Otimização de um algoritmo de ordenação

Computação Paralela

Gilberto Cunha
Tomás Carneiro

7 de junho de 2021

full code here

1. Introdução

Ao longo da UC de Computação Paralela, foram-nos apresentadas várias formas de otimizar e comparar algoritmos utilizando ferramentas de medição de métricas.

Este trabalho consiste na aplicação destas ferramentas aprendidas a um caso prático de um algoritmo de ordenação, o algoritmo **Bucket Sort**.

2. Bucket Sort

O **Bucket Sort** é um algoritmo de ordenação que pode ser descrito nos seguintes passos:

1. Distribuir os elementos do array original por diversos ”baldes”. Desta forma cada balde deverá ter um array mais pequeno que o original.
2. Ordenar os arrays de cada um dos baldes
3. Voltar a juntar os elementos (agora ordenados) de cada um dos baldes num array ordenado (que é o array original ordenado)

Abaixo, podemos observar um diagrama ilustrativo do seu funcionamento:

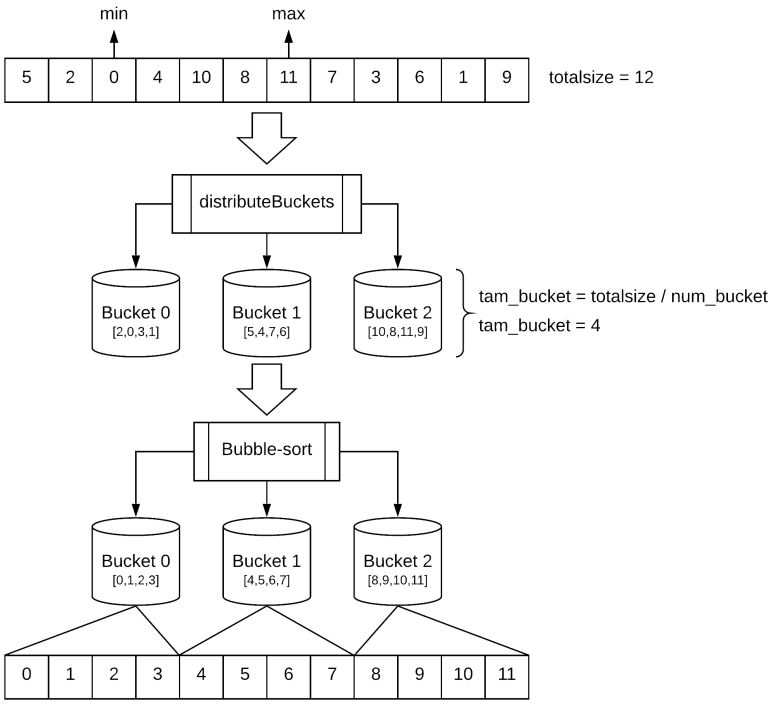

Fig. 1: Bucket Sort Algorithm representation

Este algoritmo traz vantagens dado que o tempo médio de ordenação $T(n)$ de um array de tamanho n é tipicamente acima de um tempo linear, isto é, $T(n) > \mathcal{O}(n)$. Desta forma, um aumento linear no número de baldes trará um decréscimo acima de linear no tempo de ordenação de cada balde, tornando a ordenação mais rápida. Este aumento de rapidez requer, no entanto, que haja uma boa distribuição dos elementos do array pelos baldes (se ficarem todos no mesmo balde não haverá melhoria), o que exige que se tenha alguma informação à priori relativa à distribuição dos elementos do array original (distribuição uniforme, normal, etc).

O algoritmo base utilizado encontra-se disponível em [1]. Foram, no entanto, feitas certas alterações ao código original.

Em primeiro lugar, o código foi limpo de forma a torná-lo mais fácil de ler.

O código foi também separado em diferentes ficheiros, de forma a isolar o algoritmo **Bucket Sort** completo das suas partes constituintes, as quais foram reimplementadas de várias formas distintas com o objetivo de otimizar o seu funcionamento original.

A distribuição dos elementos do array pelos baldes assumia uma distribuição **uniforme** entre 0 e um número máximo definido como parâmetro do programa, o que é uma distribuição algo restritiva. Para este algoritmo ser mais geral, este passo foi modificado para aceitar uma distribuição uniforme entre qualquer gama de valores.

Por fim, modificou-se o uso de arrays em cada balde e substituiu-se por apontadores, de forma a evitar erros de memória insuficiente para ordenação de arrays de elevada dimensão.

3. Otimizações

3.1. v1.0 - Distribuição de elementos pelos baldes

De forma a colocar cada um dos elementos num balde, e assumindo uma distribuição **uniforme** destes elementos, são colocadas gamas de valores para cada balde. No caso em que temos b baldes, um elemento máximo M e um elemento mínimo m, então o i-ésimo balde ($i \in \{0, 1, \ldots, b-1\}$) deverá aceitar valores na gama $\left[m + i \cdot \frac{M-m+1}{b}, m + (i+1) \cdot \frac{M-m+1}{b} - 1\right]$.

No algoritmo original, estas gamas são pesquisadas sucessivamente, desde a gama do último balde até ao primeiro, de forma a determinar o balde adequado para cada elemento do array:

```
for(i=0; i<tam; i++) {
    j=num_bucket-1;
    while (1) {
        if(j<0) break;
        if(v[i] >= j * ((max - min + 1) / num_bucket)) {
            b[j].balde[b[j].topo]=v[i];
            (b[j].topo)++;
            break;
        }
        j--;
    }
}
```

Para além deste código ser difícil de se ler, é necessário, para cada elemento, iterar o ciclo `while` até encontrarmos o balde adequado.

A nossa alternativa foi verificar que o balde $\hat{b}$ de cada elemento v pode ser calculado diretamente através de uma única operação, $\hat{b} = v \ // \ \frac{M-m+1}{b}$, onde a operação $//$ representa a *divisão inteira*, removendo assim a necessidade do ciclo `while` interior:

```
n = (max - min + 1) / num_bucket;
for(i=0; i<tam; i++) {
    aux = v[i];
    j = aux / n;
    b[j].balde[b[j].topo++] = aux;
}
```

O impacto desta otimização não deverá ser muito grande no tempo total de execução, uma vez que este não é o passo mais demorado do algoritmo (como veremos na subsecção seguinte). De qualquer forma, a remoção do ciclo `while` interior deverá **reduzir ligeiramente o número de intruções e o tempo de execução** do **Bucket Sort**.

3.2. v1.1 - Redução do número de iterações do Bubble Sort

Utilizando a ferramenta `perf`, e correndo a versão **v1.0** do algoritmo com o comando `perf record [executável]`, seguido de uma análise com o comando `perf annotate`, foi possível determinar que o passo mais lento neste algoritmo é a *ordenação de cada um dos baldes*, que utiliza o algoritmo de ordenação **bubble sort**. Deste modo, boas otimizações neste algoritmo de ordenação deverão levar a boas melhorias de desempenho do nosso algoritmo geral.

O bubble sort original está implementado da seguinte forma:

```
void bubble (int v[], int tam) {
    int i,j,temp,flag;
    if (tam) {
        for(j=0; j<tam-1; j++) {
            flag=0;
            for(i=0; i<tam-1; i++) {
                if(v[i+1]<v[i]) {
                    temp=v[i];
                    v[i]=v[i+1];
                    v[i+1]=temp;
                    flag=1;
                }
            }
            if(!flag) break;
        }
    }
}
```

Repare-se no seguinte pormenor: após um ciclo (exterior) do bubble sort, o último elemento do array já se encontra na posição correta, logo esta já não precisa ser percorrida no ciclo interior. Na próxima iteração o mesmo irá acontecer, e assim sucessivamente. Deste modo, podemos limitar a variável `i` superiormente não por `i<tam-1`, mas por `i<tam-j-1`. Esta alteração reduz aproximadamente para metade o número de ciclos interiores executados, sendo assim, em princípio, uma **grande redução no número de instruções e tempo de execução** do nosso algoritmo.

Para além desta otimização, a verificação do tamanho do array, efetuada antes da execução do ciclo exterior da função bubble, é agora realizada aquando da inicialização do próprio array.

A primeira otimização do bubble sort é, então, correspondente à seguinte implementação:

```
void bubble1 (int v[], int tam) {
    int i,j,temp,flag;
```

```
    for(j=0; j<tam-1; j++) {
        flag=0;
        for(i=0; i<tam-j-1; i++) {
            if(v[i+1]<v[i]) {
                temp=v[i];
                v[i]=v[i+1];
                v[i+1]=temp;
                flag=1;
            }
        }
        if(!flag) break;
    }
}
```

3.3. v1.2 - Hierarquia de memória no Bubble Sort

Quando num programa acedemos a uma posição de memória, o processador envia um bloco de memória que *começa na posição de memória acedida* para cache, para agilizar o uso de posições de memória seguintes.

Deste modo, na condição `v[i+1]<v[i]` da versão **1.2**, ao fazer load da posição `i+1` do array `v`, vai ocorrer uma *cache miss* ao aceder à posição `i`.

Para evitar estes *cache misses* pode-se simplesmente inverter a ordem desta condição para `v[i]>v[i+1]`, **melhorando ligeiramente a *localidade espacial* do programa**.

Esta alteração deverá diminuir mais no número de *cache misses* para arrays em cada balde de maior dimensão, não tendo, no entanto, um grande impacto no desempenho do algoritmo. Quando o número de inteiros no array de cada balde passar a ser inferior ao número de inteiros que se pode armazenar em cache, então não haverão cache misses nesta secção do código, sendo que o número de cache misses totais deverá reduzir drasticamente.

O ciclo da função `bubblesort` irá então ter a estrutura abaixo:

```
...
for(j=0; j<tam-1; j++) {
    flag=0;
    for(i=0; i<tam-j-1; i++) {
        if(v[i]>v[i+1]) {
            ...
```

3.4. v1.2v - Vectorização da função Bubble Sort

Em **C**, é possível indicar que uma variável **deve ficar armazenada em registo**, diminuindo desta forma o número de acessos à memória. Para além disso, instruções vetoriais juntam várias intruções em instruções únicas, sendo que **deverá diminuir o número de instruções total** do algoritmo. No entanto, não se sabe à priori se esta diminuição de instruções irá levar a uma redução do tempo de execução.

Relativamente à versão anterior, a função que executa o bubble sort terá apenas de utilizar um tipo de dados `__restrict__` e de conter mais duas intruções `C` acima da função:

```
#pragma GCC target("arch=znver2")
#pragma GCC optimize("tree-vectorize")
```

A versão **v1.2.1v** é em tudo idêntica a esta, mas vetoriza todo o algoritmo do Bucket Sort, e não apenas o Bubble Sort.

3.5. v1.3 - Substituição do algoritmo de ordenação Bubble Sort pelo Merge Sort

De forma a podermos analisar a execução deste programa de uma perspetiva diferente, implementámos o algoritmo Merge Sort, cujo funcionamento se baseia na famosa técnica da ”divisão e conquista”.
Este algoritmo divide o array inicial em vários sucessivamente mais pequenos, ordenando-os depois de forma recursiva.

4. Métricas e discussão de resultados

Os resultados apresentados são todos para um array com 10000000 de elementos e com números de 0 a 999999. Foi feito um estudo para um número variável de baldes.

Todas as métricas apresentadas foram obtidas a partir de uma média de 5 execuções do algoritmo **Bucket Sort**, alterando a disposição dos elementos do array inicial de forma aleatória. Preferencialmente seria feita a média a partir de muitas mais execuções, mas isso implicaria muito tempo de testagem. Os testes foram feitos utilizando um Ryzen 5 3600.

4.1. 10 Baldes

Tal como esperado, o número de instruções vai diminuindo com cada uma das versões implementadas. Verificou-se também uma descida muito ligeira no número *cache misses* entre a versão v1.1 e v1.2 devido à melhoria da localidade espacial. O tempo de execução apenas não melhorou nas versões vetorizadas do algoritmo. Estes resultados estão, então, de acordo com as previsões da secção anterior, baseadas no conteúdo teórico lecionado.

Versão	CC	#I	CPI	L1 Misses	Texe (μs)
v0	691849920512	772623171584	0.895	31270682363 (1.77%)	165465264
v1.0	692045807616	772627103744	0.896	31275336702 (1.76%)	165826976
v1.1	624826449920	625025351680	1.000	15628177966 (1.30%)	149603664
v1.2	622653472768	625025351680	0.996	15627398426 (1.29%)	148922752
v1.2v	669568008192	600019238912	1.116	15741922778 (1.40%)	161615840
v1.2.1v	647579369472	600015110144	1.079	15689291984 (1.47%)	155757616

Tab 1: Métricas para cada umas das versões usando 10 baldes

4.2. 100 baldes

Relativamente aos resultados anteriores, este aumento em 10x no número de baldes resultou numa diminuição geral em cerca de 10x do tempo de execução do **Bucket Sort**.

Analogamente ao caso anterior, o número de instruções e tempo de execução diminuiu de versão para versão exceto nas versões vetorizadas (também aumentou da versão v1.1 para v1.2, mas a diferença é pouca, podendo refletir apenas o baixo número de ensaios).

Versão	CC	#I	CPI	L1 Misses	Texe (μs)
v0	47597662208	76939591680	0.619	3125222973 (2.20%)	11422450
v1.0	47469797376	76656754688	0.619	3127746615 (2.20%)	11406080
v1.1	29935742976	62523387904	0.479	431725761 (0.48%)	7202907
v1.2	30685751296	62523387904	0.491	432404288 (0.47%)	7347083
v1.2v	62638120960	60017631232	1.044	489495437 (0.45%)	15174531
v1.2.1v	33220055040	60013502464	0.554	458925095 (0.60%)	8025817

Tab 2: Métricas para cada umas das versões usando 100 baldes

4.3. 1000 baldes

Novamente, de versão para versão notam-se as mesmas melhorias que para o caso dos 10 e 100 baldes. Desta vez, no entanto, as versões vetorizadas já estão bastante mais próximas das versões não vetorizadas em termos de tempo de execução.

Versão	CC	#I	CPI	L1 Misses	Texe (μs)
v0	4960982528	10513360896	0.472	18238839 (0.13%)	1196120
v1.0	4392975872	7514360832	0.585	21656447 (0.16%)	1054128
v1.1	2649972736	6273888256	0.422	15552019 (0.18%)	645708
v1.2	2659878912	6273888256	0.424	16683186 (0.19%)	642334
v1.2v	2978027264	6017953280	0.495	19291197 (0.26%)	721368
v1.2.1v	2857289216	6013846016	0.475	17815677 (0.26%)	684138

Tab 3: Métricas para cada umas das versões usando 1000 baldes

Mais uma vez, com um aumento de 10x do número de baldes viu-se uma redução em cerca de 10x do tempo de execução do algoritmo.

Repare-se também que a percentagem de *cache misses* desceu muito da versão com 100 baldes para esta versão com 1000 baldes. Com 100 baldes, cada balde tinha 10000 elementos, enquanto agora apenas tem 1000. O processador utilizado tem 36Mb de Cache, sendo que consegue armazenar na totalidade $36 \cdot 1024/4 = 9216$ inteiros. Deste modo, usando 1000 baldes todos os elementos do array de cada balde podem ficar armazenados na memória, não havendo cache misses no bubble sort.

4.4. MergeSort

Quando se compara a utilização deste algoritmo de ordenação com complexidade *assimtótica* de $\mathcal{O}(n \log n)$ com o bubble sort, de complexidade assimtótica média de $\mathcal{O}(n^2)$, verifica-se que o *Merge Sort* é, de facto, o mais eficiente dos dois. Este resultado não é tão óbvio quando se passa para um número maior de baldes (e consequentemente um menor número de elementos por balde), pois a análise assimtótica já não se aplica à complexidade dos algoritmos (e de facto o merge sort é mais lento que o bubble sort para 100000 baldes), mas na maioria dos casos o merge sort mostrou-se à mesma ser a forma mais eficiente de efetuar esta ordenação.

Versão	Num Buckets	CC	#I	CPI	L1 Misses	Texe (μs)
v1.2	10	622653472768	625025351680	0.996	15627398426 (1.29%)	148922752
v1.2	100	30685751296	62523387904	0.491	432404288 (0.47%)	7347083
v1.2	1000	2659878912	6273888256	0.424	16683186 (0.19%)	642334
v1.2	10000	564812608	653443456	0.864	40663142 (2.60%)	148862
v1.2	100000	330681376	128358040	2.576	159162437 (5.51%)	188688

Tab 4: Métricas do bubble sort

Versão	Num Buckets	CC	#I	CPI	L1 Misses	Texe (μs)
v1.3	10	442631072	746246144	0.593	39214999 (2.13%)	121119
v1.3	100	386850912	700021952	0.553	37051042 (2.21%)	106510
v1.3	1000	336912512	652398592	0.516	35818464 (2.26%)	94063
v1.3	10000	314010944	603401216	0.520	57877596 (3.45%)	97422
v1.3	100000	443504544	546822784	0.811	170862008 (4.60%)	222158

Tab 5: Métricas do merge sort

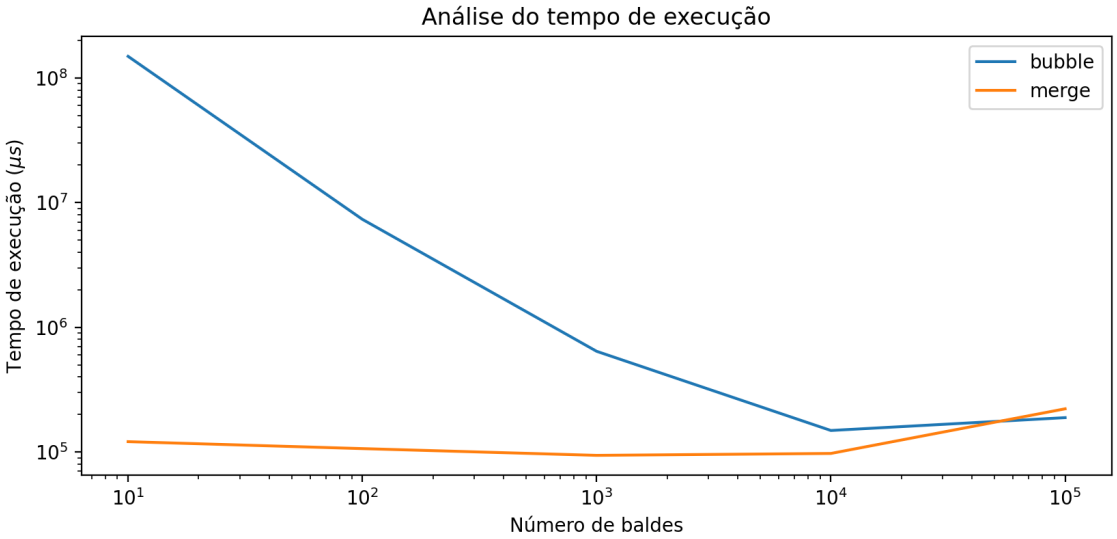

Fig. 2: Comparação dos tempos de execução para a ordenação de baldes com Bubble e Merge Sort

5. Conclusão

Conseguiu-se, com este trabalho, diminuir o número de instruções e reduzir o tempo de execução do **Bucket Sort** em até 2 vezes e diminuir ligeiramente o número de *cache misses* do programa, mantendo o algoritmo **Bubble Sort** para a ordenação dos baldes. Verificou-se também que a vetorização do código *não trouxe vantagens* para o desempenho deste algoritmo.

Com a modificação do algoritmo de ordenação dos baldes para o **Merge Sort**, verificaram-se ainda mais melhorias na redução de instruções e no tempo de execução do algoritmo, tendo este conseguido atingir, no melhor caso, uma *melhoria de 1000x no tempo de execução relativamente à melhor versão do Bubble Sort desenvolvida*. Embora para um maior número de baldes as melhorias reduzam gradualmente, o **Merge Sort** tem consistentemente um melhor desempenho que o **Bubble Sort**, com a excepção do caso em que se usou 100000 baldes.

Referências

[1] Bucket Sort, `https://pt.wikipedia.org/wiki/Bucket_sort`, Accessed: 2021-04-04.