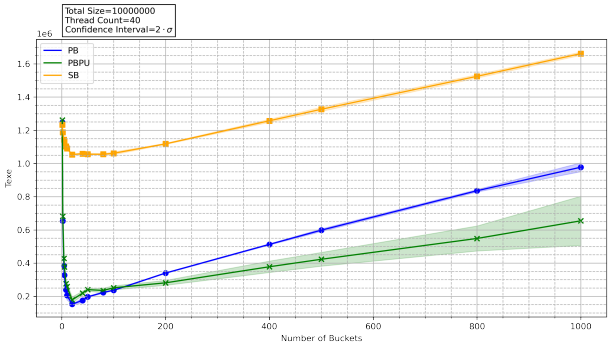

Fig. 10: Tempo de execução com o número de baldes

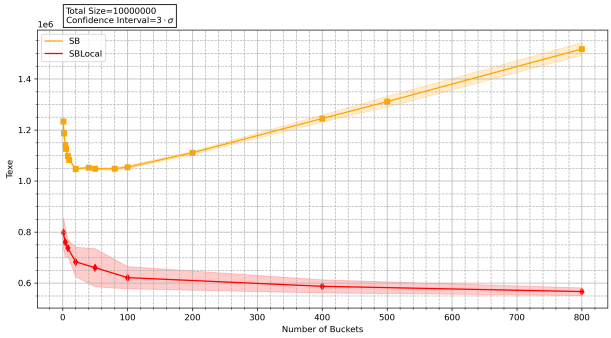

Fig. 11: Comparação do algoritmo sequencial local e na `search1`

Como podemos visualizar pelo gráfico da figura 10, os tempos de execução dos algoritmos sequencial e paralelo diferem bastante. No caso do algoritmo paralelo, este obtém os seus melhores resultados para 20 baldes, a partir dos quais o tempo de execução aumenta ligeiramente (daí não ser paralelizada a distribuição dos elementos pelos baldes).

Analisando a figura 11, podemos concluir que com o aumento do número de baldes, localmente, o algoritmo sequencial diminuiu sempre o seu tempo de execução. Não obstante, realizando os mesmos testes na máquina `search1`, esta diminuição não foi observada (os motivos desta diferença não nos são claros, requererendo análises adicionais para os detetar).

Para além disso, pode-se também verificar que o tempo de execução da máquina local é substancialmente inferior, potencialmente devido a esta ter uma melhor *single-core performance* (*Ryzen 5 3600*).

Por fim, devido à frequência fixa do CPU da `search1`, pode-se também verificar que o desvio padrão dos seus tempos de execução é consideravelmente inferior, pelo que estes tempos de execução sofrem menos flutuação devido a esta maior estabilidade.

5. Conclusão

Primeiramente, conseguiu-se diminuir o tempo de execução da nossa implementação do **Bucket Sort** em cerca de 7 vezes, comparando a sua melhor versão sequencial à melhor versão paralelizada.

Analisou-se que o ganho obtido (7 vezes) é muito inferior ao ganho teórico (40 vezes). Utilizando a Lei de Amdahl, pôde-se concluir que esta disparidade se deve à percentagem de código paralelizado (por volta de 90%). Um aumento nesta percentagem iria aumentar significativamente o ganho obtido em tempo de execução.

De modo a tentar aumentar esta percentagem de código paralelizado, paralelizaram-se também as tarefas de criação de baldes e união de elementos ordenados para o array final. No entanto, devido a estas tarefas serem pouco dispendiosas comparativamente à ordenação dos baldes e ao overhead inerente à sua paralelização, o seu ganho viu um ligeiro decréscimo.

Concluiu-se também que os algoritmos paralelos desenvolvidos escalam melhor face ao aumento do array original, dado que o seu tempo de execução aumenta mais lentamente do que no algoritmo **Bucket Sort** sequencial desenvolvido.

Por fim, verificou-se que para os algoritmos paralelos, números de baldes superiores a 20 não trouxeram benefícios para o seu tempo de execução, contrariamente ao algoritmo sequencial realizado localmente, que viu sempre um decréscimo deste tempo de execução, embora muito reduzido.