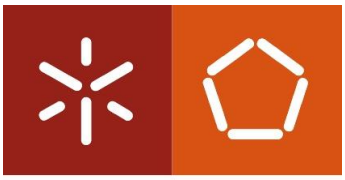

Universidade do Minho

Departamento de Informática

Paradigmas de Computação Paralela
Bucket Sort com OpenMP

João Teixeira (A85504)
José Filipe Ferreira (A83683)

6 de dezembro de 2020

Conteúdo

Capítulo 1

Introdução

O algoritmo escolhido para o projeto da unidade curricular de Computação Paralela e Distribuída foi o *Bucket Sort*.

Inicialmente desenvolvemos uma versão sequencial do projeto e procedemos ao *benchmarking* do programa resultante. Em seguida convertemos a implementação sequencial numa versão com utilização de memoria partilhada fazendo uso de *OpenMP* e comparamos o resultado com a versão sequencial desenvolvida anteriormente.

Ao longo deste relatório iremos descrever a metodologia utilizada e os resultados de *benchmarking* obtidos ao longo deste projeto.

De notar que todos os benchmarks descritos foram efeutados em nós do tipo 652 do cluster *SeARCH*, e todos os executáveis foram compilados com a versão 7.2.0 do *gcc*, com as flags *-O3 -ftree-vectorize -fopenmp -std=c11* para garantir a maior consistência entre os diferentes benchmarks. Na ausência de indicação, os benchmarks foram efetuados com um input de 100000000 de elementos aleatórios, entre -1000 e 500000.

Capítulo 2

Sequencial

O *Bucket Sort* consiste em definir um conjunto de N "baldes"inicialmente vazios. Em seguida os elementos do vetor a ser ordenado são distribuídos pelos baldes. O critério escolhido para esta distribuição foi calcular o máximo e o mínimo do vetor a ser ordenado e dividir os intervalos de valores de cada balde em intervalos do mesmo tamanho. Os elementos do vetor inicial são então distribuídos pelo respetivo balde com base no intervalo definido. Seguidamente o conteúdo de cada balde é ordenado recorrendo ao *quicksort* presente na *standard library* de C. Finalmente todos os elementos são copiados um a um para o vetor original.

Para testar se a primeira implementação sequencial produzia de facto vetores ordenados criamos um *script* que permite gerar N testes aleatórios e comparar o resultado do nosso programa com o resultado de ordenar os valores com o comando *sort* de *bash*.

Numa implementação inicial desta versão sequencial, o algoritmo consistia em iterar sobre o array dos elementos a ordenar e inserir o elemento a processar no respetivo balde. Durante a paralelização deste, identificamos alguns problemas, que serão descritos em maior pormenor no capitulo 3, que nos levou a tomar uma abordagem diferente, em vez de iterar pelo array uma vez e colocar os elementos no respetivo balde, cada balde irá iterar pelo array, escolhendo os elementos que a ele lhe pertencem.

Fazer isto aumentou a complexidade do algoritmo, passando de iterar uma vez pelo array, para N vezes, onde N é o número de baldes existentes. Como consequência disto também os tempos de execução aumentarem.

Capítulo 3

OpenMP

Fazendo uso da implementação sequencial descrita no capitulo anterior, começamos a conversão para uma versão com um modelo de memoria partilhada fazendo uso de *OpenMP*.

De modo a paralelizar a primeira versão sequencial, começamos por tratar do ciclo mais longo que itera por todos os elementos do *array*. Ao fazer uma analise mais detalhada do algoritmo reparamos na existência de uma zona critica. Esta zona critica ocorria quando era escrito cada elemento no respetivo balde. Com o objetivo de tentar resolver este problema tentamos duas abordagens distintas: a utilização de tarefas e a declaração da zona critica. Em ambas as opções obtivemos resultados piores quando comparados com a sua versão sequencial, pois tanto a gestão de zonas criticas como a gestão de tarefas é muito pesado, como visto nas duas primeiras colunas da tabela A.1.

Após analizar-mos estes resultados tentamos adaptar o algoritmo para um formato mais paralelizável através da remoção da zona critica. Nesta nova versão verificamos que os tempos de execução paralelos estavam a baixar, obtendo speedups consideráveis até às 16 *threads*. Embora o speedup esteja ligeiramente abaixo do ideal e se afastar do valor quanto maior o numero de *threads*. Isto deve-se ao número de elementos por balde não ser uniforme e o tempo de execução só conseguir ser tão baixo quanto o balde mais lento de processar, como se pode ver na tabela A.6, onde com um desequilíbrio extremo quase não existe speedup. Por isso, quanto maior o numero de *threads* em trabalho mais se notará esse desequilíbrio.

Para tentar mitigar este problema tentamos utilizar o colapsamento dos dois ciclos aninhados, o que percorre os baldes e o interior que percorre o array dos elementos. Para manter a escrita no balde dos elementos em paralelo a atualização do índice de escrita no balde era feita de forma atómica. Nesta implementação, a escalabilidade do algoritmo provou-se bastante inferior, começando a diminuir a partir das 4 *threads*, mantendo a tendência até às 32 *threads*, como visto na ultima coluna da tabela A.1.

Para tentar tornar o algoritmo ainda mais escalável decidimos variar o numero de baldes. Numa primeira abordagem testamos com um número de baldes constante para vários números de *threads*. Não encontramos nenhum valor que fosse o melhor para todos o número de *threads*, mas vimos que quantas mais *threads* utilizadas, mais benéfico era o maior número de baldes, visto na tabela A.2.

Perante esta relação tentamos testar com um valor de baldes dependente do número de *threads* utilizadas. Aqui vimos que, regra geral, utilizar o dobro do número de *threads* garantia o maior resultado, mas tendo em conta as características do nosso algoritmo, nomeadamente, o facto de por cada balde é feita uma passagem pelo array de elementos a ordenar, a partir das 32 *threads* o aumento da carga de trabalho torna-se tal que o programa começa a ficar mais lento, como visto na tabela A.3.

Juntando a informação obtida em ambos os testes, implementamos um meio termo entre ambas as abordagens, utilizando o dobro do número de *threads* para o número de baldes, e colocando um teto máximo de 32 baldes, valores para o qual obtivemos os melhores resultados médios até às 40 *threads*, visto na tabela A.4.

Apêndice A

Benchmarking

Times				
Threads	Old Critical	Old Task	New	New Collapsed
Seq	16,209999	16,209999	22,299999	22,299999
2	43,23	305,169983	11,403177	11,968859
4	100,150002	937,929993	6,880816	6,872915
8	187,929993	2227,460205	4,622566	5,965684
16	479,750031	5547,610352	2,362948	4,678829
32	1213,959961	12624,78906	2,363651	4,859851
64	1726,070068	11589,56934	2,449052	4,059251

Tabela A.1: Tempos de execução iniciais, com 10 baldes

Threads	5 buckets	10 buckets	20 buckets	40 buckets	80 bucket
2	12.520761	11.218051	12.921473	17.016363	25.398283
4	8.365767	6.739361	6.511606	8.512317	12.743460
8	5.218658	4.984042	3.952061	4.323404	6.438317
16	5.344770	3.167999	3.042659	2.632886	3.465635
32	5.248249	3.147622	2.148584	2.392601	2.836905
64	5.270648	3.161402	2.366316	2.002006	3.260201

Tabela A.2: Tempos de execução, fazendo variar o número de baldes utilizados

Threads	n_threads	2*n_threads	4*n_threads	8*n_threads	16*n_threads
2	11.395469	10.310162	10.808166	12.206050	15.356203
4	6.129920	5.490817	6.108120	7.681805	11.012663
8	3.719489	3.100843	3.883367	5.560686	9.027642
16	2.217584	2.001313	2.854720	4.599587	8.157944
32	1.824530	2.292776	3.511230	6.276313	11.743461
64	2.486487	3.223805	5.932982	10.554493	20.222078

Tabela A.3: Tempos de execução, fazendo variar o número de baldes, com base no número de threads.

Threads	30	32	34	36	38	40
16	2.085598	2.008397				
32	1.846964	1.813501	2.572607	2.442342	2.408187	2.349852
40	1.907169	1.799508	1.779662	1.718217	1.788759	1.881975
64	2.599405	2.536593	2.106488	1.911450	1.976940	1.977957

Tabela A.4: Tempos de execução, com o número de baldes a ser o dobro do número de threads, e variando o número máximo de baldes utilizados.

Threads	Time	Speedup
Seq	20,629999	1
2	10,346263	1,993956562
4	5,57325	3,701610192
8	3,288632	6,273124813
16	2,199246	9,380487222
32	1,677771	12,29607557
64	1,814721	11,36813813

Tabela A.5: Tempos finais de execução e respetivo speedup, com a versão sequencial a utilizar 4 baldes.

Threads	Time	Speedup
Seq	1,57	1
2	1,485135	1,057142953
4	1,49086	1,053083455
8	1,498286	1,047864026
16	1,504521	1,043521493
32	1,45915	1,075968886
64	1,456144	1,078190069

Tabela A.6: Tempos de execução para 10000000 elementos aleatórios entre -100 e 100, com um elemento sendo 500000