Escola de Engenharia
Universidade do Minho

Otimização do BucketSort com o MPI

Computação Paralela

Gilberto Cunha
Tomás Carneiro

7 de junho de 2021

full code here

1. Introdução

A forma usual e mais simples de conceber programas e algoritmos passa pelo desenvolvimento dos mesmos em termos sequenciais, isto é, onde quaisquer duas instruções são executadas uma após a outra.

Uma forma de acelerar certos programas sequenciais é a paralelização dos mesmos, permitindo a execução de múltiplas instruções em simultâneo. Para atingir este objetivo há dois conceitos fundamentais: os de **threads** e de **processos**.

Com a finalidade de desenvolver um algoritmo **Bucket Sort** paralelo, no trabalho anterior foi utilizado o `OpenMP` para controlo de múltiplas threads a partilhar memória num único processo.

Neste trabalho a estratégia difere: utilizar-se-á (com a mesma finalidade) o `MPI`, que permite o controlo de múltiplos processos, cada um com a sua própria memória, podendo comunicar apenas através de mensagens.

2. Paralelização do Bucket Sort

O Bucket Sort divide o array original em baldes para efetuar a sua ordenação. Os seus três passos principais são:

- Distribuição dos elementos do array pelos baldes
- Ordenação de cada balde
- União dos elementos de cada balde de volta ao array original

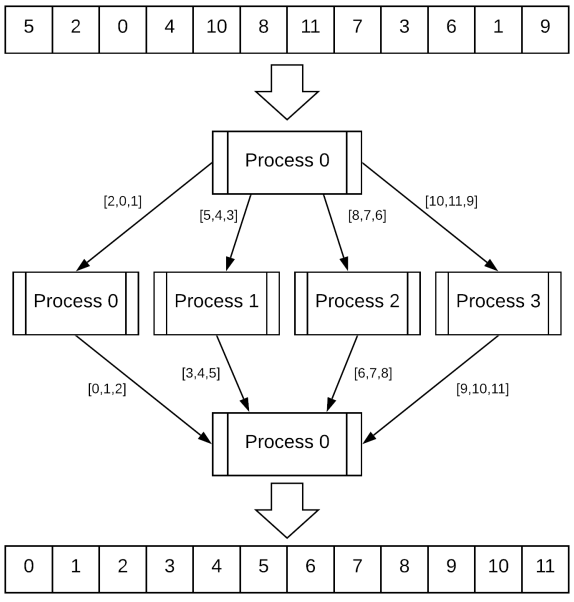

Fig. 1: Exemplo do funcionamento da implementação com diferentes processos

Numa versão paralela com o `MPI`, para diferentes processos poderem colaborar para o objetivo comum da ordenação do array original, os elementos deste array devem ser distribuídos entre os diferentes processos disponíveis. Desta forma, cada um dos processos deverá receber os seus elementos, executar o Bucket Sort sequencial e finalmente enviar os valores ordenados ao processo principal (ver figura 1).

Esta distribuição dos elementos do array pelos processos, no entanto, não é arbitrária. Semelhante à distribuição dos elementos por baldes no Bucket Sort, deve ser garantido que os elementos de um processo são necessariamente menores ou iguais aos dos processos seguintes. Caso contrário, os diferentes segmentos do array original estarão ordenados, mas o array em si continuará desordenado.

Exemplo: Considere o array 2, 1, 3, 0, a ser dividido pelos processos 0 e 1. Caso o processo 0 fique com os elementos 2, 1 e o processo 1 com 3, 0, o array ordenado final será 1, 2, 0, 3, que continua desordenado. Para uma correta distribuição, o processo 0 deverá ficar com os elementos 1, 0 e o processo 1 com 2, 3.

Para se efetuar esta distribuição de elementos do processo 0 para os restantes processos utilizou-se a função `MPI_Scatterv` (que permite enviar um número diferente de elementos para cada processo). Após ser feita a ordenação, os segmentos de array resultantes são reenviados para o processo 0 utilizando a função `MPI_Gatherv`.

3. Obtenção de Dados

Para efetuar o estudo da otimização do algoritmo paralelo relativamente ao sequencial é necessária a aquisição de métricas relativas aos tempos de execução.

Num programa **MPI**, no entanto, o tempo de execução tem três componentes distintas, sendo estas o tempo de troca de mensagens, o tempo de processamento e o tempo de espera:

$$t_{\texttt{total}} = t_{\texttt{msg}} + t_{\texttt{proc}} + t_{\texttt{wait}} \tag{1}$$

Deste modo, foram medidos utilizando a primitiva `MPI_Wtime` os tempos de mensagem, tempo de processamento e tempo total [1]. Utilizando estes três tempos é também possível inferir o tempo de espera utilizando a equação 1.

Para garantir que os resultados obtidos são representativos de várias configurações possíveis do array original, foram então medidas as médias e os desvios padrões das métricas acima referidas, ao longo de 50 execuções do algoritmo para inicializações aleatórias do array original, para cada uma das configurações do algoritmo testadas.

Foram então variados o tamanho do array, o número de baldes do Bucket Sort e o número de processos a serem utilizados.

Para obter as métricas para cada uma destas configurações foram definidas listas de valores para cada um dos parâmetros variados. Utilizando estas listas, obtiveram-se as métricas para todas as combinações possíveis de parâmetros, isto é, foi feita uma **grid search**. Estes dados foram armazenados num ficheiro `.csv`, posteriormente utilizado para fazer a análise estatística dos algoritmos e dos seus tempos de execução.

4. Métricas e discussão de resultados

4.1. Strong scaling e lei de Amdahl

O objetivo final da paralelização do algoritmo é a diminuição do seu tempo de execução. Para verificar esta diminuição foi então analisado (figura 2) o ganho do algoritmo com o aumento do número de processos, mantendo o tamanho do array fixo (**strong scaling**):

Desta análise, foi possível obter um ganho no tempo de execução do algoritmo de cerca de 4.5. Na figura 2 está também representado um intervalo de confiança de 3 desvios padrões.

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

$$S_M = \frac{1}{(1-p) + \frac{S}{p}} \tag{2}$$

[1] Todos os tempos medidos e apresentados são em segundos, tal como o output da primitiva `MPI_WTime`. Estes testes foram todos executados localmente numa máquina com um `Ryzen 5 3600`.

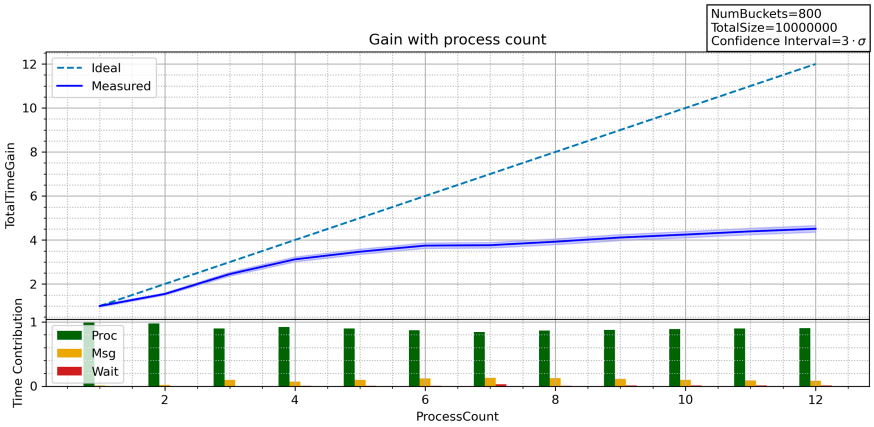

Fig. 2: Análise de *Strong scaling*

Onde S_M é o ganho medido experimentalmente, 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):

$$p(S, S_M) = \frac{S}{1-S}\left(\frac{1}{S_M} - 1\right), \quad \delta p(S, S_M) = \left|\frac{\partial p}{\partial S_M}\delta S_M\right| = \frac{S}{S_M^2(S-1)}\delta S_M \tag{3}$$

Sabendo o ganho ideal (é igual ao número de processos) e utilizando o ganho medido apresentado nas figuras 2 e 3, podemos então obter um gráfico relativo a esta fração do código paralelizado.

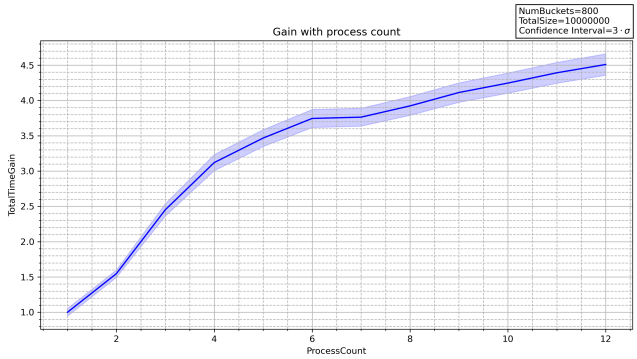

Fig. 3: Ganho em tempo de execução

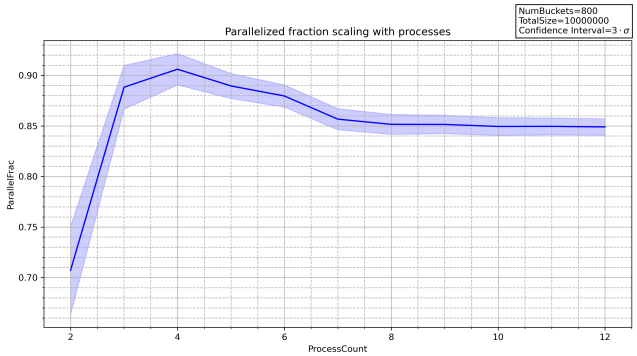

Fig. 4: Fração paralelizada do algoritmo com o número de processos

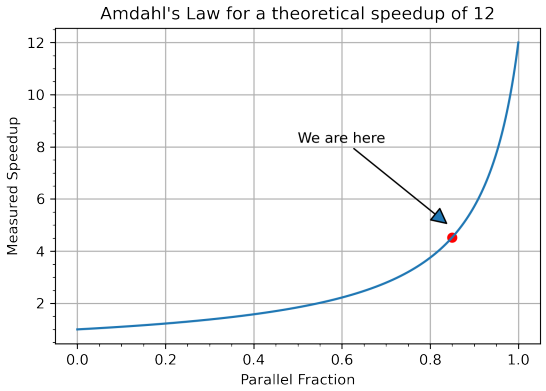

Fig. 5: Representação da Lei de Ahmdal

Pela figura 4 conseguimos verificar que por volta de 85% do código é paralelizado. Para demonstrar o motivo pelo qual uma percentagem tão elevada de código paralelizado apenas origina um ganho de 4.5 (onde o teórico é de 12), analisemos a Lei de Amdahl (figura 5):

Pela figura ao lado, podemos concluir que com um aumento da fração paralelizada do código para além dos 0.85 (ou 85%), 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.

4.2. Weak scaling

Para uma análise do *weak scaling*, ao invés de se variar o número de processos mantendo o tamanho total do array fixo, será mantido fixo o tamanho da porção do array que é enviada para cada processo. Desta forma, a ”carga” por processo na ordenação deverá ser fixa. A figura 6 representa a Eficiência $E(p)$ do *weak scaling* com o número de processos p:

$$E(p) = \frac{t(1)}{t(p)}, \quad \delta E(p) = \sqrt{\left(\frac{1}{t(p)}\delta t(1)\right)^2 + \left(\frac{t(1)}{t^2(p)}\delta t(p)\right)^2} \tag{4}$$

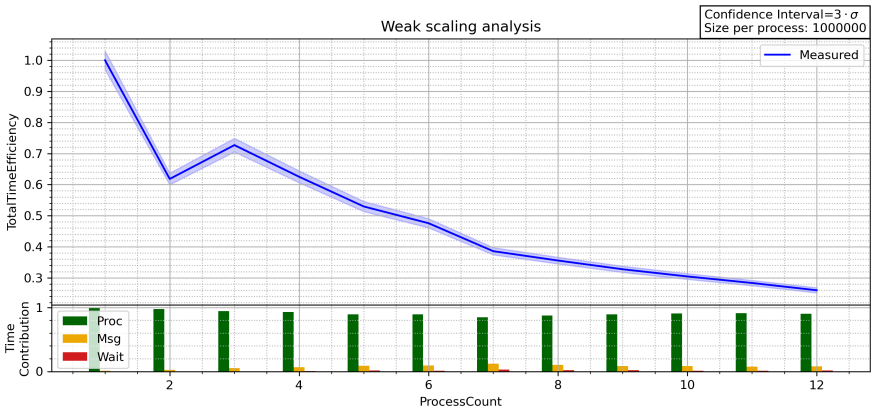

Fig. 6: Análise de *Weak scaling*

Como se pode verificar pela figura acima, a eficiência de *weak scaling* decresce consideravelmente com o aumento do número de processos.

Uma vez que a ”carga” por processo se deveria manter aproximadamente constante, e que na prática se verificou uma aumento significativo com o aumento de processos, conclui-se que o aumento deste tempo de execução se deve à distribuição dos elementos pelos processos, que é feita de forma sequencial e cujo tempo de execução aumenta linearmente com o aumento total do tamanho do array original.

Desta forma, otimizar a distribuição dos elementos do array pelos processos seria um fator de grande impacto para obter melhorias na escalabilidade fraca do algoritmo implementado.

Por fim, pode-se também verificar um aumento da percentagem de tempo ocupada pelo envio de mensagens, dado que o aumento do array original aumenta também a quantidade de informação que deve ser enviada.

4.3. Outras análises de escalonamento

Pela análise do tempo total com o tamanho do array original, o tempo de execução do algoritmo (para um número fixo de processos) é linear com o tamanho do array. Este aumento é na maior parte devido à distribuição dos elementos do array pelos processos e também à ordenação de cada processo, que se tornam mais custosas com o aumento do tamanho total do array.

É também possível de se verificar que o algoritmo paralelo (com 12 processos) tem uma melhor escalabilidade com o tamanho total do array do que o algoritmo sequencial.

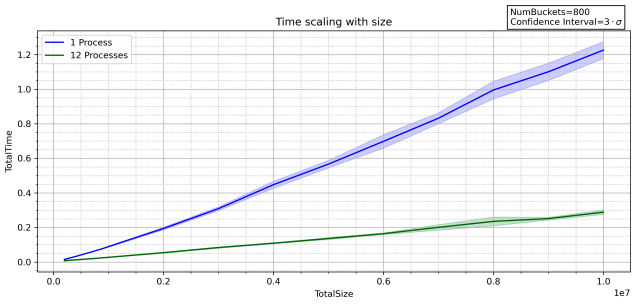

Fig. 7: Tempo de execução com o tamanho do array

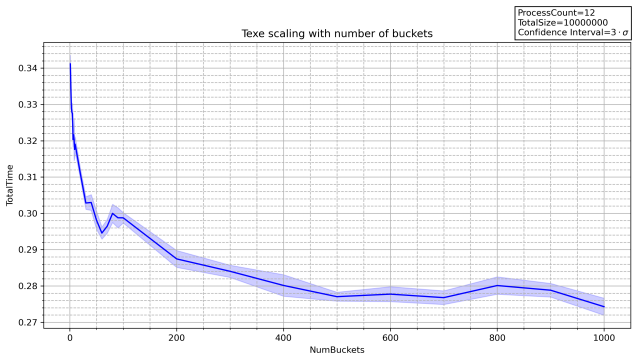

Fig. 8: Tempo de execução com o número de baldes

Pela análise da figura 8, pode-se concluir que o tempo de execução diminui com o aumento do número de baldes, atingindo o seu valor mínimo com 1000 baldes. Estes valores estão de acordo com o esperado, dado que um maior número de baldes implica um menor tempo de ordenação de cada balde.

5. Conclusão

Primeiramente, conseguiu-se diminuir o tempo de execução da nossa implementação do **Bucket Sort** em cerca de 4.5 vezes, comparando a sua versão sequencial à versão paralelizada com o `MPI`.

Analisou-se que o ganho obtido (4.5 vezes) é muito inferior ao ganho teórico (12 vezes). Utilizando a Lei de Amdahl, pôde-se concluir que esta disparidade se deve à percentagem de código paralelizado (por volta de 85%). Um aumento nesta percentagem iria aumentar significativamente o ganho obtido em tempo de execução.

Relativamente aos tempos de execução, conclui-se que a sua maior contribuição é o tempo de processamento. Foi também possível observar que o tempo de mensagem aumentou consideravelmente com o aumento do tamanho total do array, dado o aumento da quantidade de informação partilhada entre processos por mensagens.

Verificou-se também pela análise do *weak scaling* do algoritmo que este veria melhorias significativas no seu escalonamento com a otimização da distribuição dos elementos do array pelos processos.

Concluiu-se também que o algoritmo paralelo desenvolvido escala 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 também que, tal como esperado, este algoritmo tem uma melhor performance com o aumento do número de baldes, que resulta na diminuição dos elementos por balde e consequentemente também uma diminuição no tempo total de ordenação.